Comparative Genomics in Distant Taxa

Generating Total Orders of Digraphs

Der Fakultät für Mathematik und Informatik
der Universität Leipzig
angenommene

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

Doctor rerum naturalium
(Dr. rer. nat.)

im Fachgebiet

Informatik

vorgelegt von

Master of Science Informatik Fabian Gärtner (geb. Externbrink)
geboren am 08.06.1988 in Herdecke

Die Annahme der Dissertation wurde empfohlen von:

1. Prof. Dr. Peter F. Stadler, Universität Leipzig
2. Prof. Dr. Burkhard Morgenstern, Georg-August-Universität Göttingen

Die Verleihung des akademischen Grades erfolgt mit Bestehen der Verteidigung am 18. Februar 2020 mit dem Gesamtprädikat magna cum laude

To Christiane and our children Janosch,
Mathilda, and Lioba.
You are my life.

Bibliographic Description

Title:	Comparative Genomics in Distant Taxa
Subtitle:	Generating Total Orders of Digraphs
Type:	Dissertation
Author:	Fabian Gärtner
Year:	2020
Professional discipline:	Computer Science
Language:	English
Pages in the main part:	143
Chapter in the main part:	6
Number of Figures:	59
Number of Tables:	28
Number of Appendices:	4
Number of Citations:	168
Key Words:	genome assembly, supergenome, superbubble, DFS, graph ordering, vertex ordering

This thesis is based on the following publications.

F. Gärtner, C. Höner zu Siederdissen, L. Müller, and P. F. Stadler (Sept. 2018). "Coordinate Systems for Supergenomes". In: *Algorithms for Molecular Biology* 13.1, p. 15. *DOI*: 10.1186/s13015-018-0133-4.

F. Gärtner, L. Müller, and P. F. Stadler (Dec. 2018). "Superbubbles revisited". In: *Algorithms for Molecular Biology* 13.1, p. 16. *DOI*: 10.1186/s13015-018-0134-3.

F. Gärtner and P. F. Stadler (Apr. 2019). "Direct Superbubble Detection". In: *Algorithms* 12.4. *ISSN*: 1999-4893. *DOI*: 10.3390/a12040081.

Abstract

Genome sequences and genome annotation data have become available at ever-increasing rates in response to the rapid progress in sequencing technologies. As a consequence, the demand for methods supporting comparative, evolutionary analysis is also rising. In particular, adequate tools to visualize omics data simultaneously for multiple species are sorely lacking. A first and crucial move in this direction is the construction of a common coordinate system. Since genomes not only vary by rearrangements but also by massive insertions, deletions, and duplications, the use of a single reference genome is insufficient, in particular when the number of species becomes extensive. The computational problem then becomes to define order and orientations of optimal local alignments that are as co-linear as possible within all the genome sequences of the alignment. This problem is identified as the supergenome sorting problem.

The formulation of the supergenome sorting problem as a formal task is not straight forward because the common problem definitions for creating orders are not sufficient for the supergenome sorting problem. Furthermore, published specific solutions of the supergenome sorting problem do not take into account distant taxa and are insufficient for them. In this thesis, the supergenome sorting problem is solved with a novel betweenness based approach. This NP-hard problem is named Directed Colored Multigraph Betweenness Problem. The betweenness backbone is very robust against the effects of distant taxa. Thus, it is the first approach that is suitable for them.

Exact solutions are beyond reach for the problem sizes of practical interest. Thus, the thesis uses a pipeline of advanced heuristics where each heuristic addresses different aspects of the ordering problem. The pipeline can be decomposed in three main steps: First, filtering is performed to reduce noise; then, a graph is created out of the data which is transformed into an acyclic graph; the last step is then to do a specific topological sorting on the acyclic graph to create an order. The resulting order is further optimized in post-processing.

Real-life data ranging from bacterial to fly genome alignments are used to verify the results. Because the ground truth is not known, benchmarking is done by comparing the properties of the results. The results demonstrate the feasibility of the new approach to compute sound common coordinate systems. Even for the distant taxa in the fly genome alignments, the results are promising. Thus, it is now for the first time possible to create a plausible common coordinate system

for distant taxa which make downstream comparative, evolutionary analysis much more accessible.

The methodic of the new approach is oriented on a related problem: the genome assembly problem. In this well-studied problem, the task is to order small pieces of a genome to reconstruct the whole genome. With the guidance of alignments, a graph is constructed that connects adjacent segments of the genome. Then this graph is ordered with the help of heuristics. It is reasonable to represent the supergenome sorting problem as an equivalent graph. The only variation to the genome assembly problem is how detected ambivalences are solved.

Most approaches that seek to solve the genome assembly problem utilize heuristic simplifier. These simplifiers use local data to restrict the ordering in components of the graph. One type of simplifiers considers bubble-like structures. A bubble-like structure is a co-linear subgraph. These subgraphs can be transformed in linear subgraphs. Thus they are crucial for creating a unique order.

One bubble-like structure of special interest is the superbubble. Superbubbles are distinctive subgraphs in digraphs that are connected to their host graph by one particular entrance and one particular exit vertex, thus allowing them to be handled independently. This structure has many improvements over other bubble-like structures in the genome assembly problem and especially in the supergenome sorting problem. However, it is not well studied and the existing studies contain some remarkable mistakes.

This thesis represents the first in-depth theoretical analyses of superbubbles. It gathers significant results from the literature and contains numerous new results. Especially the thesis corrects some mistakes published in literature that can lead to the reporting of false-positive superbubbles. Furthermore, it describes a more straightforward, space-efficient linear-time algorithm for detecting superbubbles that only uses simple data structures.

The new approach uses in principle, a single depth-first search (DFS), provided one can guarantee that the root of the DFS is not itself located in the interior of a superbubble or is the exit vertex of a superbubble. A linear-time algorithm to determine suitable roots for a DFS-forest that is guaranteed to identify the superbubbles in a digraph correctly is described. In addition to the advantages of a more straightforward implementation, in almost all data sets performance gain of one order of magnitude can be observed.

The approach of detecting superbubbles is available for the scientific community in two forms. Firstly, a reference implementation that makes it easy to understand how the theoretical results are used to create a working program. Secondly, an optimized program that is harder to understand but performs very well. This program is an out of the box solution for detecting superbubbles. Both programs are available as open-source from GitHub.

Zusammenfassung

Genomsequenzen und Genomannotationsdaten sind aufgrund des raschen Fortschritts der Sequenzierungstechnologien mit immer höheren Raten verfügbar geworden. Infolgedessen steigt auch die Nachfrage nach Methoden, die eine vergleichende evolutionäre Analyse unterstützen. Insbesondere fehlen geeignete Werkzeuge, um Omics-Daten gleichzeitig für mehrere Spezies zu visualisieren. Ein erster und entscheidender Schritt in diese Richtung ist die Erstellung eines gemeinsamen Koordinatensystems. Da Genome nicht nur durch Translokation, sondern auch durch massives Einfügen, Löschen und Verdoppeln von Teilen variieren, ist die Verwendung eines einzelnen Referenzgenoms unzureichend, insbesondere wenn die Anzahl der Spezies umfangreich wird. Das Problem besteht dann darin, die Ordnung und Ausrichtung optimaler lokaler Sequenzalignment zu definieren, die innerhalb aller Genomsequenzen des Sequenzalignments so kollinear wie möglich sind. Dieses Problem wird als das Supergenom-Sortierproblem bezeichnet.

Die Formulierung des Supergenom-Sortierproblems als formales Problem ist nicht einfach, da die allgemeinen Problemdefinitionen zum Erstellen von Ordnungen für das Supergenom-Sortierproblem nicht ausreichend sind. Darüber hinaus berücksichtigen veröffentlichte spezifische Lösungen des Supergenom-Sortierproblems entfernte Taxa nicht und sind daher für sie unzureichend. In dieser Arbeit wird das Problem der Supergenom-Sortierung mit einem neuartigen Ansatz auf der Basis von Betweenness gelöst. Dieses NP-harte Problem heißt "Directed Coloured Multigraph Betweenness Problem". Die meisten Auswirkungen entfernter Taxa haben keinen großen einfluss auf Betweenness. Darum ist es der erste Ansatz, der für Sequenzalignments mit entfernten Taxas geeignet ist.

Genaue Lösungen sind für Problemgrößen von praktischem Interesse unberechenbar. Die Dissertation verwendet daher eine Kette fortgeschrittener Heuristiken, bei der jede Heuristik unterschiedliche Aspekte des Ordnungsproblems behandelt. Der Ansatz kann in drei Hauptschritte zerlegt werden: Zuerst wird eine Filterung durchgeführt, um das Rauschen zu reduzieren. Anschließend wird aus den Daten ein Graph erstellt, der in einen azyklisches Graph umgewandelt wird. Der letzte Schritt ist dann, eine spezifische topologische Sortierung des azyklischen Graphen vorzunehmen, um eine Ordnung zu erstellen. Die resultierende Ordnung wird in der Nachbearbeitung weiter optimiert.

Zur Überprüfung der Ergebnisse werden reale Daten verwendet, deren Komplexität von Bakterien- bis zu Fliegen-Sequenzalignments reicht. Da die wahre Evolution nicht bekannt ist, erfolgt das Benchmarking durch Vergleichen der Eigen-

schaften der Resultate. Die Ergebnisse zeigen die Machbarkeit solider gemeinsamer Koordinatensysteme mit der neuen Methode. Selbst für die entfernten Taxa in dem Fliegensequenzalignment sind die Ergebnisse vielversprechend. Damit ist es erstmals möglich, ein plausibles gemeinsames Koordinatensystem für entfernte Taxa zu schaffen, das die nachgelagerte vergleichende, evolutionäre Analyse wesentlich einfacher macht.

Die Methodik des neuen Ansatzes orientiert sich an einem verwandten Problem: dem Genom-Assemblierungs-Problem. In diesem gut untersuchten Problem besteht die Aufgabe darin, kleine Teile eines Genoms zu ordnen, um das gesamte Genom zu rekonstruieren. Unter Verwendung von Sequenzalignments wird ein Graph konstruiert, der benachbarte Teile des Genoms verbindet. Dann wird dieser Graph mit Hilfe der Heuristik geordnet. Es ist sinnvoll, das Supergenom-Sortierproblem als äquivalenten Graph darzustellen. Die einzige Änderung zum Genom-Assemblierungs-Problem besteht darin, wie erkannte Ambivalenzen gelöst werden.

Die meisten Ansätze zur Lösung des Genom-Assemblierungs-Problems, verwenden heuristische "Simplifier". Diese Simplifier verwenden lokale Daten, um die Ordnung in Teilen des Graphs einzuschränken. Eine Art von Simplifier betrachtet Bubble-Strukturen. Eine Bubble-Struktur ist ein kollinearer Teilgraph. Diese Teilgraphen können in lineare Untergraphen umgewandelt werden. Sie sind daher für die Erstellung einer eindeutigen Ordnung von entscheidender Bedeutung.

Eine Bubble-Struktur von besonderem Interesse ist die Superbubble. Superbubbles sind Teilgraphen, die durch einen bestimmten Eingangs- und einen bestimmten Ausgangsknoten mit ihrem übergeordneten Graphen verbunden sind, sodass sie unabhängig voneinander behandelt werden können. Diese Struktur hat viele Verbesserungen gegenüber anderen Bubble-Strukturen im Genom-Assemblierungs-Problem und insbesondere im Supergenom-Sortierproblem. Sie ist jedoch nicht gut untersucht und die vorhandenen Studien enthalten einige bemerkenswerte Fehler.

Diese Arbeit ist die erste gründliche theoretische Analyse von Superbubbles. Sie sammelt signifikante Ergebnisse aus der Literatur und enthält zahlreiche neue Ergebnisse. Insbesondere korrigiert sie einige in der Literatur veröffentlichte Fehler, die zu falsch erkannten Superbubbles führen können. Darüber hinaus wird ein unkomplizierterer, platzsparender, linearer Algorithmus zum Erkennen von Superbubbles beschrieben, der nur einfache Datenstrukturen verwendet.

Der neue Ansatz verwendet im Prinzip eine einzelne Tiefensuche (DFS), vorausgesetzt, man kann garantieren, dass sich die Wurzel der DFS selbst nicht im Inneren eines Superbubbles befindet oder der Ausgangsknoten eines Superbubbles ist. Ein linearer Algorithmus zur Bestimmung geeigneter Wurzeln für einen DFS-Wald, der die Superbubbles in einem Graphen garantiert korrekt identifiziert, wird beschrieben. Neben den Vorteilen einer einfacheren Implementierung ist in nahezu allen Datensätzen ein Leistungszuwachs von einer Größenordnung zu beobachten.

Der Ansatz zur Erkennung von Superbubbles ist für die wissenschaftliche Gemeinschaft in zwei Formen verfügbar. Erstens als Referenzimplementierung, die es einfach macht zu verstehen, wie die theoretischen Ergebnisse umgesetzt werden können. Zweitens als optimiertes Programm, das schwerer zu verstehen ist, aber sehr effizient funktioniert. Beide Programme sind als Open Source bei GitHub erhältlich.

Acknowledgment

First of all, I want to thank my supervisor Peter F. Stadler for all the support and patience that made this thesis possible. Also, I want to thank Lydia, who supported me as a mentor on my long way to write this thesis. Overall it was the right mix of fun and hard work that produces excellent results. A thank also goes to the people that proofread this work and made it at least a little readable: Lydia, Sarah, Sven, Berni, Steffi, and Tom.

I want to thank everybody of the Bioinformatics Lehrstuhl that makes this roller coaster ride enjoyable. Especially Petra that helps to survive the German madness of bureaucracy. Furthermore, Jens who was there to solve every problem that a system can think about. Under my colleagues, I want to point out some that help me in every phase of this ride: Tobi, Edith, Sarah, Milan, and Nancy. You are terrific people, thanks for all the great moments. Without you, I would not be considered sane anymore.

Last but not least, I want to thank my family: Christiane, Janosch, Mathilda, and Lioba. You take the challenge to fight over my time with this project. Even when you lost too often against it, you supported me further on my way. You give me the power to accomplish every hard step and finish it. You are the heroes of this story.

Contents

CHAPTER 1

Introduction

Contents

What is life?

George Harrison

Even though George Harrison has no biological intention when he asks this great question, biology tries to answer this question. There is no clear answer, and there are many possible definitions. Trifonov (2011) gives an overview providing the key point of life in a biological view:

- Life interacts with the environment
- Life is self-reproduction with variations

As far as humans know, in all life forms the same types of molecules are responsible for those functions. Proteins and ribonucleic acid (RNA) catalyze reactions and thus interact with the environment, where deoxyribonucleic acid (DNA) has the potential to self-reproduce.

1.1 Molecules of Life

DNA and RNA are very similar molecules. Both combine a sugar-phosphate backbone with different bases. This combination of sugar, phosphate, and base is called a nucleotide. The difference already indicated by the name, is that for DNA deoxyribose is used as a sugar and for RNA ribose. Regarding the bases, both have four different types. In DNA the following bases are used: cytosine (**C**), guanine (**G**), adenine (**A**), and thymine (**T**). In RNA thymine is replaced by uracil (**U**).

In both cases, the sugar-phosphate backbone linearly connects the bases. The ends of the chain can distinguished by the phosphate. By convention, DNA is read from the end with phosphate to the other. Thus simplified DNA and RNA can be described as a sequence of bases. For this, the one letter versions of the bases are used. An example would be "ATTC" which stands for a DNA molecule that starts with an adenine base, then two thymine bases, and at the end a cytosine base.

Most species use DNA to store the information. This DNA storage is called the genome. Every individual has one genome that is present in more or less every cell of it. This genome is unchanged mostly during the whole life. The key feature is the possibility to create copies of such a genome. This makes it possible to create a new cell from a cell which contains a copy of the genome.

This DNA copy mechanism makes use of the DNA structure. The DNA is not a single molecule in the cell most of the time. Two DNA molecules together form a double helix (Watson and Crick, 1953). The bases of the two strands (DNA molecules) interact in the double helix. Adenine matches with thymine and cytosine with guanine. Furthermore, the reading directions is inverse between the two strands.

The one to one relationship between the strands makes it possible to construct a strand from a template strand. In simplification copying is done by separating

the strands and then reconstructing the respective missing strand. The result is then two copies of the double helix.

From the genome, the information is transcribed into RNA. This RNA molecules can belong to different types. Broadly the types are: non-coding RNA and mRNA. The first type has many subtypes including, among others: tRNA, rRNA, snoRNA, and microRNA. All of these interact with other molecules either by structures or with sequence motives. The importance of structure has lead to advanced RNA secondary structure prediction tools (Lorenz, Bernhart, Externbrink, et al., 2012; Lorenz, Bernhart, Siederdissen, et al., 2011).

The second type, the mRNA, is translated into a protein. A protein is again a chain of smaller molecules, the amino acids. In the translation the sequence of bases determines the sequence of amino acids in the resulting protein. Proteins and especially the subcategory of enzymes also interact with other molecules and catalyze chemical reactions or change the shape of other molecules.

Thus life can be described by interacting with the environment by proteins and RNA. It further can self-reproduce itself by using copyable DNA as a genome. The associated components are shown in Figure 1 on the example of the heart.

1.2 Evolution

A last but essential part of the life definition is that the self-reproduction has variations, i.e., it is not entirely identical. This variation is significant for life. Thus, it can be changed and adapt to changes in the environment.

These variations are variations in genome copying process. Every time a genome is copied, small mistakes happen. Thus two individuals of the same species most likely do not have the same genome. Such a change in the genome is called a mutation. The mutations that change only one base are shown in Figure 2. There also exist mutations that change many bases at once. More details to this are given in Section 1.3. However, this mutations lead to large differences in the genome size.

To get this in perspective, the genome size of different species vary between less than one megabase and 670000 megabases (McGrath and Katz, 2004). The human genome has around 3286 megabases. The mutation rate, i.e., the probability of a mutation per base per generation, for humans, is estimated to be $3 \cdot 10^{-}8$ (Xue et al., 2009). Thus, there are around 100 mutations in every generation in a human genome.

However, these mutations make the difference between the species. The earliest proven live is up to 4.3 billion years old (Dodd et al., 2017). Thus, there was plenty of time to create many different species. This species creation is, in fact, a species specification.

One species evolve into two different species if the difference between them is so significant that they are no longer able to create fertile offspring together. Thus, their genomes do not mix.

This definition is very vague. However, this is a consequence of the fact that there are many different life forms, and every more detailed description would need

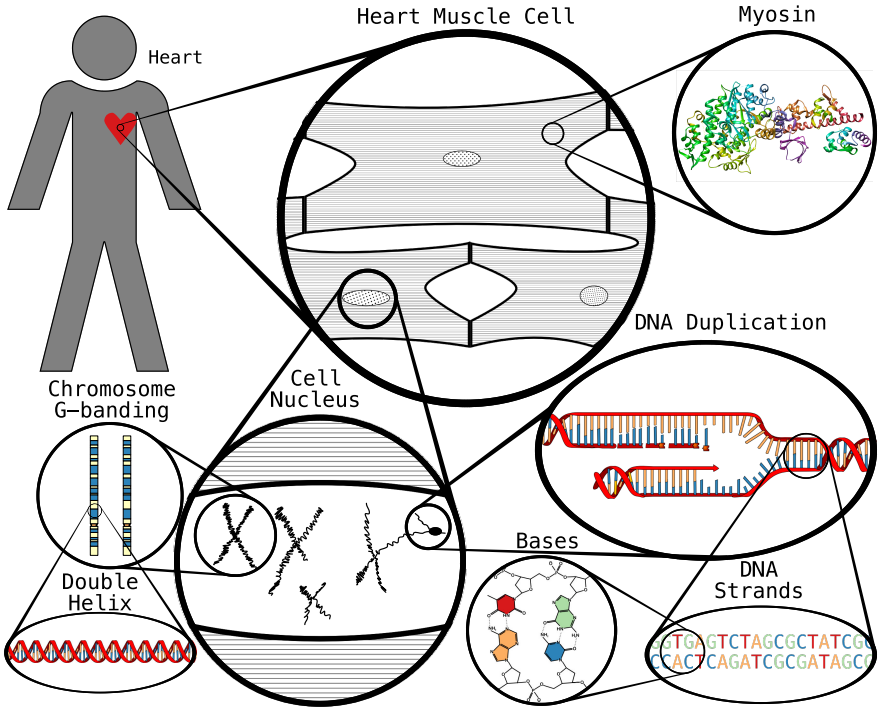


Figure 1: The concepts of life explained by the human Heart. This figure is simplified heavily to make it understandable. Every human has a heart muscle that contracts and keeps the blood in flow (top left). These muscle cells are connected (top center). Thus size changes of each cell create a forceful contraction. A size change in myosin (top right) creates the contraction. This protein creates long chains and can change its size by using energy. The structure after Risi et al. (2018) is shown. Each of the cells has a nucleus (bottom center). Beside many compartments (not shown) the genome, i.e., the DNA in the nucleus, is shown as chromosomes. The "X" like shape consists of two sister chromosomes, where one chromosome comes from the mother, and the other comes from the father. Chromosomes often are visualized with G-banding (middle left). This banding shows how strong the DNA is compressed at this region (the packing of DNA is not shown.). Beside the banding also the centromere is shown (orange). This region connects one chromosome with its sister chromosome. The normal structure of DNA is the double helix (bottom left). Thus every chromosome consists of two strands. This doubling of information is used for DNA duplication (middle right). The DNA double helix is unwound, and the strands are separated. Then for both lose strands, a complementary second strand is synthesized. Afterwards the helices again are winded, in this way two chromosomes are created from one. This process uses many protein complexes and other helper molecules. However, to keep it simple, they all are not shown. The two strands can be visualized as two complimentary strings (bottom right). Complimentary means that every **A**(orange) pairs with a **T**(red) and every **G**(green) pairs with a **C**(blue). This pairing is created by the molecular structure of the bases (bottom center).

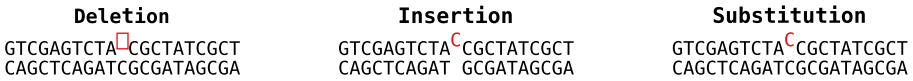


Figure 2: Single nucleotide mutations. The three possible mutations that change one base. The changes are shown in the strand on the top. Left one base is deleted. In the middle one extra-base is added (a **C**). On the right, a **G** is replaced with a **C**.

some exceptions. For example, animals need two sexual partners to create offspring. Thus, small changes that change mating behavior can separate to populations of one species. The absence of mixing the genome with time create different species that do not any longer can reproduce offspring. Such a specification is based on sexual selection (Lande and Kirkpatrick, 1988).

On the other side, for example, bacteria create offspring by cell division. Thus no need of a partner is given. However, the evolution of bacteria is even more complicated because there exists horizontal gene transfer (Gogarten and Townsend, 2005). Horizontal gene transfer transfers a part of a genome from one living individual to another living individual. Thus species definitions are more or less impossible for bacteria.

Linné (1767) has created a system to classify the species. This system, known today as taxonomy, orders different taxa hierarchically. One taxon could be a species that lives today or has lived at some time on earth. These extinct species could be described by paleontology, or they are more on a theoretical level.

These theoretical taxa are created because two living species are developed from one common ancestor. Thus, these taxa are classified at different levels. The root is the first level and is called Life. Other levels of the hierarchy are in order from most general to most specific: Domain, Kingdom, Phylum, Class, Order, Family, Genus, and Species. As an example, the human belongs to the Domain Eukaryota, the Kingdom Animalia, the Phylum Chordata, the Class Mammalia, the Order Primates, the Family Hominidae, the Genus Homo, and the Species Homo sapiens. Note that of course, more than eight differentiations happened after the first life to humans. Thus taxa exist between the here given taxa on every level.

This system may deviate from the evolution that really happen. Science only created the system based on observable features of the living species. In the beginning, this was done by the phenotype of the species, i.e., how they look. Now more specific genomics data is used. Even this data is not error free. However, it is the best representation that is known. Thus, it can change if new results are found. Data very well support the global connections, and new technologies support them further. Therefor it is reasonable to see the global structure of the taxonomy as reliable.

A simple distance measure can be applied to the taxonomy. The distance between two taxa is the number if taxa on the path from on taxon to the other in the taxonomic tree.

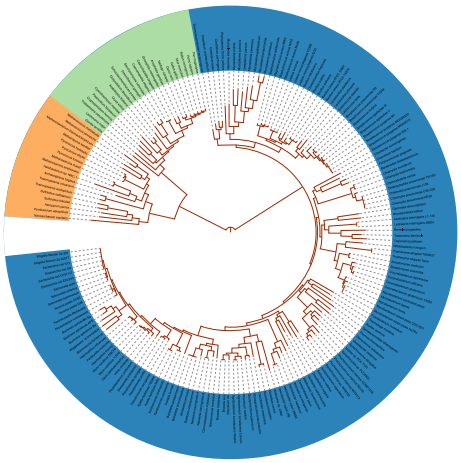


Figure 3: Modern Taxonomy. The taxonomy is created from genomic data (Ciccarelli et al., 2006). The colors belong to the three main domains of living species: orange is the archaea, green is the Eukaryota containing animals, plants, and fungi, and blue is the bacteria.

1.3 Genome Evolution

Mutations change the genome, thus, it is advisable to look at evolution on the genomic level. As stated before the genome is the information storage that changes and thus evolves over time. Thus, evolution means changes in the genome.

However, not every part of the genome evolves in the same way. Thus, the mutation rate is different for different parts of the genome. As a result, some parts are equal in more or less every living species. To understand this, a more in-depth look at how the storage works help.

The different mutation rate can be described with genes. The term appeared at the beginning of the 20th century (Johannsen, 1914). A modern definition is given by Gerstein et al. (2007) as:

> A gene is an union of genomic sequences encoding a coherent set of potentially overlapping functional products.

Thus, genes are the information that is saved in the genome. Important is that not every part of the genome is belongs to a gene. Thus, if a gene is mutated, some of the function is maybe changed, and if the mutation is not on a gene, the functions are generally unchanged.

Changes in the function are dangerous for the individual. Maybe something does not work or is less useful than before. Then the other individuals that do not have this mutation are more fit. Thus, this individual maybe creates less offspring, and the mutation is lost in the evolution.

Consequently, the mutation rate is higher on parts of the genome that are not part of a gene. However, the same effect also applies within the genes because depending on the gene, some mutation also do not influence the function.

An example of this are genes that are translated into proteins. For a protein, three bases are transformed into one amino acid. The transformation follows a specific code, the so-called genetic code. However, there are only 20 amino acids but 64 possible codons. Thus, some codons create the same amino acid. Therefore,

a mutation in the DNA could create the same protein and does not influence the function.

On the other side, some genes are so essential that changes on them are lethal for the individual. Thus these parts are remarkably conserved in most species, i.e., mostly unchanged in the genomes. An example is the HOX-gene cluster, that is heavily conserved. This conservation is so strong that, a fly can mostly function if a HOX gene is replaced with the same gene form a chicken (Lutz et al., 1996). This gives an idea why some parts of the genome change and others are comparable between different species. Thus, every analysis of the genome must consider the genes.

On an informatics point of view, the genes are annotations on the sequence that represents the DNA, i.e., the base sequence. Different tools use this equality to transfer annotations from one species to the other. Examples are `BLAST` (Altschul et al., 1990) and `Infernal` (Nawrocki and Eddy, 2013).

These searches need tuning for every gene to create reliable results. To explain this, a look at the structure of the genome helps. In most cases, the genome is not one molecule. A genome consists most times of different parts. Usually, these parts are called chromosomes.

With limited exceptions, these chromosomes are linear. The most known exceptions are mitochondria. A mitochondrion is a cell part that exists in more or less every eukaryote. The specialty is that every mitochondrion has a separate genome, that is circular. However, specific annotation tools are created to consider this circularity (Bernt et al., 2013).

In the cell, many linear molecules together create the genome. The order of these molecules is arbitrary. However, this structure is essential because besides the mutations that are created by the misplacement of bases in the coping process, another type of mutation exists. Thees mutations happen on the chromosome level. Thus, these mutations are called chromosome mutations. An overview of such mutations is given in Figure 4.

These more general structural mutations can have a more significant influence on the individual. If complete genes are missing or duplicated, this can have an influence on a much broader level than a change of a single nucleotide. However, even the rearrangements that only change the position or the direction of chromosome regions can have a significant influence on the individual.

A rearrangement can have tremendous influence for different reasons. The simplest is that a gene sequence is interrupted by the rearrangement and thus no functional version of the gene exists anymore. However, even if the borders of the rearrangement are not within any gene, there could be an effect. This effect can be understood by the fact that some mechanism of regulation work on a local basis.

An example of this could be genes that are transcribed in a row. Usually, they are required at the same time and thus are regulated jointly. Thus, if one gene of the row moved somewhere else, it is no longer transcribed if the others are transcribed. Further changes could be reached if instead another gene is placed there and transcribed. This gene could then suddenly be more active.

These global mutations happen frequently, i.e., even if two closely related species are compared, these mutations can be observed. A theory even assumes

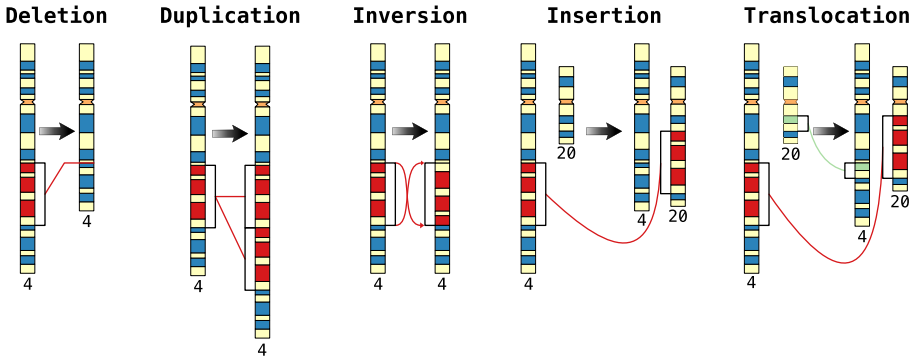


Figure 4: Chromosome Mutations. The Deletion and Duplication create or remove DNA where the other mutations only move the DNA to other positions. Note that in some literature, the Translocation is not mentioned as a chromosome mutation. However, one Translocation could be reproduced with two Insertions. Also, some literature, gather the Inversion, Insertions, and Translocation as a Rearrangement.

that duplication has a significant influence on evolution (Peer, Maere, and Meyer, 2009). Thus, a comparison of genomes between species must consider these mutations.

1.4 Alignments

A comparison of the genomes is based on the equivalence of the DNA sequences. However, after mutations are present, a simple perfect matching can not be used. Thus a matching must be computed that allow mismatches and gaps. A gap represents a deletion in one or an insertion in the other sequence. Such a matching is called an alignment.

Two types of alignments must be distinguished. The local alignment that only considers single mutations and a whole genome alignment that also considers chromosome mutations. This means, if a position i of one sequence match with position j on the other sequence in a local alignment, then position $i+1$ can only match with a position that is greater than j. In a whole genome alignment, this is not guaranteed.

For the local alignments, different methods have been created. T. Smith and M. Waterman (1981) has introduced a simple dynamic programming algorithm to compute an alignment. Later Gotoh (1982) improve this to handle gaps more realistically. However, every alignment method depends on the scoring matrix that is used.

A scoring matrix, specifies how a mismatch or a match should be scored. Furthermore, a scoring is given for the gaps. Mostly these scoring matrices are calculated from know sequences(Dayhoff, R. Schwartz, and Orcutt, 1978; S. Henikoff and J. G. Henikoff, 1992).

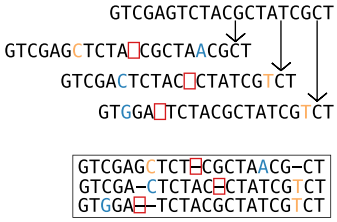


Figure 5: Local Alignment. From the sequence on the top three new sequences are created. Each of them is created with a deletion (red), an insertion (orange) and a substitution (blue). The created sequences are aligned (at the bottom). Note that even if the alignment is optimal, it does not necessarily recover the ancestral sequence.

These algorithms work well for two sequences. However, they get more complex if more sequences are considered. In general, multiple sequence alignments (MSAs) are *NP*-complete (Wang and Jiang, 1994). Kirchner, Retzlaff, and Stadler (2019) have created an exact algorithm for MSA but it is not suited for most practical problems. Thus, most MSA are created with heuristics. An example of a MSA is shown in Figure 5.

An example of such a heuristic is the progressive method (Higgins and Sharp, 1988). This method starts with pairwise alignments between all sequence pairs. This directly gives pairwise scores. This scores can be used to create a tree. Along this guide tree the alignments are combined to create the MSA. This combining uses the pairwise alignments to combine columns of the alignments. Thus, one error that is created down in the tree is conserved until the root of the tree. Thus, the progressive method works fast but stacks errors. However, it creates plausible results like all other heuristics.

In local alignments, the decision to align two bases cuts the sequence in two parts. Every base that is in front of the aligned base pair can only match with other bases before the pair and vice versa for bases behind the aligned base pair. If genes or small sequences are compared, this is reasonable, but not for whole genomes.

The problems arise from the chromosome mutations. It already appears if only rearrangements and no other mutations are considered, i.e., the sequences consist of the same parts, but the parts have different orders. Thus, the algorithm only gets parts right that have the same relative order in both sequences.

The solution to this is a genome-wide multiple sequence alignment (gMSA). The idea is straightforward: create independent local alignments, keep the best alignments, and create alignment blocks out of them. Thus, an alignment block is nothing else than a best local MSA. This splitting into blocks make it possible to detect rearrangements.

Beside that now alignments can cross, another enhancement is made. In such alignment not only one but two sequences are considered per chromosome: the two strands of the chromosome. Thus, even inversions can be detected. Only one strand is saved. This sense strand is enough because the other strand (antisense strand) can directly be computed based on the sense strand. This reverse complement is computed by reversing the reading direction and replacing every base with this compliment, i.e., **A** to **T**, **G** to **C**, and vise versa.

In an alignment block, a sequence on the antisense strand is marked. This mark is made in two different ways depending on the display form. Sometimes they are overlined, and some times they are marked with a minus in front. An example of a

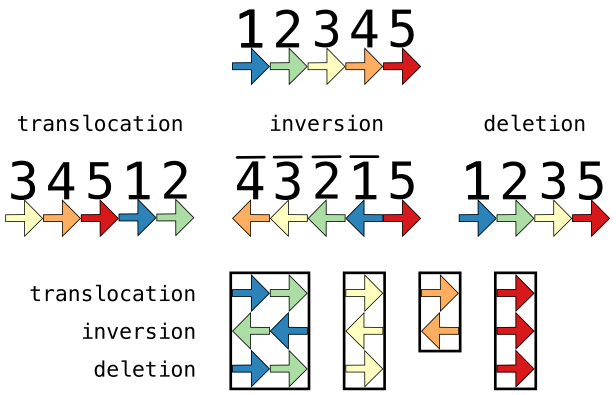


Figure 6: A gMSA. On the top a sequence with five parts is shown. Parts are indicated by colored arrows. From this sequence three mutated sequences are created: one with translocation of blue and green, one with an inversion of blue, green, yellow, and orange, and one with a deletion of orange. The direction of the arrows indicate the direction of the sequence. The gMSA is shown at the bottom. There are four alignment blocks created. Note that the antisense strand is considered also. Thus the parts of the inverse sequence are in the same blocks as the other sequences in the same color. In every sequence the blue and the green part is in the same order (or reversed ordered on the reverse strand). Thus they together form one block.

gMSA is shown in Figure 6.

gMSAs are solved with heuristics. It is simply not possible to calculate all possible local alignments between two complete genomes. The problem becomes even worse if more genomes are included. However, over the past two decades several pipelines have been deployed to construct such gMSAs, most prominently the `tba`/`multiz` pipeline (Blanchette et al., 2004; S. Schwartz et al., 2003) employed by the `UCSC` genome browser and the `Enredo`/`Pecan`/`Ortheus` (`EPO`) pipeline (Paten, Herrero, et al., 2008) featured in the `ensembl` system. For the ENCODE project data, in addition alignments generated with `MAVID` and `M-LAGAN` (Bray and Pachter, 2004) have become available, see X. Chen and Tompa (2010) for a comparative assessment.

1.5 Genome Assembly

To align any genomic data, it must exist. Genomic data is a DNA sequence of **A**, **T**, **G**, and **C**. The genomic sequences of many species are determined using two methods. First DNA sequencing is used, and afterwards a genome assembly method is used.

The first method, DNA sequencing is a practical laboratory method, in which the sequence of a part of the genome is determined. The past decade has seen the rapid progress of sequencing technologies (Gawad, Koh, and Quake, 2016).

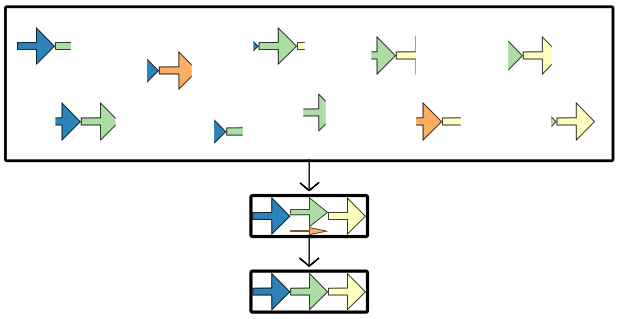


Figure 7: Genome assembly. On the top ten reads are shown. If they are assembled, they create a genome shown in the middle. Note that two of ten reads show the orange sequence as the center sequence. Where the other show a green sequence. Thus 4/5 show green and 1/5 orange. Because there are much more reads with the green sequence, the orange is seen as a sequencing error, and the finished genome contains the green sequence in the center (at the bottom).

However, all methods have in common that only parts of the genome are sequenced at a time. In many cases, these are only very short fragments of DNA. Such a fragment is called a read.

Thus, this reads must be put together to a whole genome like the pieces of a puzzle. This puzzling is done with the second method, the genome assembly. The task of genome assembly could be formulated as to order the fragments such that connected chromosomes are created.

However, such a formulation leaks one crucial detail. The sequencing methods do not work perfectly. They create sequencing errors that must be filtered. This filtering means that reads or part of reads are discarded. Thus, the problem is not ordering, it is ordering and filtering errors at the same time. An example of such an assembly is shown in Figure 7.

1.6 Supergenome

Another problem that is similar to the assembly problem is the supergenome problem (Herbig et al., 2012). The supergenome problem is to determine an universal coordinate system of all alignment blocks in a gMSA. This problem can be solved by giving an order of the alignment blocks. Then the coordinate i of an alignment block x is $\sum_{y<x}(|y|)+i$, i.e., the sum of the size of all blocks in front of it plus i. The hard part is to determine an order that keeps most genomes intact.

To a certain extent, this problem is alleviated by considering the blocks arranged w.r.t. a reference genome. For many applications, however, this does not appear to be sufficient. For sufficiently similar genomes with only a few rearrangements gMSA blocks are large or can at least be arranged so that large syntenic regions can be represented as a single aligned block. Any ordering of these large syntenic blocks then yields an informative common coordinate system.

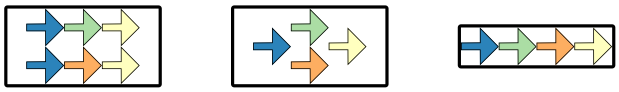


Figure 8: Supergenome problem. Left are two genomes shown. If they are aligned, they create a sequence shown in the center. Blue is at the start, and yellow at the end of the sequence in both sequences. However, one sequence have a green sequence, and the other an orange sequence in the center. Thus there exists an inaccuracy that must be resolved. It is resolved simply by placing the orange sequence behind the green (right) or vise versa.

So far, this supergenome approach has been applied only to closely related taxa. Prime examples are detailed comparative analysis of the transcriptome of multiple isolates of *Campylobacter jejuni* (Dugar et al., 2013) or the reconstruction of the phylogeny of mosses from the "nucleotide pangenome" of mitogenomic sequences (Goryunov et al., 2015). Note that some approaches to "pangenomes" are concerned with gMSAs of (usually large numbers of) closely related isolates; most of this literature, however, treats pangenomes as sets of orthologous genes (Medini et al., 2005).

The supergenome problem differs mostly to the assembly problem by the fact that in the assembly problem alternative paths are reduced to one path wherein the supergenome both paths are kept intact. An example is shown in Figure 8. This equality of the problems makes it possible to transfer methods from one problem to the other.

1.7 Orders and graphs

For a more formal view on both problems, a more in-depth look into some mathematical background is done. Two mathematical concepts are of particular interest: ordering and graphs.

Ordering is in the core of mathematics since the early days of mathematics. Something like a number makes only sense if it can be distinguished from other numbers. Therefore, ordering is based on a relation between a pair of elements. Mostly this relation is interpreted as one element is greater than the other. The relation must fulfill other properties to create specific types of ordering.

However, there many precise definitions of ordering and different types of orders. To describe them all is way beyond the scope of this work. There even exists a field of mathematics that only considers the order theory. Here, the focus is on the linear or total order. This order has the property that for every pair of elements, exactly one of the following statements is true: they are equal, or one is greater than the other.

A graph is a mathematical construct that describes relations between objects. Sylvester (1878) introduce the term graph. A graph is defined by two sets: the vertices, and the edges. Here, the vertices are the elements in the graph, and the edges connect two of these elements. Thus, an edge is a tuple of two vertices.

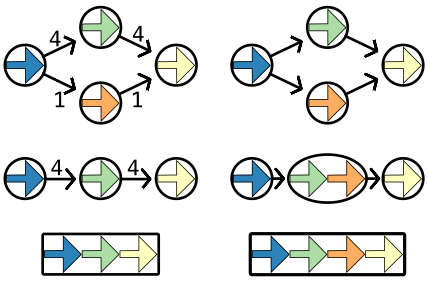


Figure 9: Genomic problems solved with graphs. The genome assembly from Figure 7 (left) and the supergenome from Figure 8 (right) as graphs. On the top, the graphs that are created from the inputs shown. The numbers on the assembly graph show how many reads support one edge. In the middle then the graphs after the ambiguity are resolved for the given problem. At the bottom is the results are shown.

The main focus of this thesis is on directed graphs or digraphs. Thus, no symmetry is needed. One object can have a connection to another without having a connection in the opposite direction. An example could be that one alignment block comes behind the other in one genome which actuality permits that in a linear genome the opposite is also correct.

1.8 Total Ordering of a Digraph

The total ordering of a digraph or the vertex ordering problem is a class of combinatorial problems that recently has received increasing attention in computer science (Bodlaender et al., 2011; Fellows et al., 2016; K. Li et al., 2015; Pardo, Martí, and Duarte, 2018). In this problem, a digraph is given, and an ordering of the vertices has to be created that keeps most of the edges valid. Here, what is valid depends on the use case.

The total ordering of a digraph can be used to solve both the assembly and the supergenome problem. This transformation is done in two steps: First, a fitting graph must be created from the inputs (the reads or the gMSA). Second, the ordering is created by solving ambiguities.

This second step is done in the genome assembly by simplifiers. Simplifiers are tools that use local information to remove ambiguities and create a single order. The use of local information make them relatively fast but also not perfect. Thus, they are heuristic by nature.

The goal of this thesis is to optimize and use simplifiers known from genome assembly to create practical solutions to the supergenome problem. For this, in Chapter 2 an overview of the literature over total orderings of digraphs is given. Then, Chapter 3 presents the superbubble simplifier and novel algorithm to detect superbubbles. The theory for a pipeline to create a supergenome of distant taxa is presented in Chapter 4. In Chapter 5 the applications of both the superbubble and the supergenome theory is shown. Chapter 6 then discusses the results and present an outlook on how the theory can be used further.

CHAPTER 2

Total graph ordering

Contents

In this chapter, an overview of methods in the literature to create a total order out of a graph is given. First, some basic definitions are provided. In Section 2.1, different order types and their features are introduced. In Section 2.2, the graph types and there basic properties are presented.

Then, different types of creating an order are given. Various ways to solve a general ordering problem are shown in Section 2.3. After this, solutions for more specific problems like the assembly problem (Section 2.4) and the supergenome problem (Section 2.5) are considered.

In the last section, Section 2.6, a more in-depth observation of graph simplifications is provided. These methods do not solve the ordering problem but are used in the assembly and supergenome problem to make large graphs less complex. However, the use of a simplifier influences the outcome of the ordering. Thus, they are part of the solution process and before they applied they have to considered carefully.

The thesis assumes some basic knowledge like sets, relations, basic operations of sets. Furthermore, the reader should be familiar with runtime complexity classes in the field of computer science. Specifically, the complexity class *NP* and the term *NP*-completeness should be known.

2.1 Ordering

This work only considers the basic types of ordering since the complete order theory would be beyond the scope of this thesis. Thus, only four order types are presented: the partial, the total, the cyclic, and the betweenness order. Also, the mathematical structure of a set is extended by these orders. Where possible the notion of Viro et al. (2008) is used.

All of the order types that are presented in the thesis are defined on finite sets. The condition of the orders also work on infinite sets, but some conclusions are not so simple on infinite sets. Furthermore, is the focus of this thesis a biological application that exist only on finite objects. Thus, if not others mention a set is assumed as finite.

2.1.1 Partial Order

A partial order is an order where not all elements must be comparable. An example is the order of ancestry. If Peter has two daughters, Sarah and Nancy, then Peter is an ancestor of them, but Sarah and Nancy have no ancestor connection. An example of a partial order is given in Figure 10.

Definition 1. *A binary relation $a \leq b$ on a set X is a partial order if it satisfies the following three conditions:* ***partial order***

- $a \leq a$ *for any* $a \in X$ *(Reflexivity)*
- *If* $a \leq b$ *and* $b \leq a$*, then* $a = b$ *for any* $a, b \in X$ *(Antisymmetry)*
- *If* $a \leq b$ *and* $b \leq c$*, then* $a \leq c$ *for any* $a, b, c \in X$ *(Transitivity)*

In a possible combination of two partial orders, one order is placed in front of the other. This gives the definition of the $+$ relation on partial orders:

Definition 2. *Let $\leq_1$ and $\leq_2$ be two partial orders on the sets X and Y. Then the order combination $\leq_1 + \leq_2 = \leq$ is a partial order on $X \cup Y$, where $a \leq b$ holds when:*

- *if $a, b \in X$ and $a \leq_1 b$,*
- *if $a, b \in Y$ and $a \leq_2 b$, or*
- *if $a \in X$ and $b \in Y$.*

Note that even though the notation $+$ is used, the operation is not commutative.

2.1.2 Total Order

A total order is a partial order that guarantees that every pair of elements can be compared. An example would be the natural numbers. For every pair of natural numbers (a, b) holds $a \leq b$ or $b \leq a$.

total order **Definition 3.** *A partial order $a \leq b$ on a set X is a total order if it satisfies the following condition:*

- *$a \leq b$ or $b \leq a$ for any $a, b \in X$ (Totality).*

Note that the order combination (Definition 2) on partial orders also work on total orders and if both original orders are total, the result is also total. An example of a total order is given in Figure 10.

It is possible to create a bijection between the first n natural numbers and any finite set with n elements by using a total order on this set. To formulate this a definition for the set of the first n natural numbers is given as:

$$[n] = \{i \mid i \in \mathbb{N} \wedge i \leq n\}. \tag{2.1}$$

Note that $\mathbb{N}$ does not include zero. Thus a total order of a finite set can be formulated as bijection:

Definition 4. *Let X be a finite set and $\varpi \colon X \to [|X|]$ a bijection. Then ϖ defines a total order $\leq$ on X in the way that:*

$$a \leq b \longleftrightarrow \varpi(a) \leq \varpi(b), \forall a, b \in X.$$

Such a bijection associates to every element a position or index in the total order. Thus, it is a permutation of the elements. In this work, a bijection is used as the notation of a total order. Total orders are the most frequently used order thus the "total" is often discarded if terming them. Thus, if later the term "order" is used, a total order is meant.

It is possible to reverse the total order of a finite set X. Let ϖ be the bijection of a total order. Then the reversed total order with the bijection $\overline{\varpi}$ is defined as: **reversed order**

$$\overline{\varpi}(v) = |X| - \varpi(v) + 1. \tag{2.2}$$

This reversed order is again a total order.

The fact that the total order is represented by a bijection opens another possibility to identify the element for each position. For this the inverse function of the bijection is used. The inverse function $\varpi^{-1}\colon [|X|] \to X$ is defined as: **inverse function**

$$\varpi^{-1}(i) = x \longleftrightarrow \varpi(x) = i \tag{2.3}$$

where $i \in [|X|]$, $x \in X$, and ϖ defines a total order on set X.

2.1.3 Cycle Order

The third type of orders are the cycle orders. These orders are very similar to total orders. The difference is that a cycle order has no end or beginning but is repeating itself. An example of this is the clock. There are only twelve hours on a clock, but after twelve, the one of the next cycle follows.

A cycle order can not be described with only two points. For the one comes six before twelve but for the ten, twelve comes first. The order of two elements depends on a third element, the "viewing point". Thus, the cycle order is a ternary relation and not a binary relation.

Definition 5. *A ternary relation $[a \leq b \leq c]$ on a set X is a cycle order if it satisfies the following four condition:* ***cycle order***

- *If $[a \leq b \leq c]$, then $[c \leq a \leq b]$ for any $a, b, c \in X$ (Cyclicity).*
- *If $[a \leq b \leq c]$ and $[a \leq c \leq b]$, then $b = c$ for any $a, b, c \in X$ (Antisymmetry).*
- *If $[a \leq b \leq c]$ and $[b \leq c \leq d]$, then $[a \leq b \leq d]$ for any $a, b, c, d \in X$ (Transitivity).*
- *$[a \leq b \leq c]$, or $[a \leq c \leq b]$, for any pairwise distinct $a, b, c \in X$ (Totality).*

The analogy to a total order is easy to see by comparing the conditions. However, it is easy to transform a cycle order into a total order. A cycle order $[a \leq c \leq b]$ on X can be transformed into a total order $a \leq b$ on X by choosing a cut point $x \in X$. Then the equivalence

$$[x \leq a \leq b] \Leftrightarrow a \leq b \tag{2.4}$$

holds for every $a, b \in X$.

A total order can analogously be transferred into cycle order. A total order $a \leq b$ on X define a cycle order $[a \leq b \leq c]$ on X in the way that:

- If $a \leq b$ and $b \leq c$, then $[a \leq b \leq c]$.

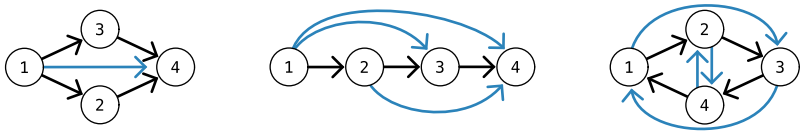


Figure 10: Order types. The left graphs shows a partial order, the center graph shows a total order, and the right graph shows a cycle order. The arrows show the relation between the objects. For partial and total orders, an arrow from 1 to 2 means that $1 \leq 2$. For the cycle order, two arrows and three distinct objects are needed for the ternary relation. The arrows from 1 to 2 and from 2 to 3 means that $[1 \leq 2 \leq 3]$. Thus, the total order is created from the partial order by adding $2 \leq 3$ and the cycle order can be represented with the total order. The orders are unique with the black edges but to fulfill the transitivity, also the blue edges are required.

- If $a \leq b$ and $c \leq a$, then $[a \leq b \leq c]$.
- If $b \leq a$ and $b \leq c$ and $c \leq a$, then $[a \leq b \leq c]$.

Note that more than one total order defines the same cycle order.

A cycle order can also be regarded as bijection. This follows directly from the fact that it can be represented as total order and a total order can be presented as bijection. Note that the cycle order can also be reversed and an inverse function exists. Both follows directly from the representation as bijection. For readability, a bijection is used as the notation for cycle orders in this work. An example of a cycle order is given in Figure 10.

2.1.4 Betweenness Order

Another version of an order is the betweenness relation. Such a relation is in some way the generalization of the cycle relation. It is a ternary relation where $[a \lesssim b \lesssim c]$ means that b is between a and c. The main difference to the previous orders is the loss of direction.

betweenness relation

Definition 6. *A ternary relation $[a \lesssim b \lesssim c]$ on a set X is a betweenness relation if it satisfies the following condition:*

- *If $[a \lesssim b \lesssim c]$, then $[c \lesssim b \lesssim a]$ for any $a, b, c \in X$ (Directionless).*

This condition directly contradicts the assumptions of the other orders. However, another way to formulate the same condition is to set both directions equivalent.

$$[a \lesssim b \lesssim c] \equiv [c \lesssim b \lesssim a] \tag{2.5}$$

With this in mind, it could be possible to define a total order that fulfills a betweenness relation. In the way that:

$$[a \lesssim b \lesssim c] \rightarrow a \leq b \leq c \vee c \leq b \leq a. \tag{2.6}$$

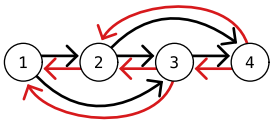


Figure 11: Betweenness Order. The ternary betweenness relation is shown with two arrows of the same color, similar to the cyclic order in Figure 10. For example, the arrows 1 to 2 and 2 to 3 mean that $[1 \leqslant 2 \leqslant 3]$.

Thus, if a total order fulfills the betweenness relation, the reverse total order must also accomplish it.

However, not every betweenness relation can be fulfilled by a total order, only a subset can. In the literature, one can find many sets of axioms that must be met by a betweenness relation such that a total order can fulfill it. An overview can be found at Fishburn (1971). A betweenness relation that can be fulfilled by a total order is termed a betweenness order.

betweenness order

Definition 7. *A betweenness relation $[a \leqslant b \leqslant c]$ on a set X is a betweenness order if it satisfies the following conditions:*

- *If $[a \leqslant b \leqslant c]$ and $[a \leqslant c \leqslant b]$, then $b = c$ for any $a, b, c \in X$ (Antisymmetry).*
- *$[a \leqslant b \leqslant c]$, $[b \leqslant a \leqslant c]$, or $[a \leqslant c \leqslant b]$, for any pairwise distinct $a, b, c \in X$ (Totality).*
- *If $[a \leqslant b \leqslant c]$ and $d \neq b$, then $[a \leqslant b \leqslant d]$ or $[d \leqslant b \leqslant c]$ for any $a, b, c, d \in X$ (Transitivity).*

Note that a cycle order also fulfills all three conditions. To be more precise if the directionless condition is replaced with the cyclicity condition, an equivalent cycle order definition is given. An example of a betweenness order is given in Figure 11.

betweenness problem

Opatrny (1979) has shown that the betweenness problem, i.e., the problem to determine for an arbitrary betweenness relation if it can be fulfilled by a total order is *NP*-complete. Later Chvátal and Wu (2011) have demonstrated that this holds even if a betweenness relation fulfills the antisymmetry condition and another transitivity condition, namely:

- If $[a \leqslant b \leqslant c]$ and $[a \leqslant d \leqslant b]$, then is $[a \leqslant d \leqslant c]$ for any pairwise distinct $a, b, c, d \in X$.

2.1.5 Ordered Set

A set is by definition an unordered structure. However, the combination of a set with a total order creates a construct termed ordered set. These ordered sets combine the features of the set with a total order. The notion is (X, ϖ) for an ordered set. Where X is a finite set and $\varpi \colon X \to [|X|]$ is the bijection that donates an order.

ordered set

Beside this notation, this thesis uses a compact representation of an ordered set. $X = \{x_1, x_2, \ldots, x_n\}$ donates an ordered set (X, ϖ) with n elements. The order bijection is than $\varpi \colon X \to [|X|], x_i \longrightarrow i$.

Since an ordered set is a set, the set operations do work on them. However, these operations do not consider the order. Thus, the result has no well-defined

order. The problem is addressed by defining the result as a regular set, i.e., an ordered set is for a set operation a set, and the order is ignored. Note that this makes it further possible to use an ordered set as one operand and a set as the other.

Besides the possibility to combine different ordered sets, it is possible to restrict an ordered set to a subset. This restriction can be done by giving the indices of the ordered subset. To be more precise, the limits, i.e., the smallest and the largest included indices is sufficient, if everything between them is also included. An ordered subset created this way is called an interval. Let (X, ϖ) be an ordered set then an interval is defined as:

interval

$$\varpi[i\!:\!j] = (\{x \mid x \in X \wedge i \leq \varpi(x) \leq j\}, \varpi') \tag{2.7}$$

where ϖ' stands for the order ϖ that is restricted to the new set. Which means that the domain of ϖ' only contains the elements of the new set, and the function ϖ' is defined as:

$$\varpi'(x) = \varpi(x) + 1 - i. \tag{2.8}$$

It exists a second type of interval, the open interval. The difference is that the open interval does not include the limits. Let (X, ϖ) be an ordered set then an open interval is defined as:

$$\varpi(i\!:\!j) = (\{x \mid x \in X \wedge i < \varpi(x) < j\}, \varpi'') \tag{2.9}$$

where ϖ'' again stands for ϖ with restricted domain and the function ϖ'' is changed to:

$$\varpi''(x) = \varpi(x) - i. \tag{2.10}$$

There is no set operation that uses an equivalent notion. Thus, it is possible to shorthand the notation. If

$$X = \{x_1, \ldots, x_n\} \to (X, \varpi) \tag{2.11}$$

then $X[x_i\!:\!x_j] = \varpi[i\!:\!j]$ and $X(x_i\!:\!x_j) = \varpi(i\!:\!j)$. Note that elements of X are used. The indirect assumption by this equivalences is $\varpi(x_i) = i$ and $\varpi(x_j) = j$.

As special case consider that ϖ is the identity function of the natural numbers, i.e., $\varpi\colon \mathbb{N} \to \mathbb{N}, x \longrightarrow x$. An interval on ϖ is then simply the natural numbers between i and j. Thus, in this case, ϖ can be omitted:

$$[i\!:\!j] = \{x \mid i \leq x \leq j\} \tag{2.12}$$

2.1.6 Cyclic Set

cyclic set

A cyclic set is the combination of a set and a cycle order. The nomenclature (X, ϖ) is similar to ordered sets. Again a bijection that represents the cycle order is used to define the cyclic set. The distinction of an ordered set and a cyclic set should be made by context. However, the short notation $\circlearrowleft c_1, \ldots, c_k \circlearrowright$ is unique for a cyclic set. Note that more than one short notation represents the same cyclic set.

An interval of a cyclic set is then called a cycle interval. The definition must be changed to reflect the cyclic nature of the order. However, the cycle interval looses as subset the cycle properties and is an ordered set. Let (X, ϖ) be a cyclic set then a cycle interval and an open cycle interval is defined as: **cycle interval**

$$\varpi[i\!:\!j] = \begin{cases} (\{x \mid x \in X \wedge i \leq \varpi(x) \leq j\}, \varpi') & \text{if } i < j \\ (\{x \mid x \in X \wedge i \leq \varpi(x) \vee \varpi(x) \leq j\}, \varpi^*) & \text{otherwise} \end{cases} \tag{2.13}$$

$$\varpi(i\!:\!j) = \begin{cases} (\{x \mid x \in X \wedge i < \varpi(x) < j\}, \varpi'') & \text{if } i < j \\ (\{x \mid x \in X \wedge i < \varpi(x) \vee \varpi(x) < j\}, \varpi^{**}) & \text{otherwise} \end{cases} \tag{2.14}$$

where ϖ' and ϖ'' is defined as in Equation 2.8 and Equation 2.10. Further ϖ^* is defined as

$$\varpi^*(x) = \begin{cases} \varpi(x) + 1 + |X| - i & \text{if } \varpi(x) \leq j \\ \varpi(x) + 1 - i & \text{otherwise} \end{cases} \tag{2.15}$$

and ϖ^{**} as

$$\varpi^{**}(x) = \begin{cases} \varpi(x) + |X| - i & \text{if } \varpi(x) < j \\ \varpi(x) - i & \text{otherwise.} \end{cases} \tag{2.16}$$

For cyclic sets a similar shorthand notation as for ordered sets can be used:

$$X = \circlearrowleft x_1, \ldots, x_n \circlearrowright \to (X, \varpi) \tag{2.17}$$

then is $X[x_i\!:\!x_j] = \varpi[i\!:\!j]$ and $X(x_i\!:\!x_j) = \varpi(i\!:\!j)$.

2.1.7 Sequence

A sequence is a structure that is similar to a totally ordered set. The difference is that it is not a set. Therefore, elements can be contained multiple times. As a sequence only exists with an order, these multiple appearances have different indexes. An example would be the Fibonacci sequence where the natural number 1 appears twice. The sequence $(1, 2 \ldots, 100)$ would be a sequence with the natural numbers $[1\!:\!100]$ where every value is equal to its index. **sequence**

2.2 Graphs

In this section, different graph types are presented. All graph types have to fulfill specific properties. This thesis only considers finite graphs. Many of the results can be identically applied to infinite graphs. However, not every result can be directly transferred, and infinite graphs have a more theoretical character. Furthermore, they are unrelated to the given biological background.

2.2.1 Undirected Graph

This work uses a definition of undirected graphs as follows:

undirected graph

Definition 8. *G is called an undirected graph if:*

- *$V(G)$ is a set of vertices and*
- *$E(G)$ is a set of edges such that $E(G) \subseteq V(G)^2$.*

This definition describes the structure of an undirected graph. Note that even though edges are considered to be order pairs, it is not differentiated here between an edge (v, u) and (u, v). The following properties are needed to explore it. They follow directly from the structure:

Definition 9. *Let G be an undirected graph. For every vertex $v \in V(G)$ the following attribute is defined:*

- *The neighbor of a vertex:*
  $\text{neighbor}(v) = \{u \mid (v, u) \in E(G)\} \cup \{u \mid (u, v) \in E(G)\}$

For every edge $e = (v, u) \in E(G)$ the following attribute is defined:

- *The vertices of an edge:*
  $V(e) = \{v, u\}$

With this definition, it is possible to traverse the graph, i.e., the connections between the objects in the graph. However, graphs represent information. Hence, the elements in the graph must represent some information. The information is saved in properties of the edges and vertices. Furthermore, any element can have more than one property. Thus, every property has two parts a key under which the property is saved and a value which contains the information.

property graph

$$\begin{aligned} \text{property}(v, k) &\text{ Is the property with key } k \text{ on vertex } v \\ \text{property}(e, k) &\text{ Is the property with key } k \text{ on edge } e \end{aligned} \tag{2.18}$$

A graph that has properties is called a property graph. However, every graph can be treated as property graph if considering vertex labels as properties. As an example, Sylvester (1878) uses the graph in chemical networks and labels the vertices with atom names. Thus, every graph in this thesis is implicitly a property graph.

The edges connect the vertices. This concept can be extended to a transitive relation.

connected relation

Definition 10. *Let G be a graph. Then the vertices $v, u \in V(G)$ are connected, notated as $u \sim v$, if*

- *the edge (u, v) or the edge (v, u) is in $E(G)$ or*
- *it exists $w \in V(G)$ such that $u \sim w$ and $w \sim v$.*

Note that this connected relation is an equivalence relation.

One equivalence class of the connected relation is called a connected component. If every vertex is connected to every other vertex only one connected component exists that includes every vertex. In this case, the graph is called connected.

connected component

Subgraphs

A graph consists of two sets. Thus, it is natural to extend the definition of subsets to graphs, i.e., a subgraph. However, the subgraph must also fulfill the graph condition. The notion of $H \subseteq G$ is used for the subgraph relation.

Definition 11. *Let G be a graph. Then H is a subgraph of G if and only if:*

subgraph

- *The vertex set is a subset:*
  $V(H) \subseteq V(G)$.
- *The edge set is a subset:*
  $E(H) \subseteq E(G)$.
- *H is a graph:*
  $E(H) \subseteq V(H)^2$.

A special case of subgraphs is the induced subgraph. An induced subgraph is determined only by its vertex set. The subgraph contains all edges of the original graph that connect vertices of the given vertex set. The notion $G[U]$ is used for the induced subgraph of G with the vertex set U.

Definition 12. *Let G be a graph. Then $H = G[U]$ is the induced subgraph of G with the vertex set U if:*

induced subgraph

- *The vertex set is a subset:*
  $V(H) = U \subseteq V(G)$.
- *The edge set is a maximal subset:*
  $E(H) = \{e \mid e \in E(G) \wedge V(e) \subseteq U\}$.

Subgraphs are one way to create a new graph from another graph. A different possibility is to manipulate the vertex and edge set directly. Thus it is possible to add and remove edges from the edge set or to add vertices to the vertex set. Note that removing vertices would mean that also the corresponding edges must be removed, which is the same as creating an induced subgraph. Thus this operation is not listed here.

These operations are equivalent to set operations (union, intersection, and difference on the edge set, and the union on the vertex set). This graph operations use the same notation as the regular set operations with the difference that the first element of the operation is a graph. Let G be a graph and E a set of edges and V a set of vertices then the following operations are defined as:

graph-set operations

$$G \cup E = H \text{ where } V(H) = V(G); E(H) = E(G) \cup E$$

$$G \cap E = H \text{ where } V(H) = V(G); E(H) = E(G) \cap E$$
$$G \setminus E = H \text{ where } V(H) = V(G); E(H) = E(G) \setminus E$$
$$G \cup V = H \text{ where } V(H) = V(G) \cup V; E(H) = E(G)$$

Note that $G \cup E$ is only allowed if $\forall (v,u) \in E : v,u \in V(G)$.

graph operations

This simple nomenclature is possible because two sets define the graph, and an edge set can be distinguished from a vertex set. However, with the same arguments, the operations are not only possible between a graph and a set, but it is also possible between two graphs. Let G_1 and G_2 be two graphs. Then the graph operations are defined as:

$$G_1 \cup G_2 = H \text{ where } V(H) = V(G_1) \cup V(G_2); E(H) = E(G_1) \cup E(G_2)$$
$$G_1 \cap G_2 = G_1[V(G_1) \cap V(G_2)]$$
$$G_1 \setminus G_2 = G_1[V(G_1) \setminus V(G_2)]$$

Note that the intersection and difference operations use an induced subgraph. Thus, these operations work with every pair of graphs. Therefore, $V(G_1) \cap V(G_2) = \emptyset$, then $G_1 \cap G_2$ is an empty graph and $G_1 \setminus G_2 = G_1$. Respectively, if $V(G_1) \cap V(G_2) = V(G_1)$, then it is vice versa.

2.2.2 Directed Graph

A directed graph or digraph is the same structure as an undirected graph. The difference is the interpretation of the structure. In an undirected graph, an edge (u,v) is similar to an edge (v,u). In a direct graph, they are different. Thus, the edges have a direction. All graph definitions apply to a digraph, but additional definitions make only sense on a digraph:

digraph

Definition 13. *Let G be a digraph. For every vertex $v \in V(G)$ the following attributes are defined:*

- *The predecessor of v:*
  $\mathrm{pre}(v) = \{u \mid (u,v) \in E(G)\}$
- *The successor of v:*
  $\mathrm{suc}(v) = \{u \mid (v,u) \in E(G)\}$
- *The in-degree of v, i.e., the number of incoming edges:*
  $\mathrm{indeg}(v) = |\{(u,v) \mid (u,v) \in E(G)\}|$
- *The out-degree of v, i.e., the number of outgoing edges:*
  $\mathrm{outdeg}(v) = |\{(v,u) \mid (v,u) \in E(G)\}|$

For every edge $e = (v,u) \in E(G)$ the following attributes are defined:

- *The tail of e:*
  $\mathrm{tail}(e) = \{v\}$

- *The head of e:*
  $\mathrm{head}(e) = \{u\}$

From the edges, a relation can be derived. The *reachability relation* (notated as $\rightsquigarrow$) links two vertices that are reachable one from the other.

Definition 14. *Let G be a graph. Then the vertex $v \in V(G)$ reach $u \in V(G)$ ($v \rightsquigarrow u$) if:* ***reachability***

- $(v, u) \in E(G)$ *or*
- *it exists $w \in V(G)$ such that $u \rightsquigarrow w$ and $w \rightsquigarrow v$.*

In this way, it is a transitive closure of the edge relation. Another way of stating the same property is:

$$v \rightsquigarrow u \Leftrightarrow \exists n \geq 0 \exists x_1, \ldots, x_n \in V(G) : (v, x_1), (x_1, x_2), \ldots, (x_n, u) \in E(G) \tag{2.19}$$

Note that if $n = 0$ no x_i exist and only the edge (v, u) is in $E(G)$. Thus, if v reach u a sequence of vertices $(v, x_1, \ldots, x_n, u)$ exists, where x_i is defined as in Equation 2.19. This sequence of vertices is termed a walk.

If a walk from v to u exists then also two other structures exist. First, a trail of digraph G is an ordered set $(X, \varpi) = \{e_1, \ldots, e_n\}$ with $X \subseteq E(G)$ such that $\mathrm{tail}(e_1) = v$ and $\mathrm{head}(e_n) = u$ and every edge has $\mathrm{head}(e_i) = \mathrm{tail}(e_{i+1})$. Note that because a trail is an ordered set every edge can only be contained once in a trail. **trail**

The second structure is a path, an ordered set of vertices that is also a walk (as a sequence). Here again, every vertex is only contained once in a path. Thus, not for every walk exists a path. Note that for every path a trail exists that represents precisely the edges that are between the vertices of the path. A path from v to u is notated as $v \to u$. **path**

It exists an equivalence between the reachability and the order of a path: Let H be a subgraph of G that has a path p in G as vertex set and the corresponding trail as edge set. Then the reachability relation of H is equivalent to the order relation of p.

A path can give an alternative definition of reachability:

$$v \rightsquigarrow u \longleftrightarrow \text{a path } v \to u \text{ exists.}$$

If reachability is defined in such a way, it can be extended to different forms. First, it can be extended in the way that two paths are demanded. These paths should be vertex-independent. Two paths are vertex-independent if they share no internal vertex, therefore they are edge-disjoint. Because they must only differ in the internal vertices, it is possible that they have the same start and end vertex. If the vertex v has at least two vertex-independent paths to u then v 2-reaches u or notate as $v \twoheadrightarrow u$. The name of the relation follows from the fact that at least two vertices must be removed until v cannot reach u anymore. This idea comes from Menger (1927). This can be formulated more formally as: **2-reachability**

$$v \twoheadrightarrow u \longleftrightarrow \exists p_1 = v \to u, p_2 = v \to u : p_1 \neq p_2 \wedge p_1 \cap p_2 = \{v, u\}. \tag{2.20}$$

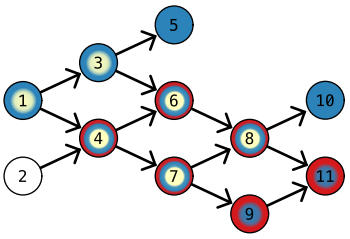


Figure 12: Different Reachable Sets. Every red vertex belongs to $[4,8]_{\rightsquigarrow}$ and every yellow vertex belongs to $[8,4]_{\reflectbox{$\rightsquigarrow$}}$. The vertices that are red and yellow belong to $[4,8]_{\rightsquigarrow} \cap [8,4]_{\reflectbox{$\rightsquigarrow$}}$, i.e., the paths from 4 to 8. All blue vertices belongs to $[1]_{\rightsquigarrow}$.

constrained reachability

The second extension is done by giving the path more constraints, i.e., give the relation more constraints. The constraint is that a specific vertex is not included in the path. If v can reach u without visiting x then the symbol $v \overset{x}{\rightsquigarrow} u$ is used.

$$v \overset{x}{\rightsquigarrow} u \longleftrightarrow \text{a path } p = v \to u \text{ with } p \cap \{x\} = \emptyset \text{ exists} \tag{2.21}$$

This relation can simply be extended to a set of vertices (X):

$$v \overset{X}{\rightsquigarrow} u \longleftrightarrow \text{a path } p = v \to u \text{ with } p \cap X = \emptyset \text{ exists} \tag{2.22}$$

reachable set

Reachability also has another aspect that can be interesting. This aspect is to get every vertex that can be reached from a specific vertex. The reflexive reachable set of v on the digraph G is defined as:

$$[v]_{\rightsquigarrow} = \{x \mid x \in V(G) \wedge v \rightsquigarrow x\} \cup \{v\} \tag{2.23}$$

Note that v is also part of the reachable set even if $v \not\rightsquigarrow v$. Another way to formulate this is that a vertex x is in $[v]_{\rightsquigarrow}$ if and only if $x = v$ or a path $v \to x$ exists.

constraint reachable set

It makes sense to define also reachable sets for the constraint reachability relation. Thus, the set of all vertices that can be reached from v without passing thorough u is defined as:

$$[v,u]_{\rightsquigarrow} = \begin{cases} \{x \mid x \in V(G) \wedge v \overset{u}{\rightsquigarrow} x\} \cup \{v,u\} & \text{if } v \rightsquigarrow u \\ \{x \mid x \in V(G) \wedge v \overset{u}{\rightsquigarrow} x\} \cup \{v\} & \text{otherwise} \end{cases} \tag{2.24}$$

Note that if not $v \rightsquigarrow u$, then the set is equivalent to $[v]_{\rightsquigarrow}$.

Such a set can also be constructed in the other direction, i.e., every vertex that can reach v without passing through u. This set is defined as:

$$[v,u]_{\reflectbox{$\rightsquigarrow$}} = \begin{cases} \{x \mid x \in V(G) \wedge x \overset{u}{\rightsquigarrow} v\} \cup \{v,u\} & \text{if } u \rightsquigarrow v \\ \{x \mid x \in V(G) \wedge x \overset{u}{\rightsquigarrow} v\} \cup \{v\} & \text{otherwise} \end{cases} \tag{2.25}$$

Note that $[v,u]_{\rightsquigarrow} \cap [u,v]_{\reflectbox{$\rightsquigarrow$}}$ is equivalent to the union of all paths that start at v and ends at u. An example for different reachable sets is shown in Figure 12.

2.2.3 Graph Representations

A graph can be represented in many different ways. Most common are the edge list, the adjacency matrix, and the incidence matrix. An edge list is simply a list of all edges as vertex pairs. An adjacency matrix M is a matrix where every row and column stand for a vertex. If an edge (v, u) exists, than there is an one in row v and column u. Otherwise, the field is zero.

$$M(v,u) = \begin{cases} 1 & \text{if } (v,u) \in E(G) \\ 0 & \text{otherwise} \end{cases} \tag{2.26}$$

A little more complex is the incidence matrix. For the incidence matrix the symbol $\mathbf{A}$ is used. Here the rows are the vertices, and the columns are the edges. If an edge $e = (v, u)$ exists, then there is a minus one in the column from edge e in the row of v and an one in the row of u. All other fields in the column are zero.

incidence matrix

$$\mathbf{A}(x, e = (v,u)) = \begin{cases} -1 & \text{if } x = v \\ 1 & \text{if } x = u \\ 0 & \text{otherwise} \end{cases} \tag{2.27}$$

2.2.4 Cycles

Another structure is of special interest for directed graphs. It is also based on paths, i.e., a path where an edge from the last to the first vertex exists. Thus a cylce order is created. This structure is called cycle. If the cycle C is created with the path $(v = c_1, \ldots, c_n = u)$ then the cyclic set $\circlearrowleft v, \ldots, u \circlearrowright$ defines the cycle. The cycle edges (denoted by $E(C)$) are $\{(c_1, c_2), \ldots, (c_{n-1}, c_n), (c_n, c_1)\}$ and the cycle vertices are given directly by the cyclic set $\circlearrowleft v, \ldots, u \circlearrowright$. Note that the order of the path represents the cycle order as shown in Subsection 2.1.3.

cycle

These cycles are important because the existence or absence of them make many problems hard or simple. Under these simple solvable problems in acyclic graphs belongs the vertex ordering problem (compare Subsection 2.3.2). Thus, directed graphs can be divided into two subtypes: the cyclic graphs and directed acyclic graph (DAG), where the latter contains no cycle. The absence of cycles means that the graph is a combination of non-contradicting paths. Since every path has a total order, the whole graph has a partial order. This partial order corresponds to the reachability relation of the DAG.

By definition, the vertices c_i are pairwise distinct and indexed consecutively along C. Recall, open cycle intervals ($C(c_i{:}c_j)$) contain only the interior of the unique path in C connecting the defining endpoints c_i and c_j. Thus, $C(c_1{:}c_2) = \emptyset$ if $(c_1, c_2) \in E(C)$ and $C(v{:}v) = C \setminus \{v\}$ for all $v \in C$. The *cycle-distance* (denoted by $d_C(c_i, c_j)$) of two vertices c_i and c_j along a cycle C is the length of the directed path, i.e., the number of edges, from c_i to c_j. More explicitly,

$$d_C(c_i, c_j) := \begin{cases} j - i & \text{if } i < j \\ j + |C| - i & \text{if } i \geq j \end{cases} = |C(c_i{:}c_j)| + 1 \tag{2.28}$$

since the number of inner vertices is one less than the number of edges. In particular, $d_C(v, v) = |C|$ for all $v \in C$. The cycle-distance $d_C(.,.)$ is not symmetric. Instead, the equality $d_C(u, v)+d_C(v, u) = |C|$ holds for all two vertices $u \neq v \in C$. Another useful consequence of the definition of $d_C(.,.)$ is:

$$d_C(v, w) < d_C(v, u) \quad \Longrightarrow \quad d_C(w, u) = d_C(v, u) - d_C(v, w) \tag{2.29}$$

Two overlapping cycles do not necessarily create a larger cycle because maybe the vertices are not longer pairwise distinct. Never the less such overlapping cycle complexes are of interest. However, they are described differently by using the reachability relation. This relation is not an equivalence relation. It is neither reflexive nor symmetric. However, it can be used to create an equivalence relation. A vertex v is strongly connected to a vertex u if $[v]_{\rightsquigarrow} = [u]_{\rightsquigarrow}$. This relation is an equivalence relation. An equivalence class of this is called a strongly connected component (SCC).

SCC

Every vertex of a cycle is in the same equivalence class. Furthermore, note that every vertex that is not part of a cycle is in its own equivalence class. This equivalence class with only one element, that has no loop (i.e., an edge to itself), is called a singleton SCC. The set of all non-singleton SCC is defined as:

$$\mathfrak{S}_G = \{S \mid S \text{ is a non-singleton SCC of } G\}. \tag{2.30}$$

acyclic component

Another handy definition is the acyclic component of a digraph G. The acyclic component is the set of all vertices that are part of a singleton SCC. The acyclic component is defined as:

$$\overline{A}_G = \{v \mid v \in V(G) \text{ and } \{v\} \text{ is in a singelton SCC } \}. \tag{2.31}$$

By construction, the induced subgraph of the acyclic component includes no cycles and every vertex that is not part of a cycle. Note that besides the name, the acyclic component must not be connected. Thus, the induced subgraph could consist of many connected components, even if the initial digraph has only one connected component.

Every vertex $v \in V(G)$ belongs by definition exactly to one SCC (S). Either is S singelton and thus $v \in \overline{A}_G$ or $S \in \mathfrak{S}_G$. Thus, $\mathfrak{S}_G$ and $\overline{A}_G$ together are a partition of $V(G)$. This partition is defined as:

$$\mathfrak{P}_G = \mathfrak{S}_G \cup \{\overline{A}_G\} \tag{2.32}$$

Examples of different graph types are shown in Figure 13. The figure includes undirected and directed graphs. However, also some subtypes of the directed graph are shown. The subtypes are DAG, cyclic graph, and tree (see next section).

2.2.5 Oriented Trees

tree

An oriented tree T is a connected DAG in which there is a single vertex, called the *root*, denoted as $\mathrm{root}(T)$, with in-degree zero, and every other vertex has in-degree one. The vertices with out-degree zero are the *leaves*. Given an edge $(u, v) \in E(T)$,

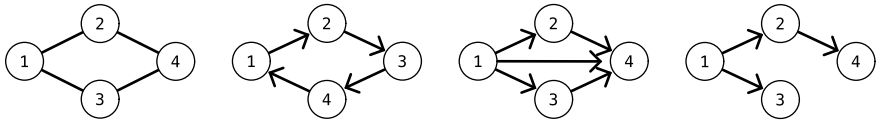


Figure 13: Basic graph types. The shown graph types are (from left to right) undirected graph, cyclic graph, DAG, and tree. The last three types are subtypes of the directed graph type. In the undirected graph, the edges have no direction. Thus, they are represented as lines. In the directed graphs, the edges have a direction. Thus, they are displayed as arrows. The arrowhead of an edge e points to the vertex $\mathrm{head}(e)$. A directed graph has either a cycle and is a cyclic graph, or it has no cycle and is a DAG. Therefore, this definition partitions the directed graphs. Thus, a tree is also a DAG.

u is the parent of v, denoted as $u = \mathrm{parent}(v)$, while v is a child of u. By definition, there is an unique directed path $p_T(v)$ from $\mathrm{root}(T)$ to $v \in V(T)$. The *ancestor partial order* $\prec$ on $V(T)$ is defined by $u \prec v$ if and only if $v \in p_T(u) \setminus \{u\}$. The *least common ancestor* ($\mathrm{lca}(x, y)$) of two vertices $x, y \in V(T)$ is the $\prec$-minimal vertex in $p_T(x) \cap p_T(y)$. The subtree rooted in v $T(v)$ is the subgraph of T induced by the vertex set $\{x \mid x \in V(T) \wedge x \prec v\} \cup \{v\}$, i.e., those that are reachable along a directed path from v.

subtree

Let assume that T is endowed with an arbitrary order of successors for each $v \in V(T)$. A subtree $T(u)$ is a prior subtree of $T(v)$ if u and v are both children of a common parent $w = \mathrm{parent}(u) = \mathrm{parent}(v)$ and u comes before v in the local ordering of the successors of w. Now, consider two vertices u and v such that u and v are incomparable w.r.t. to the ancestor order and set $w = \mathrm{lca}(u, v)$. Note that u, v, and w are pairwise distinct. Let x and y be the children of w such that $u \in T(x)$ and $v \in T(y)$. Then u is prior to v, notated as $u \lhd v$, if $T(x)$ is a prior subtree of $T(y)$, i.e., x comes before y in the local ordering of the successor set of w. The relation $\lhd$ is a partial order known as the *sibling partial order* of T.

Corollary 1. *The ancestor and the sibling orders of an oriented tree are orthogonal, i.e., for any pair of vertices exactly one of the relation $x = y$, $x \prec y$, $y \prec x$, $x \lhd y$, or $y \lhd x$ is true.*

It is well known that the two fundamental traversal orders of trees are obtained as the two natural compositions of the ancestor and the sibling partial orders. A traversal reports the vertices of the tree in a specific order. Two different types of traversal can be distinguished: either the root of a subtree is reported before the subtree (preorder) or after the subtree (postorder). Denote by ρ and π the bijections of the *preorder* and the *postorder*, respectively.

preorder and postorder

$$\begin{aligned} \rho(x) < \rho(y) \text{ iff } x \lhd y \text{ or } y \prec x \\ \pi(x) < \pi(y) \text{ iff } x \lhd y \text{ or } x \prec y \end{aligned} \tag{2.33}$$

It follows immediately that preorder and postorder together determine the ancestor

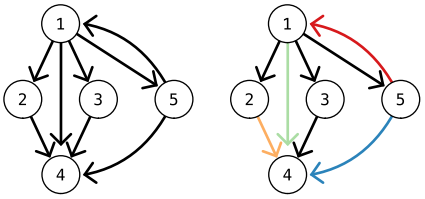


Figure 14: Tree Edge Types. On the left a graph is shown for which a search tree rooted at 1 is created. The order of the labels gives the sibling order. On the right, the edges are colored after their type w.r.t. the tree. Black: tree edge (forward edge); green: forward edge; yellow: right cross edge; blue: left cross edge; and red: back edge.

and sibling order:

$$\begin{aligned} x \triangleleft y \text{ iff } \rho(x) < \rho(y) \text{ and } \pi(x) < \pi(y) \\ x \prec y \text{ iff } \rho(x) > \rho(y) \text{ and } \pi(x) < \pi(y) \end{aligned} \tag{2.34}$$

search tree

It is possible to create trees from arbitrary digraphs. These trees are of interest because they give a special perspective on the graphs. To construct such trees recall that for every vertex beside r in the set $[r]_{\rightsquigarrow}$ of the digraph G a path from r exists. These paths can be chosen such that every $x \in [r]_{\rightsquigarrow}$ is reachable from r along a unique path, and hence there is an oriented tree T with $V(T) = [r]_{\rightsquigarrow}$ that is a subgraph of G. An oriented tree T with root $r \in V(G)$ is a *search tree* on G if there is no directed edge $(x, y) \in E(G)$ with $x \in V(T)$ and $y \notin V(T)$. An ordered tree T is a search tree if and only $V(T) = [r]_{\rightsquigarrow}$ because a vertex $y \in V(G) \setminus V(T)$ by definition cannot be reached from anywhere in $V(T)$, and thus also not from the root, while every $y \in V(T)$ is by definition reachable from the root r.

The induced subgraph $G[V(T)]$ can have more edges than the search tree T rooted at r. However, it is possible to classify every edge by T.

edge types

Definition 15. *Let G be a digraph, and T a search tree. If $(v, u) \in E(G[V(T)])$ then (v, u) is a*

(i) forward edge *iff $u \prec v$, i.e., $\rho(v) < \rho(u)$ and $\pi(v) > \pi(u)$;*

(ii) backward edge *iff $v = u$ or $v \prec u$, i.e., $\rho(v) \geq \rho(u)$ and $\pi(v) \leq \pi(u)$;*

(iii) left cross edge *iff $u \triangleleft v$, i.e., $\rho(v) > \rho(u)$ and $\pi(v) > \pi(u)$;*

(iv) right cross edge *iff $v \triangleleft u$, i.e., $\rho(v) < \rho(u)$ and $\pi(v) < \pi(u)$.*

Note that every tree edge ($e \in E(T)$) is a forward edge. The backward edge is in this thesis shortened to back edge. Examples of the different types are shown in Figure 14.

forest

A forest F is an extension of a tree. A forest consists of a finite ordered set of roots $\{r_1, \ldots, r_k\}$ and for every root r_i a tree T_i. These trees are constructed in the way that every vertex is only in one tree. This unambiguity is archived with the order of the roots:

$$V(T_{r_i}) = [r_i]_{\rightsquigarrow} \setminus \bigcup_{j<i} V(r_j). \tag{2.35}$$

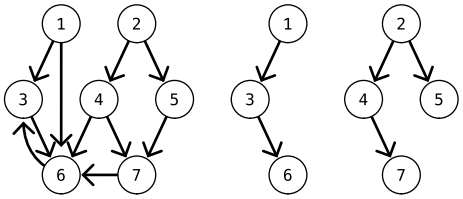


Figure 15: Search Forest. On the left side a graph is shown that can be covered with the search forest shown on the right side. The roots of the forest are 1 and 2. The tree on the first root is also a search tree. The other tree is not because the first tree already covers the vertices 6 and 3.

This can expressed as: every vertex that can be reached from a r_i is in the tree T_i unless it can be reached from a r_j with $r_j < r_i$ where $<$ is the order of the ordered root set. This construction creates an important implication:

$$(v, u) \in E(G) \wedge v \in V(T_i) \wedge u \in V(T_j) \to r_j < r_i. \tag{2.36}$$

This construction makes it possible to see the forest as a tree with an artificial root r that is the parent of the real roots $\{r_1, \ldots, r_k\}$ where the order of the real roots is the sibling order. After Equation 2.36 is every edge between subtrees of r a left cross edges. The construct directly gives the preorder and the postorder of the forest. Let ρ_i be the preorder of the tree T_i. Than the preorder ρ of the forest is defined with the help of the order combination (Definition 2 (Page 18)) as:

$$\rho = \sum_{i \in [1:k]} \rho_i. \tag{2.37}$$

Analogously is the postorder π of the forest is defined as:

$$\pi = \sum_{i \in [1:k]} \pi_i. \tag{2.38}$$

Note that the sum here means a chain of order combination. For the results it is important that this chain follow the order of the ordered root set. This is essential because the order combination is not communicative.

A forest that fulfill Equation 2.35 also fulfills

$$\bigcup_{i \in [1:k]} [r_i]_{\rightsquigarrow} = V(F). \tag{2.39}$$

Therefore, the tree with the artificial root r would be a search tree. Thus, is the forest with this construction is called a search forest. An example of a search forest is shown in Figure 15.

2.2.6 Colored Graph

A colored graph is in some way only a property graph with a specific property. Thus, all previous defined graph types exist also as a colored version. The colored graphs have two sub types: The edge colored graphs and the vertex colored graphs. In

edge colored graphs, every edge has a color property. Thus every edge belongs to a specific type of edges. In vertex colored graphs, the vertices have the color property. A graph can be both vertex and edge colored.

This property stands out of the other properties because it is often used in many different ways: to divide edge types, to mark edges or vertices, or in problem definitions (Burr, 1984).

2.2.7 Multigraph

In the here used definition of a graph G the edge set $E(G)$ is a set. Thus, every edge can be contained only once. In a multigraph this restriction is not present. It can have multiple edges between the same vertices.

However, for simplicity, in this work, it is assumed that for multigraphs all graph functions work exactly the same as for normal graphs. This could be implemented by giving every edge a hidden id. Thus, two edges (v, u) look the same but can distinguished if necessary.

This modification only apply to the edges. Thus, the neighbor set ($\mathrm{neighbor}(v)$), and the successor set ($\mathrm{suc}(v)$) of a vertex v are independent from the number of edges (v, u). Note that this is not true for $\mathrm{outdeg}(v)$, and $\mathrm{indeg}(u)$.

2.2.8 De Bruijn Graph

A specially labeled directed graph is a de Bruijn graph. This structure is introduced by De Bruijn (1946) without using the term graph. The idea is that every vertex is labeled with a sequence of letters from a defined alphabet. All sequences have the same length k. Thus they are called k-mers. An edge from a vertex v to a vertex u exists if the k-mer of v have a suffix of length $k-1$ that is a prefix of the k-mer of u. If all these k-mers and edges are present, the graph is called a complete de Bruijn graph.

Idury and M. S. Waterman (1995) utilize the idea of a de Bruijn graph, in a way that does not use the complete de Bruijn graph but subgraphs of it. These subgraphs are created by using a more extended sequence of the same alphabet or many of these longer sequences. The graphs then only contain these k-mers that are subsequences of the longer sequences. Furthermore, it only contains these edges where instances of this k-mers share the suffix (i.e., the prefix) in the longer sequences. For more details, how such a graph is used view Section 2.4. An example of a de Bruijn graph is given in Figure 16.

2.2.9 A-Bruijn Graph

A de Bruijn graph could be seen as a graph where the vertices represent an alignment, where every vertex aligns precisely one letter. The first (or last) letter of the k-mer. This letter is aligned with the rest of the k-mer as context. Thus, every time the same letter with the same context appears in the longer sequences (i.e., the same k-mer appears) this is gathered in this one vertex. This gathering is the same as creating an alignment.

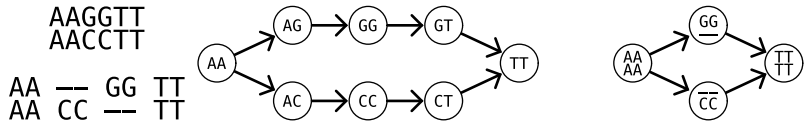


Figure 16: Bruijn graphs. Left, two sequences (top) and a plausible alignment of this sequence (bottom) are shown. In the center, the corresponding de Bruijn graph with two-mers is shown. On the right the A-Bruijn graph of the alignment is shown. Note that if consecutive vertices of the de Bruijn graph have been merged, a graph with four vertices is the result.

In this interpretation an edge from one aligned letter to the next aligned letter is then given if this letter directly follows it in the longer sequences. Thus, it connects adjacent alignments of the longer sequences.

This concept describes an A-Bruijn graph. The vertices are alignments, and the edges connect adjacents alignments. Thus, a de Bruijn graph is an A-Bruijn graph, but other alignments are possible. An example of an A-Bruijn graph is given in Figure 16. A simple practical example is a de Bruijn graphs that have different k sizes in it. Such a graph is technical not a de Bruijn graph, but an A-Bruijn graph.

Another example is something that already Idury and M. S. Waterman (1995) suggested. If one k-mer follows another k-mer and no other outgoing edge exist from this k-mer, then the graph can be simplified by merging this both vertices together without losing information (compare Subsection 2.6.2). However, if this is done, the graph is not any longer a de Bruijn graph but an A-Bruijn graph.

2.3 General ordering methods

One can find many methods of ordering something in the literate. The focus here is on the ordering of graphs or techniques that can directly be applied to a graph representation. Thus many practices of order theory are not presented here. However, such a full survey would be beyond the scope of this work.

In this section, the focus is on known methods to create a total order of graph vertices without any more in-depth knowledge of the graph and what the vertices represent. Thus, these methods are general graph methods that can be applied to any digraph.

2.3.1 Graph traversal

A graph traversal starts on a specific vertex or a set of vertices and traverses the graph by following the edges. If every vertex is only visited once the traversal creates a search forest of oriented trees. This search forest represents the edges that are used by the traversal. The literature describes two main types of graph traversal: the breadth-first search (BFS) and the depth-first search (DFS).

Depth-First Search

DFS traverses a digraph G from a root $r \in V(G)$ in the following manner: start the recursive traversal at r; (i) recursively, at $v \in V(G)$, proceed to the smallest (with respect to the sibling partial order $\lhd$), previously-unvisited successor of v; (ii) if v has no more unvisited successor, return to (i) of its "parent", i.e., the vertex $\mathrm{parent}(v)$ from which v is initially reached (R. Tarjan, 1972). Clearly, DFS generates a rooted tree T with directed edges $(\mathrm{parent}(v), v)$, which are known as the DFS-tree.

Lemma 1. *Let T be the ordered tree generated by DFS on a digraph G, and let $(v,u) \in E(G)$ with $v \in V(T)$. Then, $u \in V(T)$ and either $u = v$, $u \prec v$ (including $(v,u) \in E(T)$), $v \prec u$, or $u \lhd v$. In particular, T is a search tree on G. Furthermore, no edge $(v,u) \in E(G)$ exists such that $v \lhd u$.*

Proof. Consider a DFS reaching v. The search returns to $\mathrm{parent}(v)$ only after exhausting all successors of v; hence, any edge (v,u) either has been visited before by the DFS process or, otherwise, it is included as an edge as DFS continues into the subtree of v rooted in u. If u has been accessed before, then either $u = v$, $u \prec v$ (i.e., v has a succesor x with $x \lhd u$, u is traversed from x), $v \prec u$ (i.e., v is traversed from u), or v and u are incomparable w.r.t. $\prec$. In the latter case, there are distinct children x and y of $\mathrm{lca}(v,u)$ such that $v \in T(x)$ and $u \in T(y)$. In a DFS, $T(y)$ is traversed before $T(x)$ if y comes before x in the successor order of $\mathrm{lca}(v,u)$, and thus, $u \lhd v$.

By the construction of DFS, $x \in V(T)$ is reachable from the root $r = \mathrm{root}(T)$ along a path in G; hence, $V(T) \subseteq [r]_{\rightsquigarrow}$. Suppose there is $x \in [r]_{\rightsquigarrow} \setminus V(T)$. Along a path p from r to x, let x' be the first vertex not reachable from $V(T)$, i.e., there is an edge $(u,x') \in E(G)$ with $u \in V(T)$ and $x' \notin V(T)$, contradicting the first assertion of the lemma.

The fact that no edge $(v,u) \in E(G)$ exists with $v \lhd u$ follows directly from the first part of the lemma and Corollary 1 (Page 31). □

DFS-tree

Therefore, the following simple characterization of DFS-trees is obtained:

Corollary 2. *A search tree T on G is a DFS-tree if and only if no edge $(v,u) \in E(G[V(T)])$ is a right cross edge, i.e., $v \lhd u$.*

The DFS proceeds on $[r]_{\rightsquigarrow}$ in such a way that the preorder ρ of the DFS-tree T rooted at r corresponds to the order in which the vertices are discovered, while the postorder π describes the order in which vertices are completed, i.e., "left", by ascending back to their parent. To see this, denote by ρ' and π' the order in which vertices are discovered and completed by DFS started at r. By construction, DFS accesses the successors of v in $\lhd$-order and completes the traversal of a subtree rooted at a child v' of v before proceeding to the subtree of another child. Thus, if u and v are incomparable w.r.t. $\prec$ in T, then $\rho'(u) < \rho'(v)$ and $\pi'(u) < \pi'(v)$ if and only if $u \lhd v$ in the sibling order. It also follows directly from the definition of DFS that $\rho'(u) < \rho'(v)$ if $v \prec u$ and $\pi'(u) < \pi'(v)$ if $u \prec v$. Hence, ρ' and π' indeed coincide with the preorder ρ and the postorder π for the traversal of

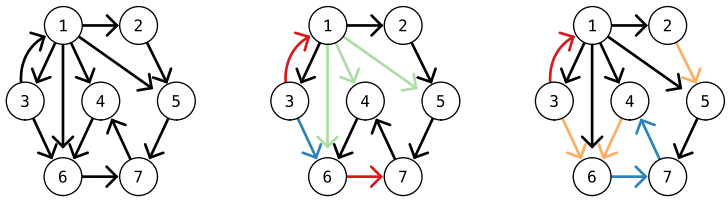


Figure 17: DFS- and BFS-Trees. On the left a graph is shown from which a DFS-tree (middle) and a BFS-tree (right) are created. The siblings are visited in the order of their labels. The edges are colored after their type w.r.t. the tree as in Figure 14 (Page 32). Note that the DFS-tree has no right cross edge and the BFS-tree has no forward edge beside the tree edges.

DFS-tree T. DFS on a graph G is therefore completely described by the oriented DFS-tree T, i.e., the sibling and ancestor order on $[r]_{\rightsquigarrow}$, and coincides with DFS on T itself.

Hence, the condition that v has been accessed before u can be expressed simply as $\rho(v) < \rho(u)$. If u and v are comparable on T, their relative order is determined by Equation 2.34 (Page 32). An example of a DFS-tree is shown in Figure 17.

DFS-forest

It is possible to create a DFS-forest by having a forest where all trees are DFS-trees. As shown in Subsection 2.2.5 the edges between the trees are all left cross edges. Thus, the DFS-forest has the same properties as a DFS-tree.

Breadth-First Search

BFS

BFS traverses a digraph G from a root $r \in V(G)$ in the following manner: put r in an empty list; (i) visit the first unvisited vertex v on the list; (ii) add the successors of v in sibling partial order $\triangleleft$ to the end of the list; if an unvisited vertex in the list exist proceed with step (i). Clearly, BFS generates a rooted tree T with directed edges $(\mathrm{parent}(v), v)$, where $\mathrm{parent}(v)$ is the vertex that put v in the list. Such a tree is known as the BFS-tree.

Lemma 2. *Let T be the ordered tree generated by BFS on a digraph G, and let $(v,u) \in E(G)$ with $v \in V(T)$. Then, $u \in V(T)$ and only if $(v,u) \in E(T)$ is $u \prec v$. In particular, T is a search tree on G.*

Proof. Consider v is the first unvisited vertex on the list. Then two possibilities exists. If u is not already in the list then the first appearance is put in the list from v, i.e., (v,u) is a tree edge. If u is already in the list then v' exists in front of v in the list such that (v',u) is a tree edge. However, as v' is in front of v, v can not be an ancestor of v'. This follows because v' is by assumption visited and thus already in the tree but v is just added now to the tree. Thus, if u is already in the list $u \prec v$ cannot be true. By the same arguments as for DFS in the proof of Lemma 1, T is a search tree. □

BFS-tree

It is possible to give a simple definition of a BFS-tree that is similar to the definition of a DFS-tree:

Corollary 3. *A search tree T on G is a BFS-tree if and only if every forward edge $(v,u) \in E(G[V(T)])$ (i.e., $u \prec v$) is a tree edge (i.e., $(v,u) \in E(T)$).*

A BFS on the BFS-tree is equivalent to a BFS on G that starts at the root of the tree. However, the order in which the BFS visits the vertices is different from the preorder and postorder of the BFS-tree. An example of a BFS-tree is shown in Figure 17.

BFS-forest

It is possible to create a BFS-forest by having a forest where all trees are BFS-trees. As shown in Subsection 2.2.5 the edges between the trees are all left cross edges. Thus, the BFS-forest has the same properties as a BFS-tree.

2.3.2 Topological sorting

topological sorting

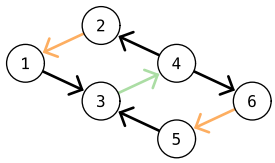
Remember that if a directed acyclic graph (DAG) is given, the reachability relation is a partial order. The extension of this partial order into a total order is called topological sorting. Assume that in the partial order i and j incomparable then this missing information is added. If $i \leq j$ or $j \leq i$ is irrelevant as long the transitivity is fulfilled. Thus, one partial order can have many extension to total orders.

DFS-topological sorting

Different types of algorithms perform topological sorting. Kahn (1962) presented the first efficient algorithm. However, as R. E. Tarjan (1976) has shown, it is possible to use a DFS for this. If F is a DFS-forest that uses the sources as roots, then the reverse postorder ($\overline{\pi}$) of F is a topological sorting of the DAG. In such a case $\overline{\pi}$ is called a DFS-topological sorting.

Not every graph is a DAG. The solution to this problem is first to extract a maximum acyclic subgraph and then to compute a topological sorting of this subgraph. An equivalent formulation asks for the removal of a minimum set of edges that close cycles. This *Maximum Acyclic Subgraph* or minimum feedback arc set problem (MFAS) is well-known to be *NP*-hard (Karp, 1972). Nevertheless fast, practicable heuristics have been devised, see e.g., Eades, X. Lin, and Smyth (1993) and Saab (2001). An example of a graph where MFAS and topological sorting is used to create an order is shown in Figure 18.

A closely related approach is the *Linear Ordering Problem* (LOP): Given a complete weighted directed graph, find a tournament with maximum total edge weights (Martí and Reinelt, 2011). It yields essentially the same model since *LOP* and MFAS can be transformed into each other quite easily (Grötschel, Jünger, and Reinelt, 1984). Cost functions designed to define consensus orderings for sets of total and partial orders have been considered in different fields starting with the work of Spearman (1904) and Kendall (1938), see also e.g. Collier and Konagurthu (2014), Fagin, Kumar, and Sivakumar (2003), and Fried et al. (2004). The reconciliation of partial orders is investigated in detail, e.g., in D. Bertrand, Blanchette, and El-Mabrouk (2009).



421653 465213

426153 462513

426513 462153

Figure 18: MFAS and Topological Sorting. Left, a graph is shown that has two cycles ($\circlearrowleft 1, 3, 4, 2 \circlearrowright$ and $\circlearrowleft 3, 4, 6, 5 \circlearrowright$). If the orange edges are removed, the graph is acyclic. However, this is not a valid solution for MFAS. Since removing the green edge can also make the graph acyclic and one edge less is removed. If the graph is acyclic after removing the green edge, it can be topological sorted. All six valid topological sortings are shown on the right. The red topological sorting are DFS-topological sortings.

2.3.3 Simultaneous consecutive ones and matrix banding

Instead of edges the incidence matrix $\mathbf{A}$ of the digraph G is considered. Then an approach to solve the problem is to sort both the vertices and their edges in such a way that, to the extent that this is possible, (i) adjacent vertices are consecutive and (ii) edges that have a vertex in common are consecutive. In more formal terms, the goal is to sort both the rows (vertices) and columns (edges) of the incidence matrix in such a way that rows and columns show all non-zero entries consecutively. A rectangular matrix $\mathbf{A}$ that admits such a pair of row and column permutations is said to have the simultaneous consecutive ones property (C1S) (Oswald and Reinelt, 2009). This is possible if G is an union of disjoint paths. Note that instead of edges one could also cover the graph with short paths $\wp_k$. In this case, column k identifies the vertices incident with path k. Again, if G is an union of paths, the path-incidence matrix satisfies C1S. It is not difficult to see that $\mathbf{A}$ satisfies C1S if and only if $\mathbf{A}$ has the well-studied consecutive ones property (Booth and Lueker, 1976; Meidanis, Porto, and Telles, 1998; Oswald and Reinelt, 2009) for both its rows and columns. Thus C1S can be checked in linear time (Booth and Lueker, 1976). Furthermore, Tucker (1972) characterization of C1S in terms of forbidden sub-matrices also carries over. Some direct connection between the consecutive ones property and the *Betweenness Problem*, are discussed in Christof, Oswald, and Reinelt (1998).

In general, $\mathbf{A}$ have not the consecutive ones property. The problem of identifying a minimal number of columns (edges) whose removal leaves a C1S matrix is *NP*-complete (Oswald and Reinelt, 2009). In practice, it may be desirable to quantify the extent of the violation of C1S in terms of intervals of consecutive zeros enclosed by two non-zeros. For instance, one may want to use $\omega = \sum_i h_i$, where the sum runs over all intervals i of consecutive zeros enclosed by two non-zeros, and $h_i \geq 0$ is some contribution that monotonically grows with the length of the zero-interval. For a given ordering of the rows and columns, the total violation is quantified as the sum of the ω values. It should be noted, however, that C1S does not imply G is an union of disjoint paths.

A related set of optimization problems is concerned with reducing the bandwidth

of matrices, i.e., the maximal distance of non-zero entries from the diagonal (in a symmetric case) or the parameter $\min(l, u) + l + u$ (for rectangular matrices); here $u = \max(\{i - j \mid \mathbf{A}_{ij} \neq 0\})$ and $l = \max(\{j - i \mid \mathbf{A}_{ij} \neq 0\})$ (Reid and Scott, 2006). In the symmetric case, several good heuristics are known, starting with the Cuthill and McKee (1969) and Gibbs, Poole, and Stockmeyer (1976) algorithms even though the problem is *NP*-hard (Feige, 2000), while the general case has received much less attention (Reid and Scott, 2006). Bandwidth reduction methods do not eliminate "bad" edges that eventually determine bandwidth. The resulting ordering of rows and columns thus may be very inaccurate locally.

2.3.4 Hamiltonian Path

A Hamiltonian path is a path that visits every vertex exactly once. It is similar to a Hamiltonian cycle, i.e., a cycle that visits every vertex, beside the start/end vertex, exactly once. Both can be transformed with a polynomial reduction into each other. Kirkman and Cayley (1856) and Hamilton (1856) independently described the concept of such a cycle. Later Karp (1972) shows that finding a Hamiltonian cycle and thus a Hamiltonian path is *NP*-complete. An example of such a cycle is shown in Figure 19.

A Hamiltonian path, as an ordered set, defines a total order of the vertices directly. Thus it can be used to create an order of vertices in graphs if one exists. An example where a Hamiltonian path is used this way is to determine a genome assembly based on an overlap graph (See Section 2.4). Another example would be the evolution of gene clusters described by Prohaska et al. (2017).

2.3.5 Eulerian Paths

The Eulerian path problem differs in many aspects from the other problems presented here. It goes back to Euler (1741). It tries to find a trail that contains every edge exactly once. Note that the term Eulerian path is historically given even if in the modern notion, the result is not necessarily a path. Comparable to the Hamiltonian path a cycle version is given: An Eulerian cycle is an Eulerian path that starts and ends in the same vertex. An example of such a cycle is shown in Figure 19.

Thus the result of the Eulerian path is a total order on the edges, not the vertices. An edge ordering can contain every vertex many times thus it does not correspond to an unique vertex ordering. On the other side a vertex ordering does not consider every edge and thus it also does not correspond to an unique edge ordering. However, depending on the problem, it may be possible to formulate an ordering problem in both ways, as edge ordering or vertex ordering. An example of this is the genome assembly problem (See Section 2.4).

Except the most of the other problems the Eulerian path problem can relatively simply be solved. Determining if an Eulerian cycle exists and the construction of it can be done in linear time (Hierholzer and Wiener, 1873). The critical point is that an Eulerian cycle exists exactly if the graph can be decomposed into edge-disjoint cycles (and is connected). Thus, every vertex has the same in-degree and out-degree.

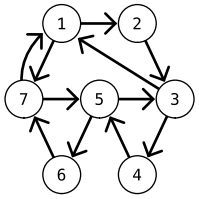


Figure 19: The Hamiltonian cycle and an Eulerian cycle. A graph is shown that has a hamiltonian cycle $\circlearrowleft 1, 2, 3, 4, 5, 6, 7 \circlearrowright$ and am eulerian cycle $\circlearrowleft 1, 2, 3, 4, 5, 6, 7, 1, 7, 5, 3 \circlearrowright$. For both cycles, 1 is used as the start. Note that every other vertex would also work because they are cycles.

However, generating a graph having an Eulerian path from a graph without a Eulerian path is complex. For this task methods like discarding edges if in- and out-degree are unequal or discarding sources and sinks until only one of each is left are used. Examples of this can be found in Section 2.4.

2.4 Genome Assembly

The genome assembly problem arose when genome sequencing arrived as widely used technology in the laboratories. The problem is that sequencing produces relative short sequence fragments (called reads), but scientists are interested in the complete genome sequence, which is much larger. Thus, the task is to reconstruct a longer sequence from short pieces. It is possible to solve this because the genomes are over-sampled, i.e., the fragments overlap, and thus, relative positions can be determined. A genome assembler is a program that addresses this problem. However, this task is difficult because sequencing works not errorless, and many genomes have repeats in the sequence. Thus, the genome assembly problem is *NP*-hard (Nagarajan and Pop, 2009).

The naive solution can be summed up to the overlap–layout–consensus paradigm. This paradigm is used in the first assemblers, like the Genome Assembly Program (GAP) (Bonfield, K. F. Smith, and Staden, 1995). In the overlap phase, the assemblers calculate overlaps between all reads. In the layout phase, the relative order is used to create a total order. The consensus phase then transforms the total order in a consensus sequence. This sequence is the genome assembly. Like GAP most of the assemblers that use this paradigm need human interaction in at least the layout step. The reader can find an overview of this type of assemblers at J. R. Miller, Koren, and Sutton (2010).

This idea later is enhanced by using graphs in the layout steps. Then creating a layout is equivalent to finding a Hamiltonian path in such an overlap graph. However, the computation of all overlaps is time consuming because every read must be compared with any other read. Thus the application of such graphs is limited. Pevzner, Tang, and M. S. Waterman (2001) paraphrase this circumstance graphically:

> Children like puzzles, and they usually assemble them by trying all possible pairs of pieces and putting together pieces that match. Biologists assemble genomes in a surprisingly similar way, the major difference being that the number of pieces is larger.

Idury and M. S. Waterman (1995) propose a new approach that use de Bruijn graphs as the data structure to overcome the pairwise comparison. The construction of the de Bruijn graphs begins with the transformation of the reads in smaller fragments of size k. Such a fragment is termed a k-mer. In the de Bruijn graph, a k-mer is a vertex. The edges are created by connecting two overlapping k-mers following the de Bruijn graph definition (Subsection 2.2.8).

graph simplifier

The genome assembly problem can be solved effectively with a de Bruijn graph. Idury and M. S. Waterman (1995) further show that addressing the genome assembly problem is similar to finding an Eulerian path in the graph. However,the size of the problem is too large for direct solving. The solution to this problem is reducing the size and complexity of the graph. Graph simplifications reach this reduction. Their characteristic is to address a local graph structure and to replace it with a more straightforward structure.

The first and most trivial of this simplifiers is the "elimination of singletons" (Idury and M. S. Waterman, 1995). A singleton is a vertex that has precisely one predecessor and one successor. Such a vertex adds no variants in the Eulerian path determination. Thus, it is replaced by an edge from the predecessor to the successor. The simplifier reduces the number of vertices in the graph but does not change any topological properties of it.

However, to effectively solve the Eulerian path problem, the topological features must be reduced. Two other simplifiers are addressing the topology of the graph. The first one is "elimination of forks" (Idury and M. S. Waterman, 1995). A fork is a vertex that has one incoming (outgoing) edge and more than one outgoing (incoming) edge. Such a vertex can exists only once in the Eulerian path. Otherwise, extra incoming (outgoing) edges would be needed. Thus, the simplifier discards all outgoing (incoming) edges beside one.

The second topology simplifier is "elimination of crosses" (Idury and M. S. Waterman, 1995). A cross is a vertex with more than one incoming and outgoing edge. Such a vertex can and should be more than once in the Eulerian path. The solution here is that a pair of an incoming and an outgoing edge are combined. This combination creates (like the Elimination of singletons) a new edge and removes the edges from the cross.

Both topology simplifier need to have a local heuristic that prefers one edge respectively pair of edges over others. In Idury and M. S. Waterman (1995) the more supported edge (pair of edges) is used. More supported means in this context that it appears in more reads. However, on a theoretical point of view, this heuristic could be replaced by any local decidable heuristic that is further optimized.

Every applied simplifier creates the possibility of further simplification. Thus, the simplifiers are repeated on the graph until no new changes happen. If the heuristics work correctly, the result is a graph with one edge that contains all data. However, Idury and M. S. Waterman (1995) already show a limitation of their heuristic. It cannot solve the repeats that are larger than k. Thus, the result of real data is not one edge but a small graph of cycles.

Further heuristics are applied until these repeats are solved and the graph reduces to one edge. This one edge then contains all information about the order of the genome assembly. This order is then transformed into an assembly using

alignments. In other words, the genome assembly follows directly from the ordering of the graph, and this ordering is created only by local heuristics.

This idea later is used to build the first de Bruijn graph-based assembler. The assembler is named after the Eulerian path "Euler" (Pevzner, Tang, and M. S. Waterman, 2001). It uses the above-described techniques combined with a preluded error correction. This error correction simplifies the graph and pushes the quality of the result. Thus, Euler superseded the overlap–layout–consensus assemblers.

Velvet (Zerbino and Birney, 2008) is another genome assembler that utilizes de Bruijn graphs. It uses different graph simplifications than Euler. First, like in Euler, the singletons are simplified. Then in Velvet two simplifiers exists that use the graph topology. First, "tips" are removed. A "tip" is a sink or source that connect only to one other vertex. Secondly, bubbles are reduced. Two independent paths that start and end in the same vertices form a bubble. A bubble is a colinear part of the graph with at least two alternative paths. An exact definition is given later in Subsection 2.6.3.

The bubble detection and simplification is the essential approach of Velvet. However, it takes too much time to compute all bubbles precisely. Thus, Velvet applies a heuristic that constrains the bubble detection. The idea is that the alternative paths in bubbles result from sequencing errors. Therefore, these paths are merged by alignments into one path.

Velvet further reduces graph complexity by discarding low supported edges. Again supported means that it is not present in many reads. This discarding step can introduce errors if some sequences of the genome are underrepresented in the reads. Thus this step is applied after the bubble simplification.

The last step in Velvet's pipeline contains the decomposition of cycles, i.e., repeats on the genome. Velvet uses data that is not present in the de Bruijn graph. Such data includes the full reads, mate pairs, and apriori statistics. Then the pipeline starts over and tries to simplify the graph further until only one vertex is left.

Many other genome assemblers use this approach with little variations. The basic ideas of filtering and correction of the reads, building the de Bruijn graph, simplify the graph and solve the repeats is the same in most de Bruijn graph assemblers. Many assemblers also use the three simplifier ideas: merge singletons, remove tips, and solve bubbles. The reader can find an overview of assemblers in El-Metwally et al. (2013).

2.5 Supergenome

Coordinatization of supergenomes is answering how multiple sequence alignment (MSA)-blocks of a genome-wide multiple sequence alignment (gMSA) can be ordered in a way that facilitates comparative studies of genome annotation data. To be more general, a particular interest is in large animal and plant genomes and large phylogenetic ranges. Therefore, short MSA-blocks and abundant genome rearrangements, leaving only short sequences of MSA-blocks that are perfectly syntenic between all genomes involved, can be assumed. The problem of optimally

sorting the MSA-blocks can be regarded as a quite particular variant of a vertex ordering problem. In the computational biology literature, furthermore, several graph-based methods have been proposed to solve the problem of sorting MSA-blocks for supergenomes, see e.g. Haussler et al. (2018), Kececioglu (1993), Nguyen, Hickey, Zerbino, et al. (2015), Paten, Earl, et al. (2011), Paten, Herrero, et al. (2008), and Pevzner, Tang, and Tesler (2004).

Several specialized graph structures have been introduced recently to tackle the problem of ordering sequence blocks, which is a problem very similar to the coordinatization of supergenomes. Among these constructions are A-Bruijn graphs, Enredo graphs, and Cactus graphs (see Kehr et al. (2014) for a review). A key insight of Kehr et al. (2014) is that these graph representations are equivalent in the sense that they can be transformed into each other. They differ, however, in additional information extracted from the input alignment that is stored as vertex and edge labels. In the following two models are explained: the sequence graphs and the bidirected graphs.

2.5.1 Sequence graphs

The sequence graphs of Haussler et al. (2018) are related closely to the A-Bruijn graphs. The critical difference is that the orientation of the sequences in MSA-blocks is used to determine the direction of the edge. Two adjacent intervals with negative orientation thus imply an edge that is reversed compared to the A-Bruijn graph. This situation is problematic in the case where the orientation of the intervals is switched. In Haussler et al. (2018), a preprocessing step is performed to minimize the number of such edges. The sequence graph approach is designed for the comparison of human genomes of different individuals. In such a scenario, the resulting information loss is small and does not present a practical problem.

The natural formulation of the total ordering problem on a sequence graph is to find a vertex ordering that minimizes weighted feedback edges and the average cut width. These optimization criteria ensure that the successor relations are kept mostly intact and at the same time, successors are placed close to each other in the solution. Both problems, the *Minimum Feedback Arc Set Problem* (Karp, 1972) and the *Average Cut-Width Minimization Problem* (Gavril, 1977; Makedon, Papadimitriou, and Sudborough, 1985; Martí, Pantrigo, et al., 2013), are known to be *NP*-hard. Cut-width minimization problems ask for a linear ordering of the vertices of a graph such that the average or the maximum number of edges spanning across the gap between a pair of consecutive vertices is minimized. Conceptually, cut-width problems are quite similar to bandwidth problems (Barth et al., 1995). In Haussler et al. (2018) a heuristic is presented that first extracts a totally ordered "backbone" and then inserts the remaining vertices into the backbone order that is kept intact in the process. While the presence of a global common backbone order is a well-founded assumption for pangenomes of a single or very closely related species, it is violated in general graphs.

2.5.2 Bidirected graphs

The bidirected graphs of Nguyen, Hickey, Zerbino, et al. (2015) have the same underlying graph as an A-Bruijn graph. While the A-Bruijn graph uses a standard directed graph structure, bidirected graphs encode directional information independently in the endpoints of each edge, distinguishing three cases: (i) adjacent MSA-blocks have the same orientation, (ii) the connected MSA-blocks switch from minus to plus orientation, or (iii) *vice versa*. The latter two cases indicate a change of orientation between two changes. In Nguyen, Hickey, Zerbino, et al. (2015), this basic structure is extended by additional transitive edges given by a legal path through the graph with an exponential weight function. Then the task is to find a consistent (non-conflicting) set of edges with maximal weight, i.e., such that a Hamiltonian path along or with support of this edges exist. This form of the weight function takes the biological background into account. Due to the genetic linkage, close MSA-blocks are rarely separated. This gives higher weight to locally correct MSA-block positions and orientations. In general, this problem is *NP*-hard. While this sorting problem is *NP*-hard (Nguyen, Hickey, Zerbino, et al., 2015) in general, for a set of closely related genomes, the effort is particularly suitable.

2.6 Graph simplifier

The simplifier is a concept that originated in the genome assembly. More or less every graph-based assembler has its simplifier or modifies an already existing one. Here are the ground ideas of the basic types given and explained. These basic types are dead end simplifier, linear simplifier, bubble simplifier, and superbubble simplifier.

2.6.1 Dead ends

This type of simplifier origins directly from Idury and M. S. Waterman (1995) which is the first work that uses de Bruijn graphs. The idea is that in a genome assembly, many dead ends can be produced by false reads where only a small number of real ends exists. For every chromosome exists one source and one sink, every additional sink or source is a sequencing error.

Thus these dead ends are removed, if they are not sufficiently supported. What sufficiently supported means depends on the assembler. It must especially be prevented that real ends are discarded, i.e., the chromosomes are shortened every time. However, most of them use such a simplifier. How an end is qualified also depends on the environment.

Three types can be distinguished. The ones that are connected only to one vertex. The second type includes ends that connect to more than one vertex but an order is given of the other vertices. The last type are ends that connect to more vertices with an unclear order. This unclear order could be created by cycles or if the vertices cannot be compared by reachability. Examples of the three types are shown in Figure 20.

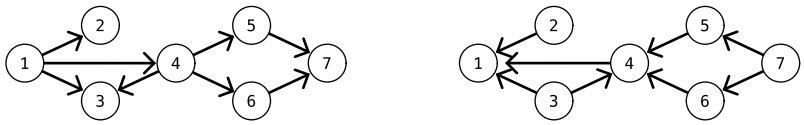


Figure 20: Different End Types. Different types of sinks (left) and sources (right) are shown. Both can be differentiated into three types. The ends that are connected only to one other vertex (vertex 2 in both graphs). The second type comprises the ends that are connected to more than one vertex, but these vertices have a specific order (vertex 3 in both graphs). The last type comprises the ends that connected to more than one vertex with unclear order (vertex 7 in both graphs).

Ends that connect only to one vertex can be removed or collapsed with this vertex. The ends that have more than one predecessor are more complex. If an order exists, they can be handled like an end that is only connected with the last neighbor in the order. However, if such an order does not exist, the end cannot be classified as a dead end or real end. Depending on how the order is determined, it may be the one or the other.

2.6.2 Consecutive Vertices

The concept of consecutive vertices has already present be presented in Idury and M. S. Waterman (1995). If a vertex v has only vertex u as successor and v is the only predecessor of u then these two vertices are called consecutive vertices. More formal this means that the following equation holds:

$$[v, u]_{\rightsquigarrow} = [u, v]_{\leftsquigarrow} = \{v, u\}. \tag{2.40}$$

This forms a linear part in the graph, and the two vertices can be merged without loss of information.

Some assemblers do this as a preprocessing and not as a simplification step (Zerbino and Birney, 2008). However, this form of preprocessing cannot help if other simplifiers create new consecutive vertices. Thus, in most cases, it is more effective for simplification to repeat this process. However, it is also more time-consuming.

The simplicity of the structure makes this process valid for every graph ordering, completely independent of the optimization function. Thus, it is present in every genome assembler. For supergenomes, it is in some cases ignored because by definition every MSA-block has more than one successor. However, this can change by other simplifier or by filtering on the alignment. Thus, consecutive vertices should be considered.

2.6.3 Bubbles

Bubbles or colinear parts of the graph are considered already in Idury and M. S. Waterman (1995). However, the name bubble first is used in Zerbino and Birney (2008) for genome assembly. Bubble structures in a digraph have become the focus

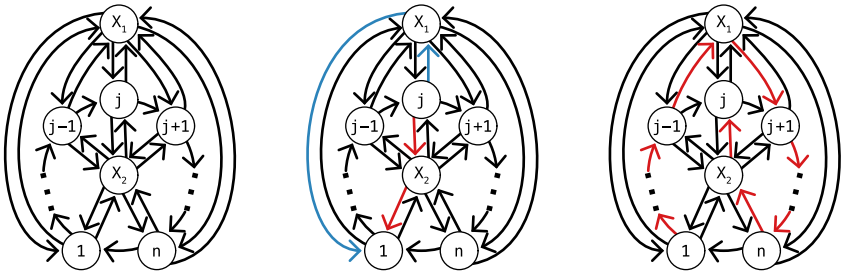


Figure 21: Quadratic number of bubbles. An artificial graph that consists of a directed cycle with n vertices ($\circlearrowleft 1, \ldots, n \circlearrowright$) and two extra vertices X_1 and X_2 is shown on the left. Every vertex of the cycle has besides the cycle edge an outgoing edge to both extra vertices and an incoming edge from both. Thus the graph has $n+2$ vertices and $5n$ edges (n cycle edges, $2n$ edges to the extra vertices, and $2n$ edges from the extra vertices). However, every pair of none extra vertices creates a bubble. Two simple paths would be from the entrance to X_1 or X_2 then to the exit (shown in the center). Thus, there are n^2 bubbles. Each of these bubbles contains the complete graph. The reason for this is that, a Hamilton path from every vertex to every other vertex exist. Such a Hamilton path from i to j can be constructed in the following way: Starting by i following the cycle until $j-1$, follow the edges $(j-1, X_1)$ and $(X_1, j+1)$ (skiping j on the cycle), follow again the cycle until $i-1$, and follow the edges $(i-1, X_2)$ and (X_2, j). In this way every vertex is visited once. An example of such a Hamilton path is shown on the right, the path consist out of the red edges.

of a growing interest of research because of their role in genome assembly and related topics; see, e.g., Paten, J. M. Eizenga, et al. (2018) and the references therein. See Acuña et al. (2017) for a formal analysis. A bubble is defined as:

bubble

Definition 16. *Let G be a digraph and let (s,t) be an ordered pair of distinct vertices with the set $U = [s,t]_{\rightsquigarrow} \cap [t,s]_{\leftsquigarrow}$. Then the induced subgraph $G[U]$ is a* bubble *in G if and only if $s \rightsquigarrow t$. Then s, t, and $U \setminus \{s,t\}$ are called the* entrance, exit, *and* interior *of the bubble. The induced subgraph $G[U]$ is denoted by $\prec s,t \succ$ if it is a bubble with entrance s and exit t.*

If such a bubble is found the different paths in them can be unified until only one path is left. How this is done depends then again on the assembler. However, this simple definition of a bubble hides complex problems.

The first problem that appears is that the detection of a bubble is not possible in linear time. The simple reason for this is that there are quadratic possibilities of bubbles (compare Figure 21). Thus, the detection can be time-consuming on large graphs. To overcome this some assembler gives limits to the bubble detection (Zerbino and Birney, 2008). That creates a linear algorithm with a constant factor that misses some bubbles.

The second problem is that bubbles can substantially overlap (compare Figure 21). Graphs exist where every bubble contains the complete graph. If this

is the case, bubble structures give more or less no information about the local structure. As a consequence, even if the entrance and the exit of the bubble are known, the computation of the interior of the bubble is time-consuming. Thus, again assemblers work with constant limits. However, this means that maybe not the complete bubble is found and a path that should also be unified is ignored.

The last problem is, that over the interior of the bubble nothing is known. There could exist cycles or other complex structures that cannot be unified. Thus, the complete effort to detect a bubble and get every vertex in it could be useless because a contained cycle makes the unification of the bubble impossible.

The assembler tries to solve the problems with sophisticated heuristics that produce excellent results for the assembly problem. Such an approach works relatively well because de Bruijn graphs are well formatted and have limits for every neighborhood even if they are complete. Thus, a worst case graph like the one in Figure 21 is not possible as a de Bruijn graph. However, this is not helpful for the ordering of arbitrary digraphs or the supergenome problem. Thus another bubble-like structure is more interesting: the superbubble.

Before a more in-depth look into superbubbles, it makes sense in the scope of this work to define a supertype of a bubble. Such a bubble-like subgraph is called bubbloid and is defined as:

bubbloid **Definition 17.** *Let G be a digraph and let (s,t) be an ordered pair of distinct vertices with the set $U = [s,t]_{\rightsquigarrow} \cap [t,s]_{\leftsquigarrow}$. Then the induced subgraph $G[U]$ is a* bubbloid *in G if and only if $s \rightsquigarrow t$. Then s, t, and $U \setminus \{s,t\}$ are called the* entrance, exit, *and* interior *of the bubbloid. The induced subgraph $G[U]$ is denoted by $\prec s,t \succ$ if it is a bubbloid with entrance s and exit t.*

Clearly, every bubble is also a bubbloid. Furthermore, also, consecutive vertices are bubbloids. On one side, this definition is so vague that there is no benefit in calculating any bubbloid. On the other side, the general structure makes it the superclass of different bubble-like structures, including superbubbles.

2.6.4 Superbubbles

Superbubbles are another type of bubble-like structures. They are introduced by Onodera, Sadakane, and Shibuya (2013) as an extension of bubbles. However, bubbles are not the supertype of superbubbles. There exist bubbles that are not superbubbles and vice versa. An example for both is shown in Figure 22. However, both are subtypes of bubbloids.

Here the definition of Gärtner, Höner zu Siederdissen, et al. (2018) is used that is a simple rephrasing of the language used in Onodera, Sadakane, and Shibuya (2013). This rephrasing is done by first considering a more general class of structure which are obtained by omitting the minimality criterion:

superbubbloid **Definition 18.** *Let G be a digraph, (s,t) be an ordered pair of distinct vertices, and $U \subseteq V(G)$. Then the induced subgraph $G[U]$ is a* superbubbloid *in G if the following three conditions are satisfied:*

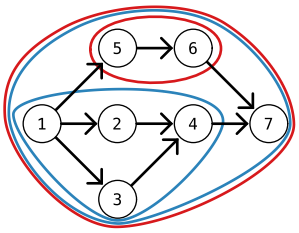


Figure 22: Comparison of Bubbles and Superbubbles. In the graph, the bubbles are marked blue, and the superbubbles are marked red. The bubble $\prec 1, 4 \succ$ is not a superbubble because the edge $(1, 5)$ exists. The superbubble $\langle 5, 6 \rangle$ corresponds to consecutive vertices and is not a bubble because $5 \not\rightsquigarrow 6$. However, the superbubble $\langle 1, 7 \rangle$ is also a bubble $\prec 1, 7 \succ$. Furthermore, besides the mini superbubbles (like $\langle 5, 6 \rangle$), every superbubble is also a bubble.

(S1) $s \rightsquigarrow t$ *(Reachability condition).*

(S2) $U = [s, t]_{\rightsquigarrow} = [t, s]_{\rightsquigarrow}$ *(Matching condition).*

(S3) $G[U]$ *is acyclic (Acyclicity condition).*

Then s*,* t*, and* $U \setminus \{s, t\}$ *are called the* entrance*,* exit*, and* interior *of the superbubbloid. The induced subgraph* $G[U]$ *is denoted by* $\langle s, t \rangle$ *if it is a superbubbloid with entrance* s *and exit* t*.*

Note that the reachability condition is redundant. If the matching condition is fulfilled, the reachability condition is also fulfilled. However, in the work of Onodera, Sadakane, and Shibuya (2013), the definition is not formal, and thus, both conditions are given. In this work, the reachability condition is kept to be consistent with the previous works.

A superbubble is a superbubbloid that is minimal in the following sense:

Definition 19. *A superbubbloid* $\langle s, t \rangle$ *is a superbubble if there is no* $s' \in [s, t]_{\rightsquigarrow} \setminus \{s\}$ *such that* $\langle s', t \rangle$ *is a superbubbloid (Minimality condition).* ***superbubble***

The conditions are represented graphically in Figure 23.

Even if bubbles are not the supertype of superbubbles, most of the superbubbles are also bubbles with only one class of exception. This class is called the mini superbubbles and have the property of an empty interior, i.e., they only consist of two vertices: the entrance and the exit. However, such a mini superbubble corresponds to two consecutive vertices. Thus, a sub-relation can be created by defining the superbubbles as a subset of the union of all consecutive vertex pairs and bubbles (compare Figure 22). **mini superbubbles**

Superbubbles are an important class of subgraphs in the context of assembly for many reasons. Beside the fact that they remove the need of a distinct consecutive vertices simplifier, they also tackle most of the drawbacks of bubbles. This is done for the cost of reducing the number of found subgraphs that can be handled. However,

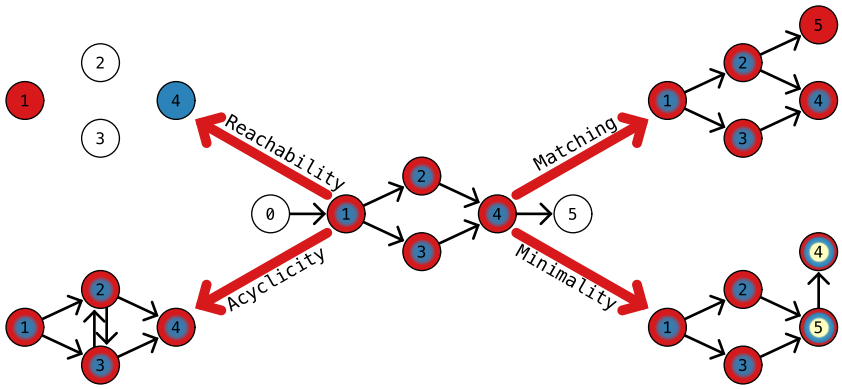


Figure 23: Superbubble Conditions. In the middle, a graph with the superbubble $\langle 1, 4\rangle$ is shown. Every vertex that is in $[1, 4]_{\rightsquigarrow}$ is marked red and all vertices in $[4, 1]_{\rightsquigarrow}$ are marked blue. In the corners, transformations of this superbubble are shown that violate each a superbubble condition. The top left violates the reachability condition because 1 cannot reach 4. The top right violates the matching condition because 5 can be reached from 1 but cannot reach 4. The acyclicity condition is violated in the bottom left because a cycle with 2 and 3 exists. The minimality condition is violated in the bottom right because $\langle 1, 4\rangle$ is a superbubbloid but $\langle 5, 4\rangle$ (yellow marked) is also a superbubbloid. Thus, $\langle 1, 4\rangle$ is not a superbubble.

as already Onodera, Sadakane, and Shibuya (2013) have shown, superbubbles, in assembly graphs, help to simplify the graphs. Furthermore, a portion of the missed bubbles is not usable for simplification because of there structure.

The first treated drawback is that there are only linear superbubbles in the graph. This follows directly from the fact that every vertex can at most be the exit of one superbubble. Thus, linear time detection is possible, and in fact, linear time detection algorithms exist. Since the definition of superbubbles is entirely based on reachability, multiple edges are irrelevant and can be omitted altogether. Hence, only simple digraphs are considered.

Onodera, Sadakane, and Shibuya (2013) proposed a simple $\mathcal{O}(|V(G)|(|E(G)| + |V(G)|))$-time algorithm that, for each candidate entrance s, explicitly retrieves all superbubbles $\langle s, t\rangle$ in G.

The combination of the work of Sung et al. (2015) with the improvement of Brankovic et al. (2016) results in the state of the art $\mathcal{O}(|E(G)| + |V(G)|)$-time algorithm. The concept of a superbubble is extended to bi-directed and bi-edged graphs, called ultrabubble in Paten, J. M. Eizenga, et al. (2018) and Rosen, J. Eizenga, and Paten (2017). The enumeration algorithm for ultrabubbles in Paten, J. M. Eizenga, et al. (2018) has a worst case complexity of $\mathcal{O}(|E(G)| \cdot |V(G)|)$, and hence does not provide an alternative for directed graphs.

The second treated drawback are overlaps. Superbubbles can only overlap in two ways. Either is one of the overlapping superbubbles included in the other, or the exit of one superbubble is an entrance of the other superbubble. The second

case is not a relevant overlap. Thus, such overlaps can be ignored. The first case is also not problematic. Depending on the solving algorithm, it could be possible to ignore the inner vertices or solve first the inner and then the outer.

The last treated drawback is that the bubbles can contain cycles and thus are hard to solve. A superbubble cannot contain any cycles by definition. Thus, a superbubble is a directed acyclic graph (DAG), which means that every found superbubble can be used.

Beside the usability in assembly, superbubbles can be used for any ordering method that is presented in Section 2.3. For graph traversals, topological sorting, and simultaneous consecutive ones property (C1S), a superbubble can be seen as an atomic unit. Thus, the optimal solution of a superbubble can be precomputed and then handled as one entry. Such an approach can help to calculate a faster result for the complex problems.

It is a little different if the Hamiltonian and the Eulerian paths are considered. The existence of a non-mini superbubble means that the solution may be impossible. For a Hamiltonian path, this directly follows because multiple paths exist, but the entrance and exit can only be visited once. For an Eulerian path, the entrance and exit can be visited more often. However, the superbubble must completely consist of x edge independent paths. The entrance must have x incoming edges, and the exit must have x outgoing edges. Otherwise, no Eulerian path can exist.

Thus, a modified superbubble simplifier can be used in every ordering problem, including the supergenome problem. More details on how it is used in the supergenome problem can be found at Chapter 4. However, first, a more in-depth look into the theory of superbubbles is given in Chapter 3. Chapter 3 also presents two novel detection algorithms that have benefits over the existent ones.

CHAPTER 3

Superbubbles

Contents

In this chapter the properties of superbubbles are analyzed in detail. The first part is based on Gärtner, Müller, and Stadler (2018). Section 3.7 and following are based on Gärtner and Stadler (2019). This two works jointly create a novel state of the art detection algorithm for superbubbles.

3.1 State of the Art

The state of the art algorithm for detecting superbubbles is the combination of two works: Sung et al. (2015) and Brankovic et al. (2016). However, the algorithm has some missing details that lead to false positive reports of superbubbles. An in-depth analysis of the algorithm give a deeper understanding of its pitfalls.

Recall that the vertex set of a graph (G) has a partition $\mathfrak{P}_G$ (Equation 2.32 (Page 30)), which consist of the non-singleton strongly connected components (SCCs) and the acyclic component. The key observation of Sung et al. (2015) can be stated as

Proposition 1. *Every superbubble $\langle s, t\rangle$ in G is an induced subgraph of $G[S]$ for some $S \in \mathfrak{P}_G$.*

It ensures that it is sufficient to search separately for superbubbles within $G[S]$ for $S \in \mathfrak{P}_G$. However, these induced subgraphs may contain additional superbubbles that are created by omitting the edges between different components. In order to preserve this information, the individual components S are augmented by artificial vertices (Sung et al., 2015). The augmented component S is then converted into a DAG. Within each DAG the superbubbles can be enumerated efficiently. With the approach of Sung et al. (2015), this yields an overall $\mathcal{O}(|E(G)| \cdot \log(|E(G)|))$-time algorithm, the complexity of which is determined by the extraction of the superbubbles from the component DAGs. The partitioning of G into the components $G[S]$ for $S \in \mathfrak{P}_G$ and the transformation into DAGs can be achieved in $\mathcal{O}(|E(G)| + |V(G)|)$-time. Recently, Brankovic et al. (2016) showed that superbubbles can be found in linear time within a DAG. Their improvement uses the fact that the DAG can always be topologically sorted in such a way that superbubbles appear as contiguous blocks. In this ordering, furthermore, the candidates for entrance and exit vertices can be narrowed down considerably. For each pair of entrance and exit candidates (s, t), it can then be decided in constant time whether $G[[s, t]_{\rightsquigarrow}]$ is indeed a superbubble. Using additional properties of superbubbles to further prune the candidate list of (s, t) pairs results in $\mathcal{O}(|E(G)| + |V(G)|)$-time complexity.

A careful analysis showed that the conversion into a DAG is erroneous. Sung et al. (2015) proposed the construction of an auxiliary DAG H that not only retains the superbubbles of $G[S]$ but also introduces additional ones. By the unique structure of H, these additional superbubbles can be filtered.

sung graph **Definition 20** (Sung graph)**.** *Let G be a strongly connected digraph with a search tree T with root x. The vertex set $V(H) = V' \cup V'' \cup \{a, b\}$ consists of two copies $v' \in V'$ and $v'' \in V''$ of each vertex $v \in V(G)$, an artificial source a, and an artificial sink b. The edge set of H comprises four classes of edges: (i) edges*

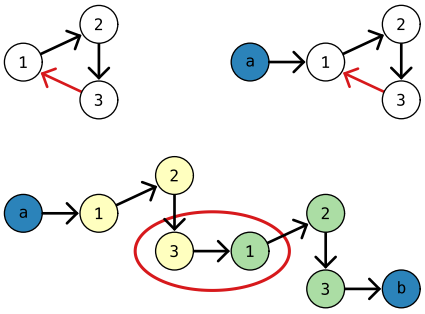


Figure 24: False-positive superbubble returned by Sung et al. (2015). On the top left a SCC is shown that is also a connected component. On the top right a similar SCC is shown with the difference that the artificial vertex a is connected to 1. In the search trees of the graphs rooted by 1 and a the red edge is the only back edge. Both graphs create the same Sung graph (bottom). Then among others the superbubble $\langle 3, 1\rangle$ is reported, which is correct for the first graph but an error for the second.

(u', v') and (u'', v'') whenever (u, v) is a forward edge in G w.r.t. T. (ii) edges (u', v'') whenever (u, v) is a back edge in G. (iii) edges (a, v') whenever (a, v) is an edge in G and (iv) edges (v'', b) whenever (v, b) is an edge in G.

The Sung graph H is a connected DAG since all back edges are removed within each copy of $V(G)$ and only edges from the first copy to the second exists. Further details can be found at Sung et al. (2015).

The errors appear if two graphs (with different sets of superbubbles) create the same Sung graph, see Figure 24. These do not constitute a fatal problem because they can be recognized easily in linear total time simply by checking the tails of incoming and heads of outgoing edges. It is nevertheless worthwhile to analyze the issue and to seek a direct remedy.

3.2 Weak Superbubbles

The main aim of this section is to prove moderate generalizations of the main results of Sung et al. (2015) and Onodera, Sadakane, and Shibuya (2013). To this end, it is convenient to rephrase some parts of the superbubbloid (Definition 18 (Page 48)) and the superbubble (Definition 19 (Page 49)) definition. The rephrase is on the reachability (S1) and matching conditions (S2) for the vertex set U of superbubbloid with entrance s and exit t.

Lemma 3. *Let G be a digraph, $U \subset V(G)$ and $s, t \in U$. Then* (S1) *and* (S2) *(of Definition 18) hold for U if and only if the following four conditions are satisfied*

(S.i) *Every $u \in U$ is reachable from s.*

(S.ii) *t is reachable from every $u \in U$.*

(S.iii) *If $u \in U$ and $w \notin U$ then every $w \to u$ path contains s.*

(S.iv) *If $u \in U$ and $w \notin U$ then every $u \to w$ path contains t.*

Proof. Suppose (S1) and (S2) are true. By (S2) is $U := [s, t]_{\rightsquigarrow} = [t, s]_{\rightsquigarrow}$. Then $u \in [s, t]_{\rightsquigarrow}$ and $u \in [t, s]_{\rightsquigarrow}$ implies, by definition of the reachable sets, that (S.i)

and (S.ii) hold. If $w \notin U$ it is not reachable from s without passing through t. Since every u is reachable from s without passing through t, it would be $w \in U$ if w is reachable from any $u \in U$ on a path not containing t, hence (S.iv) holds. Similarly, since t is reachable from u without passing through s, it would be $w \in U$ if v could be reached from w along a path that does not contain s, i.e. (S.iii) holds.

Now suppose (S.i), (S.ii), (S.iii), and (S.iv) hold. Clearly, both (S.i) and (S.ii) already imply (S1). Since $u \in U$ is reachable from s by (S.ii) and every path reaching $w \notin U$ passes through t by (S.iii), it is $U = [s,t]_{\rightsquigarrow}$. By (S.i), t is reachable from every $u \in U$ and by (S.iv) t can by reached from $w \notin U$ only be passing through s, i.e., $U = [t,s]_{\leftsquigarrow}$, i.e., $[s,t]_{\rightsquigarrow} = [t,s]_{\leftsquigarrow}$. □

Corollary 4. *Suppose U, s, and t satisfy* (S.i), (S.ii), (S.iii), *and* (S.iv). *Then every path connecting s to $u \in U$ and u to t is contained within U.*

Proof. Assume, for contradiction, that there exists an $u \to t$ path containing a vertex $w \notin \langle s,t \rangle$. By definition of the set $[s,t]_{\rightsquigarrow}$, $w \notin [s,t]_{\rightsquigarrow}$ is not reachable from $u \in [s,t]_{\rightsquigarrow}$ without passing through t first, i.e., w cannot be part of an $u \to t$ path. □

Corollary 4 shows subgraphs satisfying (S1) and (S2) related to reachability structures. This structures are explored in some detail in Ronse (2014) and Tankyevych, Talbot, and Passat (2013). In the following it is useful to consider:

(S.v) If (u,v) is an edge in U then every $v \to u$ path in G contains both t and s.

In the following, it is shown that (S.v) acts as a slight relaxation of the acyclicity condition (S3) of Definition 18 (Page 48).

Lemma 4. *Let G be a digraph, $U \subseteq V(G)$ and $s,t \in U$. If U is the vertex set of the superbubbloid $\langle s,t \rangle$, it satisfies* (S.v). *If U satisfies* (S.i), (S.ii), (S.iii), (S.iv), *and* (S.v), *then $G[U] \setminus \{(s,t)\}$, the subgraph induced by U without the potential edge from t to s, is acyclic.*

Proof. Suppose U is the vertex set of a superbubbloid with entrance s and exit t. If (u,v) is an edge in U, then $v \neq s$ by (S3). Since v is reachable from s within U, no $v \to s$ path can exist within U, since otherwise there would be a cycle, contradicting (S3), that any $v \to s$ path passes through t. There are two cases: If there exists $(t,s) \in E(G)$, any path containing this edge trivially contains both s and t. The existence of the edge (t,s) contradicts (S3). Otherwise, any $t \to s$ path contains at least one vertex $x \notin U$. By (S.iii) and (S.iv) every $v \to x$ path contains t and every $x \to u$ path contains s and t, respectively. Hence the first statement holds.

Conversely, suppose (S.v) holds, i.e., every directed cycle Z within U contains s and t. Suppose (t,s) is not contained in Z, i.e., there is vertex $u \in U \setminus \{s,t\}$ such that $(t,u) \in E$. By (S.ii), t is reachable from u without passing through s, and every $u \to t$ path is contained in U by Corollary 4. Thus there is a directed cycle within U that contains u and t but not s, contradicting (S.v). Removing the edge (t,s) thus cuts every directed cycle within U, and hence $G[U] \setminus \{(s,t)\}$ is acyclic. □

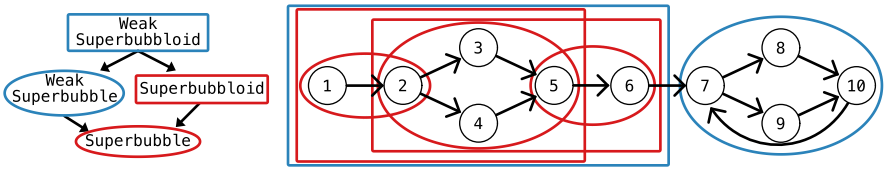


Figure 25: Overview of the different superbubbloid types. (Left) Hierarchy of the superbubbloid types. The arrows show a subset relation between the sets of the bubble-like structures. Thus, every weak superbubble and every superbubbloid are also weak superbubbloids. However, not every superbubbloid is a weak superbubble or vice versa. A superbubble on the other side is also a weak superbubble and a superbubbloid (and a weak superbubbloid). On the right is a graph including all types of superbubbloids. The bubble-like structures are shown with the same coding as in the hierarchy, i.e., weak bubble-like structures are blue, and none weak structures are red. Furthermore, bubbloids are marked with a rectangle and bubbles with a circle. The representation is based on the hierarchy. Thus, if a superbubble is marked, it is not marked as weak superbubble or superbubbloid. Note that every superbubbloid in the weak superbubbloid $\langle 1, 6\rangle$ is either a superbubbloid or a superbubble (compare Lemma 5 and Corollary 8).

Although the definition of Onodera, Sadakane, and Shibuya (2013) (our Definition 19 (Page 49)) is also used in Sung et al. (2015), the notion of a superbubble is tacitly relaxed in Sung et al. (2015) by allowing an edge (t, s) from exit to entrance, which contradicts the acyclicity condition (S3). This suggests the following definition:

Definition 21 (Weak Superbubbloid)**.** *Let G be a digraph, $U \subseteq V(G)$ and $s, t \in U$. The subgraph $G[U]$ induced by U is a* weak superbubbloid *if U satisfies* (S.i)*,* (S.ii)*,* (S.iii)*,* (S.iv)*, and* (S.v)*.* ***weak superbubbloid***

A weak superbubbloids with entrance s and exit t is denoted by $\langle s, t\rangle$ and write $[s, t]_{\rightsquigarrow}$ for its vertex set. As an immediate consequence of Definition 21 and Lemma 4 it is:

Corollary 5. *A weak superbubbloid $\langle s, t\rangle$ is a superbubbloid in G if and only if $(t, s) \notin E(G)$.*

The possibility of an edge connecting t to s plays a role below, hence the focus is on weak superbubbloids in this contribution.

Definition 22. *A weak superbubbloid $\langle s, t\rangle$ is a* weak superbubble *if there is no interior vertex s' in $\langle s, t\rangle$ such that $\langle s', t\rangle$ is a weak superbubbloid.* ***weak superbubble***

Figure 25 shows a graph in which all (weak) superbubbloids and superbubbles are indicated.

3.3 Properties of (Weak) Superbubbles

In this section the theory of superbubbles in digraphs is revisited in some more detail. Although some of the statements below have appeared similarly in the literature (Brankovic et al., 2016; Onodera, Sadakane, and Shibuya, 2013; Sung et al., 2015), this thesis gives concise proofs and takes care to disentangle properties that depend on minimality from those that hold more generally.

First it is observed that a weak superbubbloid contained within another weak superbubbloid must be a superbubbloid because the existence of an edge from exit to entrance contradicts (S.v) for the surrounding weak superbubbloid. This fact is recorded as

Lemma 5. *If $\langle s,t\rangle$ and $\langle s',t'\rangle$ are weak superbubbloids with $s',t' \in V(\langle s,t\rangle)$ and $\{s',t'\} \neq \{s,t\}$, then $\langle s',t'\rangle$ is a superbubbloid.*

The result is important in the context of minimal (weak) superbubbloids below. Another immediate consequence of Lemma 4 is

Corollary 6. *Let $\langle s,t\rangle$ be a weak superbubbloid in G. If there is an edge (u,v) in $\langle s,t\rangle$ that is contained in a cycle, then every edge in $\langle s,t\rangle$ is contained in a cycle containing s and t.*

Proof. By (S.v) there is a cycle running through s and t. Let (u,v) be an edge in $\langle s,t\rangle$. Since u is reachable from s and v reaches t within U, there is a cycle containing s, t, and the edge (u,v). □

Next a few technical results are derived that set the stage for considering minimality among weak superbubbloids.

Lemma 6. *Assume that $\langle s,t\rangle$ is a weak superbubbloid and let u be an interior vertex of $\langle s,t\rangle$. Then $\langle s,u\rangle$ is a superbubbloid if and only if $\langle u,t\rangle$ is a superbubbloid.*

Proof. Suppose$\langle s,u\rangle$ is a weak superbubbloid. Set $W_{ut} := ([s,t]_{\rightsquigarrow} \setminus [s,u]_{\rightsquigarrow}) \cup \{u\}$ and consider $w \in W_{ut}$. The subgraph induced by W_{ut} is an induced subgraph of $\langle s,t\rangle \setminus \{(t,s)\}$. Hence it is acyclic and in particular $(t,u) \notin E(G)$, i.e., (S.v) and even (S3) holds. Since $t \notin [s,u]_{\rightsquigarrow}$ every path from s to t runs through u. Since w is reachable from s there is a path from s through u to w, i.e., w is reachable from u. Thus (S.i) holds. (S.ii) holds by assumption since t is reachable from w. Now suppose $v \notin W_{ut}$ and $w \in W_{ut}$. If $v \notin [s,t]_{\rightsquigarrow}$, then every $v \to w$ path passes through s and then through u, the exit of $\langle s,u\rangle$ before reaching w. If $v \in [s,t]_{\rightsquigarrow}$, then $v \in [s,u]_{\rightsquigarrow} \setminus \{u\}$ and thus every $v \to w$ path passes through u as the exit of $\langle s,u\rangle$. Hence W_{ut} satisfies (S.iii). If $v \in [s,t]_{\rightsquigarrow}$, then $v \in [s,u]_{\rightsquigarrow} \setminus \{u\}$ and thus every $w \to v$ path passes through s. By (S.v) there is no $w \to s$ path within $\langle s,t\rangle \setminus \{(t,s)\}$, and thus any $w \to v$ includes (t,s) or a vertex $y \notin [s,t]_{\rightsquigarrow}$. By construction, all $w \to y$ paths contain t, and thus all $w \to v$ paths also pass through t and W_{ut} also satisfies (S.iv).

Conversely suppose $\langle u,t\rangle$ is a superbubbloid. It must be shown that $W_{su} := ([s,t]_{\rightsquigarrow} \setminus [u,t]_{\rightsquigarrow}) \cup \{u\}$ induces a superbubbloid. The proof strategy is very similar. As above is observed that (S.v), (S.i), and (S.ii) are satisfied. Now consider $v \notin W_{su}$ and $w \in W_{su}$. If $v \notin [s,t]_{\rightsquigarrow}$ then every $v \to w$ path contains s; otherwise $v \in [u,t]_{\rightsquigarrow} \setminus \{u\}$ and $v \to w$ passes through t and thus also through s by Corollary 4, thus (S.iii) holds. If $v \in [s,t]_{\rightsquigarrow}$, then $v \in [u,t]_{\rightsquigarrow} \setminus \{u\}$, in which case every $w \to v$ path passes through u. Otherwise $v \notin [s,t]_{\rightsquigarrow}$ then every $w \to v$ runs through $t \in [s,t]_{\rightsquigarrow}$ and thus in particular also through u. Hence (S.iv) holds. □

Lemma 7. *Let $\langle w,u\rangle$ and $\langle s,t\rangle$ be two weak superbubbloids such that u is an interior vertex of $\langle s,t\rangle$, s is an interior vertex of $\langle w,u\rangle$, w is not contained in $\langle s,t\rangle$ and t is not contained in $\langle w,u\rangle$. Then the intersection $\langle s,u\rangle = \langle w,u\rangle \cap \langle s,t\rangle$ is also a superbubbloid.*

Proof. First consider the intersection $\langle s,u\rangle$. $u \in V(\langle s,t\rangle)$ is reachable from s, hence (S1) holds. Furthermore $\langle s,u\rangle$ is an induced subgraph of $\langle s,t\rangle \setminus \{(t,s)\}$ and hence again acyclic (S3). Set $W_{su} := [w,u]_{\rightsquigarrow} \cap [s,t]_{\rightsquigarrow}$ and consider $v \in W_{su}$. First, note that v is reachable from s by definition of $\langle s,t\rangle$ and u is reachable from v by definition of $\langle w,u\rangle$. Let $x \notin W_{su}$ and $v \in W_{su}$. If $x \notin [s,t]_{\rightsquigarrow}$ then every $x \to v$ path passes through s; if $x \in [s,t]_{\rightsquigarrow}$ then $x \notin [w,u]_{\rightsquigarrow}$ (and $v \in [w,u]_{\rightsquigarrow}$) and thus every $x \to v$ path passes through w. Since $w \notin [s,t]_{\rightsquigarrow}$, every $x \to v$ path contains s. If $x \notin [w,u]_{\rightsquigarrow}$, then every $v \to x$ path passes through u; otherwise $x \in [w,u]_{\rightsquigarrow}$ but $x \notin [s,t]_{\rightsquigarrow}$, thus every $v \to x$ path passes through $t \notin [w,u]_{\rightsquigarrow}$ and hence also through u. Thus W_{su} is a superbubbloid. □

The following results are included for completeness, although it is irrelevant for the algorithmic considerations below.

Lemma 8. *Let $\langle w,u\rangle$ and $\langle s,t\rangle$ be defined as in Lemma 7. Then the union $\langle w,t\rangle = \langle w,u\rangle \cup \langle s,t\rangle$ is superbubbloid if and only if the induced subgraph $\langle w,t\rangle$ satisfies* (S.v).

Proof. Since $\langle w,s\rangle$, $\langle s,u\rangle$, $\langle u,t\rangle$ are superbubbloids, t is reachable from w, i.e., (S1) holds. By the same token, every $v \in W_{wt} := [w,u]_{\rightsquigarrow} \cup [s,t]_{\rightsquigarrow}$ is reachable from w or s and reaches u or t. Since s is reachable from w and t is reachable from u, every $v \in W_{wt}$ is reachable from w and reaches t. Now consider $x \notin W_{wt}$ and $v \in W_{wt}$. If $v \in [w,u]_{\rightsquigarrow}$ every $x \to v$ path passes through w; if $v \in [s,t]_{\rightsquigarrow}$, it passes through $s \in [w,u]_{\rightsquigarrow}$ and thus also through w. If $v \in [s,t]_{\rightsquigarrow}$, then every $v \to x$ path passes through t. If $v \in [w,u]_{\rightsquigarrow}$ it passes through $u \in [s,t]_{\rightsquigarrow}$ and thus also through t. Thus W_{wt} satisfies (S2). Thus $\langle w,t\rangle$ is a weak superbubbloid if and only if (S.v) holds. □

Lemma 9. *Let $\langle s,t\rangle$ be a weak superbubbloid in G with vertex set $[s,t]_{\rightsquigarrow}$. Then $\langle s,t\rangle$ is a weak superbubbloid in the induced subgraph $G[W]$ whenever $[s,t]_{\rightsquigarrow} \subseteq W$.*

Proof. Conditions (S.i), (S.ii), and (S.v) are conserved if G is restricted to $G[W]$. Since every $w \to u$ and $u \to w$ path with $u \in [s,t]_{\rightsquigarrow}$ and $w \notin [s,t]_{\rightsquigarrow}$ within W is

also such a path in $V(G)$, it is concluded that (S.iii) and (S.iv) are satisfied w.r.t. W whenever they are true w.r.t. the larger set $V(G)$. □

The converse is not true. The restriction to induced subgraphs thus can introduce additional (weak) superbubbloids. As the examples in Figure 24 (Page 55) show, it is also possible to generate additional superbubbles.

The "non-symmetric" phrasing of the minimality condition in Definition 19 (Page 49) and Definition 22 (Brankovic et al., 2016; Onodera, Sadakane, and Shibuya, 2013; Sung et al., 2015) is justified by Lemma 6: If $\langle s, t\rangle$ and $\langle s, t'\rangle$ with $t' \in V(\langle s, t\rangle)$ are superbubbloids, then $\langle t', t\rangle$ is also a superbubbloid, and thus $\langle s, t\rangle$ is not a superbubble. As a direct consequence of Lemma 5, furthermore, it is:

Corollary 7. *Every superbubble is also a weak superbubble.*

Lemma 6 also implies that every weak superbubbloid, which is not a superbubble itself, can be decomposed into consecutive superbubbles:

Corollary 8. *If $\langle s, t\rangle$ is a weak superbubbloid, then it is either a weak superbubble or there is a sequence of vertices v_k with $s = v_1, \ldots, v_k = t$, $k \geq 3$, such that $\langle v_i, v_{i+1}\rangle$ is a superbubble for all $i \in \{1, \ldots, k-1\}$.*

A useful consequence of Lemma 7, furthermore, is that superbubbles cannot overlap at interior vertices since their intersection is again a superbubbloid and thus neither of them could have been minimal (compare Definition 19 (Page 49)). Furthermore, Lemma 6 immediately implies that $\langle w, s\rangle$ and $\langle u, t\rangle$ are also superbubbloids, i.e., neither $\langle w, u\rangle$ nor $\langle s, t\rangle$ is a superbubble in the situation of Lemma 7.

3.4 Superbubbles and SCC

Theorem 1. *Every weak superbubbloid $\langle s, t\rangle$ in G is an induced subgraph of $G[S]$ for some $S \in \mathfrak{P}_G$.*

Proof. First assume that $\langle s, t\rangle$ contains an edge (u, v) that is contained in a cycle. Then by (S.v), there is a cycle through s and t and thus in particular a $t \to s$ path. For every $u \in U$, there is a path within U from s to t through u by (S.i), (S.ii), and Corollary 4 (Page 56). Thus $\langle s, t\rangle$ is contained as an induced subgraph in a strongly connected component (SCC) $G[S]$ of G. If there is no edge in $\langle s, t\rangle$ that is contained in a cycle, then every vertex in $\langle s, t\rangle$ is a SCC on its own. $\langle s, t\rangle$ is therefore an induced subgraph of the acyclic component $G[\overline{A}_G]$. □

Theorem 1 establishes Proposition 1 (Page 54), the key result of Sung et al. (2015), in sufficient generality. It guarantees that every weak superbubbloid and thus every superbubble in G is completely contained within one of the induced subgraphs $G[S]$, $S \in \mathfrak{P}_G$. It does not guarantee, however, that a superbubble in $G[S]$ is also a superbubble in G. This is already noted in Sung et al. (2015). This fact leads to augmenting the induced subgraph $G[S]$ of G by an artificial source a and an artificial sink b.

Definition 23. *The augmented graph $\tilde{G}_S$ is constructed from $G[S]$ by adding the artificial source a and the artificial sink b. There is an edge (a, x) in $\tilde{G}_S$ whenever $x \in S$ has an incoming edge from another component in G and there is an edge (x, b) whenever $x \in S$ has an outgoing edge to another component of G.*

augmented graph (SCC)

Since $G[\overline{A}_G]$ is acyclic, a has only outgoing edges and b only incoming ones, it follows that the augmented graph $\tilde{G}_{\overline{A}_G}$ is also acyclic.

Lemma 10. *$\langle s, t\rangle$ is a weak superbubbloid in G if and only if it is a weak superbubbloid of $\tilde{G}_S$ for $S \in \mathfrak{P}_G$ that does not contain an auxiliary source a or an auxiliary sink b.*

Proof. First assume that $\langle s, t\rangle$ is an induced subgraph of the SCC $G[S]$ of G. By construction, $G[S]$ is also a SCC of $\tilde{G}_S$. Thus reachability within S is the same w.r.t. G and $\tilde{G}_S$. Also by construction, a vertex $w \notin S$ is reachable from $x \in S$ in G if and only if b is reachable from x in $\tilde{G}_S$. Similarly, a vertex $x \in S$ is reachable from $w \notin S$ if and only if x is reachable from a. Hence $\langle s, t\rangle$ is a (weak) superbubbloid w.r.t. G if and only if it is a weak superbubbloid w.r.t. $\tilde{G}_S$. For the special case that $\langle s, t\rangle$ is an induced subgraph of the acyclic component $\tilde{G}_{\overline{A}_G}$ it can be argued in exactly the same manner.

For SCC S, the graph $\tilde{G}_S$ contains exactly three SCC whose vertex sets are S and the singletons ($\{a\}$ and $\{b\}$). Since (a, b) is not an edge in $\tilde{G}_S$, every weak supperbubbloid in $\tilde{G}_S$ is contained in $G[S]$ and hence contains neither a nor b. Superbubbloids containing a or b cannot be excluded for the acyclic component $\tilde{G}_{\overline{A}_G}$, however. □

It is possible, therefore, to find the weak superbubbloids of G by computing the weak superbubbloids not containing an artifical source or sink vertex in the augmented graphs.

3.5 Superbubbles maintaining DAG

This section utilizes *search trees*. Thus, a small recap of the properties of a search trees is given. For more details see Subsection 2.2.5. A search tree T of the digraph G is defined by its root r. The tree is then the combination of paths $r \to v$ ($p_T(v)$) for every $v \in [r]_{\rightsquigarrow}$. An edge (v, w) is a back edge if $w \in p_T(v)$.

Lemma 11. *Let G be a strongly connected digraph, $\langle s, t\rangle$ be a weak superbubbloid in G, $r \notin V(\langle s, t\rangle)$, and T a search tree rooted at r. Then the induced subgraph $\langle s, t\rangle$ of G contains no back edge w.r.t. T except possibly (t, s).*

Proof. By (S.iii) it is clear that every path $p_T(v)$ with $v \in V(\langle s, t\rangle)$ contains s. Thus, a subtree $T(s)$ contains every vertex of $\langle s, t\rangle$. Thus, a back edge (v, u) with $v \in V(\langle s, t\rangle)$ has two possibilities either $u \in p_T(s) \setminus \{s\}$ or $u \in s \to v$. In the first case $u \notin V(\langle s, t\rangle)$. If $v = t$ then the edge (v, u) is not part of $\langle s, t\rangle$. If $v \neq t$ the edge (v, u) contradicts (S.iv). Thus, $\langle s, t\rangle$ is not a weak superbubbloid.

The second case $u \in s \to v$ means that $u \in V(\langle s,t\rangle)$. If $v = t$ and $u = s$ the back edge represents the given exception of (t,s). If $v \neq t$ or $u \neq s$ then the back edge (v,u) creates a cycle with $u \to v \subseteq s \to v$. Note that this cycle does not contain t if $v \neq t$ or does not contain s if $u \neq s$. Therefore, this contradicts (S.v) and $\langle s,t\rangle$ can not be a weak superbubbloid. Thus, there are no back-edges in $\langle s,t\rangle \setminus \{(t,s)\}$. □

Note that the main point of the proof is that the complete superbubble is in a subtree of s. This is also true if s is the root of T.

Lemma 11 is the key prerequisite for constructing an acyclic graph that contains all weak superbubbles of $\tilde{G}_S$. Similar to the arguments above, the back edges cannot simply be ignored. Instead, again edges are added to the artificial source and sink vertices.

auxiliary graph (DAG) **Definition 24.** *Given a search tree T with a root r that is neither an interior vertex nor the exit of a weak superbubbloid of $\tilde{G}_S$, the auxiliary graph $\hat{G}_S$ is obtained from $\tilde{G}_S$ by replacing every back edge (v,u) with respect to T in $\tilde{G}_S$ with both an edge (a,u) and an edge (v,b).*

Note that Definition 24 implies that all back edges (v,u) of $\tilde{G}_S$ are removed in $\hat{G}_S$. As a consequence, $\hat{G}_S$ is acyclic. The construction of $\hat{G}_S$ is illustrated in Figure 26.

Lemma 12. *Let S be a SCC of G and let T be a search tree on $\tilde{G}_S$ with a root that is neither an interior vertex nor the exit of a weak superbubbloid of G. If $\hat{G}_S$ is constructed with T from $\tilde{G}_S$ then $\langle s,t\rangle$ with $s,t \in S$ is a weak superbubble of G contained in $\tilde{G}_S$ if and only if $\langle s,t\rangle$ is a superbubble in $\hat{G}_S$ that does not contain the auxiliary source a or the auxiliary sink b.*

Proof. Assume that $\langle s,t\rangle$ is a weak superbubble in $\tilde{G}_S$ that does not contain a or b. Lemma 10 ensures that this is equivalent to $\langle s,t\rangle$ being a weak superbubble of G. By Lemma 11, $\langle s,t\rangle$ contains no back edges in $\tilde{G}_S$, with the possible exception of the edge (t,s). Since $\tilde{G}_S$ and $\hat{G}_S$ by construction differ only in the back edges, the only difference affecting $\langle s,t\rangle$ is the possible insertion of edges from a to s or from t to b. Neither affects a weak superbubble, however, and hence $\langle s,t\rangle$ is a (weak) superbubble in $\hat{G}_S$.

Now assume that $\langle s,t\rangle$ is a superbubble in $\hat{G}_S$ with vertex set U and $a,b \notin U$. Since $\hat{G}_S$ is constructed from $\tilde{G}_S$, reachability w.r.t. to $\hat{G}_S$ implies reachability w.r.t. $\tilde{G}_S$. Therefore U satisfies (S.i) and (S.ii) also w.r.t. $\tilde{G}_S$. Therefore, if $\langle s,t\rangle$ is not a weak superbubble in $\tilde{G}_S$ then there must be a back edge (x,v) or a back edge (x,v) with v in the interior of $\langle s,t\rangle$. The construction of $\hat{G}_S$, however, ensures that $\hat{G}_S$ then contains an edge (a,v) or (v,b), respectively, which would contradict (S.iii), (S.iv), or acyclicity (in case $x \in U$) and hence (S.v). Therefore, $\langle s,t\rangle$ is a weak superbubble in $\tilde{G}_S$. □

The remaining difficulty is to find a vertex w that can safely be used as root for the search tree T. In most cases, one can simply set a as root since Lemma 10

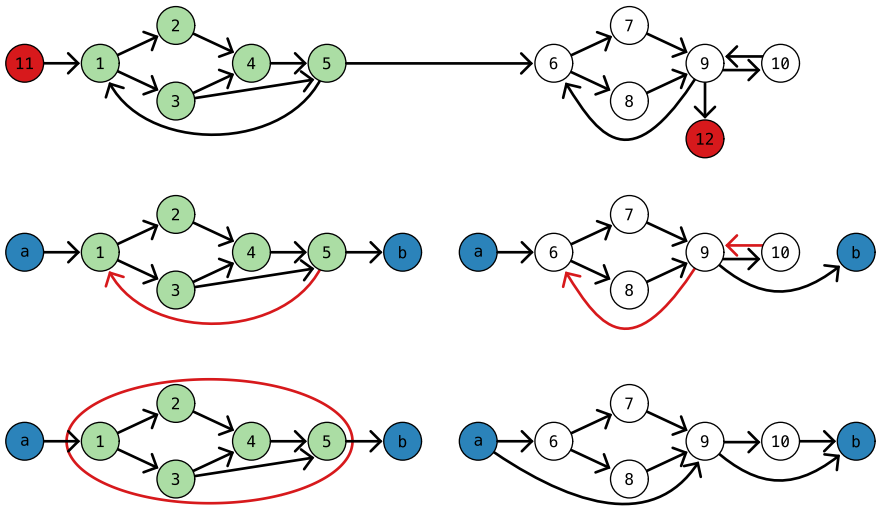


Figure 26: Example for the construction of $\hat{G}_S$. On the top the starting graph G is shown. The graph G has two non-singleton SCCs (indicated by the green and white vertices, respectively). Also, there are two singleton SCCs (red vertices) from which $\tilde{G}_{\overline{A}_G}$ is constructed. However, as these two vertices are not connected, the trivial $\tilde{G}_{\overline{A}_G}$ is not shown. The middle panel shows the graphs $\tilde{G}_S$. Each is obtained by adding the artificial source and sink vertices a and b (blue vertices). The edges $(11, 1)$, $(5, 6)$ and $(9, 12)$ in G form connections between the SCCs and the acyclic component, respectively. Hence they are replaced by corresponding edges from an artificial source a or to an artificial sink b according to Definition 23. The bottom panel shows the graphs $\hat{G}_S$ obtained with the help of search trees rooted at a. The back edges (red edges) $(5, 1)$, $(9, 6)$, and $(10, 9)$ are replaced with the corresponding edge from a and to b as described by Definition 24. The $\hat{G}_S$ graphs have the same weak superbubbles as G. Thus, the weak superbubble $\langle 1, 5\rangle$ is reported. Note that in $\hat{G}_S$ no discrimination between week superbubbles and superbubbles are possible thus the $\langle 1, 5\rangle$ must be filtered after the detection.

ensures that a is not part of a weak superbubbloid of G. However, there is no guarantee that an edge of the form (a, w) exists, in which case $\tilde{G}_S$ is not connected. Thus, another root for the search tree must be chosen. A closer inspection shows that three cases have to be distinguished:

(A) a has an out-edge. In this case a is chosen as the root of the search tree.

(B) a has no edge, but b has an in-edge. In this case it is possible to identify vertices that can only be entrances of a superbubble. These can then be connected with the artificial source vertex without destroying a superbubble.

(C) Neither a nor b have edges. The case requires special treatment, as explained below.

In order to handle case (B), the following is used:

Lemma 13. *Let a and b be the artificial source and sink of $\tilde{G}_S$. Let a' and b' be a successor of a and a predecessor of b, respectively. Then*

i) a' is neither an interior vertex nor the exit of a superbubble.

ii) A predecessor a'' of a' is neither an interior vertex nor an entrance of a superbubble.

iii) b' is neither an interior vertex nor the entrance of a superbubble.

iv) A successor b'' of b' is neither an interior vertex nor an exit of a superbubble.

Proof. If a' is contained in a superbubble, it must be the entrance, since otherwise its predecessor, the artificial vertex a would belong to the same superbubble. If a'' is in the interior of an entrance, the a' would be an interior vertex of a superbubble, which is impossible by (i). The statements for b follow analogously. □

Corollary 9. *If b has an in-edge in $\tilde{G}_S$, then every successor $b'' \neq b$ of every predecessor b' of b can be used as a root of the search tree. At least one such vertex exists.*

Proof. By assumption, b has at least one predecessor b'. Since $G[S]$ is strongly connected, b' has at least one successor $b'' \neq b$, which by Lemma 13(iv) is either not contained in a superbubble or is the entrance of a superbubble. □

The approach sketched above fails in case (C) because there is no efficient way to find a root for the search tree that is guaranteed not to be an interior vertex or the exit of a (weak) superbubbloid presented yet. This is done in Section 3.8. However, for the moment the construction of Sung et al. (2015) is used.

The Sung graph (Definition 20 (Page 54)) H contains two types of weak superbubbloids: those that contain no back edges w.r.t. T, and those that contain back edges. Members of the first class do not contain the root of T by Lemma 11 and hence are also superbubbles in G. Every weak superbubble of this type is present (and is detected) in both V' and V''. A weak superbubble with back edge has a "front part" in V' and a "back part" in V'' and appears exactly once in H. The vertex sets V' and V'' are disjoint. It is possible that H contains superbubbles that have duplicated vertices, i.e., vertices v' and v'' deriving from the same vertex in V. These candidates are removed together with one of the copies of superbubbles appearing in both V' and V''. This filtering step is referred to as *Sung filtering* as it is proposed in Sung et al. (2015).

This construction is correct in case (C) if there are no other edges connecting $G[S]$ within G. The additional connections to a and b introduced to account for edges that connect $G[S]$ to other vertices in G, may fail. To see this, consider an interior vertex v' in a superbubble $\langle s, t \rangle$ with a back edge. It is possible that its original has an external out edge and thus b should be connected to v'. This is not accounted for in the construction of H, which required that V' is connected to a

Algorithm 1: Top level organization of the computation of superbubbles in a digraph G. It reduces the problem to the problem of identifying all superbubbles in a collection of directed acyclic graphs (DAGs). For the identification of superbubbles in a DAG the algorithm *DAGsuperbubble* is used. This is presented in Section 3.6. However, the same result can be created by using the algorithm from Brankovic et al. (2016).

Require: digraph G
  compute all SCC S and the acyclic component $\overline{A}_G$ of G.
  for all S **do**
    if $G[S]$ is a connected component of G **then**
      choose arbitrary root x in $G[S]$
      construct search tree T with root x
      construct Sung graph $H(S)$
      construct DFS-topological sorting $\overline{\pi}$ for $H(S)$
      DAGsuperbubble$(H(S), \overline{\pi})$
      filter superbubbles with Sung filter
    else
      construct auxiliary graph $\tilde{G}_S$
      choose a or b'' as root x
      construct search tree T with root x
      construct $\hat{G}_S$
      construct DFS-topological sorting $\overline{\pi}$ for $\hat{G}_S$
      DAGsuperbubble$(\hat{G}_S, \overline{\pi})$
  construct auxiliary graph $\tilde{G}_{\overline{A}_G}$
  construct DFS-topological sorting $\overline{\pi}$ for $\tilde{G}_{\overline{A}_G}$
  DAGsuperbubble$(\tilde{G}_{\overline{A}_G}, \overline{\pi})$

only, and V'' is connected to b only. These "missing" edges may introduce false positive superbubbles as shown in Figure 24 (Page 55).

This is not a dramatic problem because it is easy to identify the false positives: it suffices to check whether there is an edge (x, w) or (w, y) with $w \notin [s, t]_{\rightsquigarrow}$, $x \in [s, t]_{\rightsquigarrow} \setminus \{t\}$ and $y \in [s, t]_{\rightsquigarrow} \setminus \{s\}$. Clearly, this can be achieved in linear total time for all superbubble candidates $[s, t]_{\rightsquigarrow}$, providing an easy completion for the algorithm of Sung et al. (2015).The alternative construction eliminates the need for this additional filtering step.

Lemma 14. *The (weak) superbubbles in a digraph G can be identified in $\mathcal{O}(|V(G)| + |E(G)|)$ time using Algorithm 1 provided the (weak) superbubbles in a DAG can be found in linear time.*

Proof. The correctness of Algorithm 1 is an immediate consequence of the discussion above. Let us briefly consider its running time. The SCCs of G can be computed in linear, i.e., $\mathcal{O}(|V(G)| + |E(G)|)$ time (Nuutila and Soisalon-Soininen, 1994; Pearce, 2016; R. Tarjan, 1972). The acyclic component $\overline{A}_G$ as well as its connected components (Hopcroft and R. Tarjan, 1973) are obtained in linear time. The

construction of directed (to construct search tree T) or undirected depth-first search (DFS) (to construct a DFS-topological sorting) also require only linear time (R. Tarjan, 1972; R. E. Tarjan, 1976), as does the classification of forward and back edges. The construction of the auxiliary DAGs $\hat{G}_S$ and $H(S)$ and the determination of the root for the DFS is then also linear in time. Since the vertex sets considered in the auxiliary DAGs are disjoint in G, it is concluded that the superbubbles can be identified in linear time in arbitrary digraphs if the problem can be solved in linear time in a DAG. □

The algorithm of Brankovic et al. (2016) shows that this is indeed the case.

Corollary 10. *The (weak) superbubbles in a digraph G can be identified in $\mathcal{O}(|V(G)| + |E(G)|)$ time using Algorithm 1.*

In the following section, a different linear time algorithm for superbubble finding is presented that may be more straightforward than the approach in Brankovic et al. (2016), which heavily relies on range queries.

3.6 Superbubbles in a DAG

The identification of (weak) superbubbles is drastically simplified in DAGs since acyclicity, i.e., (S3), and thus (S.v), can be taken for granted. In particular, therefore, every weak superbubbloid is a superbubbloid. A key result of Brankovic et al. (2016) is the fact that there are vertex orders for DAGs in which all superbubbles appear as intervals. The proof of Proposition 2 does not make use of the minimality condition hence it can be stated the result here more generally for superbubbloids and arbitrary DFS-topological sorting on G:

Proposition 2 (Brankovic et al. (2016))**.** *Let G be a DAG and let $\overline{\pi}$ be a DFS-topological sorting of G. Suppose $\langle s, t\rangle$ is a superbubbloid in G. Then*

i) Every interior vertex u of $\langle s, t\rangle$ satisfies $\overline{\pi}(s) < \overline{\pi}(u) < \overline{\pi}(t)$.

ii) If $w \notin \langle s, t\rangle$ then either $\overline{\pi}(w) < \overline{\pi}(s)$ or $\overline{\pi}(t) < \overline{\pi}(w)$.

OutParent (DAG) and OutChild (DAG)

The following two functions are introduced also in Brankovic et al. (2016):

$$\mathbf{OutParent}(v) := \begin{cases} -1 & \text{if no } (u,v) \in E(G) \text{ exists,} \\ \min(\{\overline{\pi}(u) \mid (u,v) \in E(G)\}) & \text{otherwise.} \end{cases}$$

$$\mathbf{OutChild}(v) := \begin{cases} \infty & \text{if no } (v,u) \in E(G) \text{ exists,} \\ \max(\{\overline{\pi}(u) \mid (v,u) \in E(G)\}) & \text{otherwise.} \end{cases} \tag{3.1}$$

The definition is slightly modified here to assign values also to the sink and source vertices of the DAG G. The functions return the predecessor and successor of

v that is furthest away from v in terms of the DFS-topological sorting $\overline{\pi}$. It is convenient to extend this definition to intervals by setting

$$\begin{aligned} \mathbf{OutParent}([i\!:\!j]) &:= \min(\{\mathbf{OutParent}(v) \mid v \in \overline{\pi}[i\!:\!j]\}) \\ \mathbf{OutChild}([i\!:\!j]) &:= \max(\{\mathbf{OutChild}(v) \mid v \in \overline{\pi}[i\!:\!j]\}) \end{aligned} \tag{3.2}$$

A main result of this section is that superbubbles are characterized completely by these two functions, resulting in an alternative linear-time algorithm for recognizing superbubbles in DAGs that also admits a simple proof of correctness. To this end few simple properties of the $\mathbf{OutParent}(.)$ and $\mathbf{OutChild}(.)$ functions for intervals are needed. The first observation is that $[k\!:\!l] \subseteq [i\!:\!j]$ implies the inequalities

$$\begin{aligned} \mathbf{OutParent}([k\!:\!l]) &\geq \mathbf{OutParent}([i\!:\!j]) \\ \mathbf{OutChild}([k\!:\!l]) &\leq \mathbf{OutChild}([i\!:\!j]) \end{aligned} \tag{3.3}$$

A key observation is the following

Lemma 15. *If* $\mathbf{OutChild}([i\!:\!j-1]) \leq j < \infty$ *then*

i) $\overline{\pi}^{-1}(j)$ *is the only successor of* $\overline{\pi}^{-1}(j-1)$*;*

ii) $\overline{\pi}^{-1}(j)$ *is reachable from every vertex* $v \in \overline{\pi}[i\!:\!j-1]$*;*

iii) every path from some $v \in \overline{\pi}[i\!:\!j-1]$ *to a vertex* $w \notin \overline{\pi}[i\!:\!j-1]$ *contains* $\overline{\pi}^{-1}(j)$*.*

Proof. (i) By definition $\overline{\pi}^{-1}(j-1)$ has at least one successor (otherwise $\mathbf{OutChild}([i\!:\!j-1]) = \infty$). On the other hand, all successors of vertices in $\overline{\pi}[i\!:\!j-1]$ are by definition not later than j. Hence $\overline{\pi}^{-1}(j)$ is the only possibility.

(ii) It is proven by induction w.r.t. the length of the interval $[i\!:\!j-1]$. If $i = j-1$, i.e., a single vertex, the assertion (ii) is obviously true. Now assume that the assertion is true for $[i+1\!:\!j-1]$. By definition of $\mathbf{OutChild}(.)$, i has a successor in $\overline{\pi}[i+1\!:\!j]$, from which $\overline{\pi}^{-1}(j)$ is reachable.

(iii) Again, it is proven by induction. The assertion holds trivially for single vertices. Assume that the assertion is true for $[i+1\!:\!j-1]$. By definition of $\mathbf{OutChild}(.)$, every successor u of $\overline{\pi}^{-1}(i)$ is contained in $\overline{\pi}[i+1\!:\!j]$. By induction hypothesis, every path from u to a vertex $w \notin \overline{\pi}[i+1\!:\!j]$ contains $\overline{\pi}^{-1}(j)$, and also all paths from $\overline{\pi}^{-1}(i)$ to $w \notin \overline{\pi}[i\!:\!j]$ run through $\overline{\pi}^{-1}(j)$. □

It is important to notice that Lemma 15 depends crucially on the fact that $\overline{\pi}$, by construction, is a DFS-topological sorting. It does not generalize to arbitrary topological sortings.

Replacing successor by predecessor in the proof of Lemma 15 it is obtained:

Lemma 16. *If* $\mathbf{OutParent}([i+1\!:\!j]) \geq i > -1$ *then*

i) $\overline{\pi}^{-1}(i)$ *is the only predecessor of* $\overline{\pi}^{-1}(i+1)$*;*

ii) every vertex $v \in \overline{\pi}[i+1\!:\!j]$ *is reachable from* $\overline{\pi}^{-1}(i)$*;*

iii) every path from $w \notin \overline{\pi}[i+1:j]$ to a vertex $v \in \overline{\pi}[i+1:j]$ contains $\overline{\pi}^{-1}(i)$.

The functions $\mathbf{OutParent}(.)$ and $\mathbf{OutChild}(.)$ have also interesting properties if they are considered for an individual vertex of a superbubbloid.

Lemma 17. *Let $\langle s,t\rangle$ be a superbubbloid in a DAG G, v is an interior vertex of $\langle s,t\rangle$, and w a vertex not in $\langle s,t\rangle$. Then*

i) $\overline{\pi}(s) \leq \mathbf{OutParent}(v)$ and $\mathbf{OutChild}(v) \leq \overline{\pi}(t)$.

ii) $\overline{\pi}(s) \leq \mathbf{OutParent}(t)$ and $\mathbf{OutChild}(s) \leq \overline{\pi}(t)$.

iii) $\mathbf{OutParent}(w) < \overline{\pi}(s)$ or $\mathbf{OutParent}(w) \geq \overline{\pi}(t)$, and $\mathbf{OutChild}(w)) \leq \overline{\pi}(s)$ or $\mathbf{OutChild}(w) > \overline{\pi}(t)$.

Proof. (i) The matching property (S2) implies that for every successor x and predecessor y of an interior vertex v there is a path within the superbubble from s to x and from y to t, respectively. The statement now follows directly from the definition.
(ii) The argument of (i) applies to the successors of s and the predecessors of t.
(iii) Assume, for contradiction, that $\overline{\pi}(s) \leq \mathbf{OutParent}(w) < \overline{\pi}(t)$ or $\overline{\pi}(s) < \mathbf{OutChild}(w) \leq \overline{\pi}(t)$. Then Proposition 2 implies that w has a predecessor v' or successor v'' in the interior of the superbubble. But then v' has a successor (namely w) outside the superbubble, or v'' has a predecessor (namely w) inside the superbubble. This contradicts the matching condition (S2). □

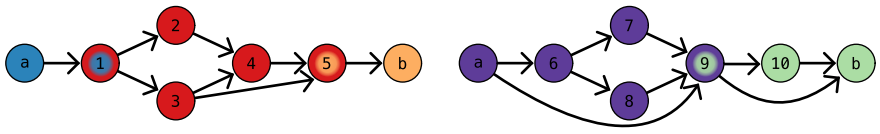
superbubbloid (DAG) **Theorem 2.** *Let G be a DAG and let $\overline{\pi}$ be a DFS-topological sorting on G. Then $\langle s,t\rangle$ is a superbubbloid if and only if the following conditions are satisfied:*

(D1) $\mathbf{OutParent}([\overline{\pi}(s)+1:\overline{\pi}(t)]) = \overline{\pi}(s)$ *(predecessor property)*

(D2) $\mathbf{OutChild}([\overline{\pi}(s):\overline{\pi}(t)-1]) = \overline{\pi}(t)$ *(successor property)*

Proof. Suppose $\mathbf{OutParent}([\overline{\pi}(s)+1:\overline{\pi}(t)])$ and $\mathbf{OutChild}([\overline{\pi}(s):\overline{\pi}(t)-1])$ satisfy (D1) and (D2). By (D1) and Lemma 15(ii) t is reachable from every vertex in v with $\overline{\pi}(s) \leq \overline{\pi}(v) < \overline{\pi}(t)$. Thus the reachability condition (S1) is satisfied. Lemma 15(iii) implies that any vertex w with $\overline{\pi}(w) < \overline{\pi}(s)$ or $\overline{\pi}(w) > \overline{\pi}(t)$ is reachable from v only through a path that runs through t. The topological sorting then implies that w with $\overline{\pi}(w) < \overline{\pi}(s)$ is not reachable from s at all since w is not reachable from t. Hence $[s,t]_{\rightsquigarrow} = \overline{\pi}[\overline{\pi}(s):\overline{\pi}(t)]$. By (D2) and Lemma 16(ii) every vertex v with $\overline{\pi}(s) < \overline{\pi}(v) \leq \overline{\pi}(t)$, i.e., is reachable from s. Lemma 16(ii) implies that v is reachable from a vertex w with $\overline{\pi}(w) < \overline{\pi}(s)$ or $\overline{\pi}(w) > \overline{\pi}(t)$ only through paths that contain s. The latter are not reachable at all since s is not reachable from w with $\overline{\pi}(w) > \overline{\pi}(t)$ in a DAG. Thus $[t,s]_{\overleftsquigarrow} = \overline{\pi}[\overline{\pi}(s):\overline{\pi}(t)] = [s,t]_{\rightsquigarrow}$, i.e., the matching condition (S2) is satisfied.

Now suppose (S1) and (S2) hold. Lemma 17 implies that $\mathbf{OutParent}([\overline{\pi}(s)+1:\overline{\pi}(t)]) \geq \overline{\pi}(s)$. Since some vertex $v' \in V(\langle s,t\rangle)$ must have s as its predecessor is $\mathbf{OutParent}([\overline{\pi}(s)+1:\overline{\pi}(t)]) = \overline{\pi}(s)$, i.e., (D1) holds. Analogously, Lemma 17 implies $\mathbf{OutChild}([\overline{\pi}(s):\overline{\pi}(t)-1]) \leq \overline{\pi}(t)$. Since there must be some $v' \in V(\langle s,t\rangle)$ that has t as its successor, is $\mathbf{OutChild}([\overline{\pi}(s):\overline{\pi}(t)-1]) = \overline{\pi}(t)$, i.e. (D2) holds. □



1	2	3	4	5	6	7
a	1	2	3	4	5	b
−1	1	2	2	3	4	6
2	4	5	6	6	7	∞

Position
Vertex
OutParent
OutChild

1	2	3	4	5	6	7
a	6	7	8	9	10	b
−1	1	2	2	1	5	6
5	4	5	5	7	7	∞

Figure 27: Superbubble representation with a DFS-topological sorting. On the top, two DAGs are shown (Compare Figure 26 (Page 63)). Below the DAGs is the DFS-topological sorting. First line of the table is the position ($\overline{\pi}(v)$) of the column in the DFS-topological sorting. Second line is the vertex (v) on this position. Third line is the $\mathbf{OutParent}(v)$ value. Last line is the $\mathbf{OutChild}(v)$ value. Intervals that together fulfill (D1) and (D2), thus create a superbubble are marked in the same color (not black). The same colors are used to mark the superbubbles in the graph. Note that because (D1) uses $\overline{\pi}(s)+1$ and (D2) uses $\overline{\pi}(t)-1$ the intervals are shifted by one. Note that the superbubble $\langle 1,5\rangle$ is the only weak superbubble that contains no artificial vertex (a and b).

An example that shows such intervals along the corresponding DAG can be found in Figure 27.

In the following it is proven that the possible superbubbloids and superbubbles can be found efficiently, i.e., in linear time using only a DFS-topological sorting and the corresponding functions $\mathbf{OutChild}(.)$ and $\mathbf{OutParent}(.)$. As an immediate consequence of (D2) and Lemma 15, the following necessary condition for exits exists:

Corollary 11. *The exit t of superbubbloid $\langle s,t\rangle$ satisfies* $\mathbf{OutChild}(\overline{\pi}^{-1}(\overline{\pi}(t)-1))=\overline{\pi}(t)$.

The minimality condition of Definition 19 (Page 49) is used to identify the superbubbles among the superbubbloids.

Lemma 18. *If t is the exit of a superbubbloid, then there is also the exit of a superbubble $\langle s,t\rangle$ whose entrance s is the vertex with the largest value of $\overline{\pi}(s)<\overline{\pi}(t)$ such that* (D1) *and* (D2) *are satisfied.*

Proof. Let $\langle s,t\rangle$ be a superbubbloid. According to Definition 19 (Page 49), $\langle s,t\rangle$ is a superbubble if there is no superbubbloid $\langle s',t\rangle$ with $\overline{\pi}(s)<\overline{\pi}(s')<\overline{\pi}(t)$, i.e., there is no vertex s' with $\overline{\pi}(s')>\overline{\pi}(s)$ such that (D1) and (D2) are satisfied. □

Lemma 19. *Suppose $\overline{\pi}(s)\leq\overline{\pi}(v)<\overline{\pi}(t)$ and $\mathbf{OutChild}(v)>\overline{\pi}(t)$. Then there is no superbubbloid with entrance s and exit t.*

Proof. Suppose $\langle s,t\rangle$ is a superbubbloid. By construction, $\mathbf{OutChild}([\overline{\pi}(s){:}\overline{\pi}(t)-1]) \geq \mathbf{OutChild}(v) > \overline{\pi}(t)$, contradicting (D2). □

Corollary 12. *If $\langle s,t\rangle$ is a superbubble, then there is no superbubbloid $\langle s',t'\rangle$ with exit $t' \in \overline{\pi}[\overline{\pi}(s)+1{:}\overline{\pi}(t)-1]$ and entrance s' with $\overline{\pi}(s') < \overline{\pi}(s)$.*

Proof. This is an immediate consequence of Lemma 7 (Page 59), which shows that the intersection $\langle s,t\rangle \cap \langle s',t'\rangle$ would be a superbubbloid, contradicting minimality of $\langle s,t\rangle$. □

Corollary 13. *If $\langle s,t\rangle$ and $\langle s',t'\rangle$ are two superbubbles with $\overline{\pi}(t') < \overline{\pi}(t)$ then either $\overline{\pi}(s') < \overline{\pi}(t') \leq \overline{\pi}(s) < \overline{\pi}(t)$, or $\overline{\pi}(s) < \overline{\pi}(s') < \overline{\pi}(t') < \overline{\pi}(t)$.*

Thus superbubbles are either nested or placed next to each other, as already noted in Onodera, Sadakane, and Shibuya (2013). Finally, it is considered how to identify false exit candidates, i.e., vertices that satisfy the condition of Corollary 11 but have no matching entrance s.

Lemma 20. *Let $\langle s,t\rangle$ be a superbubble and suppose t' is an interior vertex of $\langle s,t\rangle$. Then there is a vertex v with $\overline{\pi}(s) \leq \overline{\pi}(v) < \overline{\pi}(t')$ such that $\mathbf{OutChild}(v) > \overline{\pi}(t')$.*

Proof. Suppose, for contradiction, that no such vertex v exists. Since $\langle s,t\rangle$ is superbubble by assumption, it follows that $\mathbf{OutParent}([\overline{\pi}(s)+1{:}\overline{\pi}(t')]) = \overline{\pi}(s)$ is correct and so (D1) satisfied for $\langle s,t'\rangle$. After no such v exists also $\mathbf{OutChild}([\overline{\pi}(s){:}\overline{\pi}(t')-1]) \leq \overline{\pi}(t')$ is correct and so (D2) is satisfied. Thus $\langle s,t'\rangle$ is superbubbloid. By Lemma 6 (Page 58) $\langle t',t\rangle$ is also a superbubbloid, contradicting the assumption. □

Taken together, these observations suggest to organize the search by scanning the vertex set for candidate exit vertices t in reverse order. For every such t, one would then search for the corresponding entrance s such that the pair s,t fulfills (D1) and (D2) from Theorem 2. Using Equation 3.3 one can test (D2) independently for each v by checking whether $\mathbf{OutChild}(v) \leq \overline{\pi}(t)$. Checking for (D1) requires that the interval $[\overline{\pi}(s)+1{:}\overline{\pi}(t)]$ is considered. The value of its $\mathbf{OutParent}(.)$ function can be obtained incrementally as the minimum of $\mathbf{OutParent}(v)$ and the $\mathbf{OutParent}(.)$ interval of the previous step:

$$\begin{aligned}\mathbf{OutParent}([\overline{\pi}(v){:}\overline{\pi}(t)]) = \\ \min(\mathbf{OutParent}(v), \mathbf{OutParent}([\overline{\pi}(v)+1{:}\overline{\pi}(t)]))\end{aligned} \tag{3.4}$$

By Lemma 18, the nearest entrance s to the exit t completes the superbubble. The tricky part is to identify all superbubbles in a single scan. Lemma 19 ensures that no valid entrance can be found for exit t' if a vertex v with $\mathbf{OutChild}(v) > \overline{\pi}(t')$ is encountered. In this case t' can be discarded. Lemma 20 ensures that a false exit candidate t' within a superbubble $\langle s,t\rangle$ candidate cannot "mask" the entrance s belonging to t, i.e., there is necessarily a vertex v satisfying $\mathbf{OutChild}(v) > \overline{\pi}(t')$ with $\overline{\pi}(s) < \overline{\pi}(v)$.

Algorithm 2: Detecting superbubbles in a DAG The *DAGsuperbubble* algorithm. It utilizes the helper functions $\mathbf{OutChild}(.)$ and $\mathbf{OutParent}(v)$ as defined in Equation 3.1 (Page 66). The values of different $\mathbf{OutParent}(.)$ must be saved dependent on the exit candidate. Thus the value is saved in the map outmap.

DAGsuperbubble (Algorithm)

Require: DAG G with DFS-topological sorting $\overline{\pi}$
  empty stack $\mathbb{S}$
  empty map outmap
  for $k = n...1$ **do**
    $v = \overline{\pi}^{-1}(k)$
    $child \leftarrow \mathbf{OutChild}(v)$
    if $child = k + 1$ **then**
      push $\overline{\pi}^{-1}(k+1)$ onto $\mathbb{S}$
    else
      while $child > \overline{\pi}(\mathrm{TOP}(\mathbb{S}))$ **do**
        $t \leftarrow \mathrm{POP}(\mathbb{S})$
        $\mathrm{outmap}[\mathrm{TOP}(\mathbb{S})] \leftarrow \min(\mathrm{outmap}[t], \mathrm{outmap}[\mathrm{TOP}(\mathbb{S})])$
    if $\mathrm{outmap}[\mathrm{TOP}(\mathbb{S})] = k$ **then**
      report $\langle v, \mathrm{TOP}(\mathbb{S})\rangle$
      $\mathrm{POP}(\mathbb{S})$
    $\mathrm{outmap}[v] \leftarrow \mathbf{OutParent}(v)$
    $\mathrm{outmap}[\mathrm{TOP}(\mathbb{S})] \leftarrow \min(\mathrm{outmap}[\mathrm{TOP}(\mathbb{S})], \mathrm{outmap}[v])$

It is natural therefore to use a stack $\mathbb{S}$ to hold the exit candidates. Since the $\mathbf{OutParent}(.)$ interval explicitly refers to an exit candidate t, it must be re-initialized whenever a superbubble is completed or the candidate exit is rejected. More precisely, the $\mathbf{OutParent}(.)$ interval of the previous exit candidate t must be updated. This is achieved by computing

$$\mathbf{OutParent}([\overline{\pi}(v):\overline{\pi}(t)]) = \min(\mathbf{OutParent}([\overline{\pi}(v):\overline{\pi}(t')]), \mathbf{OutParent}([\overline{\pi}(t')+1:\overline{\pi}(t)])) \tag{3.5}$$

To this end, the value $\mathbf{OutParent}([\overline{\pi}(t')+1:\overline{\pi}(t)])$ is associated with t when t' is pushed onto the stack. The values of $\mathbf{OutParent}(.)$ intervals are not required for arbitrary intervals. Instead, only intervals are needed of the form $\mathbf{OutParent}([\overline{\pi}(t')+1:\overline{\pi}(t)])$ with consecutive exit candidates t' and t. Hence a single integer associated with each candidate t suffices. This integer is initialized with $\mathbf{OutParent}(t)$ and is then advanced as described above to $\mathbf{OutParent}([\overline{\pi}(v):\overline{\pi}(t)])$.

Algorithm 2 presents this idea in a more formal way. An example of the process of Algorithm 2 is shown in Figure 28.

Lemma 21. *Algorithm 2 identifies the superbubbles in a DAG G.*

Proof. Every reported candidate satisfies (D1) since $\mathbf{OutParent}([\overline{\pi}(s)+1:\overline{\pi}(t)]) = \overline{\pi}(s)$ is used to identify the entrance for the current t. Since $v \in \overline{\pi}[\overline{\pi}(s):\overline{\pi}(t)-1]$ is checked for every $\mathbf{OutChild}(v) \leq \overline{\pi}(t)$, (D2) holds due to Equation 3.3 (Page 67)

since by Lemma 15 (Page 67) this is equal to test the interval. Hence every reported candidate is a superbubbloid. By Lemma 18 $\langle s, t\rangle$ is minimal and thus a superbubble. Lemma 20 ensures that the corresponding entrance is identified for every valid exit t, i.e., that all false candidate exits are rejected before the next valid entrance is encountered. □

Lemma 22. *Algorithm 2 has time complexity* $\mathcal{O}(|V(G)| + |E(G)|)$.

Proof. Given the DFS-topological sorting $\overline{\pi}$, the `for` loop processes every vertex exactly once. All computations except $\mathbf{OutChild}(v)$, $\mathbf{OutParent}(v)$, and the `while` loop take constant time. This explicitly includes the calculation of the minimum of two integer values that are needed to update the intervals. The values of $\mathbf{OutChild}(v)$ and $\mathbf{OutParent}(v)$ are obtained by iterating over the outgoing or incoming edges of v, respectively, hence the total effort is $\mathcal{O}(|V(G)| + |E(G)|)$. Every iteration of the `while` loop removes one vertex from the stack $\mathbb{S}$. Since each vertex is pushed on $\mathbb{S}$ at most once, the total effort for the `while` loop is $\mathcal{O}(|V(G)|)$. The total running time therefore is $\mathcal{O}(|V(G)| + |E(G)|)$. □

Recalling the DFS-topological sorting $\overline{\pi}$ can also be obtained in $\mathcal{O}(|V(G)| + |E(G)|)$ it directly follows:

Corollary 14 (Brankovic et al. (2016))**.** *The superbubbles in a DAG can be identified in linear time.*

3.7 Superbubbles and DFS

The combination of Algorithm 1 (Page 65) and Algorithm 2 creates a new linear time approach for detecting superbubbles. This approach is published in Gärtner, Müller, and Stadler (2018). However, the creation of $\hat{G}_S$ depends on a search tree and the detection of the superbubbles in a DAG depends on a depth-first search (DFS)-topological sorting. Such sorting is a reversed post order of a DFS-forest, which is a search forest. Thus, the question arises if the creation of $\hat{G}_S$ is necessary or if one single DFS-forest that combines both ideas is enough. In this section, it is shown that one DFS-forest is enough.

The following observation, which slightly generalizes the previous analysis in Section 3.5 and Section 3.6, forms the basis of the direct superbubble detection in arbitrary digraphs. The key ingredient are DFS-trees.

Lemma 23. *Let G be a digraph and $[s,t]_{\rightsquigarrow}$ the vertex set of a weak superbubbloid $\langle s, t\rangle$ in G, and suppose r is not an interior vertex or the exit of $\langle s, t\rangle$. Then, either $[r]_{\rightsquigarrow} \cap [s,t]_{\rightsquigarrow} = \emptyset$ or $[s,t]_{\rightsquigarrow} \subseteq [r]_{\rightsquigarrow}$.*

Proof. (i) Every digraph can be decomposed into strongly connected components (SCCs) and acyclic components. If $x \in [r]_{\rightsquigarrow}$, then every vertex reachable from x is also contained in $[r]_{\rightsquigarrow}$. Thus, in particular, every SCC of G is either contained in $[r]_{\rightsquigarrow}$or disjoint from $[r]_{\rightsquigarrow}$. Theorem 1 (Page 60) ensures that every superbubbloid is either contained in a SCC S or the acyclic component $\overline{A}_G$ of G. Now, suppose

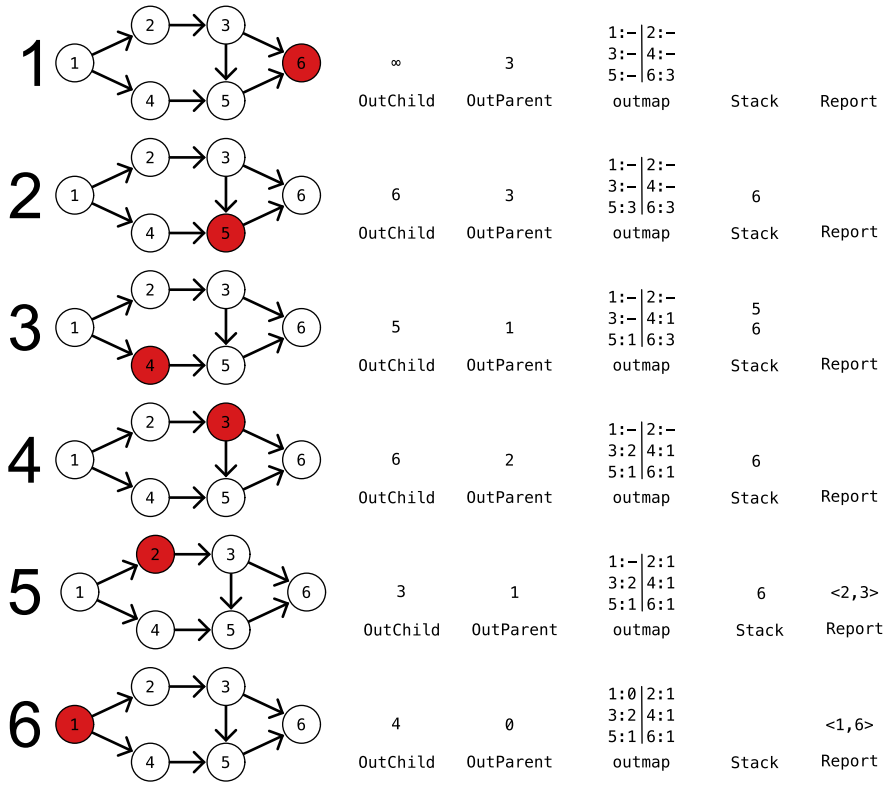


Figure 28: An example of Algorithm 2 applied to a DAG with six vertices. Every line shows one step of the **for** loop. On the left the DAG is shown where the current vertex (v) is marked red. The position of a vertex in the DFS-topological sorting corresponds to its label. In each step, values for $\mathbf{OutChild}(v)$ and $\mathbf{OutParent}(v)$ are calculated.
In the first step only the value of 6 is set. In the second step 6 is pushed on the stack, and the value of 5 is set. In the third step 5 is pushed on the stack, the value of 4 is set, and the value of 5 is updated. The fourth step discards 5 from the stack and updates thus the value of 6. Also, the value of 3 is set. In the fifth step 3 is pushed on the stack. Then the superbubble $\langle 2, 3\rangle$ is reported and thus 3 promptly removed again. Furthermore, the value of 2 is set. In the last step, the superbubble $\langle 1, 6\rangle$ is reported.

$[r]_{\rightsquigarrow} \cap [s,t]_{\rightsquigarrow} \neq \emptyset$, and let $x \in [s,t]_{\rightsquigarrow}$ be the first vertex of the DFS in $\langle s,t\rangle$. By definition (of weak superbubbloids) $x = s$, since no other vertex in $[s,t]_{\rightsquigarrow}$ is reachable from outside $[s,t]_{\rightsquigarrow}$, and the DFS assumption does not start at an interior vertex or the exit of $\langle s,t\rangle$. The reachability axiom (S.ii) ensures that every $u \in [s,t]_{\rightsquigarrow}$ is reached by the DFS whenever $s \in [r]_{\rightsquigarrow}$, i.e., $[s,t]_{\rightsquigarrow} \subseteq [r]_{\rightsquigarrow}$. □

Lemma 23 can be seen as a variant of Theorem 1 (Page 60).

Corollary 15. *Let G be a digraph and $[s,t]_{\rightsquigarrow}$ the vertex set of a weak superbubbloid $\langle s,t\rangle$ in G. Let $r_1,\ldots,r_k \in V(G)$ be such that none of the r_i are an interior or an exit vertex of $\langle s,t\rangle$. Set $W_j := [r_j]_{\rightsquigarrow} \setminus \bigcup_{i=1}^{j-1}[r_i]_{\rightsquigarrow}$. Then, either $[s,t]_{\rightsquigarrow} \cap W_j = \emptyset$ or $[s,t]_{\rightsquigarrow} \subseteq W_j$.*

Proof. By Lemma 23, $[s,t]_{\rightsquigarrow}$ is either contained in the intersection of two or more reachable sets $[r_j]_{\rightsquigarrow}$ or is disjoint from it. As an immediate consequence, it is also either contained in the difference of two reachable sets or disjoint from it. □

Corollary 16. *Let G be a digraph and $[s,t]_{\rightsquigarrow}$ the vertex set of a weak superbubbloid $\langle s,t\rangle$ in G. Let F be a search forest with the root set $\{r_1,\ldots,r_k\}$ that is created form the trees T_i. If none of the r_i are an interior or an exit vertex of $\langle s,t\rangle$ then $[s,t]_{\rightsquigarrow} \subseteq V(T_i)$ for one T_i.*

Proof. This follows directly from Corollary 15 and the definition of a search forest (see Subsection 2.2.5). □

Lemma 24. *Let G be a digraph; let $[s,t]_{\rightsquigarrow}$ be the vertex set of a weak superbubbloid $\langle s,t\rangle$ in G; let T be a DFS-tree on G with root $r \notin [s,t]_{\rightsquigarrow} \setminus \{s\}$; and let π be the postorder w.r.t. T. Then:*

(i) The induced subgraph $G[[s,t]_{\rightsquigarrow}]$ contains no back edges w.r.t. T, except possibly (t,s).

(ii) If $[s,t]_{\rightsquigarrow} \subseteq V(T)$, then $\{\pi(u) \mid u \in [s,t]_{\rightsquigarrow}\} = [\pi(t){:}\pi(s)]$ is an interval w.r.t. to π.

Proof. (i) The statement is trivial if $\langle s,t\rangle$ is not contained in T. If $\langle s,t\rangle$ resides in the acyclic component $\overline{A}_G$ of G, there are no back edges because $\overline{A}_G$ cannot contain back edges by acyclicity. If $\langle s,t\rangle$ is contained in a SCC S, the proof of Lemma 11 (Page 61) also implies assertion (i) because the DFS-tree T, in particular, contains a search tree of S as a subtree and back edges in G can only be located within a SCC.

(ii) Since the DFS generating T enters $\langle s,t\rangle$ through s and leaves it through t, the preorder ρ of T satisfies $\rho(s) < \rho(t)$. Since t is reachable from every $u \in [s,t]_{\rightsquigarrow}$, any DFS reaches t before completing any $u \in [s,t]_{\rightsquigarrow}$; hence, t precedes any other $u \in [s,t]_{\rightsquigarrow}$ in postorder, i.e., $\pi(u) > \pi(t)$. Since $u \in [s,t]_{\rightsquigarrow}$ is not reachable without passing through s, every other vertex in $u \in [s,t]_{\rightsquigarrow}$ precedes s in postorder, i.e., $\pi(s) > \pi(u)$. Now, suppose there is some $w \notin [s,t]_{\rightsquigarrow}$ with $\pi(s) > \pi(w) > \pi(t)$. Then, w must be reachable from s along a directed path that

does not pass through t, a contradiction to the definition of weak superbubbloids. Hence, the vertices of a superbubbloid form an interval in postorder of the DFS-tree T. □

Statement (ii) rephrases Proposition 2 (Page 66), although it is not assumed that G is a DAG. Conceptually, Lemma 24 suggests that it might not be necessary to first identify the SCCs of G or the construct acyclic auxiliary digraphs in order to find all weak superbubbles. Corollary 16 then ensures that a single DFS-forest is sufficient.

Next it is shown how to retrieve all weak superbubbles of a digraph G that are located within the induced subgraph $G[[r]_{\rightsquigarrow}]$ of G. To this end, a slightly modified version of the algorithm *DAGsuperbubble* (Algorithm 2 (Page 71)) is used. It is originally designed to operate on acyclic auxiliary graphs with a single source. Thus, it could be assumed that a DFS-tree rooted on this source reached all vertices. Here, it is intended to apply it to the unmodified input graph, which is neither acyclic, nor guaranteed to have a single source. It, therefore, needs to be modified to deal appropriately with back edges within the DFS-tree and the existence of vertices outside the DFS-tree. To this end, vertices in $[r]_{\rightsquigarrow}$ that cannot be contained in a superbubble have to be identified. By Lemma 24, there are two possible obstructions for a vertex u: (i) u has an edge that is a back edge w.r.t. the DFS-tree; (ii) u is incident to an edge (x, u) or (u, x) where $x \notin [r]_{\rightsquigarrow}$.

The basic idea of *DAGsuperbubble* is to identify minimal intervals in reverse postorder $\overline{\pi}$ of the DFS-tree T that satisfy conditions equivalent to membership in a superbubbloid. These conditions are expressed in terms of a pair of helper functions with the help of reverse postorder $\overline{\pi}$. As in Section 3.6, $\mathbf{OutParent}(v)$ denotes the first vertex (w.r.t. reverse postorder) in T from which v can be reached. Similarly, $\mathbf{OutChild}(v)$ is the last child vertex reachable from v.

OutParent (DFS) and OutChild (DFS)

$$\mathbf{OutParent}(v) := \begin{cases} -1 & \text{if no } (u,v) \in E(G) \text{ exists} \\ -1 & \text{if } (u,v) \in E(G) \wedge u \notin [r]_{\rightsquigarrow} \\ -1 & \text{if a back edge } (u,v) \text{ exists} \\ \min(\{\overline{\pi}(u) | (u,v) \in E(G)\}) & \text{otherwise} \end{cases}$$

$$\mathbf{OutChild}(v) := \begin{cases} \infty & \text{if no } (v,u) \in E(G) \text{ exists} \\ \infty & \text{if } (v,u) \in E(G) \wedge u \notin [r]_{\rightsquigarrow} \\ \infty & \text{if a back edge } (v,u) \text{ exists} \\ \max(\{\overline{\pi}(u) | (v,u) \in E(G)\}) & \text{otherwise} \end{cases} \tag{3.6}$$

Compared to the original functions on DAGs in Equation 3.1 (Page 66) more corner cases exist. The functions can be extended to intervals as shown in Equation 3.2 (Page 67).

Theorem 2 (Page 68) derives a characterization of weak superbubbloids in terms of $\mathbf{OutParent}([\overline{\pi}(s)+1 : \overline{\pi}(t)])$ and $\mathbf{OutChild}([\overline{\pi}(s) : \overline{\pi}(t)-1])$ for the case of acyclic digraphs. Here, this condition is generalized to general graphs using the

modified definition of $\mathbf{OutParent}(v)$ and $\mathbf{OutChild}(v)$. The difference is that back edges and edges connecting to the outside of the DFS-tree are considered. In either case, the corresponding vertices are marked by -1 or ∞, respectively, to indicate that they cannot be part of superbubbloids.

superbubbloid (DFS) **Theorem 3.** *Let G be a digraph; let T be a DFS-tree on G with root r that is not an interior vertex or exit of a weak superbubbloid; and denote by $\overline{\pi}$ the reverse postorder on T. Then, $\langle s,t\rangle$ is a weak superbubbloid in G whose vertex set $[s,t]_{\rightsquigarrow}$ satisfies $[s,t]_{\rightsquigarrow} \cap [r]_{\rightsquigarrow} \neq \emptyset$ if and only if the following conditions are satisfied:*

(F1) $\mathbf{OutParent}([\overline{\pi}(s)+1 : \overline{\pi}(t)]) = \overline{\pi}(s)$ *(predecessor property)*

(F2) $\mathbf{OutChild}([\overline{\pi}(s) : \overline{\pi}(t)-1]) = \overline{\pi}(t)$ *(successor property)*

Proof. It is shown in Theorem 2 (Page 68) that the statement is true for DAGs. First note that by Lemma 23 (Page 72), every weak superbubbloid intersecting $[r]_{\rightsquigarrow}$ is contained in $[r]_{\rightsquigarrow}$, i.e., in $V(T)$. For the purpose of the proof, consider the auxiliary graph $\hat{G}[V(T)]$ with edge set $E(\hat{G}[V(T)]) = E(G[V(T)]) \setminus \{e \mid e \text{ is a back edge w.r.t. } T\}$. By construction, $\hat{G}[V(T)]$ is acyclic, and every vertex is in T. Thus, every superbubbloid $\langle s,t\rangle$ (with vertex set $[s,t]_{\rightsquigarrow}$) in $\hat{G}[V(T)]$ is characterized by conditions (D1) and (D2) from Theorem 2 (Page 68). It is, furthermore, a weak superbubbloid in G if and only if the following conditions hold:

(i) For every $u \in [s,t]_{\rightsquigarrow} \setminus \{s\}$, there is no edge $(x,u) \in E(G)$ such that $x \notin [s,t]_{\rightsquigarrow}$;

(ii) For every $v \in [s,t]_{\rightsquigarrow} \setminus \{t\}$, there is no edge $(c,x) \in E(G)$ such that $x \notin [s,t]_{\rightsquigarrow}$; and

(iii) $G[[s,t]_{\rightsquigarrow}]$ is without the possible edge (t,s) acyclic.

Only edges not contained in $\hat{G}[V(T)]$ need to be considered for conditions (i) and (ii), because no such edges exist within $\hat{G}[V(T)]$ due to the assumption that $[s,t]_{\rightsquigarrow}$ is a weak superbubbloid in $\hat{G}[V(T)]$. For (iii), only the back edges are of interest. By definition, a back edge creates a cycle in $\hat{G}[V(T)]$. A back edge (v',u') with $u' \in [s,t]_{\rightsquigarrow}$ would violate (iii) if $v' \in [s,t]_{\rightsquigarrow}$ or (i) if $v' \notin [s,t]_{\rightsquigarrow}$. Analogously, if $v' \in [s,t]_{\rightsquigarrow}$ and $u' \notin [s,t]_{\rightsquigarrow}$, then (ii) is violated. Thus, a weak superbubbloid cannot contain the head or tail of a back edge. Only for condition (i), it is needed to consider the case that $x \notin V(T)$.

(F1) can be satisfied only if $\mathbf{OutParent}(u) > -1$ for every $u \in [s,t]_{\rightsquigarrow} \setminus \{s\}$. Analogously, (F2) can only be true if $\mathbf{OutChild}(u) < \infty$ for all $u \in [s,t]_{\rightsquigarrow} \setminus \{t\}$. Hence, it suffices to rule out false positive weak superbubbloids in G by ensuring that every vertex u that violates one of the three conditions also violates (F1) or (F2). This is achieved by setting $\mathbf{OutParent}(u) = -1$ for a vertex u if there is an edge (x,u) such that $x \notin V(T)$ or (x,u) is a back edge; analogously, is $\mathbf{OutChild}(v) = \infty$ for all v with an incident edge (v,x) such that $x \notin V(T)$ or (v,x) is a back edge. Equation 3.6 implements exactly these conditions. Thus, only weak superbubbloids fulfill (F1) and (F2).

Conversely, it suffices to note that by Lemma 24(ii), every weak superbubbloid forms a contiguous interval w.r.t. the postorder π of T and, thus, also w.r.t. the reverse postorder $\overline{\pi}$ of T. □

Superbubble (Algorithm)

Let *Superbubble* be the algorithm *DAGsuperbubble* with the modified functions $\mathbf{OutParent}(.)$ and $\mathbf{OutChild}(.)$ as described above. By construction, *Superbubble* identifies minimal intervals of $\overline{\pi}$ that satisfy (F1) and (F2); see Figure 29 for an illustration. Since the modification of $\mathbf{OutParent}(.)$ and $\mathbf{OutChild}(.)$ only amounts to setting additional entries to -1 or ∞, respectively, the performance remains unaffected. According to Theorem 3, the minimal intervals satisfying (F1) and (F2) are exactly the minimal weak superbubbloids and, thus, by definition, the weak superbubbles.

Corollary 17. *Let G be a digraph, and let T be a DFS-tree on G with a root r that is not an interior vertex or exit of a weak superbubble. Then, Superbubble correctly identifies exactly the weak superbubbles $\langle s,t\rangle$ in G whose vertex set satisfies $[s,t]_{\rightsquigarrow} \cap [r]_{\rightsquigarrow} \neq \emptyset$.*

It is straightforward to extend this result to a DFS-forest that covers $V(G)$ entirely.

Corollary 18. *Let G be a digraph, and let F be a DFS-forest on G comprising DFS-trees T_i with roots r_i, $1 \leq i \leq k$, none of which is an interior vertex or exit of a weak superbubble. Let $\overline{\pi}$ be the reverse postorder on F. Then, Superbubble correctly identifies exactly the weak superbubbles $\langle s,t\rangle$ in G. Furthermore, given the roots r_i, Superbubble has a running time of $\mathcal{O}(|E(G)| + |V(G)|)$.*

Proof. Correctness follows immediately from Corollary 17, the construction of the forest, and Corollary 16. During the DFS, each out-edge is considered exactly once, and each vertex is traversed twice. The number k of required roots is limited by $|V(G)|$. For each vertex v, checking whether $\mathbf{OutParent}(v) = -1$ or $\mathbf{OutChild}(v) = \infty$ requires checking all neighbors only; hence, the total effort is no more than $\mathcal{O}(|E(G)|+|V(G)|)$. The linear time complexity of *DAGsuperbubble*, finally, is proven in Lemma 22 (Page 72). □

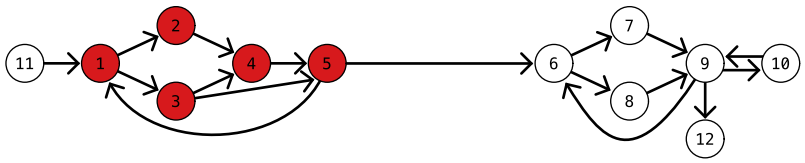
It is important to note the correctness of *Superbubble*, Corollary 17, crucially depends on the correct choice of the roots r_i of the DFS-forest. The remaining problem thus is to find a suitable sequence of roots $r_1, \ldots, r_k$.

legitimate root

Definition 25. *A vertex $r \in V(G)$ is a* legitimate root *if for every weak superbubble $\langle s,t\rangle$ in G with vertex set $[s,t]_{\rightsquigarrow}$, either is $[s,t]_{\rightsquigarrow} \subseteq [r]_{\rightsquigarrow}$ and $t \prec s$ (in the ancestor order of a search tree with root r), or $[s,t]_{\rightsquigarrow} \cap [r]_{\rightsquigarrow} = \emptyset$.*

The discussion can be summarized in the following form:

Corollary 19. *The algorithm Superbubble detects all weak superbubbles in G if and only if there is a set $\{r_1, \ldots, r_k\}$ of legitimate roots such that the DFS-forest F with the roots $\{r_1, \ldots, r_k\}$ covers $V(G)$.*



Position	1	2	3	4	5	6	7	8	9	10	11	12
Vertex	11	1	2	3	4	5	6	7	8	9	10	12
OutParent	−1	−1	2	2	3	4	−1	7	7	−1	10	10
OutChild	2	4	5	6	6	∞	9	10	10	∞	∞	∞

Figure 29: Illustration of the algorithm *Superbubble* on a digraph G with cycles. The top panel shows the input digraph. The DFS-tree T is rooted at 11 and covers $[r]_{\rightsquigarrow} = V(G)$. The table below gives the values of $\mathbf{OutParent}(v)$ and $\mathbf{OutChild}(v)$ as a function of the reverse postorder $\overline{\pi}$ of T. The colored intervals corresponds to the intervals that fulfill (F1) and (F2). The weak superbubble $\langle 2, 5\rangle$ is the only weak superbubble. Compare Figure 27 (Page 69).

Corollary 20. *A vertex $r \in V(G)$ is a legitimate root if and only if r is neither an interior nor an exit of a weak superbubble.*

Proof. By Corollary 18, a root is legitimate if it is not the exit or an interior vertex of a weak superbubble. Conversely, if r is an interior vertex or the exit of $\langle s, t\rangle$, then a DFS-tree rooted in r reaches the entrance s either not at all or there is no search tree with root r such that $t \prec s$, since by definition of a weak superbubble, the exit t is found before s along every path from r to s. □

Lemma 25. *Let G be a digraph and $v \in V(G)$ a source, i.e., a vertex with in-degree zero. Then, v is a legitimate root.*

Proof. Since v is not reachable from any other vertex, it is only reachable by DFS if the traversal starts in v. By the same argument, v is neither an interior vertex, nor the exit of a weak superbubble and, thus, is a legitimate root. □

Unfortunately, there is no guarantee that a digraph G has source vertices, and even if they exist, not every vertex of G is necessarily reachable from them.

3.8 Superbubbles and Cycles

Every vertex in a digraph G that is not reachable from a source in G must be reachable from a cycle in G. Thus identifying legitimate roots in arbitrary cycles leads directly to a root set for the DFS-forest. Thus, a matching between cycle properties and superbubbles can be created.

This matching use one key observation that is independent of cycles. It connects the 2-reach relation in Equation 2.20 (Page 27) with superbubbles:

Corollary 21. *Let G be a digraph, $\langle s,t\rangle$ a weak superbubble in G, and let $v \in V(\langle s,t\rangle) \setminus \{t\}$. If $v \twoheadrightarrow u$ then $u \in V(\langle s,t\rangle)$.*

Proof. Recall that $v \twoheadrightarrow u$ if at least two vertex-independent paths $v \to u$ exists. Assume for contradiction $u \notin V(\langle s,t\rangle)$. After the paths are vertex-independent, t could only be on one of the paths which contradicts that $\langle s,t\rangle$ is a weak superbubble. □

3.8.1 $\overset{C}{\rightsquigarrow}$-Covers and $\overset{C}{\rightsquigarrow}$-Cuts

As "consequence" of Corollary 21, it is useful to know whether two vertices on a cycle C are reachable also via a directed path that is disjoint from C. This idea can be formalized as a binary relation on C.

Definition 26. *Let G be a digraph and C a cycle in G and $v,u \in V(G)$. Then, u is C-*reachable *from v, in symbols $v \overset{C}{\rightsquigarrow} u$, if there is a path $p = \{v = v_0, \dots v_h = u\}$ such that $h \geq 1$ and $v_i \notin C$ for $0 < i < h$.* ***C-reachable***

Note that this definition is a special case of the constrained reach definition in Equation 2.22 (Page 28). C-reachability is defined not only for vertices in the "reference cycle" C. It satisfies a restricted transitivity property: If $w \in V(G) \setminus C$, $v \overset{C}{\rightsquigarrow} w$, and $w \overset{C}{\rightsquigarrow} u$, then $v \overset{C}{\rightsquigarrow} u$. Another interesting observation is that $v \overset{C}{\rightsquigarrow} v$ implies that there is a directed cycle C' such that $C \cap C' = \{s\}$. As an immediate consequence of Definition 26, it follows:

Lemma 26. *Let G be a digraph, C a cycle in G, $c_1, c_2 \in C$ such that $c_1 \overset{C}{\rightsquigarrow} c_2$, and $d_C(c_1,c_2) > 1$. Then, $c_1 \twoheadrightarrow c_2$, i.e., c_1 and c_2 are connected by (at least) two edge-disjoint directed paths. In particular, $\{(v_1, v_{i+1}) \mid 0 \leq i < h\} \cap E(C) = \emptyset$.*

Definition 27. *Let C be a cycle in the digraph G and $v,u \in C$. Then, $C(v\!:\!u)$ is* $\overset{C}{\rightsquigarrow}$-covered *if $v \overset{C}{\rightsquigarrow} u$.* ***$\overset{C}{\rightsquigarrow}$-covered***

The open interval $C(v\!:\!u)$ based on the cyclic set C is termed C-interval in the following. As an immediate consequence of the definition, $v \overset{C}{\rightsquigarrow} v$ implies that $C \setminus \{v\}$ is covered, while nothing is covered if $(v,u) \in E(C)$. ***C-interval***

Definition 28. *Let C be a cycle in the digraph G, and $C(v\!:\!u)$ and $C(x\!:\!y)$ two C-intervals on C. Then:*

(i) $C(v\!:\!u)$ *is* ***included*** *in* $C(x\!:\!y)$ *if* $C(v\!:\!u) \subsetneq C(x\!:\!y)$,

(ii) $C(v\!:\!u)$ *and* $C(x\!:\!y)$ *are* ***disjoint*** *if* $C(v\!:\!u) \cap C(x\!:\!y) = \emptyset$, *and*

(iii) $C(x\!:\!y)$ ***extends*** $C(v\!:\!u)$ *if* $d_C(x,u) < d_C(v,u)$ *and* $d_C(x,u) < d_C(x,y)$.

In particular, if $C(x\!:\!y)$ extends $C(v\!:\!u)$, then $x \in C(v\!:\!u)$, since the interval boundaries themselves are not considered part of the C-intervals. For each pair of distinct C-intervals, thus exactly one of the following four statements is true: (a) the C-intervals are disjoint; (b) one C-interval is contained in (i.e., a proper

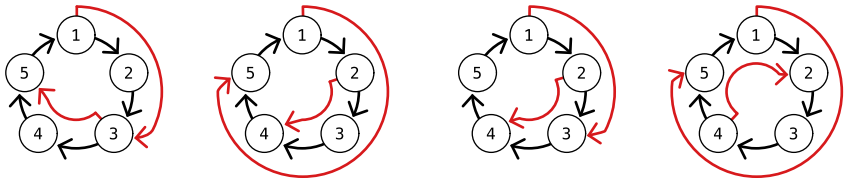


Figure 30: Relationships of distinct C-intervals. The four possibilities for the relative location of two distinct C-intervals (red arrows) are shown on a cycle C with five vertices $(1, 2, 3, 4, 5)$. From left to right: the C-intervals $C(1:3)$ and $C(3:5)$ are *disjoint*; $C(1:5)$ *includes* $C(2:4)$; $C(2:4)$ *extends* $C(1:3)$, but not *vice versa*; $C(1:5)$ and $C(4:2)$ *extend each other*. Together, the two C-intervals cover C.

subset of) the other one; (c) one C-interval, say $C(x:y)$, extends the other one, but not *vice versa*, i.e., $x \in C(v:u)$ and $y \notin C(v:u)$; (d) both C-intervals extend each other, i.e., $x, y \in C(v:u)$. Note that in case (c), the interval boundaries are arranged in the order $v - x - u - y - v$ along the cycle, while in case (d), the arrangement is $v - y - x - u - v$ along C. Figure 30 illustrates the four cases.

In the following, two notations are used:

$$\begin{aligned} \mathfrak{Q}(C) &:= \{C(u:v) \mid u \overset{C}{\rightsquigarrow} v,\ u, v \in C\} \\ Q(C) &:= \bigcup_{B \in \mathfrak{Q}(C)} B = \{w \mid \exists B \in \mathfrak{Q}(C) : w \in B\} \end{aligned} \tag{3.7}$$

for the set of all $\overset{C}{\rightsquigarrow}$-covered intervals and the set of all $\overset{C}{\rightsquigarrow}$-covered vertices of C, respectively. Note that $\emptyset \in \mathfrak{Q}(C)$ since $v \overset{C}{\rightsquigarrow} u$ holds for $(v, u) \in E(C)$. By the same argument, there is at least one interval $C(v:u) \in \mathfrak{Q}(C)$ for each $u \in C$, albeit some or even all of these may be empty.

$\overset{C}{\rightsquigarrow}$-cover **Definition 29.** *A subset $\mathfrak{B} \subseteq \mathfrak{Q}(C)$ is a* $\overset{C}{\rightsquigarrow}$-cover *of C if $\bigcup_{B \in \mathfrak{B}} B = Q(C)$, and $\mathfrak{B}$ is a* total $\overset{C}{\rightsquigarrow}$-cover *of C if $\bigcup_{B \in \mathfrak{B}} B = C$. C is* totally $\overset{C}{\rightsquigarrow}$-covered *if C has a total $\overset{C}{\rightsquigarrow}$-cover.*

total $\overset{C}{\rightsquigarrow}$-cover

Note that C is totally $\overset{C}{\rightsquigarrow}$-covered if and only if $Q(C) = C$. An example of a total cover is shown in Figure 31.

$\overset{C}{\rightsquigarrow}$-cut vertex **Definition 30.** *A vertex in $v \in K(C) := C \setminus Q(C)$ is a* $\overset{C}{\rightsquigarrow}$-cut vertex.

An example of a $\overset{C}{\rightsquigarrow}$-cut vertex is shown in Figure 32. Obviously, C is either totally $\overset{C}{\rightsquigarrow}$-covered or it has a non-empty set $K(C)$ of $\overset{C}{\rightsquigarrow}$-cut vertices.

clean $\overset{C}{\rightsquigarrow}$-cover **Definition 31.** *Let C be a cycle in the digraph G. A* $\overset{C}{\rightsquigarrow}$-cover *$\mathfrak{B}$ of C is* clean *if $B \in \mathfrak{B}$ and $B' \subsetneq B$ implies $B' \notin \mathfrak{B}$.*

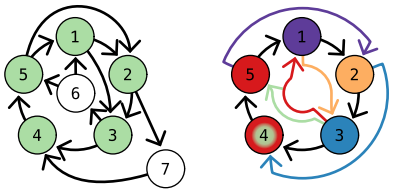


Figure 31: An example of a total $\overset{C}{\rightsquigarrow}$-cover. On the left is a graph with a cycle (green vertices). On the right, $\overset{C}{\rightsquigarrow}$-covered intervals are shown with colored edges. The vertices in the intervals have the same color. After every vertex is covered the cover is a total cover.

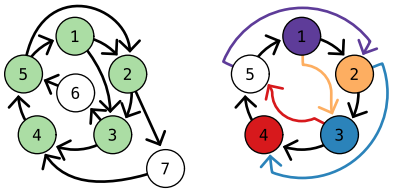


Figure 32: An example of a total $\overset{C}{\rightsquigarrow}$-cut vertex. On the left a graph with a cycle (green vertices) is shown. On the right, $\overset{C}{\rightsquigarrow}$-covered intervals are shown with colored edges. The vertices in the intervals have the same color. As the vertex 5 is not covered, it is a $\overset{C}{\rightsquigarrow}$-cut vertex.

In other words, in a clean $\overset{C}{\rightsquigarrow}$-cover, no $\overset{C}{\rightsquigarrow}$-covered interval is contained within another one.

Corollary 22. *Let C be a cycle in the digraph G, and let $\mathfrak{B}$ be a clean $\overset{C}{\rightsquigarrow}$-cover. Then, either $\mathfrak{B} = \{\emptyset\}$ or, for every $C(v{:}u) \in \mathfrak{B}$, $d_C(v,u) > 1$.*

Proof. Recall that $C(v{:}u) = \emptyset$ if and only if $d_C(v,u) = 1$. Thus, $\mathfrak{Q}(C) = \{\emptyset\}$ if and only if there is no $C(v{:}u) \in \mathfrak{B}$ with $d_C(v,u) > 1$. Since the empty set is a subset of every other set, $d_C(v,u) > 1$ for every $C(v{:}u) \in \mathfrak{B}$ unless $\mathfrak{B} = \mathfrak{Q}(C) = \{\emptyset\}$. □

Lemma 27. *Let C be a cycle in the digraph G. Then, $\mathfrak{Q}(C)$ contains a* clean $\overset{C}{\rightsquigarrow}$-cover $\mathfrak{B}$.

Proof. Let $\mathfrak{B} \subseteq \mathfrak{Q}(C)$ be a set of $\overset{C}{\rightsquigarrow}$-covered intervals that together $\overset{C}{\rightsquigarrow}$-cover $Q(C)$. Suppose $\mathfrak{B}$ is not clean. Then, there are two intervals $C(p{:}q) \in \mathfrak{B}$ and $C(v{:}u) \in \mathfrak{B}$ such that $C(p{:}q) \subsetneq C(v{:}u)$. Then, $\mathfrak{B}' = \mathfrak{B} \setminus \{C(p{:}q)\}$ still $\overset{C}{\rightsquigarrow}$-cover $Q(C)$. The removal of such redundant intervals can be repeated until no further removable interval can be found. By Definition 31, the remaining $\overset{C}{\rightsquigarrow}$-cover is clean. □

Definition 32. *Let C be a cycle in a digraph G. Then:*

$$\mathfrak{L}(C) = \{C(v{:}u) \in \mathfrak{Q}(C) \mid \forall C(v{:}u') \in \mathfrak{Q}(C) : d_C(v,u') \leq d_C(v,u)\}$$

By definition, $\mathfrak{L}(C)$ consists of all $\overset{C}{\rightsquigarrow}$-covered intervals for which there is no larger $\overset{C}{\rightsquigarrow}$-covered interval with the same starting point. Since every $C(p{:}q) \in \mathfrak{Q}(C) \setminus \mathfrak{L}(C)$ is contained in a interval with the same starting point, $\mathfrak{L}(C)$ is a $\overset{C}{\rightsquigarrow}$-cover of C. Thus, Lemma 27 implies:

Corollary 23. *Let C be a cycle in a digraph G. Then, there is clean cover $\mathfrak{B} \subseteq \mathfrak{L}(C)$.*

Lemma 28. *Let C be a cycle in the digraph G. A clean $\overset{C}{\rightsquigarrow}$-cover $\mathfrak{B}$ of C is total if and only if every $B \in \mathfrak{B}$ is extended by at least one $B' \in \mathfrak{B}$.*

Proof. Suppose, for contradiction, that $C(v:u) \in \mathfrak{B}$ is not extended by any $B \in \mathfrak{B}$. Then, any interval $B' \in \mathfrak{B}$ $\overset{C}{\rightsquigarrow}$-covering u would have to contain $C(v:u)$, contradicting the assumption that $\mathfrak{B}$ is clean. Thus, u is a $\overset{C}{\rightsquigarrow}$-cut vertex, and hence, $\mathfrak{B}$ is not total. If $\mathfrak{B}$, therefore it is non-empty, and every $B \in \mathfrak{B}$ is extended by some $B' \in \mathfrak{B}$.

Conversely, suppose c is a $\overset{C}{\rightsquigarrow}$-cut vertex of C. If $C(v:c) \in \mathfrak{B}$ for some v, then the first part of the proof implies that $C(v:c)$ is not extended by any $B' \in \mathfrak{B}$. If $\mathfrak{B}$ contains no interval $C(v:c)$, then consider the vertex u such that $C(v:u) \in \mathfrak{B}$ for which $d_C(u,c)$ is minimal. Since c is a $\overset{C}{\rightsquigarrow}$-cut vertex, there is no extension of $C(v:u)$, since any such extension B' would either contradict the minimality of $d_C(u,c)$ or $\overset{C}{\rightsquigarrow}$-cover c, thereby contradicting the assumption that c is a $\overset{C}{\rightsquigarrow}$-cut vertex. Thus, u is again a $\overset{C}{\rightsquigarrow}$-cut vertex. As shown in the first part of the proof, $C(v:u)$; therefore, it is not extended by any $B \in \mathfrak{B}$. To concluded: unless $\mathfrak{B}$ is a total $C(v:u)$-cover, there is an interval $B \in \mathfrak{B}$ without an extension. □

Another interesting type of $\overset{C}{\rightsquigarrow}$-cover is the minimal cover. Even if they are not useful for the superbubble detection, an analysis is given in the following.

minimal $\overset{C}{\rightsquigarrow}$-cover

Definition 33. *Let C be a cycle in C. A* $\overset{C}{\rightsquigarrow}$-cover $\mathfrak{B}$ *is* minimal *if*

(i) $\mathfrak{B}$ is clean,

(ii) every $\overset{C}{\rightsquigarrow}$-covered interval $B \in \mathfrak{B}$ is extended by at most one $B' \in \mathfrak{B}$.

Lemma 29. *Let C be a cycle in G. Then $\mathfrak{Q}(C)$ contains a* minimal $\overset{C}{\rightsquigarrow}$-cover $\mathfrak{B}$.

Proof. That a clean cover $\mathfrak{B} \subseteq \mathfrak{Q}(C)$ exists is shown in Lemma 27. Suppose $C(v:u)$ is extended by both $C(p:q) \in \mathfrak{B}$ and $C(p':q') \in \mathfrak{B}$, i.e., $p, p' \in C(v:u)$ and w.l.o.g., assume $d_C(v,q) \leq d_C(v,q')$. Thus $C(v:u)$ and $C(p:q)$ together $\overset{C}{\rightsquigarrow}$-cover a subset of the vertices $\overset{C}{\rightsquigarrow}$-covered by the union of $C(v:u)$ and $C(p':q')$. Therefore, $C(p:q)$ can be omitted from $\mathfrak{B}$ without affecting the set of $\overset{C}{\rightsquigarrow}$-covered vertices. □

Corollary 24. *Let C be a cycle in G. A minimal $\overset{C}{\rightsquigarrow}$-cover $\mathfrak{B}$ of C is total if and only if every $B \in \mathfrak{B}$ is extended by exactly one $B' \in \mathfrak{B}$.*

Proof. The correctness follows directly from Lemma 28 and Definition 33. □

Examples of a clean and a minimal $\overset{C}{\rightsquigarrow}$-cover are shown in Figure 33.

The following lemma provides us with a convenient way to obtain a total $\overset{C}{\rightsquigarrow}$-cover.

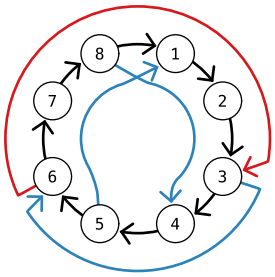


Figure 33: An example of a clean and a minimal $\overset{C}{\rightsquigarrow}$-cover. A cycle with eight vertices is shown. The colored edges create $\overset{C}{\rightsquigarrow}$-covered intervals. The blue $\overset{C}{\rightsquigarrow}$-covered intervals together create a clean and minimal $\overset{C}{\rightsquigarrow}$-cover. The blue $\overset{C}{\rightsquigarrow}$-covered intervals and the red $\overset{C}{\rightsquigarrow}$-covered interval together are still a clean $\overset{C}{\rightsquigarrow}$-cover but not minimal. Note that the red $\overset{C}{\rightsquigarrow}$-covered interval has the maximal number of elements of all $\overset{C}{\rightsquigarrow}$-covered intervals.

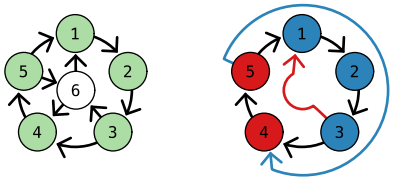


Figure 34: An example of a single-vertex $\overset{C}{\rightsquigarrow}$-cover. On the left a graph with a green highlighted cycle is shown. On the right the single-vertex $\overset{C}{\rightsquigarrow}$-cover is shown. It consists of two $\overset{C}{\rightsquigarrow}$-covered intervals that are created by the paths $\{3, 6, 1\}$ (red) and $\{5, 6, 4\}$ (blue).

Lemma 30. *Let C be a cycle in digraph G, $v \notin C$, and $c_1, c_2, c_3, c_4 \in C$ with $d_C(c_1, c_3) \leq d_C(c_1, c_2) < d_C(c_1, c_4)$. Then, $c_1 \overset{C}{\rightsquigarrow} v$, $c_2 \overset{C}{\rightsquigarrow} v$, $v \overset{C}{\rightsquigarrow} c_3$, and $v \overset{C}{\rightsquigarrow} c_4$ imply that $\mathfrak{B} := \{C(c_1 : c_4), C(c_2 : c_3)\}$ is a total clean (minimal) $\overset{C}{\rightsquigarrow}$-cover of C.*

Proof. By construction, $C(c_1 : c_4)$ and $C(c_2 : c_3)$ are $\overset{C}{\rightsquigarrow}$-covered intervals. By definition, $c_2 \in C(c_1 : c_4)$, and $C(c_2 : c_3)$ extends $C(c_1 : c_4)$. Since $c_3 \in C(c_1 : c_4)$, the two intervals cover all of C. Furthermore, the cover $\mathfrak{B}$ consists of only two intervals that are not subsets of each other; thus, it is clean and minimal. □

This type of total clean $\overset{C}{\rightsquigarrow}$-cover is referred to as a *single-vertex $\overset{C}{\rightsquigarrow}$-cover* of C. An example is shown in Figure 34.

single-vertex $\overset{C}{\rightsquigarrow}$-cover

3.8.2 Legitimate Roots from $\overset{C}{\rightsquigarrow}$-Cover and $\overset{C}{\rightsquigarrow}$-Cuts

Recall that after Theorem 1 (Page 60) every superbubble is either contained in a strongly connected component (SCC) or disjoint of any non-singelton SCC. The following results on the interaction of cycles and superbubbles are a generalization of this observation. The acyclicity condition (S.v) can be restated in the following way:

Lemma 31. *Let $\langle s, t\rangle$ be a weak superbubbloid in the digraph G and $u \in V(\langle s, t\rangle)$. Then, every cycle containing u also contains s and t.*

Proof. If $u \neq s$, then all predecessors of u are contained in $\langle s, t\rangle$. Similarly, if $u \neq t$, then all successors of u are contained in $\langle s, t\rangle$. Since every cycle through u contains both in- and successors of u, it, in particular, contains an edge e in $\langle s, t\rangle$. (S.v) now implies any cycle through e contains both s and t. □

Lemma 32. *Let C be a cycle in the digraph G, and let $\mathfrak{B}$ be a total clean $\overset{C}{\rightsquigarrow}$-cover of C. If $C(v:u) \in \mathfrak{B}$, then u is neither an interior, nor an exit of a weak superbubble, i.e., u is a legitimate root.*

Proof. Assume, for contradiction, that u is an interior or the exit of the superbubble $\langle s,t\rangle$. Since C is totally $\overset{C}{\rightsquigarrow}$-covered by assumption, Corollary 22 implies $d_C(v,u) > 1$. Thus, by Lemma 26 (Page 79), $v \rightrightarrows u$. Therefore, Corollary 21 (Page 79) implies $\langle s,t\rangle$ contains $C(v:u)$.

Since $\mathfrak{B}$ is a total clean $\overset{C}{\rightsquigarrow}$-cover of C, there is an interval $C(p:q) \in \mathfrak{B}$ that extends $C(v:u)$, i.e., $p \in C(v:u)$, and hence, p is an inner vertex of $\langle s,t\rangle$. Therefore, $\langle s,t\rangle$ contains $C(p:q)$, and it again has an extending $\overset{C}{\rightsquigarrow}$-interval. Repeating the argument, it is concluded that every vertex of $Q(C)$ is an inner vertex of $\langle s,t\rangle$. Since the cover $\mathfrak{B}$ is total, $Q(C) = C$, i.e., the cycle C consists entirely of interior vertices of $\langle s,t\rangle$, i.e., C is a proper subset of $\langle s,t\rangle$. This contradicts the acyclicity condition (S.v). □

Corollary 25. *Let C be a cycle in the digraph G. Suppose C is totally $\overset{C}{\rightsquigarrow}$-covered, and let $C(v:u) \in \mathfrak{L}(C)$ such that $d_C(v',u') \leq d_C(v,u)$ for all $C(v':u') \in \mathfrak{L}(C)$. Then, u is a legitimate root.*

Proof. The longest $\overset{C}{\rightsquigarrow}$-interval $C(v:u) \in \mathfrak{L}(C)$ by construction cannot be contained within another $\overset{C}{\rightsquigarrow}$-interval. Therefore, $C(v:u)$ is contained in the clean cover $\mathfrak{B} \subseteq \mathfrak{L}(C)$ of Corollary 23. By Lemma 32, its endpoint u is a legitimate root. □

Note that the $C(v:u)$ interval from Corollary 25, i.e., the interval with the maximal number of elements must not be contained in a minimal $\overset{C}{\rightsquigarrow}$-cover (compare Figure 33).

Let us now turn to cycles with $\overset{C}{\rightsquigarrow}$-cut vertices:

Lemma 33. *Let C be a cycle of the digraph G, and let c be a $\overset{C}{\rightsquigarrow}$-cut point of C, i.e., $c \in K(C)$. Then, c is not an interior vertex of any weak superbubble.*

Proof. Assume, for contradiction, that c is an interior vertex of a weak superbubble $\langle s,t\rangle$. Then, there is a path p from s to t not passing through c. Otherwise, $\langle s,c\rangle$ is a superbubbloid, contradicting the assumption that $\langle s,t\rangle$ is a weak superbubble; see Lemma 6 (Page 58). Along p, let u be the last vertex on C before c, and let v be the first vertex on C after c. Thus, $u \overset{C}{\rightsquigarrow} v$. Therefore, c is $\overset{C}{\rightsquigarrow}$-covered in C, a contradiction. □

The example in Figure 35 shows that it is possible that every entrance of a superbubble is at the same time the exit of another superbubble. Such graphs do not have any legitimate root. Nevertheless, it is easily possible to obtain all the superbubbles. To this end, fix a $\overset{C}{\rightsquigarrow}$-cut vertex c for some cycle C in G, and consider the auxiliary digraph $G^{\#}$ obtained from G by splitting c into two vertices c' and c'' so that c' retains only the in-edges and c'' retains only the out-edges.

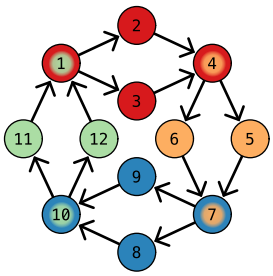


Figure 35: A digraph G without any legitimate root. In G are 16 isomorphic cycles containing eight of the 12 vertices, all of which contain $\{1, 4, 7, 10\}$. The superbubbles $\langle 1, 4\rangle$ (red),$\langle 4, 7\rangle$ (orange), $\langle 7, 10\rangle$ (blue), and $\langle 10, 1\rangle$ (green) cover G entirely, i.e., every entrance of a superbubble is also the exit of another one, and all other vertices are interior vertices of a superbubble.

Lemma 34. *Let C be a cycle in the digraph G, $c \in C$ a $\overset{C}{\rightsquigarrow}$-cut vertex, and $G^{\#}$ the digraph obtained from G by splitting c. If $\langle s, t\rangle$ is a weak superbubble in G, then it is also a weak superbubble in $G^{\#}$, where c as an entrance in G corresponds to c' in $G^{\#}$ and c as an exit in G corresponds to c'' in $G^{\#}$. Conversely, every weak superbubble $\langle s, t\rangle$ with $\{s, t\} \neq \{c'', c'\}$ in $G^{\#}$ is also a weak superbubble in G.*

Proof. For the proof, consider the auxiliary graph $\tilde{G}^{\#}$ that is constructed by inserting the edge (c', c'') into $G^{\#}$ (compare Figure 36). Then, there is an one to one relationship between the set of paths in G and the set of paths that do not start or end with the edge (c', c'') in $\tilde{G}^{\#}$, which is constructed as follows: If p starts at c in G, it starts in c'' in $\tilde{G}^{\#}$; if p ends at c in G, it ends at c'' in $\tilde{G}^{\#}$; and if p runs through c in G, then it runs through the edge (c', c'') in $\tilde{G}^{\#}$. The one to one correspondence of weak superbubbles now follows immediately from the equivalence of the path systems in G and $\tilde{G}^{\#}$ since reachability is the same for every pair u, v, with c as the starting point corresponding to c'' and c as the endpoint corresponding to c'. Thus, G and $\tilde{G}^{\#}$ have the same superbubbles, except possibly for the ones with $\{s, t\} = \{c'', c'\}$ in $\tilde{G}^{\#}$. Now, consider a depth-first search (DFS)-tree on $G^{\#}$ rooted in c''. The edge (c', c'') is not a tree edge and necessarily appears as a back edge. Since c is a $\overset{C}{\rightsquigarrow}$-cut vertex, c' and c'' are not interior vertices of any weak superbubble in $\tilde{G}^{\#}$. Thus, the edge (c', c'') does not affect any weak superbubble of $\tilde{G}^{\#}$, and thus, $G^{\#}$ and $\tilde{G}^{\#}$ have the same weaksuperbubbles, except possibly the ones with $\{s, t\} = \{c', c''\}$. □

The only potential difference between the weak superbubbles of G and $G^{\#}$ is, therefore, the possibility that $G^{\#}$ contains $\langle c', c''\rangle$ or $\langle c'', c'\rangle$ as an additional weak superbubble. Of course, it is easy to detect and remove the additional weak superbubble. Since c'' is a source in $G^{\#}$, *Superbubble* can be applied to $G^{\#}$ and remove the possible spurious weak superbubble $\langle c'', c'\rangle$ in order to obtain the correct set of weak superbubbles of G. In contrast to the auxiliary digraph constructions suggested in Sung et al. (2015), $G^{\#}$ contains only a single extra vertex instead of doubling the size. More importantly, however, it is not necessary to construct $G^{\#}$ explicitly. Instead, one can modify the DFS starting at c in G in the following manner: if c is encountered for the first time as a successor of a tree vertex u, then c'' is inserted with parent u and no further successors, with only a constant overhead. The algorithm *Superbubble* applied to $G^{\#}$ extracts the minimal intervals satisfying (F1) and (F2) from Theorem 3 (Page 76) w.r.t.

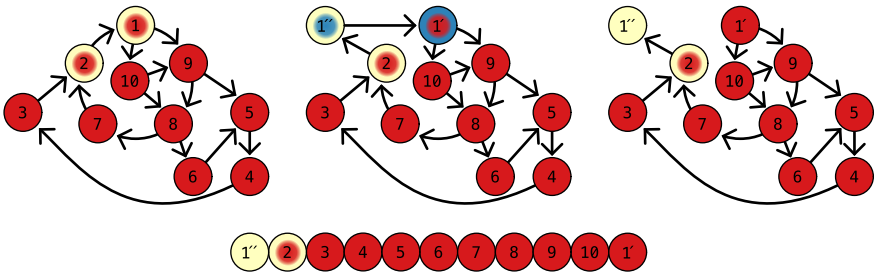


Figure 36: Relationships of G, $\tilde{G}^{\#}$, and $G^{\#}$. On the top are shown from left to right: G, $\tilde{G}^{\#}$, and $G^{\#}$. G contains a cycle $C = \circlearrowleft 1, 9, 8, 7, 2 \circlearrowright$. On this cycle is 1 a $\overset{C}{\rightsquigarrow}$-cut vertex. Furthermore, G contains two superbubbles: $\langle 1, 2\rangle$ (red vertices) and $\langle 2, 1\rangle$ (yellow vertices). Every path in G also exists in $\tilde{G}^{\#}$ by replacing 1 in a path by the two vertices $1'$ and $1''$. Thus, $\tilde{G}^{\#}$ has the same superbubbloids as G except for the superbubble $\langle 1'', 1'\rangle$ (blue vertices) and the superbubbloid $\langle 1', 1''\rangle$. In $G^{\#}$ the edge $(1'', 1')$ is missing which destroys the paths that contain 1 as an internal vertex. However, as 1 is a $\overset{C}{\rightsquigarrow}$-cut vertex such a path can not be included in a superbubbloid of G. Thus, $G^{\#}$ contains the same superbubbloids as G except $\langle 1', 1''\rangle$. In $G^{\#}$ is $1'$ a legitimate root and thus a DFS-tree rooted in $1'$ can be used for superbubble detection. The postorder of such a tree is shown at the bottom. Note that the same postorder is created in G with a modified DFS-tree of *Superbubble#* rooted in 1.

Superbubble# (Algorithm)

the reverse postorder $\overline{\pi}$ of the DFS-tree rooted as c', and thus correctly identifies the weak superbubbles of $G^{\#}$. The modified DFS on G rooted at c by construction yields the same DFS-tree on $G^{\#}$, and thus the same reverse postorder. Together with setting $\mathbf{OutChild}(c'') = \mathbf{OutChild}(c)$, $\mathbf{OutParent}(c') = \mathbf{OutParent}(c)$, $\mathbf{OutChild}(c') = \infty$, and $\mathbf{OutParent}(c'') = -1$, *Superbubble* operating on the modified DFS-tree thus correctly identifies the weak superbubbles in $G^{\#}$. This algorithm, which is equivalent to applying *Superbubble* to $G^{\#}$, is named *Superbubble#*. An example of postorder created with such a modified DFS-tree is shown in Figure 36.

quasi-legitimate root

Definition 34. *Let G be a digraph. Then, $r \in V(G)$ is a* quasi-legitimate root *if either:*

(i) r is source in G,

(ii) r is the end point of an interval $C(v{:}r) \in \mathfrak{B}$ of a total clean $\overset{C}{\rightsquigarrow}$-cover of some cycle C in G, or

(iii) r is $\overset{C}{\rightsquigarrow}$-cut vertex of some cycle C in G.

Our discussion so far can be summarized as:

Corollary 26. *Algorithm Superbubble# correctly identifies the superbubbles in $G[[r]_{\rightsquigarrow}]$ if and only if r is a quasi-legitimate root.*

As an immediate consequence of Lemma 32 and Lemma 33, every cycle contains a quasi-legitimate root. Recalling that every vertex in the digraph G can be reached either from a source vertex or from a cycle, it finally can be obtained:

Theorem 4. *Every digraph G contains a set of quasi-legitimate roots $\{r_1, \ldots, r_k\}$. Given these roots, the algorithm `Superbubble#` correctly identifies all superbubbles of G in linear time.*

It remains to show, therefore, that a suitable set of roots can be identified in linear time. Clearly, this is possible for the sources. For superbubbles that cannot be reached from a source vertex, a suitable set of cycles needs to be identified.

3.8.3 Finding start cycles

First an overview of the connection between DFS and cycles is given.

Lemma 35. *Let F be an arbitrary DFS-forest of the digraph G with the root set $\{r_1, \ldots, r_k\}$ that is created from the trees T_i, and let C be a cycle in G. Then, $C \cap V(T_i) \neq \emptyset$ implies $C \subseteq V(T_i)$, and there is a $v \in C$ such that $C \subseteq V(T_i(v))$.*

Proof. Let r_i be the first root of F that can reach any vertex of C. Then, by definition of a cycle, $C \subseteq [r_i]_{\rightsquigarrow}$. Thus, $C \subseteq V(T_i)$. Furthermore, let v be the first vertex that is reached from r_i in the DFS. Then, every other vertex of C is reached from v in the DFS. Thus, $C \subseteq V(T_i(v))$. □

The same is true for strongly connected components (SCCs):

Lemma 36. *R. Tarjan (1972) (corollary 11)*
Let S be a SCC in G, and let T be a DFS-tree with $S \subseteq V(T)$. Then, there is a vertex $v \in S$ such that $S \subseteq V(T(v))$. Then is v the root of the SCC S in T.

Our aim is now to find a set of "start cycles" such that every cycle C is reachable from at least one of these start cycles.

Lemma 37. *Let T be a DFS-tree on the digraph G rooted in r, and let W be the set of $\prec$-maximal vertices w that have an incoming back edge (v, w). Then, (i) $w \in W$ is contained in a cycle, and (ii) every cycle $C \subseteq V(T)$ is satisfied $C \subseteq V(T(w))$ for some $w \in W$.*

Proof. Property (i) is an immediate consequence of the definition of DFS. Now, suppose $v \notin V(T(w))$ for some $w \in W$. Then, by construction, none of the vertices along the path from the root r to v have an incoming back edge, and thus, neither v, nor one of its ancestors are contained in a cycle. Thus, if $x \in C$ for some cycle $C \subseteq V(T)$, then a vertex $w \in W$ exists such that $x \in V(T(w))$, and thus,$C \subseteq V(T(w))$. □

Note that $W = \emptyset$ if T does not contain a cycle. Since the vertex set of every cycle in the digraph G is necessarily contained in one of the constituent trees of a DFS-forest, it can be immediately obtained:

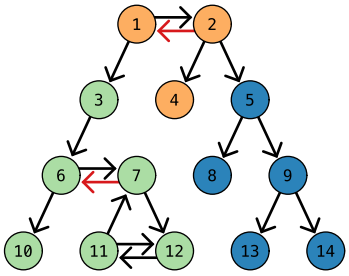


Figure 37: An example of cycle detection with a DFS-forest. The DFS-forest contains three trees and the ordered root set consist of: 5 (blue tree), 3 (green tree), and 1 (orange tree). The reported cycles are $\circlearrowleft 6, 7 \circlearrowright$ and $\circlearrowleft 1, 2 \circlearrowright$. They are identified by the red back edges. Note that the cycle $\circlearrowleft 7, 12, 11 \circlearrowright$ is never reported but this is fine because it is reachable from $\circlearrowleft 6, 7 \circlearrowright$.

Corollary 27. *Let F be a DFS-forest on the digraph G, and let W by the set of $\prec$-maximal vertices w that have an incoming back edge (v, w). Then, (i) $w \in W$ is contained in a cycle, and (ii) every cycle C in G is satisfied $C \subseteq V(T(w))$ for some $w \in W$ and some $T \in F$.*

An example of such a forest and the reported cycles are shown in Figure 37.

Lemma 38. *A set of cycles $\{C_1, \ldots, C_n\}$ from which all cycles in G are reachable can be constructed in $\mathcal{O}(|E(G)| + |V(G)|)$ time.*

Proof. The DFS-forest F on the digraph G is obtained in $\mathcal{O}(|E(G)|+|V(G)|)$ time. The set W is easily identified by a preorder traversal of F omitting a subtree as soon as a vertex w has an incoming back edge. The worst-case effort is $\mathcal{O}(|V(G)|)$ since only the forest is traversed, not the entire digraph G. Given W and the associated back edges (v_k, w_k) identified in the previous steps for each $w_k \in W$, the cycle C_k is explicitly retrieved by following the parent links of F from v_k back to w_k in $\mathcal{O}(|V(G)|)$ time. □

Lemma 38 ensures that a sufficient set of cycles can be found in linear time. More precisely, using the sources of G and a quasi-legitimate root r_i in each cycle G_i as roots, the algorithm *Superbubble#* correctly identifies all superbubbles in G in linear time. It remains to show that a quasi-legitimate root can be identified in a cycle C_i.

3.8.4 Identification of Quasi-Legitimate Roots

The obvious approach to identify quasi-legitimate roots is to construct a clean $\overset{C}{\rightsquigarrow}$-cover. The obvious starting point is $\mathfrak{L}(C)$ (Definition 32 (Page 81)) since it requires the construction of no more than the $|C|$ $\overset{C}{\rightsquigarrow}$-path. This can be achieved in polynomial time, e.g., using an independent DFS-tree rooted at $c \in C$ that ignores the edges of C. This naive approach, however, exceeds linear time even for a single cycle.

For $c \in C$, a modified DFS-tree T_c is constructed by excluding all other vertices of C from G. By construction, $u \in C$ is $\overset{C}{\rightsquigarrow}$-reachable from c if and only if T_c contains an predecessor u' of u, i.e., there is an edge $(u', u) \in E(G)$ with $u' \in V(T_c)$.

For each $v \in V(T_c)$, the vertices $\min_c(v)$ and $\max_c(v)$ are of interest. Both are vertices in C such that they are $\overset{C}{\rightsquigarrow}$-reachable from v. They minimize (maximize) $d_C(c,.)$ for all vertices that are $\overset{C}{\rightsquigarrow}$-reachable from v. These can be recursively computed on T_c by traversing T_c in postorder. For each $v \in V(T_c)$, $\min_c(v)$ and $\max_c(v)$ are obtained by comparing the $\min_c(u)$ and $\max_c(u)$ values for the successor (u) of v along T, and the vertices reachable directly from v. More precisely, at each leaf v of T_c, $\max_c(v)$ is initialized by the vertex $c' \in C$ such that $(v,c') \in E(G)$ and c' maximizes $d_C(c,c')$. At each inner vertex v of T_c, $\max_c(v)$ is computed as the vertex c' maximizes $d_C(c,c')$ from the following set of candidates: $\{\max_c(u) \mid (v,u) \in E(T_c)\} \cup \{u \in C \mid (v,u) \in E(G)\}$. The vertex (c') $\overset{C}{\rightsquigarrow}$-reachable from c with the maximal value of $d_C(c,c')$ is thus $\max_c(c)$. The same computations are used for $\min_c(v)$, except that $d_C(c,c')$ is minimized instead of maximized. The computations of T_c and values of $\min_c(v)$ and $\max_c(v)$ clearly can be performed in linear time. Repeating this for each $c \in C$, however, in general, it exceeds linear time since the length $|C|$ is not bounded in general.

Most of the information stored in T_c can be reused. However, before this is proven, a small corollary is given that is very helpful for this:

Corollary 28. *Let G be a digraph; let C a cycle in G; and let $c_1, c_2, c_3 \in C$. Then, $d_C(c_1,c_2) \leq d_C(c_1,c_3)$ if and only if $c_1 \in C(c_3 : c_2) \cup \{c_3\}$.*

Proof. If c_1, c_2, and c_3 are pairwise distinct, the l.h.s. is true if the path from c_1 to c_2 is a subpath of the path from c_1 to c_3, i.e., $c_1 \notin C(c_2 : c_3)$ and, thus, $c_1 \in C(c_3 : c_2) \cup \{c_3\}$. The converse is obvious. The statement is trivial for $c_1 = c_3$. If $c_2 = c_3$, the l.h.s. is always true, while on the r.h.s., $C(c_2 : c_2) \cup \{c_2\} = C \ni c_1$. For $c_1 = c_2$, both the l.h.s. and the r.h.s. are satisfied only if $c_2 = c_3$. □

The crucial observation is the following:

Lemma 39. *Let C be a cycle of the digraph G; consider two distinct cycle vertices $c_1, c_2 \in C$; and let $v \notin C$ with $c_1 \overset{C}{\rightsquigarrow} v$ and $c_2 \overset{C}{\rightsquigarrow} v$. If $d_C(c_2, \min_{c_1}(v)) \leq d_C(c_2, \max_{c_1}(v))$, then $\min_{c_2}(v) = \min_{c_1}(v)$ and $\max_{c_2}(v) = \max_{c_1}(v)$. Otherwise, $\mathfrak{B} = \{C(c_1 : \max_{c_1}(v)), C(c_2 : \min_{c_1}(v))\}$ forms a single-vertex $\overset{C}{\rightsquigarrow}$-cover.*

Proof. For simplicity, set $c_3 = \min_{c_1}(v)$ and $c_4 = \max_{c_1}(v)$. By definition of $\min_{c_1}(.)$ and $\max_{c_1}(.)$, (1) $d_C(c_1,c_3) \leq d_C(c_1,c_4)$, and (2) for every $c \in C$ satisfying $v \overset{C}{\rightsquigarrow} c$, $c \in C(c_3 : c_4) \cup \{c_3, c_4\}$. Starting from Property (1), Corollary 28 implies $c_1 \in C(c_4 : c_3) \cup \{c_4\}$. As a consequence, for every $c \in C(c_3 : c_4) \cup \{c_4\}$, $d_C(c_1,c) = d_C(c_1,c_3) + d_C(c_3,c)$. Since $d_C(c_1,c_3)$ is just a constant, $d_C(c_1,a) \leq d_C(c_1,b)$ implies $d_C(c_3,a) \leq d_C(c_3,b)$ for all $a,b \in d_C(c_3,c_4) \cup \{c_4\}$.

First, assume $d_C(c_2,c_3) \leq d_C(c_2,c_4)$. Then, Corollary 28 implies $c_2 \in C(c_4 : c_3) \cup \{c_4\}$. The same arguments as for c_1 show that $d_C(c_2,a) \leq d_C(c_2,b)$ implies $d_C(c_3,a) \leq d_C(c_3,b)$, which in turn implies $d_C(c_2,a) \leq d_C(c_1,b)$ for all $a,b \in C(c_3 : c_4) \cup \{c_4\}$. Because of property (2), this implication can be used in particular for every $c \in C$ for which $v \overset{C}{\rightsquigarrow} c$ might hold. Therefore, the same two vertices

minimize and maximize $d_C(c_1, v)$ and $d_C(c_2, v)$, and thus, $\min_{c_2}(v) = \min_{c_1}(v)$ and $\max_{c_2}(v) = \max_{c_1}(v)$.

Now, suppose $d_C(c_2, c_4) < d_C(c_2, c_3)$. Then, $c_3 \neq c_4$ (otherwise, the distances would be equal), and Corollary 28 implies $c_2 \in C(c_3 : c_4) \cup \{c_3\}$. Since $c_1 \in C(c_4 : c_3) \cup \{c_4\}$, is $d_C(c_1, c_3) \leq d_C(c_1, c_2) < d_C(c_1, c_4)$. By Lemma 30 (Page 83), $\mathfrak{B} := \{C(c_1 : c_4), C(c_2 : c_3)\}$ is a single-vertex cover of C. □

The use of Lemma 39 is that it allows either to use the $\min_{c_i}(v)$ and $\max_{c_i}(v)$ values also for c_2, or a single-vertex $\overset{C}{\rightsquigarrow}$-cover is obtained, which immediately provides us with a legitimate root according to Lemma 32 (Page 84). Thus, the computation of $\min_{c_i}(v)$ and $\max_{c_i}(v)$ can be stopped when a single-vertex cover is encountered. Up to this point, the values of $\min_{c_i}(v)$ and $\max_{c_i}(v)$ are independent of c_i by Lemma 39.

The difficulty is to compute the $\min_{c_1}(v)$ and $\max_{c_1}(v)$ for all $v \in V(T_c)$ correctly. How to handle tree edges is already explained above. Forward edges in T_c do not effectively contribute, because the same information (minimization or maximization over values of $d_C(c, .)$) is also propagated stepwise along the tree-edges. Cross edges, on the other hand, could add information. Postorder traversal ensures, however, that the pertinent information at their starting points is already computed in time to include them to compute the correct value, i.e., include the cross-edges in the minimization/maximization step.

Back edges are problematic if they belong to the same SCC S as $C \subsetneq C$. In this case, they can be reached from a cycle vertex $c \in C$ and they reach a cycle vertex $u \in C$. Such back edges, therefore, influence which cycle vertices are reachable. To handle this information, S is split into parts that are SCCs under the use of $\overset{C}{\rightsquigarrow}$-reachability. More precisely, define a $\overset{C}{\rightsquigarrow}$-SCC as a SCC on the induced subgraph $G[V(G) \setminus C]$.

Consider the auxiliary graph G_c with vertex set $(V(G) \setminus C) \cup \{c\}$ and all edges of $G[V(G) \setminus C]$, as well as all edges (c, u) with $u \in V(G) \setminus C$. Then, the SCCs of G_c are exactly the $\overset{C}{\rightsquigarrow}$-SCCs and the single vertex c. By construction, T_c is also a DFS-tree for G_c. Thus, Tarjan's DFS-based SCC-detection algorithm (see Lemma 36) on T_c identifies the $\overset{C}{\rightsquigarrow}$-SCC as the SCC of G_c. To mimic the traversal on G_c instead on $G[(V(G) \setminus C) \cup \{c\}]$, the graph on which T_c is originally defined, it suffices to ignore the back edge leading to the root, i.e., edges of the form (u, c) for $u \in V(G) \setminus C$. It is thus not necessary to construct the graph G_c explicitly.

The definitions of $\min_c(.)$ and $\max_c(.)$ imply:

Corollary 29. *Let C be a cycle in the digraph G; let T_c be a modified DFS-tree rooted at $c \in C$; and let S be a $\overset{C}{\rightsquigarrow}$-SCC with $S \subseteq V(T_c)$. Then, $\min_c(v)$ and $\max_c(v)$ are independent of v for every $v \in S$.*

This begs the question of whether the v-independent values of $\min_c(v)$ and $\max_c(v)$ can be obtained while traversing G. A partial answer is provided by:

Corollary 30. *Let C be a cycle in the digraph G; let T_c be a modified DFS-tree rooted at $c \in C$; and let v be the root of a $\overset{C}{\rightsquigarrow}$-SCC. Suppose the values of $\min_c(w)$*

and $\max_c(w)$ *are known for* $w \notin V(T_c(v))$. *Then,* $\min_c(v)$ *and* $\max_c(v)$ *are obtained correctly by postorder traversal of* T_c *considering all tree and cross edges.*

Proof. The only missing information could be a back edge (u, w) with $u \in V(T_c(v))$ and $v \prec w$. Such a back edge cannot exist because v is by assumption the root of a $\overset{C}{\rightsquigarrow}$-SCC, and thus, there is no cycle including u, v, and $w \in G[V(G) \setminus C]$. □

This observation yields a simple solution to obtain the correct entries for $\min_{c_i}(v')$ and $\max_{c_i}(v')$ for every $v' \in S$: determine the $\overset{C}{\rightsquigarrow}$-SCC and its root v, and set $\min_c(v') \leftarrow \min_c(v)$ and $\max_c(v') \leftarrow \max_c(v)$.

R. Tarjan (1972) showed that SCC can be found efficiently by DFS. Below, the approach is slightly modified to operate on a given DFS-tree. Therefore, Tarjan's SCC algorithm is briefly outlined; for full details, look at R. Tarjan (1972): First, the vertices are enumerated in preorder. Then, a postorder traversal is used to compute, for each v, the *lowlink* $\ell(v)$, which is recursively defined as:

$$\begin{aligned}\ell(v) := \min\big(\{\ell(w) \mid (v,w) \text{ is a tree- or unfinished cross-edge}\} \cup \\ \{\rho(w) \mid (v,w) \text{ is a back edge}\} \cup \{\rho(v)\}\big)\end{aligned} \tag{3.8}$$

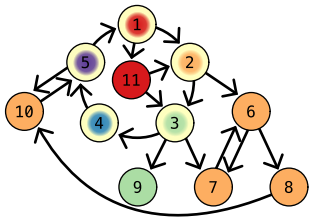
A cross edge is only included if it is "unfinished", i.e., if its endpoint w has not been reported as part of a previously-completed SCC. A vertex v is the root of a SCC if $\ell(v) = \rho(v)$. Tarjan's SCC algorithm now uses a stack to iterate over every vertex of the SCC S to mark them as finished. This cannot be done in the same way in a predefined DFS-tree.

The stack can be replaced, however, by an equally-efficient iterative method: Starting from v with $\ell(v) = \rho(v)$, simple traverse $T_c(v)$ starting at v; report all "unfinished" vertices as members of the SCC; and omit every subtree rooted in a "finished" vertex. To see that this is correct, note that $\ell(w) \neq \rho(w)$ for all $w \in S \setminus \{v\}$, and hence, w is "unfinished" when the postorder traversal encounters v. Lemma 36 (Page 87) implies that there is a path $\{v, w_1, \dots w_h = w\}$ from v to w in T_c, with $w_i \in S$ and thus also "unfinished". Thus, if u is "finished", so are all its descendants, and the subtree $T_c(u)$ does not need to be considered. The only difference from Tarjan's SCC algorithm tree traversal is to retrieve S, which considers every edge of T once and thus runs in a total time of $\mathcal{O}(|V(G)|)$. The discussion is summarized as:

Lemma 40. *The modified version of Tarjan's SCC algorithm correctly identifies all SCCs in* T *in* $\mathcal{O}(|E(G)| + |V(G)|)$ *time.*

Since the correct values of $\min_c(u)$ and $\max_c(u)$ are computed by postorder traversal of T_c, they are already available when the root v of a $\overset{C}{\rightsquigarrow}$-SCC is encountered. Thus, identification of the $\overset{C}{\rightsquigarrow}$-SCC and the computation of $\min_c(u)$ and $\max_c(u)$ can be combined in the same tree traversal. The same tree traversal also guarantees that for every cross edge (u, w), either (i) u and w are in the same $\overset{C}{\rightsquigarrow}$-SCC or (ii) the values of $\min_c(w)$ and $\max_c(w)$ are computed correctly.

Now, consider the vertex c_j along C, and suppose it is not encountered a single-vertex $\overset{C}{\rightsquigarrow}$-cover so far. Let T_j be the DFS-tree rooted in c_j that ignores all



v	c	$\min_c$	$\max_c$	π	ρ	l
11	1	2	3	1	2	2
1	1	2	3	2	1	1
10	2	5	5	2	5	5
8	2	5	5	3	4	4
6	2	5	5	4	2	2
7	2	5	5	1	3	2
2	2	5	5	5	1	1
9	3	–	–	1	2	2
3	3	5	5	2	1	1
4	4	5	5	1	1	1
5	5	5	5	1	1	1

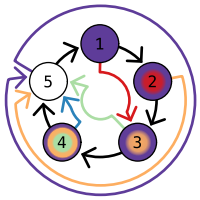


Figure 38: An example of the calculation of $\mathfrak{L}(C)$. On the left a graph is shown with a cycle $C = \circlearrowleft 1, 2, 3, 4, 5 \circlearrowright$ (yellow vertices). For every cycle vertex c in C, a modified DFS-tree T_c is constructed. The processing sequence is clock-wise starting with 1. Every vertex belongs only to the tree of the cycle vertex with the same color. Note that the trees of 4 (blue) and 5 (purple) only contain the cycle vertex. In the table values of the vertices are given that are calculated in the creation of $\mathfrak{L}(C)$. The values are: the vertex (v), the cycle vertex from which it is first reach (c), the calculated $\min_c(v)$, the calculated $\max_c(v)$, the position in the postorder of T_c (π), the position in the preorder of T_c (ρ), and the lowlink value for SCC detection ($\ell(v)$). The vertices are given in the order in which they are finished. If no SCC is present the order corresponds to π. However, if a SCC is detected, the vertex is finished after the root of the SCC (like vertex 7). The vertices have the background color of the tree they belong to (T_c), the color corresponds to c. On the right the resulting $\mathfrak{L}(C)$ is shown, with the colored edges (color of c) that correspond to the $\overset{C}{\rightsquigarrow}$-interval $C(c\colon \max_c(c))$. The vertices that are contained in the $\overset{C}{\rightsquigarrow}$-interval have the same color as the edge. Here, 5 ia a $\overset{C}{\rightsquigarrow}$-cut vertex and thus a quasi-legitimate root.

vertices already included in a previous DFS-tree. As for c_i, $\min_{c_j}(v)$ and $\max_{c_j}(v)$ with $v \in T_j$ can be compute along this tree. Then, $\min_{c_j}(v)$ either equals $\min_{c_j}(v)$ computed on T_j or $\min_{c_i}(u)$ for some u such that $(v, u) \in E(G)$, depending on which has the smaller value of $d_C(c_j, .)$, and $\max_{c_j}(v)$ either equals $\max_{c_j}(v)$ computed on T_j or $\max_{c_i}(j)$, depending on which has the larger value of $d_C(c_j, .)$. Note that $\min_{c_i}(v)$ and $\max_{c_i}(v)$ do not actually depend on i. In a practical implementation, it is simply stored in dependence of v. The index c_i only is used to keep track of the individual, disjoint DFS-trees T_i rooted in c_i in the arguments.

After processing all vertices of C,either a single-vertex $\overset{C}{\rightsquigarrow}$-cover of C is found, or for every $c_j \in C$, the largest $\overset{C}{\rightsquigarrow}$-covered interval $C(c_j\colon \max_{c_j}(c_j))$ is known. Thus, it is directly concluded:

$$\mathfrak{L}(C) := \{C(c_j\colon \max_{c_j}(c_j)) \mid c_j \in C\} \tag{3.9}$$

In particular, it is shown that for each C, $\mathfrak{L}(C)$ or a single-vertex cover can be constructed in linear time. An example of such a calculation is given in Figure 38.

To detect a quasi-legitimate root, it is necessary to first decide whether C has a total $\overset{C}{\rightsquigarrow}$-cover or a non-empty set $K(C)$ of $\overset{C}{\rightsquigarrow}$-cut vertices exists. To this end, a clean $\overset{C}{\rightsquigarrow}$-cover $\mathfrak{B}$ can be used efficiently. Recall that by Lemma 28 (Page 82),

every interval in a clean $\overset{C}{\rightsquigarrow}$-cover is extended by at least one other interval from the $\overset{C}{\rightsquigarrow}$-cover. Since a clean $\overset{C}{\rightsquigarrow}$-cover contains at most $|C|$ intervals, it is easy to check in linear time whether a $\overset{C}{\rightsquigarrow}$-cut vertex exists: starting from an arbitrary $C(v\!:\!u) \in \mathfrak{B}$, the upper bound of the $\overset{C}{\rightsquigarrow}$-covered part of C that starts at the successor of v is initialized by $x := d_C(v,u)$. For every $C(v'\!:\!u') \in \mathfrak{B}$ with $d_C(v,v') < x$, it is checked whether $d_C(v,v') > d_C(v,u')$, in which case a total cover is found, and otherwise, x is updated with $\max(x, d_C(v,u'))$. If no total cover is found when the intervals are exhausted, then x is a $\overset{C}{\rightsquigarrow}$-cut vertex (see the proof of Lemma 28 (Page 82)). With the $C(c_j\!:\max_v(u))$ stored, e.g., as array $a[v]$, a total cover or the $\overset{C}{\rightsquigarrow}$-cut vertex x is found in $\mathcal{O}(|C|)$ operations.

In practice, however, there is no clean $\overset{C}{\rightsquigarrow}$-cover computed yet. However, $\mathfrak{L}(C)$ can be computed in linear time. By Corollary 23 (Page 82), there is a clean $\overset{C}{\rightsquigarrow}$-cover $\mathfrak{B} \subset \mathfrak{L}(C)$. Thus, the same procedure is used. The redundant intervals in $\mathfrak{L}(C)$ are contained, by definition, within intervals belonging to $\mathfrak{B}$, and thus, they do not change the results provided the initial interval $C(v\!:\!u)$ is contained in the clean cover $\mathfrak{B}$. By Corollary 25 (Page 84), this is true for the longest interval $C(v\!:\!u) \in \mathfrak{L}(C)$. Since $\mathfrak{L}(C)$ contains at most $|C|$ intervals, the longest interval and a cut point or the validation of a total cover can be computed in $\mathcal{O}(|C|)$. If $\mathfrak{L}(C)$ is a total $\overset{C}{\rightsquigarrow}$-cover, the longest interval $C(v\!:\!u)$ is contained in a total clean cover, and thus, u is a legitimate root by Corollary 25 (Page 84). An example of the calculation of a total cover or cut vertex is shown in Figure 39. Thus, a quasi-legitimate root u can be retrieved in $\mathcal{O}(|C|)$ time. The entire procedure is summarized in Algorithm 3.

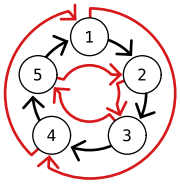
Lemma 41. *Given a cycle C in the digraph G, Algorithm 3 identifies a quasi-legitimate root in C in linear time w.r.t. the size of $G[[c]_{\rightsquigarrow}]$ for a $c \in C$, the induced subgraph of G reachable from C.*

Proof. The correctness of the algorithm follows from the discussion in the previous paragraphs. The construction of DFS-trees T_j together is linear in the size of $G[[c]_{\rightsquigarrow}]$ for a $c \in C$ since each edge in $G[[c]_{\rightsquigarrow}]$ is considered once. The recursive computation along each T_j is also linear. Since the T_j are disjoint, the total effort is still linear. □

Finally, note that by construction, no vertex in $G[[c]_{\rightsquigarrow}]$ for any $c \in C$ reaches any cycle C' disjoint from $G[[c]_{\rightsquigarrow}]$. Hence, when processing the next cycle C', the vertices (and edges) already visited in the context of processing C are irrelevant, and thus, $G[[c]_{\rightsquigarrow}]$ can be disregarded. In other words, the DFS for the next cycle can be performed in the same digraph G, with all previously processed induced subgraphs marked as finished. This ensures an overall linear running time for the identification of starting points for all cycles C_i as in Lemma 38 (Page 88).

Algorithm 3: Computing a quasi-legitimate root. The `get_root` algorithm. In the first part it computes $\mathfrak{L}(C)$ or a single-vertex $\overset{C}{\rightsquigarrow}$-cover. Then is $\mathfrak{L}(C)$ used to determine that $\overset{C}{\rightsquigarrow}$-cut vertex or that a total $\overset{C}{\rightsquigarrow}$-cover exists. If a total $\overset{C}{\rightsquigarrow}$-cover (including single-vertex $\overset{C}{\rightsquigarrow}$-cover) exists a legitimate root is returned otherwise a $\overset{C}{\rightsquigarrow}$-cut vertex as a quasi-legitimate root is returned.

Require: digraph G and cycle C
  for $c \in C$ **do**
    create DFS-tree T_c with root c by ignoring finished and cycle vertices with preorder ρ.
    while v traverses T_c in postorder **do**
      $\ell(v) \leftarrow \rho(v)$
      for $(v, u) \in E(G)$ **do**
        if $u \in C$ **then**
          Update $\min_c(v)$ with u
          Update $\max_c(v)$ with u
        else if (v, u) is a back edge **then**
          Update $\ell(v)$ with $\rho(u)$
        else
          if $d_C(c, \min_c(v)) > d_C(c, \max_c(v))$ **then**
            return legitimate root $\min_c(v)$
          Update $\min_c(v)$ with $\min_c(u)$
          Update $\max_c(v)$ with $\max_c(u)$
          if u is unfinished **then**
            Update $\ell(v)$ with $\ell(u)$
      if $\ell(v) = \rho(v)$ **then**
        for u in $\overset{C}{\rightsquigarrow}$-SCC with root v **do**
          $\min_c(u) \leftarrow \min_c(v)$
          $\max_c(u) \leftarrow \max_c(v)$
          Set u as finished
  Set $u \in C$ such that $d_C(c, \max_c(c)) \leq d_C(c, \max_u(u))$ for every $c \in C$
  $x = d_C(c, \max_u(u))$
  for $c \in C$ in cycle order starting from the successor of u **do**
    if $d_C(u, c) = x$ **then**
      return quasi-legitimate root c
    if $d_C(u, c) > d_C(u, \max_c(c))$ **then**
      return legitimate root $\max_u(u)$
    $x = \max(x, d_C(u, \max_c(c)))$



c	$d_C(c, \max_c(c))$
1	3
2	1
3	2
4	2
5	2

c	$d_C(1,c)$	$d_C(1, \max_c(c))$	x
2	1	2	3
3	2	4	4
4	3	5	5
5	4	1	

Figure 39: An example of the calculation of a total $\overset{C}{\rightsquigarrow}$-cover or $\overset{C}{\rightsquigarrow}$-cut vertex. On the left a graph is shown with the cycle $C = \circlearrowleft 1, 2, 3, 4, 5 \circlearrowright$. The red edges connect $c \in C$ with $\max_c(c)$ and thus represent the $\overset{C}{\rightsquigarrow}$-intervals of $\mathfrak{L}(C)$. The first step is to determine the interval with the maximal value of $d_C(c, \max_c(c))$. The values for every c are shown in the table in the center. The maximum is the interval $C(1{:}4)$. The next step processes the cycle vertices c clock-wise starting from the successor of 1, which is 2. The processing is shown in the right table. For every vertex it is first checked if the distance $d_C(1, c)$ is smaller than x, otherwise a cut vertex is found. Secondly, it is checked if $d_C(1, c) < d_C(1, \max_c(c))$, otherwise a total cover is found. After the checks, x is updated with the maximum of x and $d_C(1, \max_c(c))$. This makes it possible to check every already covered vertex. When $c = 5$ a total cover is found, and thus $\max_1(1) = 4$ is reported as quasi-legitimate root. Note that the $\overset{C}{\rightsquigarrow}$-interval $C(2{:}3)$ is not part of a clean $\overset{C}{\rightsquigarrow}$-cover but have no influence on the calculation.

3.9 Linear Superbubble Detection

With this, a simple linear time and space superbubble detection algorithm is presented.

Theorem 5. *Algorithm 4 correctly identifies the superbubbles of a digraph G in linear time.*

Proof. Theorem 4 (Page 87) ensures that for every digraph G, there is a set R of quasi-legitimate roots such that, given R, the algorithm *Superbubble#* identifies all superbubbles of G in linear time. Every vertex in $V(G)$ is reachable from a source or a cycle in G. By Lemma 25 (Page 78), all sources are legitimate roots. Thus, all superbubbles that overlap with the vertex set $S = \bigcup_{s \text{ is a source}} [s]_{\rightsquigarrow}$ can be detected. Lemma 38 (Page 88) shows that a set of cycles can be constructed in linear time from which all vertices in $V(G) \setminus S$ can be reached by DFS. Algorithm 3 identifies a quasi-legitimate root in a cycle (Lemma 41). As discussed in the text following Lemma 41, the effort for this step is again linear in size of G. Algorithm 4 therefore correctly identifies the superbubbles of a digraph G and does so in $\mathcal{O}(|V(G)| + |E(G)|)$ time. □

The constant factor is small for Algorithm 4. In the worst case every vertex is contained in three DFSs and is also visited by the traversal on the corresponding DFS-trees. The cycle detection traverses a vertex at most two times, for the root

Algorithm 4: Identification of all superbubbles in an arbitrary digraph G. The algorithm uses `get_root` (Algorithm 3) to compute a quasi-legitimate root for any used cycle. Furthermore, it utilizes *`Superbubble#`* to identify superbubbles. Note that there is no difference if *`Superbubble#`* is applied to F or to the tress of F independent of each other.

Require: Digraph G
  $R \leftarrow$ all sources in G
  generate a random DFS-forest $\hat{F}$
  find set W of $\prec$-maximal vertices with a back edge in $\hat{F}$
  generate set $\mathfrak{C}$ of cycles from W with $\hat{F}$
  for all cycles $C_k \in \mathfrak{C}$ **do**
    run `get_root`(C_k, G) to identify quasi-legitimate root r_k
    add r_k to R
  generate DFS-forest F with root set R
  run *`Superbubble#`* on F

generation in a cycle again two times, and for *`Superbubble#`* only one time. Thus, there is not much space for optimization left.

CHAPTER 4

Supergenome

Contents

This chapter is based on Gärtner, Höner zu Siederdissen, et al. (2018). First a motivation is given to solve the supergenome problem. After this an in-depth analysis of the concept of the supergenome and its relationship to genome-wide multiple sequence alignments (gMSAs) is shown. The next section then combinatorial optimization problems, from Chapter 2, are reviewed that are related closely to the "supergenome sorting problem", and argue that the most appropriate modeling leads to a special type of the betweenness ordering problem. Furthermore, a novel heuristic solution is introduced, that is geared towards very large input alignments and proceeds by step-wise simplification of the supergenome multigraph.

4.1 Motivation

The dramatic decrease of sequencing costs has enabled an ever-accelerating flood of genomic and transcriptomic data (1000 Genomes Project Consortium, 2015) that in turn have lead to the development of a variety of methods for data analysis. Despite recent efforts to study transcriptome evolution at large scales (Hezroni et al., 2015; S. Lin et al., 2014; Necsulea and Kaessmann, 2014; Neme and Tautz, 2016; Washietl, Kellis, and Garber, 2014) the capability to analyze and integrate -omics data in large-scale phylogenetic comparisons lags far behind data generation. One key aspect of this shortcoming is the current lack of powerful tools for visualizing comparative -omics data. Available tools such as `hal2AssemblyHub` (Nguyen, Hickey, Raney, et al., 2014b) or `progressiveMauve` (Darling, Mau, and Perna, 2010) have been designed with closely related species or strains in mind. The visualizations become difficult to interpret for multiple species and larger evolutionary distances, where homologous genomic regions may differ substantially in their lengths, an issue that becomes more pressing the larger the regions of interest become. A common coordinate system for multiple genomes is not only a convenience for graphical representations of -omics data. It would also greatly facilitate the systematic analysis of all genomic features that are not sufficiently local, to be completely contained within individual multiple sequence alignment (MSA)-blocks of a gMSA.

Still, gMSAs are the natural starting point. A common feature of gMSAs is that they are composed of a large number of MSA-blocks. At least in the case of MSAs of higher animals and plants the individual MSA-blocks are typically (much) smaller than individual genes. As a consequence, they are not ready-to-use for detailed comparative studies, e.g. of transcriptome or epigenome (Xiao, Cao, and Zhong, 2014) structure. In the gMSA-based splice site maps of Nitsche et al. (2015), for example, it is easy to follow the evolution of individual splice junctions as they are localized within a MSA-block. At the same time it is difficult to collate the global differences of extended transcripts, which may span hundreds of MSA-blocks. Further it is hard to relate changes in transcript structure with genomic rearrangements, insertions of repetitive elements or deletion of chunks of sequence.

This chapter concerned with the coordinatization of supergenomes, i.e., the question how MSA-blocks of a gMSA can be ordered in a way that facilitates comparative studies of genome annotation data. In contrast to previous work on

supergenomes this work is particularly interested in large animal and plant genomes and in wide phylogenetic ranges. Therefore it is assumed that the alignment consist of short MSA-blocks and abundant genome rearrangements, leaving only short sequences of MSA-blocks that are perfectly syntenic between all genomes involved.

4.2 Genome-wide multiple sequence alignments

In this work an assembly is simply a set of sequences representing chromosomes, scaffolds, reftigs, contigs, etc. In the following, *contig* is used to refer to any of such genomic sequences. For each of these constituent sequences, the usual coordinate system defining sequence positions is used. Since deoxyribonucleic acid (DNA) is double stranded, a piece of genomic sequence either is contained directly ($\sigma = +1$) in the assembly or is represented by its reverse complement ($\sigma = -1$). The quintuple $(\mathcal{G}, c, i, j, \sigma)$ identifies the *sequence interval* from positions i to j on contig c of genome assembly $\mathcal{G}$ with reading direction σ, where w.l.o.g., $i \leq j$.

MSA-block

Most comparative methods require gMSAs as input. A gMSA $\mathfrak{A}$ is composed of *MSA-blocks*, each of which consists of a MSA of sequence intervals. For the purposes of this thesis it its sufficient to characterize a MSA-block by the coordinates of its constituent sequence intervals. That is, a MSA-block $B \in \mathfrak{A}$ has the form $B = \{(\mathcal{G}_u, c_u, i_u, j_u, \sigma_u) \mid u \in \text{rows of } B\}$ where the index u runs over all rows of the MSA-block. It is convenient to allow MSA-blocks also to consist of a single interval only, thus referring to a piece of sequence that has not been aligned. Note that at this stage, it is not assumed that a MSA-block contains only one interval from each assembly.

The projection $\tau_{\mathcal{G}}(B)$ extracts from a MSA-block B the union of its constituent sequence intervals belonging to assembly $\mathcal{G}$. If the assembly $\mathcal{G}$ is not represented in the MSA-block B, is $\tau_{\mathcal{G}}(B) = \emptyset$. The projection operation collapses pairs of overlapping sequence intervals $\alpha = (\mathcal{G}, c, i, j, \sigma)$ and $\beta = (\mathcal{G}, c, i', j', \sigma')$ with $i \leq i' \leq j \leq j'$ into a single interval: $\alpha \cup \beta = (\mathcal{G}, c, i, j', +1)$ without regard of the orientation, which is set to $+1$ and has *no impact* on the algorithms of this chapter.

The projection $\tau_{\mathcal{G}}(\mathfrak{A})$ of $\mathfrak{A}$ onto one of its constituent assemblies $\mathcal{G}$ is the union of the sequence intervals from $\mathcal{G}$ that are contained in its MSA-blocks, i.e., $\tau_{\mathcal{G}}(\mathfrak{A}) = \bigcup_{B \in \mathfrak{A}} \tau_{\mathcal{G}}(B)$.

genome-wide multiple sequence alignments

Definition 35. *Let $\mathfrak{A}$ be a gMSA.*

(i) $\mathfrak{A}$ *is* complete *if $\tau_{\mathcal{G}}(\mathfrak{A}) = \mathcal{G}$, i.e., if each position in each assembly is represented in at least one MSA-block.*

(ii) $\mathfrak{A}$ *is* irredundant *if $\tau_{\mathcal{G}}(B') \cap \tau_{\mathcal{G}}(B'') = \emptyset$ for any two distinct MSA-blocks B' and B'' of $\mathfrak{A}$, i.e., if every sequence interval from assembly $\mathcal{G}$ is contained in at most one MSA-block.*

(iii) $\mathfrak{A}$ *is* injective *if no MSA-block comprises more than one interval from each of its constituent assemblies.*

Clearly, every given gMSA can be completed by simply adding all unaligned sequence intervals as additional MSA-blocks.

Just like a contig c in a genome assembly $\mathcal{G}$, each MSA-block $B \in \mathfrak{A}$ has an internal coordinate system defined by its columns. The used notation is $B[k]$ for column k in MSA-block B, $\mathrm{columns}(B)$ for a set of all columns of B, and $|\mathrm{columns}(B)|$ for the number of columns in B. If $\mathfrak{A}$ is irredundant, then there are functions $f_{\mathcal{G},c}$ that map position i within $(\mathcal{G}, c)$ to a corresponding gMSA coordinate $B[k]$. If $\mathfrak{A}$ is complete, the individual $f_{\mathcal{G},c}$ can be combined to a single function $f : (\mathcal{G}, c, i) \mapsto B[k]$. Completeness implies that every position $(\mathcal{G}, c, i)$ is represented in the gMSA, and irredundancy guarantees that the relation between assembly and alignment coordinates is a function by ensuring that $(\mathcal{G}, c, i)$ corresponds to at most one alignment column. The following definition is therefore equivalent to the notion of a supergenome introduced in Herbig et al. (2012).

supergenome **Definition 36.** *A gMSA $\mathfrak{A}$ is a* supergenome *if and only if it is complete, irredundant, and injective.*

The most commonly used gMSAs cannot be completed to supergenomes. The MSAs produced by the `multiz` pipeline are usually not irredundant: different intervals of the "reference sequence" may be aligned to the same interval of another assembly. While `multiz` (Blanchette et al., 2004) alignments are injective this is in general not the case with the `EPO` (Paten, Herrero, et al., 2008) alignments. In these, multiple paralogous sequences from the same genome may appear in one MSA-block.

Now consider a gMSA $\mathfrak{A}$ and an arbitrary total order $\leq$ of the MSA-blocks of $\mathfrak{A}$. Then there is an unique function ϕ that maps the column $B[k]$ injectively to the interval $[1\!:\!n]$, where $n = \sum_{B \in \mathfrak{A}} |\mathrm{columns}(B)|$ is the total number of columns in $\mathfrak{A}$ such that $\phi(B[k]) < \phi(B'[k'])$ whenever $B < B'$ or $B = B' \wedge k < k'$. If $\mathfrak{A}$ is a supergenome, then the composition $\phi(f)$ is clearly an injective function from a genome assembly $\mathcal{G}$ to $[1\!:\!n]$. Then is $\phi(f(\mathcal{G}, c, i))$ the *coordinate* of position i of contig c of assembly $\mathcal{G}$ in the ordered supergenome $(\mathfrak{A}, \leq)$.

As pointed out in Herbig et al. (2012), the existence of a coordinate system for the supergenome $\mathfrak{A}$ is independent of the MSA-block order $\leq$. However, the order $\leq$ is crucial for the practical use of the coordinate system. An example of the composition $\phi(f)$ is shown in Figure 40.

4.3 gMSA as Graph

The natural starting point for considering adjacency and betweenness of MSA-blocks are their constituent intervals $(\mathcal{G}, c, i, j, \sigma)$ on a fixed assembly $\mathcal{G}$ and contig c. Intervals have a natural partial order defined by $(\mathcal{G}, c, i, j, \sigma) < (\mathcal{G}, c, k, l, \sigma)$ whenever $i < k$ and $j < l$. Two intervals are incomparable in this *interval order* if and only if one is contained in the other. Note that the interval order allows comparable intervals to overlap. Further intervals are incomparable if they belong to different contigs and/or assemblies.

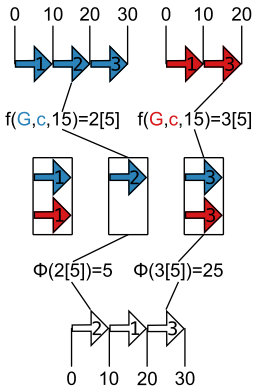


Figure 40: Example of the composition $\phi(f)$. On the top, two contigs are shown. The contigs compose of three different MSA-blocks. Which sequence interval (arrow) belongs to which MSA-block is indicated by the number. For simplicity is assumed that every MSA-block has length ten and contains no gaps. The function f maps every position in a contig on a position of a MSA-block. In the middle, the MSA-blocks are shown. The function ϕ then maps every position in a MSA-block on a position of the ordered supergenome(shown at the bottom). ϕ uses an order of the MSA-blocks. However, the existence of ϕ is independent of the meaningfulness of the order. The composition $\phi(f)$ maps every position of a contig to a position of the ordered supergenome. For example, position 15 of the blue contig maps to position 5 of the ordered supergenome.

Given three intervals $\alpha = (\mathcal{G}, c, i', j', \sigma')$, $\beta = (\mathcal{G}, c, i'', j'', \sigma'')$, and $\gamma = (\mathcal{G}, c, i, j, \sigma)$ (on the same genome assembly and contig), then γ is *between* the two distinct intervals α and β if $\alpha < \gamma < \beta$ or $\beta < \gamma < \alpha$.

Given a collection of intervals on the same assembly $\mathcal{G}$ and contig c, then $\alpha = (\mathcal{G}, c, i', j', \sigma')$ and $\beta = (\mathcal{G}, c, i'', j'', \sigma'')$ are *adjacent* if there is no interval γ between α and β. Then α is a *predecessor* of β if α and β are adjacent and $\alpha < \beta$. Analogously, α is a *successor* of β if α and β are adjacent and $\beta < \alpha$. This can be used on properties of a supergenome.

Lemma 42. *Let $\mathfrak{A}$ be a supergenome and consider the the collection $\{\tau_{\mathcal{G}}(B) \mid B \in \mathfrak{A}\}$ of intervals on a given $\mathcal{G}$. Then (i) no two intervals overlap, (ii) the interval order $<$ is a total order on every contig c, (iii) every interval has at most one predecessor and one successor, and hence is adjacent to at most two intervals, and (iv) if γ is adjacent to both α and β, then γ is between α and β.*

Proof. Property (i) follows directly from the supergenome conditions. As a consequence, any two intervals on a fixed contig c are comparable, i.e., the restriction of interval order $<$ to c is a total order, hence (ii) follows. Since only intervals on the same contig can be adjacent, (iii) is an immediate consequence of (ii) and the fact that the number intervals is finite. Property (iv) is now a trivial consequence of the fact that by (iii) α and β must be the predecessor and successor of γ. □

A key construction in this contribution is the notion of betweenness relations for MSA-blocks.

betweenness (alignment)

Definition 37. *Given three MSA-blocks $A, B, C \in \mathfrak{A}$, then C* is between *A and B with respect to $\mathcal{G}$ if $\tau_{\mathcal{G}}(C)$ is between $\tau_{\mathcal{G}}(A)$ and $\tau_{\mathcal{G}}(B)$. The ternary relation $\mathscr{C}(\mathfrak{A})$ is defined by $[A \lesssim C \lesssim B] \in \mathscr{C}(\mathfrak{A})$ whenever C is between A and B for some assembly $\mathcal{G}$ in $\mathfrak{A}$.*

Note that contradicting betweenness relations resulting from different genome assemblies $\mathcal{G}$ are expected, i.e., the relation $\mathscr{C}(\mathfrak{A})$ in general do not satisfy the properties of a betweenness relation. This issue is discussed below.

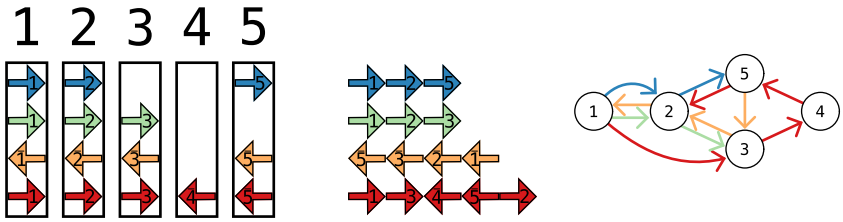


Figure 41: An example of the projection of an artificial gMSA to a supergenome graph. On the left a gMSA $\mathfrak{A}$ is shown, which comprises five MSA-blocks $1, \ldots, 5$, each consisting of up to four intervals from the four genome assemblies (distinguished by colors). Note that the intervals that are shown as arrows represents respective the position in the genome assemblies (not shown). Thus the order of the intervals in the genome assembly can be reconstructed. This reconstruction is shown in the center. The interval order implies that a path with this order exists in the supergenome graph. Alternatively, more formally it implies a separate predecessor relation among MSA-blocks, with a colored arrow from 1 to 2 in the supergenome graph implying that 1 is a predecessor of 2 w.r.t. the assembly indicated by the color. $\Gamma(\mathfrak{A})$ has all MSA-blocks of $\mathfrak{A}$ as its vertices. The predecessor relations among the MSA-blocks define the colored, directed edges. The resulting supergenome graph is shown right.

Definition 38. *Two MSA-blocks $A, B \in \mathfrak{A}$ are* adjacent *if there is an assembly $\mathcal{G}$ such that $\tau_{\mathcal{G}}(A)$ and $\tau_{\mathcal{G}}(B)$ are adjacent w.r.t. $\{\tau_{\mathcal{G}}(C) \mid C \in \mathfrak{A}\}$.*

It is useful to regard $\mathfrak{A}$ with its adjacency relation as a graph. In order to keep track of the individual contigs, an edge-colored multigraph is used, with $\mathcal{G}$ serving as edge color.

supergenome graph

Definition 39. *The* supergenome graph $\Gamma(\mathfrak{A})$ *of a gMSA $\mathfrak{A}$ is the directed, edge-colored multigraph whose vertices are the MSA-blocks of $\mathfrak{A}$ and whose directed edges (A, B) connect a MSA-block A to a MSA-block B with color $\mathcal{G}$ whenever the sequence interval $\alpha \in A$ is a predecessor of the sequence interval $\beta \in B$ in assembly $\mathcal{G}$.*

An example of the projection $\Gamma(\mathfrak{A})$ of a gMSA $\mathfrak{A}$ is illustrated in Figure 41. The projection of $\Gamma(\mathfrak{A})$ to a constituent assembly $\mathcal{G}$ is a (not necessarily induced) subgraph. As an immediate consequence of Lemma 42, each projection is a disjoint union of directed paths, each of which represents a contig. Conversely, every colored directed multigraph whose restriction to a single color is a set of vertex-disjoint directed paths is a supergenome graph. It therefore makes sense to talk about a supergenome graph Γ without explicit reference to an underlying alignment $\mathfrak{A}$. Note that Γ is only a colored version of an A-Bruijn graph (Subsection 2.2.9).

The structure of the *supergenome graph* strongly depends on the evolutionary history of the genomes that it represents. In the absence of genome rearrangements (i.e., if the only genetic changes are substitutions, insertions (including duplications), and deletions) then all genomes remain colinear with their common ancestor. In

other words, a single, canonical global alignment (Giegerich, 2000) describes a common coordinate system that is unique up to the (arbitrary) order of contigs and each trace of insertions and deletions (Sankoff, 1983). In terms of the MSA-block adjacency relation, each MSA-block has at most two adjacent neighbors in this scenario.

Genome rearrangements are by no means infrequent events (Belda, Moya, and Silva, 2005; Drillon and Fischer, 2011; Fischer et al., 2006; Friedberg, Darling, and Yancopoulos, 2008), and thus cannot be neglected. Every breakpoint introduced by a genome rearrangement operation, be it a local reversal or a cut-and-join type dislocation, introduces an ambiguous adjacency, i.e., a MSA-block that has two or more predecessors or successors. The task of identifying an appropriate ordering of the MSA-blocks therefore is a non-trivial one for realistic data, even in the absence of alignment errors.

4.4 Modeling the "Supergenome Sorting Problem"

Informally, consider the supergenome sorting problem (SSP) as the task of finding an order $\leq$ (or, equivalently, a permutation) of the MSA-blocks of $\mathfrak{A}$ *such that the orders of the constituent assemblies are preserved as much as possible*. Somewhat more precisely, to find an order $\leq$ on the vertex set of the supergenome graph $\Gamma(\mathfrak{A})$ that as many of its directed edges as possible are "consistent" with the order $\leq$. It is not clear from the outset, however, how "consistency" should be defined for the application. A large number of related models have been proposed and analyzed in the literature that make this condition precise in different ways, leading to different combinatorial optimization problems. A brief review of some paradigmatic approaches can be found at Section 2.3. Here is discussed how fitting they are for the SSP.

4.4.1 Hamiltonian Paths

A plausible attempt is to view the SSP as a variant of the Hamiltonian path problem on the *supergenome graph* Γ. A Hamiltonian path defines a total order of the vertices and therefore a solution to the SSP. This is idea is similar to the use of Hamiltonian graphs for genome assembly from read overlap graphs (El-Metwally et al., 2013). There are several quite obvious difficulties, however. First, it is not sufficient to consider only paths that are entirely confined to pass through the adjacencies. The simplest counterexample consists of only four MSA-blocks B_1, B_2, B_3 ,B_4 and three assemblies $\mathcal{G}_1$, $\mathcal{G}_2$, $\mathcal{G}_3$. Given eight sequence intervals $\beta_{k,l} = (\mathcal{G}_l, c_{k,l}, i_{k,l}, j_{k,l}, +1) \in B_k$ the following alignment is constructed:

	B_1	B_2	B_3	B_4
$g_1=$	$\beta_{1,1}$			$\beta_{4,1}$
$g_2=$	$\beta_{1,2}$	$\beta_{2,2}$		$\beta_{4,2}$
$g_3=$	$\beta_{1,3}$		$\beta_{3,3}$	$\beta_{4,3}$

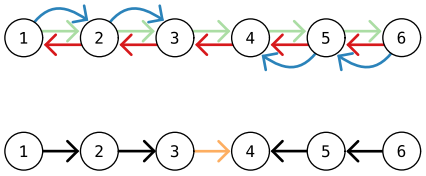


Figure 42: MFAS creating a suboptimal solution to the SSP. Due to the arbitrariness of the orientation of the edges, the best solution of the SSP may contain cycles, which by definition is excluded in MFAS. Top: *supergenome graph* representation of an artificial alignment. Bottom: Simplified solution of the (uniformly weighted) MFAS. To turn the graph into a DAG must at least six edges deleted Two such solutions exist, differing only by the orientation of the orange arrow. The corresponding topological sorting breaks the genome into two distinct colinear pieces with opposite orientation. There is, however, a consistent order of the entire graph – the linear left-to-right or right-to-left order is consistent.

This situation arises in practice e.g. if B_2 and B_3 are two independent, unrelated inserts between B_1 and B_4. The MSA-block adjacency graph is the graph

$$B_2 - B_1 - B_4 - B_3,$$

which violates the desired betweenness relation $[\beta_{1,3} \lesseqgtr \beta_{3,3} \lesseqgtr \beta_{4,3}]$.

In this case there are only two biologically correct solutions: $B_1 < B_2 < B_3 < B_4$ (or the inverse order) and $B_1 < B_3 < B_2 < B_4$ (or its inverse). In either case, the solution contains two consecutive MSA-blocks (B_2 and B_3) that are not adjacent in the MSA-block graph. This example also serves to demonstrate that the MSA-block graph alone does not contain the complete information on the supergenome. It appears that in addition the *betweenness* relation among the MSA-blocks (i.e., that both B_2 and B_3 are between B_1 and B_4) is needed.

4.4.2 Feedback Arc Sets and Topological Sorting

An other possibility to determine a well-defined order of the vertices of the supergenome graph Γ is to first solve minimum feedback arc set problem (MFAS) and then to compute a topological sorting of this subgraph.

The key problem of modeling the SSP in terms of MFAS is highlighted in Figure 42. It shows that even if undirected adjacencies would allow for a perfect solution, it may not be uncovered directly by the MFAS approach.

4.4.3 Simultaneous Consecutive Ones and Matrix Banding

The simultaneous consecutive ones property (C1S) could also be applied to Γ to create an order. However, this have some problems. First, recover that this does not imply Γ is an union of disjoint paths, i.e., that Γ is a valid supergenome graph. Secondly, the solution is very inaccurate locally. Which means that even if a global ordering is for the interesting detail level no satisfying solution is found.

4.4.4 Bidirected Graphs

The sets of genomes considered by Nguyen, Hickey, Zerbino, et al. (2015) are particularly suitable for this kind of calculations. The genomes that are considered for pangenome construction typically are related closely , or even of the same species. Thus one can expect many paths with high weights of the conserved consensus (Haussler et al., 2018). This approach also fits well to the analysis of genomic regions that are under constraint to maintain syntenic order for functional reasons, such as the MHC locus used as an example in Nguyen, Hickey, Zerbino, et al. (2015). In distantly related genomes, however, synteny tends not to be well preserved. In addition, this work is interested in particular in data sets that contain genomes in preliminary draft forms, i.e., linkage information that is at least partially limited to short contigs or scaffolds. As a consequence there is less confidence in linkage and orientation information than one can expect in a typical pangenome scenario.

4.4.5 Sequence Graphs

The sequence graphs after Haussler et al. (2018) have two main issues. First the information lost that is given if to adjacent intervals change the orientation. It is likely to become an issue, for large phylogenetic distances with frequent genome rearrangements. The second issue is that a common backbone order is assumed. The presence of such a backbone is violated substantially for phylogenetically diverse data.

4.5 Betweenness Problems

In this work the SSP is interpreted as a betweenness (ordering) problem rather than a vertex ordering problem on a directed graph. Instead of (oriented) adjacencies, which are defined on pairs of MSA-blocks, one considers the relative order of three MSA-blocks. Recall the betweenness problem:
Betweenness Problem (Chor and Sudan, 1998; Opatrny, 1979): *Given a finite set X and a collection $\mathscr{C}(X)$ of betweenness triples from X, is there a total order on X such that $\forall[i\lesssim j\lesssim k] \in \mathscr{C}(X)$ either $i < j < k$ or $i > j > k$?*

The *Betweenness Problem* can be adapted to model the SSP by means of a suitable cost function b designed to penalize violations of the betweenness relation. Consider a total order γ used to coordinatize the supergenome. Therefore, γ is a bijective function that represents this order.

For $i < j < k$ set $b_{\gamma,\mathcal{G}}(i,j,k) = 1$ if the projections of the three MSA-blocks $\gamma(i)$, $\gamma(j)$, and $\gamma(k)$ exist and violate the betweenness relation for a given assembly $\mathcal{G}$, i.e., if $\tau_{\mathcal{G}}(\gamma(i))$, $\tau_{\mathcal{G}}(\gamma(j))$ and $\tau_{\mathcal{G}}(\gamma(k))$ are located on the same contig and $\tau_{\mathcal{G}}(\gamma(j))$ is not located between $\tau_{\mathcal{G}}(\gamma(i))$ and $\tau_{\mathcal{G}}(\gamma(k))$. Otherwise set $b_{\gamma,\mathcal{G}}(i,j,k) = 0$. A natural cost function is now the total number of betweenness violations **betweenness violation**

$$b(\gamma) := \sum_{\mathcal{G}\in G(\mathfrak{A})} \sum_{i<j<k} b_{\gamma,\mathcal{G}}(i,j,k)\,, \tag{4.1}$$

where $G(\mathfrak{A})$ is the set of genome assemblies that are contained in $\mathfrak{A}$. If genome evolution preserve gene order, i.e., only local duplications and deletions are allowed, the betweenness relation of the ancestral state would be preserved, guaranteeing a perfect solution γ with $b(\gamma) = 0$.

Since this decision problem is *NP*-complete (Chor and Sudan, 1998; Opatrny, 1979), so is the problem of optimizing $b(\gamma)$ *NP*-hard. The cost function $b(\gamma)$ involves the sum over all triples of MSA-blocks and thus is fairly expensive to evaluate. It is interesting in practice, therefore, to consider a modified cost function that restricts the sum in Equation 4.1 to local information. This idea leads us to the rather natural extension of the *betweenness problem* to colored multigraphs.

betweenness (graph)

Definition 40. *Given a directed* colored multigraph Γ*, the triple* $[i \lesssim j \lesssim k]$ *is part of the collection* $\mathscr{C}(\Gamma)$*, iff there are edges* $(i, j) \in E(\Gamma)$ *and* $(j, k) \in E(\Gamma)$ *with color* $\mathcal{G}$*.*

Directed Colored Multigraph Betweenness Decision Problem: Given the directed colored multigraph Γ, is there a total order on $V(\Gamma)$ such that $\forall [i \lesssim j \lesssim k] \in \mathscr{C}(\Gamma)$ either $i < j < k$ or $i > j > k$?

The reformulation as an optimization problem that maximizes the number of edges is straightforward:

directed colored multigraph betweenness problem

Directed Colored Multigraph Betweenness Problem: Given a directed colored multigraph Γ, find a total order on V such that $E^* \subseteq E(\Gamma)$ is maximal under the condition that $\forall [i \lesssim j \lesssim k] \in \mathscr{C}(V(\Gamma), E^*)$ either $i < j < k$ or $i > j > k$.

This problem can be viewed as an analog of the *Minimum Feedback Arc Set* problem (Eades, X. Lin, and Smyth, 1993) for betweenness data. It has not be studied so far.

Lemma 43. *The (decision version of the) Directed Colored Multigraph Betweenness Problem is NP-complete.*

Proof. Every set $\mathscr{C}(\Gamma)$ of triples can be obtained from an edge-colored multigraph Γ (with vertices corresponding to MSA-blocks and colored edges corresponding to adjacencies deriving from a genome identified by the color). Thus, the total order on the vertices of Γ is a solution of the *Directed Colored Multigraph Betweenness Problem* if and only if the answer to the *NP*-complete *Betweenness Problem* is positive. □

In the example of Figure 42 the optimal solution of the *Directed Colored Multigraph Betweenness Problem* retains all adjacencies and creates an unique coordinatization (up to orientation) that leaves all MSA-blocks ordered as drawn.

Note that here a direct graph is used as the backbone. It could also be used an undirect graph. In fact, the irrelevance of direction is an important feature of betweenness. The undirected version (Colored Multigraph Betweenness Problem) is equivalently formulated. The difference is the definition of $\mathscr{C}(\Gamma)$ for undirected graphs that ignores the direction. Thus, even Lemma 43 can be applied with the same proof to the undirected version.

Even if the undirected solution is in some way more natural, the directed version is used because the supergenome graph is directed. However, after it is constructed from paths, the solution is independent of the version used.

4.5.1 Seriation

An alternative framework for solving the SSP by construction of a preferred ordering is seriation. The Robinson seriation problem (Robinson, 1951) starts from a dissimilarity measure $d : X \times X \to \mathbb{R}$, and seeks a total order γ on X that satisfies the inequality

Robinsonian dissimilarities

$$\max\{d(\gamma(i),\gamma(j)), d(\gamma(j),\gamma(k))\} \leq d(\gamma(i),\gamma(k))\,. \tag{4.2}$$

A dissimilarity d for which an ordering γ exists that satisfies Equation 4.2 for all $\gamma(i) < \gamma(j) < \gamma(k)$ is called *Robinsonian*. It is worth noting that Robinsonian dissimilarities are intimately related with pyramidal clustering problems (P. Bertrand, 2008; P. Bertrand and Diatta, 2017).

The seriation problem (Liiv, 2010; Robinson, 1951) consists of finding a total order for which the given pairwise distances violates the Robinson conditions as little as possible. To link this seriation problem with the *Directed Colored Multigraph Betweenness Problem* or *Betweenness Problem* a collections $\mathscr{C}(X)$ of triples is considered such that

$$[i \lesseqgtr j \lesseqgtr k] \in \mathscr{C}(X) \longrightarrow \max\{d(\gamma(i),\gamma(j)), d(\gamma(j),\gamma(k))\} < d(\gamma(i),\gamma(k)) \tag{4.3}$$

Clearly, if the dissimilarity is Robinsonian, then γ defines a total order on X that solves the *Betweenness Problem* for $(X, \mathscr{C}(X))$.

The relevant optimization task in this context is to minimize the number of ordered triples that violate Equation 4.2. A variety of heuristics for this problem have been developed, see e.g. Hahsler, Hornik, and Buchta (2008). It is important to note, however, that in the setting the distance between MSA-blocks is not defined directly. In order to obtain a seriation problem that approximates the SSP a heuristic is needed that summarizes the distances between two MSA-blocks in all genomes and reflects the betweenness relationships. For pangenome-like models, the cost function advocated in Nguyen, Hickey, Zerbino, et al. (2015) is a very plausible choice.

4.6 Graph Simplification

Each of the plausible models for the "Supergenome Sorting Problem" discussed in the previous sections leads to *NP*-hard computational problems. The size of typical genome-wide alignments by far exceeds the range where exact solutions can be hoped for, except possibly for the smallest and most benign examples such as the ones used as examples in Herbig et al. (2012). Therefore, only fast heuristics can be used. This section focus on the conceptual ideas behind the simplification steps. More detailed implementation details are given in Section 4.7.

Nevertheless it is possible to isolate certain sub-problems that can be solved exactly and independently of the remainder of the input graph. Since "linearized"

portions of the vertex set can be contracted to a single vertex set, this leads to a reduction of problem size.

Lemma 44. *If the supergenome graph Γ is a directed acyclic graph (DAG) then topological sorting of Γ solves the Directed Colored Multigraph Betweenness Problem.*

Proof. In this case betweenness is established exactly by the directed paths in the DAG. Hence any topological sorting preserves all betweenness triples of Γ and thus presents a perfect solution to the *Directed Colored Multigraph Betweenness Problem* as well. □

This simple observation suggests to identify subgraphs with DAG structure and to replace them with a representative for each replaced DAG. These can later be replaced by the solution that is created with topological sorting. Note that this does not necessarily preserve optimality. It is conceivable that a local DAG structure has to be broken up into two disjoint subsets that are integrated in larger surrounding structures in a way that requires reversal of the edge directions in one or even both parts. Nevertheless, if the local DAG structures are sufficiently isolated they are likely to be part of the optimal solution as an unit. The motif that describes such a local DAG structure is a superbubble. Thus, a superbubble simplifier is applied to the supergenome graph.

An other applied simplifer is a dead end simplifier. To recap a dead end is a source or sink vertice v in the supergenome graph with only a single neighbor u. These can be sorted together with their unique neighbor u. Γ is thus simplified by contracting v and u, i.e., placing the source v immediately before u and sink v immediately after u.

In some cases it is helpful to reverse the direction of the coordinate system of a single species. This is in particular the case if a single genome is reversed compared to all others. The inversion of an entire path does not change the solution of the *Directed Colored Multigraph Betweenness Problem* but can make it easier to apply some of the reduction heuristics discussed above. In particular, if the relative orientation of the coordinatizations could be fixed in an optimal manner, the betweenness problem reduces to a much easier topological sorting problem. Finding this optimum, however, is equivalent to the *Betweenness Problem*, which is a NP-hard. Hence, again a local heuristics is used.

mini-cycle **Definition 41.** *Let Γ be a supergenome graph. A pair of vertices $v, w \in V(\Gamma)$ such that there are edges (v, w) and (w, v) in $E(\Gamma)$ is a* mini-cycle.

Mini-cycles naturally are removed by removing one of the two edge directions between v and w. More precisely, the less supported direction of an edge is dropped. The estimate for support is evaluated in a region around a mini-cycle since adjacent mini-cycles may yield contradictory majority votes.

mini-cycle complex **Definition 42.** *Two mini-cycles are connected with each other if they share a vertex. A mini-cycle complex $\mathfrak{C}$ is a maximal connected set of mini-cycles.*

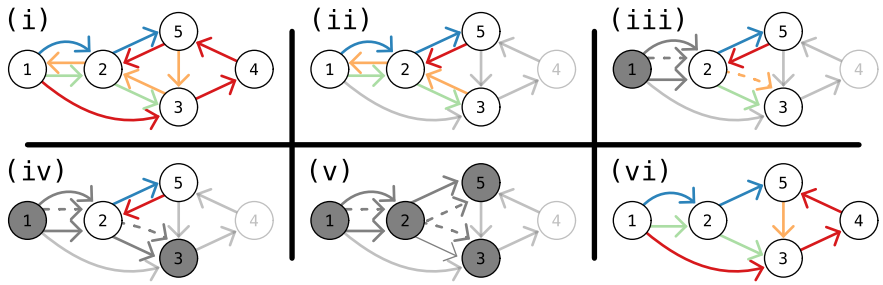


Figure 43: Step-wise resolution of a complex of mini-cycles. (i): Starting point. **(ii):** The mini-cycle complex is highlighted. The complex is created from the mini-cycles $\circlearrowleft 1, 2 \circlearrowright$, $\circlearrowleft 2, 3 \circlearrowright$, and $\circlearrowleft 2, 5 \circlearrowright$. Note that the edges $(1, 3)$ and $(5, 3)$ are not contained in the complex. The best supported directions are between $(1, 2)$. **(iii):** This direction between $(1, 2)$ is set. The orange edges are reversed (marked by dashed lines). The adjacency $\{1, 2\}$ is decided and is no longer considered (marked with dark grey). **(iv):** In this step the best supported direction is $(2, 3)$ and the graph is updated correspondingly. **(v):** Adjacency $\{2, 5\}$ is left. No direction is superior. Since vertex 2 is solved previously it is used. This leads to direction $(2, 5)$. **(vi):** Then the completed complex is decided and the edges that contradict with the decisions are removed. Note that the circle $\circlearrowleft 3, 4, 5, 3 \circlearrowright$ that is not part of the complex is not removed.

Lemma 45. *The mini-cycle complexes of a supergenome graph Γ form a unique partition of the set of all mini-cycles. Any two classes of this partition are vertex and edge disjoint.*

Proof. Consider the the graph H whose vertices are the mini-cycles, and there are edges between any two mini-cycles that share at least one vertex. Then every mini-cycle complex $\mathfrak{C}$ is a connected component of H. Since the connected components of a graph are uniquely defined, disjoint, and form a cover, they partition the vertex set of H. Every mini-cycle, furthermore, forms a connected subgraph of Γ by construction. Since any two mini-cycles that contain a common vertex belong to the same mini-cycle complex, two mini-cycle complexes cannot have a vertex in common. This implies that they are also edge disjoint. □

The mini-cycle complexes therefore can be resolved independently of each other. The target is to remove edges that create cycles in order to obtain a DAG that can then be subjected to topological sorting. However, this topological sorting is a solution of the *Directed Colored Multigraph Betweenness Problem* for a subgraph. This is still a hard problem, so that a heuristic approach is again utilized. This step only attempt to remove mini-cycles. Cycles that connect mini-cycle complexes with each other or with other vertices in the graph are untouched and have to be dealt with in a subsequent step.

The local sorting within a complex $\mathfrak{C}$ is achieved by considering adjacencies. To this end each adjacency is annotated with the number of edges and the ratio of the edges in the two directions. The best supported edges are identified as those with a

high multiplicity and a strong bias for one direction over the other. This choice of a direction is propagated. If a directed edge has more than one possible successor, first propagate along the one with the largest support for the proposed direction. The issue now is when exactly to stop propagating this information. Clearly, it is forbidden to orient an edge that would close a directed cycle. Any such edge is instead seeded with the reverse directional information.

As result of this procedure it is possible that parts of a directed path from a given genome received contradictory orientations in different regions. If this is the case, the edge crossing the boundary between the differently oriented regions must be removed. Finally, the heuristic may terminate and still leave some edges unoriented. This indicates that the orientations are contradictory and need to be reversed. An example of the mini-cycle resolution process is shown in Figure Figure 43.

4.7 Supergenome Pipeline

The complete pipeline to create the supergenome can be divided into five parts:

i) Curation of input data sets

ii) Graph simplification one

iii) DAG construction

iv) Graph simplification two

v) Seriation

(i) is described in Subsection 4.7.1, (ii-iv) is described in Subsection 4.7.2, and (v) is described in Subsection 4.7.3. A more detailed overview in form of a flow diagram is given at Figure 44.

4.7.1 Curation of input data sets

Three genome-wide multiple sequence alignments (gMSAs) are investigated here. The smallest set, referred to as $\mathbf{B}$ (bacteria) below, is an alignment of four *Salmonella enterica* serovars. This alignment is produced with `Cactus` (Paten, Earl, et al., 2011) using the *Salmonella enterica Newport* genome as reference and comprises $13\ 416$ MSA-blocks, $50\ 932$ sequence fragments, and $18\ 047\ 456$ nucleotides. The medium-size set, termed $\mathbf{Y}$ (yeast), is an alignment of seven yeast species that uses the *Saccharomyces cerevisiae* genome as references. It comprises $49\ 795$ MSA-blocks composed of $275\ 484$ sequences fragments that contains $71\ 517\ 259$ nucleotides. The third, much larger set $\mathbf{F}$ (fly) is an alignment of 27 insect species that uses the *Drosophila melanogaster* genome as references. It comprises $2\ 112\ 962$ MSA-blocks composed of $36\ 139\ 620$ sequence fragments hat contains $2\ 172\ 959\ 429$ nucleotides. More detailed information of the data sets are given in Appendix A.

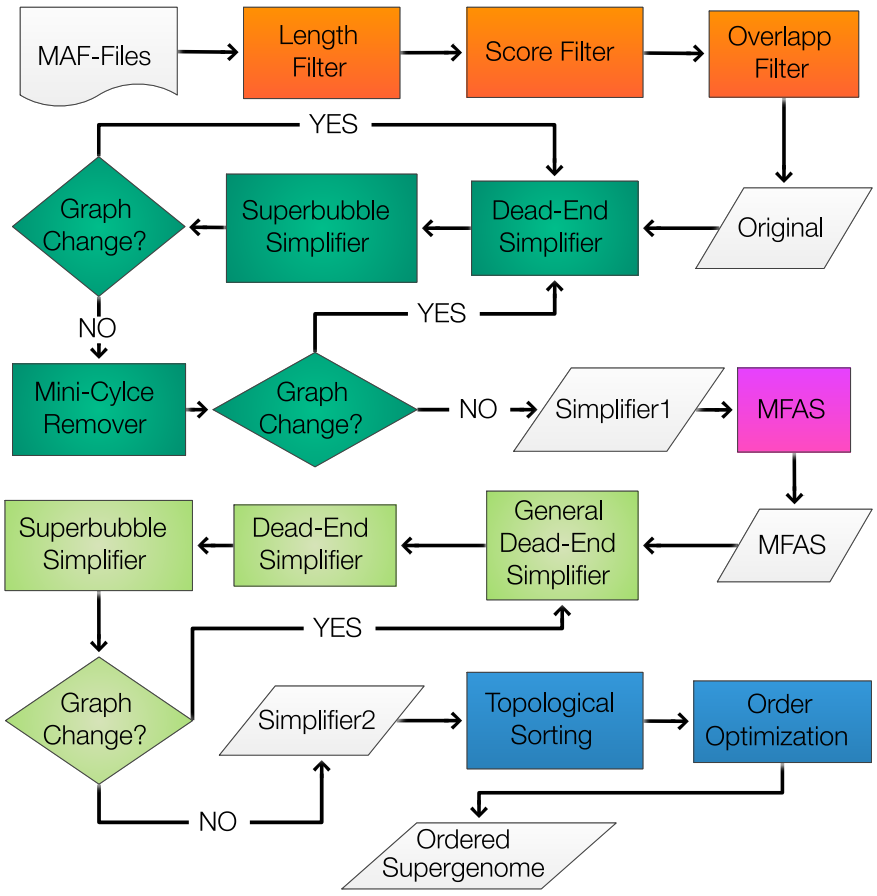


Figure 44: Flow diagram of the complete supergenome pipeline. The pipeline can be divided into five steps: First the input data is filtered (orange), secondly the first round of simplifier is applied to the supergenome graph (dark green), thirdly all cycles are removed (pink), fourth the second round of simplifier is applied (bright green), and the last step is the seriation of the supergenome (blue). Note that after every step an observable data set exists. They are used for the analysis of the pipeline (Chapter 5).

The two large gMSAs are produced by the `multiz` pipeline and are downloaded from the `UCSC` genome browser (Kent et al., 2002). They are, as discussed above, injective but not irredundant. In order to remove spurious MSA-blocks the input MSA-blocks are filtered with respect to first length, then score, and finally mutual overlap. Very short MSA-blocks are almost certainly either spurious matches or they are inserted to bridge gaps between larger MSA-blocks. Consequently, they convey little or no useful information. Therefore, all MSA-blocks are removed with a length ≤ 10 nt.

Since gMSAs tend to contain also very poorly aligned regions a minimum similarity is required, expressed here in the form of sum-of-pairs `blastz` scores (Chiaromonte, Yap, and W. Miller, 2001). Since these scale linearly with the number of columns $|\operatorname{columns}(B)|$ of the MSA-block B and the number $\binom{r}{2}$ of pairwise alignments formed by the r rows in B, is normalized with $\binom{r}{2}\ell(B)$ to obtain a similarity measure that is independent of the size of the MSA-block. Based on the parametrization of `blastz`, the threshold is set to a normalized score of -30, which corresponds to the gap extension penalty.

The coordinatization of supergenomes depend on the uniqueness of coordinate projections. Three major reasons are observed for overlaps, i.e., genomic regions that appear in more than one alignment: (i) the sequence is duplicated in some species. Then `multiz` tends to align the corresponding unduplicated sequence to both duplicates. (ii) Spurious similarities in particular in poorly conserved regions may lead to alignments containing a sequence element twice at the expense of the second copy. (iii) Short overlaps at the end of MSA-blocks may appear due to difficulties in determining the exact ends of alignable regions. The first two causes introduce undesirable noise and uncertainties. Therefore, all such overlapping MSA-blocks are removed. Since there is no easy way to determine which one of two overlapping MSA-blocks is correct, both copies are removed. The third case, in contrast, does not disturb the relative order of MSA-blocks and thus can be ignored. The overlap filter is applied after low quality alignments already have been removed from the data set.

An overlap of 20 nt is tolerated at the borders of MSA-blocks. This cutoff is designed to remove ambiguous alignments, while avoiding the removal of MSA-blocks that overlap by a few nucleotides owing to overlapping extensions of local `blastz` seeds. In addition sequences that completely overlap other sequences are removed regardless of their size to further reduce the noise introduced by spurious alignments. A stringent procedure is applied and all MSA-blocks are removed that contain sequences tagged for removal. In practice, this step removes only a tiny fraction of the MSA-blocks and thus does not significantly influence the coverage of the retained data.

The initial data filtering steps removed almost 35% (40%, 30%) of the MSA-blocks from data set $\mathbf{F}$ ($\mathbf{Y}$, $\mathbf{B}$). The majority are eliminated because of their minimal length. About 8.5% (27%, 0%) of the MSA-blocks are removed because they contained non-unique sequences. The sequences in the MSA-blocks that are removed with all filters contain less then 15% (26%, 0.4%) of the nucleotides in the alignment. Hence more than 85% (74%, 99%) of the sequence information of

the alignment is intact and the quality of the data is significant better. A more detailed summary of the filtering is compiled in Appendix B.

4.7.2 Graph simplification and DAG construction

The algorithmic ideas and their justifications for the graph reduction steps have already been discussed in Section 4.6. This section briefly address implementation issues as well as particular choices of cost functions and parameters that are discussed in a more general setting above.

The filtered data is used to create an initial supergenome graph. Then the three different graph simplifiers are iterated until no further reduction steps can be applied: the mini-cycle remover, the dead end simplifier, and the superbubble simplifier. The individual simplifiers are straightforward implementations of the basic ideas outlined above. The mini-cycle remover first identifies the mini-cycles, aggregates them into non-overlapping complexes, and then proceeds to remove contradictory edges in a greedy manner. The dead end simplifiers first check for each vertex in the input graph whether it is a valid sink or source. If a vertex is valid, it is merged with the neighbor. The superbubble simplifiers detect superbubbles with the algorithm *Superbubble#* described in Chapter 3. Then it merge the superbubbles to one vertex. A more detailed description of the simplifiers is given in Section B.2.

The mini-cycle remover works more effectively on a single big complex than on many small ones separated by narrow gaps. The other two simplifiers therefore are applied until a fixed point is reached to close some of these gaps. The entire procedure is iterated until the mini-cycle remover cannot change the graph any further.

Once a fixed point is reached directed cycles are removed. This amounts to solving the minimum feedback arc set problem (MFAS), which is known to be *NP*-hard (Karp, 1972). Given the size of the input graphs, the approach is bounded to linear-time heuristics. Here `Algorithm GR` (Eades, X. Lin, and Smyth, 1993) is used because it is known to work particularly well on sparse graphs. Cycle removal typically creates new possibilities to simplify the graph. For instance, a sink is created whenever the last outgoing edge of a vertex is removed. The new dead end can then be simplified further. The graph simplifiers are applied again after the cycle removal step.

The mini-cycle remover is not used in this second phase because it is not applicable to directed acyclic graphs (DAGs) by construction. Instead, a generalized version of the dead end simplifier is used in which a source s may have more than a single successor v, provided v is a predecessor of all other successors of s. The position of source s in the DAG is determined by v and thus s can be placed immediately before v. The corresponding arrangements for a sink and its predecessor is treated analogously.

4.7.3 Seriation

Finally, the common coordinate system is created by seriation of the DAG. The resulting supergenome, i.e. linear order of the vertices of the graphs corresponds to a linear order of all MSA-blocks. In particular, vertices resulting from a simplifier may contain more than one MSA-block. Those MSA-blocks are sorted already and thus are inserted as a single block. Seriation is naturally divided into two steps. First, topological sorting is used to calculate an initial linear ordering from the DAG. It is desirable that, if possible, two nodes v and w are placed consecutively whenever there is an edge (v, w) in the final DAG. This can be archived in with a DFS-topological sorting. Furthermore, the sibling order follows the support of the successors. In the way that the less supported successor comes first and the best supported successor the last.

The order obtained in this manner may not be optimal w.r.t. its agreement with the order of the blocks in the genomes. It provides a good starting point, however, for the final optimization step, which iw phrased as minimizing the number of triplets (i, j, k) for which the Robinson condition, Equation 4.2 (Page 107), is violated. The following distance measure is used:

$$d(i,k) = \begin{cases} \frac{1}{|(i,k)|} & \text{if an edge } (i,k) \text{ exists,} \\ \min_{i<j<k}\{d(i,j) + d(j,k)\} & \text{if a path from } i \text{ to } k \text{ through } j \text{ exists,} \\ \infty & \text{if no path from } i \text{ to } k \text{ exists,} \end{cases} \tag{4.4}$$

where $|(i,k)|$ is the number of edges from i to k. Since d is a good measure of co-linearity only for short distances, the path length is limited in Equation 4.4 to a small number of l edges. Here l is set to 10 in the implementation. In addition this reduces the effort of computing the distances from $\mathcal{O}(|V(\Gamma)|^2)$ to $\mathcal{O}(|V(\Gamma)|)$ as a consequence of the sparsity of the input graph Γ.

A gradient descent-like optimization algorithm is used to minimize the number of triplets for which the Robinson condition, is violated. Two nodes are siblings if they either share a predecessor in the DAG or if they are both sources. The move set for the gradient descent consists of swaps of siblings only. In addition, it is allowed to move a node directly in front of its sibling. The gradient descent is computed exhaustively by generating and evaluating each potential move. Since non-overlapping swaps do not influence each other, greedily a maximal set of non-overlapping swaps is execute in a single optimization step.

CHAPTER 5

Applications

Contents

In this chapter, applications are presented that use the theory of the previous chapters. First, in Section 5.1 applications of Chapter 3 is shown. Secondly, in Section 5.2 applications of Chapter 4 is demonstrated.

5.1 Superbubbles

The theory of superbubbles has two main applications. On one side, the new detection algorithm and its performance gain over the previous algorithm. On the other side, superbubbles can be used as graph properties. Both are discussed in this section.

5.1.1 Implementation

The novel detection algorithm from Algorithm 1 (Page 65) and Algorithm 4 (Page 96) are implemented in `Python` and are available as *Linear Superbubble Detector* (*LSD*). *LSD* can be installed using **pip**[1]. The source code is available on `GitHub`[2]. It is intended as a reference implementation emphasizing easy understanding rather than a performance-optimized production tool. The underlying graph structures make use of `NetworkX` (Hagberg, Schult, and Swart, 2008), which has the benefit that many input formats can be parsed.

`SUPBUB`[3] (Brankovic et al., 2016) is the only other publicly available implementation of a superbubble detector. Unfortunately, it has some bugs e.g., in the handling of successors in the depth-first search (DFS) tree that leads to problems with superbubbles with a back edge. Furthermore, an analysis of the code shows, that the construction of the auxiliary graphs strictly follows Sung et al. (2015). Hence it cannot serve as a reference implementation.

In order to compare the new approach to the state-of-the-art algorithm, the workflow of Sung et al. (2015) and Brankovic et al. (2016) is re-implemented using the same python libraries. This allows a direct comparison that focuses on the algorithms rather than the differences between programming languages and compilers. The partitioning workflow can be subdivided into two separate tasks: (1) the construction of the DAGs, and (2) the recognition of superbubbles within the DAG. For the first task, the new approach and the algorithm of Sung et al. (2015) (augmented by a simple linear-time filter to detect the false positives) are compared. For the second part, the new stack-based approach is compared with the range-query method of Brankovic et al. (2016).

This gives four algorithm combinations that are based on the partitioning workflow: Algorithm 1 (Page 65) and Algorithm 2 (Page 71) (LSD), Sung et al. (2015) approach (with extra filtering) and Algorithm 2 (Page 71) (S + LSD), Algorithm 1 (Page 65) and Brankovic et al. (2016) approach (LSD + B), and Sung et al. (2015) approach (with extra filtering) and Brankovic et al. (2016) approach (S + B). The last one is the state-of-the-art algorithm.

[1] `https://pypi.org/project/LSD-Bubble/`
[2] `https://github.com/Fabianexe/Superbubble`
[3] `https://github.com/Ritu-Kundu/Superbubbles`

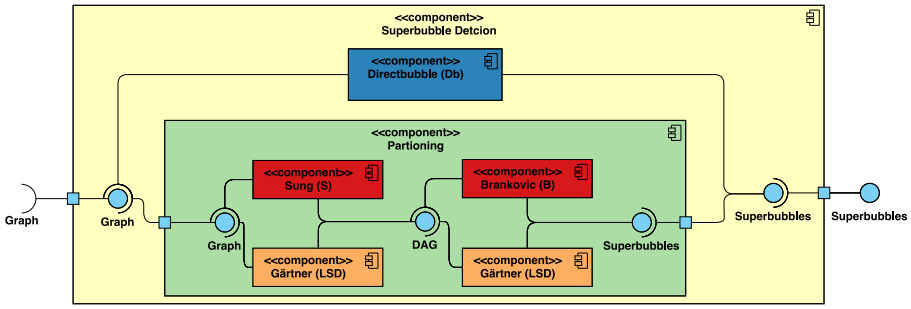


Figure 45: Component Diagram of Linear Superbubble Detector. The task to detect superbubbles (yellow) has a graph as input and superbubbles as output. There are to different approaches to do this: a direct approach (blue) and the partitioning workflow (green). The later uses two subcomponents: First, the graph is transformed into DAGs, and then the superbubbles are detected on this DAGs. The state-of-the-art algorithm (red) consists of two components from the literature. For each, a novel component is presented (orange). Since the subcomponents are interchangeable, four combinations in the partitioning workflow exist. Thus, five different approaches can be used to detect superbubbles.

Beside this in Algorithm 4 (Page 96) a third new approach is presented. This approach does not need the extra partitioning. This approach is called *Directbubble* afterwards. An overview as component diagram is shown in Figure 45.

The implementation of *Directbubble* deviate from the presentation in Chapter 3 in two minor details. First, instead of using the reverse postorder of the DFS-tree, directly the postorder is used. Further the corresponding (trivial) redefinitions of the helper functions $\mathbf{OutChild}(.)$ and $\mathbf{OutParent}(.)$ is used. Second, the determination of the cycles, the identification of the roots, and the identification of the superbubbles is not completely separated. Instead, cycle search, root detection, and superbubble identification is immediately performed for each DFS-tree. Since cycles and superbubbles necessarily completely are contained within the DFS-trees, this does not affect the correctness of the algorithm. As a by-product, a speedup is obtained by a constant factor because cycles reachable within a given DFS-tree are marked as "already processed" in the superbubble detection step and hence are not (superfluously) considered as candidate additional roots.

LSD is used as reference implementation and to benchmark the runtime (See Subsection 5.1.2). It is not useful for a productive implementation. To fill this gap a second implementation in *C++* is created. This version implements only *Directbubble* and is called *CLSD*[4]. It is optimized to work fast and memory efficient. Additional, a third implementation of *Directbubble* in Java exists as part of the supergenome pipeline.

[4] `https://github.com/Fabianexe/clsd`

5.1.2 Runtime

Table 1 summarized the empirical results for test data of different sizes taken from the Stanford Large Network Dataset Collection (Leskovec and Krevl, 2014). Also the unsimplified supergenome graph of data set $\mathbf{Y}$ is tested. Although the running times are comparable, is *LSD* consistently performs better than the alternative in the partitioning scheme for both tasks. The combined improvement of *LSD* is at least a factor of two in the examples tested here.

For most data sets, an approximately three-fold speedup of *Directbubble* compared to *LSD* is observed. The exception is the Slashdot data set for which no performance gain is observed.

To understand this outlier, it is necessary to understand the source of the speedup in the other test cases. In a typical case, both *Directbubble* and *LSD* performed three depth-first searches: in *LSD*, they are used to determine strongly connected components (SCCs), create auxiliary graphs, and detect superbubbles. *Directbubble* uses them to identify the cycles, quasi-legitimate roots, and finally the superbubbles. Both need to handle exceptional cases. *LSD* requires the construction of the Sung graph if a SCC coincides with a connected component of the input graph (rather than being just part of it). Since the Sung graph is twice the size of the SCC, this roughly doubles the running time. *Directbubble* behaves exceptionally for vertices that are reachable from a source. In this case, the detection of cycles and quasi-legitimate roots in cycles is skipped, incurring a substantial speedup. If a graph has neither a SCC that is also a connected component, nor large subgraphs reachable from a source, then *LSD* and *Directbubble* essentially performed the same computations and thus performed very similarly. The Slashdot data set is such a case. Typically, however, directed graphs have some sources so that *Directbubble* outperforms its competitors on most real-life graphs. All results and methods are available in the git repository[5].

This performance study shows *Directbubble* to be the new best algorithm for detection superbubbles. With the data sets used here, it gives a best speedup of twelve (Amazon) and a worst case speedup of two (Slashdot) discussed above. Thus, is the the predicted speedup on the most graphs higher then two.

5.1.3 Appearance of Superbubbles

The search for superbubbles only makes sense if superbubbles exist. Thus, a significant result are graph types containing superbubbles and graph simulation models containing superbubbles.

Onodera, Sadakane, and Shibuya (2013) stated the existence of superbubbles in assembly graphs. In supergenome graphs, a quantity statement on a broad base is missing. That superbubbles may exist in social networks is hinted by Table 1, where superbubbles are detected in some social networks. This assumption is verified with an evaluation of a more extensive set of graphs from the Stanford Large Network Dataset Collection (Leskovec and Krevl, 2014).

[5] `https://github.com/Fabianexe/Superbubble`

Table 1: Comparison of running times. The following five combinations of algorithms are: *Db* (*Directbubble*) refers to the new approach described in Algorithm 4 (Page 96). *LSD* (using the auxiliary graphs $\hat{G}_C$ and the stack-based superbubble detector). S + LSD combines the Sung graphs as auxiliary graphs (Sung et al., 2015) with *LSD* stack-based detector plus a post-filter for the false positives. *LSD + B* uses the *LSD* graph construction with the range-query-based detector of (Brankovic et al., 2016), and *S + B* uses Sung graphs together with the range-query-based detector, as well as the necessary post-filters. All computations are performed on a 2.5-GHz quad-core Intel Core i7 processor (Turbo Boost up to 3.7 GHz) with 6-MB shared L3 cache and 16 GB of 1600-MHz DDR3L onboard memory. Test data sets are taken from the Stanford Large Network Dataset Collection (Leskovec and Krevl, 2014) and the supergenome graph $\mathbf{Y}$. For each tested graph, the number of vertices N, the numbers of edges M, and the number S of superbubbles is listed.

Data	N	M	S	**Running Times (s)**				
				Db	*LSD*	*S+LSD*	*LSD+B*	*S+B*
$\mathbf{Y}$	49,795	130,993	325	1	3	4	5	9
EU Mail	265,214	420,045	13,285	5	14	16	30	32
Slashdot	82,168	948,464	0	16	16	30	22	37
Amazon	403,394	3,387,388	3	13	59	93	84	159
Google	875,713	5,105,039	6,477	26	95	147	152	255
Wikipedia	2,394,385	5,021,410	4,737	52	160	164	382	418

For the simulation models, the expectation is that they do not reflect the results of real data sets. This expectation is based on the fact that the models are created without considering superbubbles.

The absolute number of superbubbles are not significant. The size of a superbubble must also be considered. A large superbubble is more interesting than a mini superbubble. Thus, the number of vertices that are covered by a superbubble is used instead of the number of superbubbles.

However, different graphs have different numbers of vertices. Thus the number must be normalized by the number of vertices. Since every vertex can only be either covered by a superbubble or not, a value between zero and one is the result.

It makes further sense to determine the vertices that are covered by non-trivial superbubbles, i.e., none mini superbubbles. The reason for this is that mini superbubbles have less structure then non-trivial superbubbles. However, even the largest graph can have no superbubbles. Thus special attention is given if at least one superbubble exists in the graph.

First, some supergenome data sets are considered. For this, eight data sets from the UCSC website (other than the three used in Chapter 4) are downloaded. Then the alignments are filtered, and the original graph is created following the procedure in Figure 44 (Page 111). The results of these eight data sets are shown in Figure 46.

Each of the eight data sets contains mini superbubbles and non-trivial superbubbles. In every graph, the number of vertices that are covered by non-trivial superbubbles is higher than the number of vertices that are covered only by mini

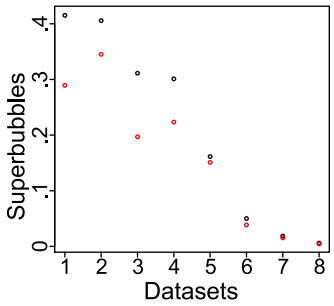


Figure 46: The superbubbles in different supergenome graphs. The data sets and the exact values are described in Appendix D. The number of superbubbles is normalized by the number of the vertices. The black symbols are all superbubbles, and the red symbols are the non-trivial superbubbles. The data sets are ordered by the normalized number of superbubbles.

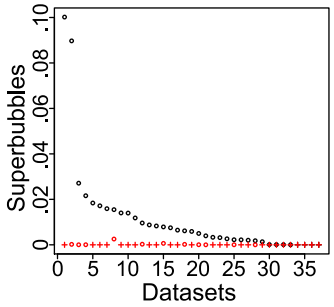


Figure 47: The superbubbles in different graphs of the Stanford Large Network Dataset Collection. The data sets and the exact values are described in Appendix D. The number of superbubbles is normalized by the number of the vertices. The black symbols are all superbubbles, and the red symbols are the non-trivial superbubbles. The data sets are ordered by the normalized number of superbubbles. If a value is zero a $+$ is used as the symbol and otherwise a $\circ$.

superbubbles.

The last three data sets have significantly more species than the first five data sets. In the first five data sets, are three to seven species and in the last three data sets 20 or 30 species. The superbubble reduction in the last three data sets hints that the fraction of covered vertices decrease if more species are included in the data set. In fact, except for the sixth data set, this rule holds.

However, even in the last data set more then sixty thousand superbubbles exists, and the largest contains over a hundred vertices. Thus, it makes sense to assume that in every supergenome data set, superbubbles exist. Where in more complex data sets, the relative number is reduced.

Now, since the existences of superbubbles in supergenome graphs are shown, other graph types are considered. The Stanford Large Network Dataset Collection (Leskovec and Krevl, 2014) is used as the source of other real graphs. Thirty-seven different data sets are downloaded from there website, and superbubbles are detected in them. The results are shown in Figure 47.

These graphs contains significantly fewer vertices covered by superbubbles. Besides the first two data sets, each data set has less than three percent of the vertices covered by superbubbles. In only eleven of the data sets non-trivial superbubbles exists. Furthermore, if non-trivial superbubbles exist, they only cover a tiny fraction of the vertices.

However, in only four data sets, it is impossible to detect any superbubble. Thus, the majority of graphs contains superbubbles. Therefore, it is plausible that simulated graphs of this type should contain a small fraction of superbubbles, and even a smaller fraction of non-trivial superbubbles should also be included.

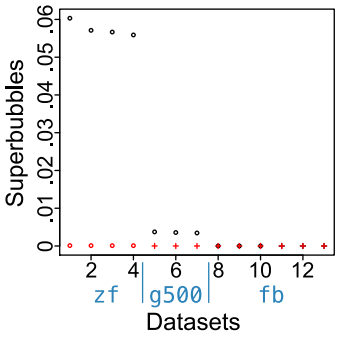


Figure 48: The superbubbles in different graphs of the LDBC Graphalytics. The data sets and the exact values are described in Appendix D. The number of superbubbles is normalized by the number of the vertices. The black symbols are all superbubbles, and the red symbols are the non-trivial superbubbles. The data sets are ordered by the normalized number of superbubbles. If a value is zero a $+$ is used as the symbol and otherwise a $\circ$. Three different models are used. The models are indicated in blue under the data sets.

The next step is to check if simulated data can reproduce this results. First, are published simulated data is used. The LDBC Graphalytics benchmark (Capotă et al., 2015) has created a default set of simulated data on which graph algorithms can be tested. They use three different models. The first model is an old model (g500) that is from another benchmark: the graph500 benchmark (Murphy et al., 2010). The second model is the facebook model (fb) that is the initial model of the benchmark (Capotă et al., 2015). Both models work by scaling up a specific graph that is observed in reality. The last model (zf) is a graph that has a Zipf distribution as degree distribution (Metcalf and Casey, 2016). All these generate undirected graphs for the benchmark. However, to use the graphs, the downloadable edge list is interpreted as directed graphs. The results of some of these data sets are shown in Figure 48.

The data sets have three distinct types of results: First, one type has $5 - 6\%$ of the vertices covered by superbubbles. This type also has a small fraction of non-trivial superbubbles. The second type has around 0.3% covered and no non-trivial superbubbles. The remaining data sets have no superbubbles or almost no superbubbles.

These types correspond to the different models, where zf have most superbubbles, g500 have some and fb no superbubbles. All models have fewer superbubbles than the supergenome graphs. Thus, these cannot be used as models for such graphs. However, the zf and the g500 model have comparable results to the Stanford Large Network Dataset Collection. Thus, these models are suitable to simulate data for such graphs.

The literature also knows other graph models that are not part of the LDBC Graphalytics benchmark. Thus, simulated graphs are created by using some of the models. The first model (BA) is the model of Barabási and Albert (1999) that uses the preferential attachment method to create graphs. The second model (ER) is the model of Erdös and Rényi (1959). In this model, every edge has the same probability to exists. The model of Watts and Strogatz (1998) is used as the last model (WS). It produces graphs with short average path lengths and high clustering coefficient. The results are shown in Figure 49.

An accidental resemblance by low vertex numbers is prevented by fixing the number of vertices to 100.000, and repeating each parameter combination three times. Furthermore, for the ER model, different edge probabilities are tested, and

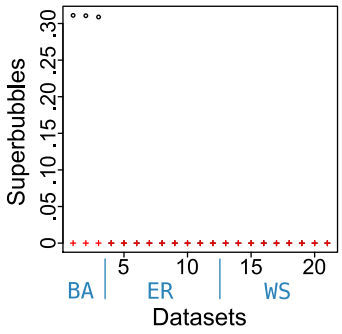


Figure 49: The superbubbles in different simulated graphs. The data sets and the exact values are described in Appendix D. The number of superbubbles is normalized by the number of the vertices. The black symbols are all superbubbles, and the red symbols are the non-trivial superbubbles. The data sets are ordered by the normalized number of superbubbles. If a value is zero a $+$ is used as the symbol and otherwise a $\circ$. Three different models are used. The models are indicated in blue under the data sets.

for the WS model, different rewiring probabilities are tested. Thus, altogether 21 data sets are tested. However, only the BA model create any superbubble but no non-trivial superbubbles.

Thus, concerning superbubbles, the BA model looks suitable for social graphs, but the ER and WS model does not cover any real-world graph class. Of course, all models have more parameters as here are tested; thus, maybe in different settings, more superbubbles can be created.

However, none of the three models should be fitting to create any non-trivial superbubble. Only a rear random event could create small non-trivial superbubble. Thus, these models also are not fitting for the supergenome graphs.

5.1.4 Simulate Supergenome Distribution

As shown before the literature leaks a graph simulation model that creates superbubble distributions comparable to supergenome graphs. Thus, it makes sense to take a more in-depth look at how such distributions can be created.

Supergenome graphs represent, in some way the evolution of the species in the background genome-wide multiple sequence alignment (gMSA). Thus, it is possible to use a genome simulation tool like simuG (Yue and Liti, 2019) to generate genomes and create a gMSA afterward from this simulated genomes. However, this creates computational overhead for simulating a graph. Furthermore, the indirect control over the graph makes it hard to adjust it to conform basic properties of the graph like vertex count.

Thus, a more abstract method is chosen to simulate this data. Every genome represents an order of the multiple sequence alignment (MSA)-blocks which must not contain every MSA-block. Note that this ignores the existence of chromosomes or contigs, but this is a possible simplification. Thus, a supergenome graph is constructed from a small number of paths.

These paths are not constructed entirely randomly. They evolve from one original order by mutations and specifications. Thus, this evolution is represented in a tree of orders. The root represents a simple identity order of a specific size. Then this order creates two new order by appealing two independent sets of mutations. This process can be repeated on the resulting two children and on this way a full binary tree of a given height can be constructed.

The tree leaves are then used in the graph as paths. Thus a tree of height three creates a graph with eight paths. By this process, a simplified evolution is simulated. The open questions are: which mutations should be considered and how these mutations are applied.

If mutations are ignored, the result would be a graph that corresponds to one path because all orders would be equivalent. Mini superbubbles would completely cover such a graph. Thus, to recover real data mutations should generate non-trivial superbubbles and reduce mini superbubbles. Two types of mutations are considered independently: the single nucleotide mutations and the chromosome mutations.

First, chromosome mutations are considered. Recap that four chromosome mutations exist: A deletion that removes some MSA-blocks, an inversion that inverse the order of some MSA-blocks, the insertion that moves one MSA-block to another position, and the translocation that interchange two MSA-blocks. Note that the term MSA-block is used here. In general the mutations are defined over parts of the genome but in the simulation mutated parts are restricted to the size of MSA-blocks. This is a valid simplification because only mutations can create MSA-blocks.

An overview of these mutations and their consequences in the supergenome graph are shown in Figure 50. The only mutations that result in graphs with superbubbles are the deletion and the insertion. However, in the case of the insertion, it is a mini superbubble that would also exist without any mutation. Thus, the only chromosome mutation that can create a non-trivial superbubble is a deletion. Note that the other mutations could create superbubbles if, in every sequence, a chromosome mutation happen. However, this needs a specific combination of mutations and thus are not very likely.

Furthermore, such mutations could quickly destroy non-trivial superbubbles by creating a cycle. Thus, it is a valid assumption that chromosome mutations are more likely reducing the number of superbubbles and not creating more non-trivial superbubbles. A result of this observation is that the non-trivial superbubbles must be mostly create by single nucleotide mutations.

The first question is how these mutations can influence the graph at all. A single mutation in a MSA-block should not influence at all. This one change is reflected in the MSA but nothing else changes. However, if many of the mutations appear in one MSA-block, the gMSA does not any longer represent the sequence as one MSA-block but two different MSA-blocks.

Thus, the single nucleotide mutations can be observed as the change of the MSA-block to a new MSA-block. Alternative a sequence could be not characterized as a new MSA-block but as an already existing MSA-block. This characterization can happen if, by chance, the mutations create a sequence that is similar to an already existing sequence.

Note that if a mutation creates a new MSA-block, the number of vertices is incremented by one. On the other side, mutating the last appearance of a MSA-block reduces the number of vertices by one. With other words, the resulting number of vertices differ from the number of MSA-blocks in the root of the tree. If every mutated MSA-block characterize as an existing MSA-block, the extreme case would be that all orders consist only out of one MSA-block. However, this is not

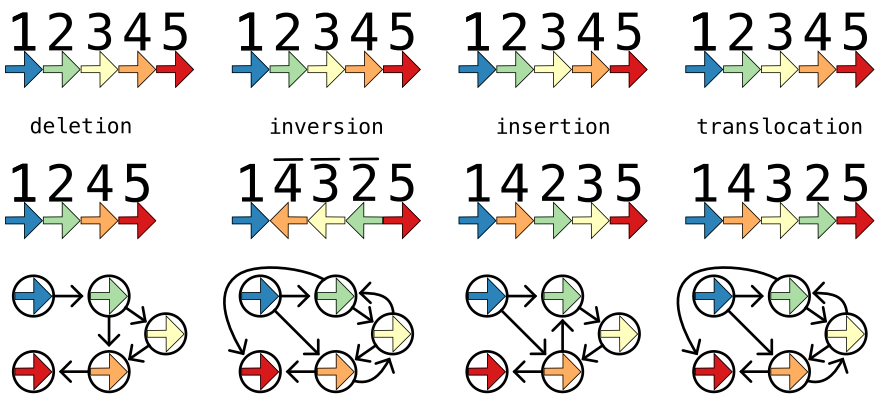


Figure 50: The different chromosome mutations and the resulting supergenome graphs. For every chromosome mutation beside the duplication an example is shown. On the top the starting sequence is given then the sequence that is created by the mutation. At the button than the created supergenome graph is shown that contains both sequences. This graphs shows for simplicity no multi edges. Note that only the graph of the deletion contain a non-trivial superbubble. The duplication is not shown because it would only add one edge (from the end to the beginning of the duplication) that creates a cycle in the graph.

very likely. On the other side, if only new MSA-blocks are created, the extreme would by that all orders do not share any MSA-block. Thus the graph consists of independent paths and is entirely covered by mini superbubbles.

To show that this model is capable of generating enough non-trivial superbubbles, graphs are simulated by using it. The simulation has four parameters. The number of vertices at the start, this number is fixed to $100,000$ like in the previous simulations. The second parameter is the number of generated orders, i.e., the three height. The parameter values 4, 8, 16, and 32 are tested. This span includes the values of the given real data sets. The third parameter is then the ratio of newly created MSA-blocks by every mutation. For this parameter are three values tested: 0% thus only old MSA-blocks are used, 90% thus every tens mutation matches an already existing MSA-block, and 99% thus one in hundred mutations match an existing MSA-block. The last parameter is then the mutation rate, i.e., how many of the MSA-blocks in the order are changed. For this $1\%'$ to 20% is used. Which means that from 100 MSA-blocks one or up to 20 MSA-blocks change. The results are shown in Figure 51.

This simulation shows that single nucleotide mutations can create graphs that are mostly covered by non-trivial superbubbles. Thus, this explains where such a large number of non-trivial superbubbles arises from in the supergenome graphs.

To look in more detail in the results, first the fraction of newly created MSA-blocks is analyzed. If no new MSA-blocks are created, no non-trivial superbubbles appear at all. This result is not surprising since every mutation creates a cycle in the graph. For the other two percentage, the larger percentage creates equal or

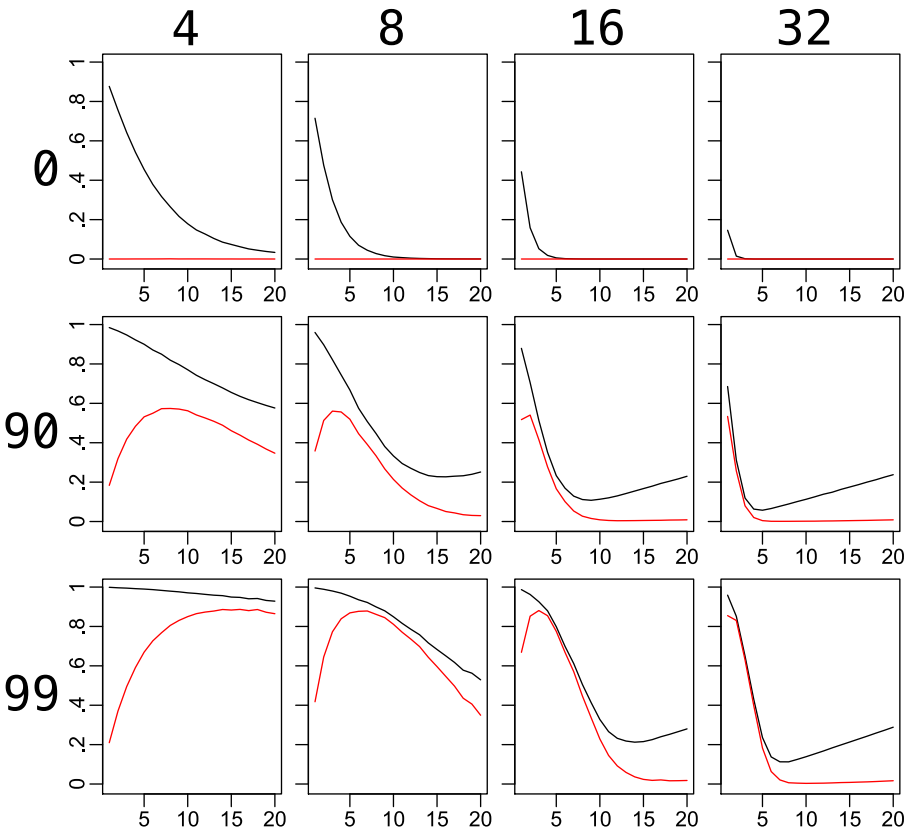


Figure 51: The superbubbles in different simulated mutation graphs. The simulation results for single nucleotide mutations. Each graphic shows for a fixed number of orders and new MSA-block creation fraction the results for mutation rates from 1 to 20. The mutation rate is shown on the x-axis, and the y-axis shows the fraction of vertices covered by superbubbles. The black line corresponds to vertices covered by all superbubbles and the red line corresponds to vertices covered by non-trivial superbubbles. Each row uses the percentage given in front of the row for newly created MSA-blocks in a mutation event. Every column uses the number of orders shown on the top.

more superbubbles and non-trivial superbubbles than the other. This result can again be explained with the lower number of created cycles.

Note that the percentage of one percent to randomly reassemble another MSA-block is still very high compared for real data. Since real data is created with alignments and alignment tools, try to prevent this reassemble. However, in reality, similar effects could be created by a combination of duplication events and single nucleotide mutations.

The second parameter is the number of orders. The result here is that higher numbers create fewer superbubbles than lower numbers with fixed other parameters. This result reassembles the same observation in real data sets. It can be explained with the reflection that the larger tree stacks more mutations and thus has the same effect as a higher mutation rate. On the other side, more genomes also mean that the more edges are in the graph. Thus, by the same number of vertices, they have a higher degree. A higher degree reduces the chances for non-trivial superbubbles.

The last parameter is the mutation rate. Here the behavior is more complicated. After the simulation of only old MSA-blocks simply losing all superbubbles, they are not considered in the following. Too low mutation rates create non-trivial superbubbles but not enough to mimic the real data sets. However, the number rises with greater mutation rate. This fast reaches a maximum and then the number of covered vertices fast decrease. As mentioned before the number of genomes works as a multiplier for the number of mutations. Thus the described effect can be seen by lower mutation rates for larger numbers of genomes.

For the more massive data sets of 16 and 32 genomes and high mutations rate the effect is such strong that new mini superbubbles are created. This behavior is the result of the above describe separation of the paths. If every path is independent of the other paths, the graph is covered with mini superbubbles, but no non-trivial superbubble exists.

The simulation shows that single nucleotide mutations can create non-trivial superbubbles, and they cover more of the graph than the real data sets. That more superbubbles are created makes sense because the chromosome mutations that can destroy some of the superbubbles are missing in this simulation. Furthermore, it shows that in more massive data sets, it can be expected that the fraction of covered vertices is small.

After all, the chromosome mutations are permutations on the orders. They can be added to the given simulation. Adding this aspect could create a more precise supergenome graph generator. However, it is not directly clear how such a permutation must look like to simulate realistic chromosome mutations. Furthermore, it is hard to validate which ratio chromosome and single nucleotide mutations on a MSA-block-level must have.

5.2 Supergenome

In this thesis a new heuristic algorithm to extract a common coordinate system for a supergenome from a genome-wide multiple sequence alignment is devised. The procedure has been tested on three alignments of very different size and difficulty:

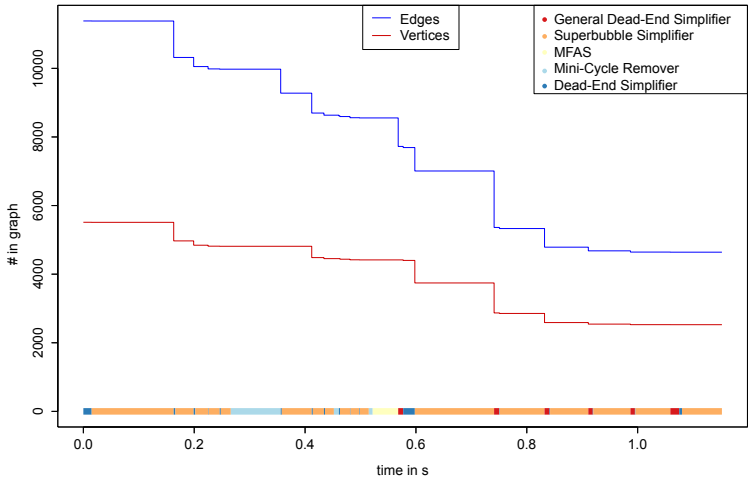


Figure 52: Simplification process for data set $\mathbf{B}$. The size of the graph (number of edges and vertices in the graph) is shown while the simplifier and the *MFAS* are applied. The running time is computed on an Intel(R) Xeon(R) CPU E7-8860 processor with 32 Gb RAM. At the bottom, the different processes are shown as color-coded bars.

an easy instance comprising four closely related bacterial species, an intermediate size problem composed of seven yeast genomes, and the alignment of 27 insect genomes as the most difficult instance.

5.2.1 Performance of individual components

The heuristic algorithm outlined above is composed of several largely independent components. It is of interest, therefore to consider their relative impact on the final results. Most edges are removed by the mini-cycle remover, with a small contribution of `Algorithm GR`. On the other hand, the largest reduction of the vertex set is due to the merges identified by the superbubble simplifier. More quantitative information is compiled in Figure 52, Figure 53, Figure 54 and in Appendix C. The simplifiers reduce the graph size by about an order of magnitude in both the number of vertices and edges, reducing it in size and complexity to a point where the seriation heuristic operates efficiently. The relative improvement is smallest for the bacterial data set.

Since the *Directed Colored Multigraph Betweenness Problem* cannot be solved exactly in reasonable time for instances with sizes that are of interest for the application at hand, it is impossible to measure performance relative to the exact solution. The multigraphs obtained from real-life alignments contain a large number of conflicting edges. In the most difficult data set, $\mathbf{F}$, for instance, the final order keeps more than 95% of the initial edges.

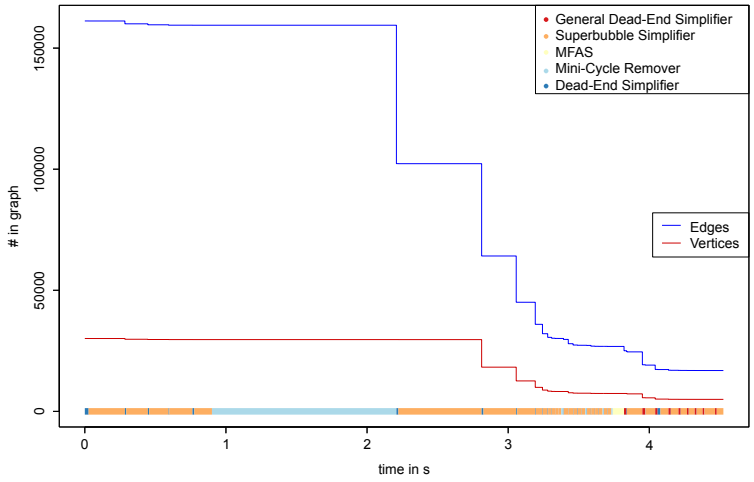


Figure 53: Simplification process for data set $\mathbf{Y}$. The size of the graph (number of edges and vertices in the graph) is shown while the simplifier is applied and the *MFAS* are solved. The running time is computed on am Intel(R) Xeon(R) CPU E7-8860 processor with 32Gb RAM. At the bottom, the different processes are shown as color-coded bars.

5.2.2 Assessment of the quality of supergenomes

Since no ground truth is available for this problem and the construction of simulated benchmarks for genome wide multiple sequence alignments would be a research project in its own right, here it is resort to measuring quantities that are informative about the final choice of the coordinate system.

A straightforward measure is the distribution of distances in the output coordinate system of MSA-blocks that are contiguous in at least one input genome. Since the length of MSA-blocks is not from interest, distance is measured here not in terms of sequence length but in terms of the number of MSA-blocks, so that adjacent MSA-blocks have distance 0. It is important here to keep track of the reading directions: contiguity with the same reading direction corresponds to preservation of the original genomic coordinates, while a change in reading direction indicates change of the order. Thus preserved and inverted reading direction are distinguished in the quantitative analysis.

Among the best-conserved features in the genome are open reading frames (ORFs), due to the strong selection pressures acting to preserve the corresponding proteins. As an immediate consequence it is expected that ORFs almost always are preserved. This should be reflected also by the supergenome coordinates, i.e., MSA-blocks belonging to the same ORF should have only a small number (smaller then five) of other MSA-blocks between them and retain their relative order. For higher eukaryotes, near perfect adjacency of coding blocks is not expected, however, because larger introns are subject to local rearrangements. To quantify the proximity of MSA-blocks of an ORF, the distances between all adjacent MSA-blocks are

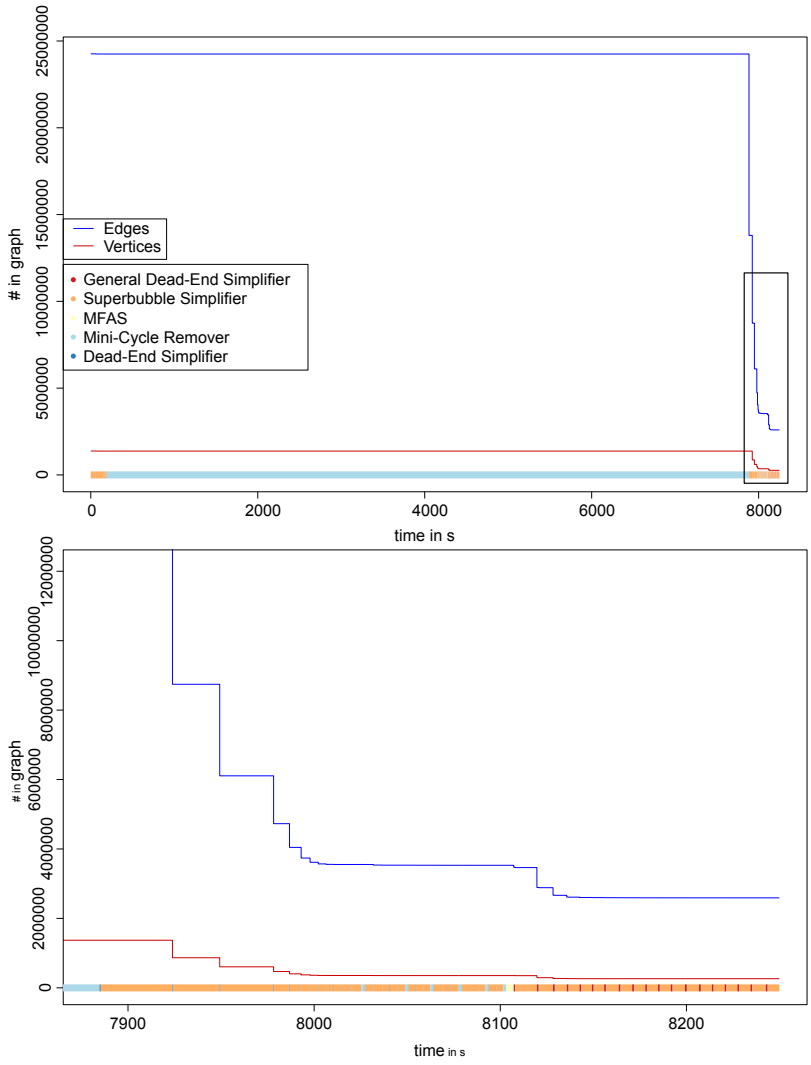


Figure 54: Simplification process for data set $\mathbf{F}$. The size of the graph (number of edges and vertices in the graph) is shown while the simplifier is applied and the *MFAS* is solved. The running time is computed on an Intel(R) Xeon(R) CPU E7-8860 processor with 32gb ram. At the bottom, the different processes are shown as color-coded bars.

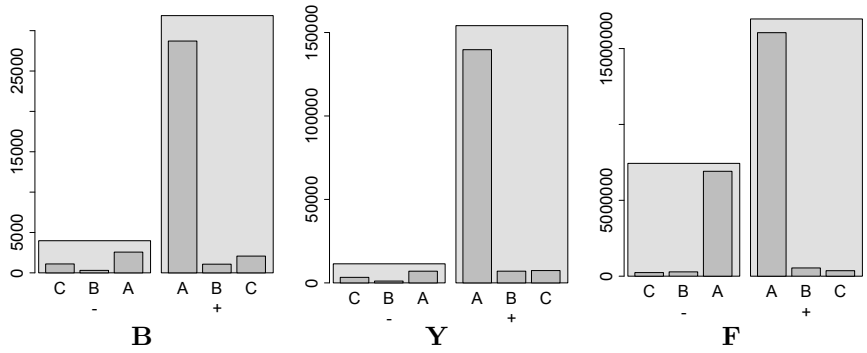


Figure 55: Distribution of MSA-block-wise distances of consecutive MSA-blocks in the original genomes. Data are shown separated for inverted (-) and preserved (+) orientation of consecutive MSA-blocks (light gray). As expected, the number of inverted MSA-blocks increases with the difficulty of the input alignment. In particular, there is a substantial number of local inversions in the insect data set $\mathbf{F}$. Both the inverted (-) and preserved (+) bin are subdivided further into a bin of adjacent MSA-blocks (A), MSA-blocks with a distance of 1-5 MSA-blocks (B), and more distantly placed MSA-blocks (C), in the supergenome.

determined as described above and their absolute values are added up to yield a single characteristic value. In addition the number of exons that are "broken up" in the sense that consecutive pieces do not have consecutive coordinates or are placed in reverse order in the supergenome is counted. Coding genes and exons are taken as annotated for the corresponding genomes. Note that in particular for large, intron-rich genomes such as the insect data set $\mathbf{F}$ this is an additional source of errors.

5.2.3 Quality of supergenome coordinate systems

The quality of the coordinate systems strongly depends on the quality of the input alignments. A detailed discussion of issues with the input alignments can be found in Appendix C. Here, the focus is on an assessment of the coordinate systems themselves.

In order to check the overall quality of the solution a betweenness graph is computed from the supergenome coordinate systems. This is done by starting with a graph without edges. First, all edges that are supported by the total order of the supergenome are added. This is followed by edges that contradict the total order but do not create contradicting betweenness triples. Note that this graph is not necessary optimal but a good approximation that can easily be computed. The edge set of this graph is compared to the edge set of the initial graph. Good solutions are expected to retain most of the edges. For the three data sets are 95.3%, 97.5%, and 99.4% of the edges are retained in data sets $\mathbf{B}$, $\mathbf{Y}$, and $\mathbf{F}$, respectively.

The distribution of MSA-block-wise distances in the supergenome of MSA-blocks

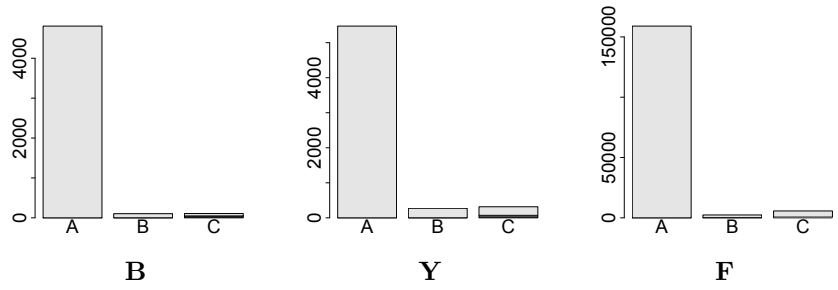


Figure 56: Distribution of MSA-block-wise distances between MSA-blocks that contain ORFs for the bacteria and yeast data set. For the insect data individual exons are considered since rearrangements as well as alignment errors within introns are not infrequent. The black bar indicates the number of broken ORFs/exons. The data is binned in three distance ranges: a distance of 0 (A), a distance of $1-100$ (B), and a distance larger than 100 (C).

that are consecutive in the original genome serves as a simple measure of preserved synteny. The results are summarized in Figure 55 and presented in full detail in Appendix C. Another measure is how many of the input orders are preserved. To measure this consider every MSA-block and all successors from the different genomes. For the bacterial data set $\mathbf{B}$ 89% of the successors preserve the order and 80% also preserve the adjacency. For the yeast data set $\mathbf{Y}$ it is observed that 93% of the successors preserve the order and 84% also preserve adjacency. This is a very encouraging result, taking in mind that every true genome rearrangement necessarily introduces at least one non-adjacency. Even in the much larger and more difficult fly set $\mathbf{F}$ 70% of the successors are preserved the order and 66% also preserve adjacency. The overwhelming majority of non-contiguous successors are placed in the adjacent but order-reversed position, reflecting the level of local rearrangements in the insect data set. This is entirely reasonable given the much larger number of species and their larger phylogenetic depth compared to the yeast data. Taken together, these numbers already indicate that the supergenome coordinates are meaningful and indeed are likely an useful starting point for large-scale multi-genome comparisons.

Restricting the attention to coding sequences yields a more stringent quality measure. As bacteria have essentially no introns, it is expected that nearly all MSA-blocks belonging to the same ORF retain both adjacency and order. In the bacterial data set $\mathbf{B}$ 96% of the ORFs are in one stretch with no interruption and less then 1% of the ORF are broken. Since yeasts have few and short introns (Spingola et al., 1999) it is expected that data set $\mathbf{Y}$ is also very well-behaved in this respect. It contains $6\,062$ ORFs annotated for *Saccharomyces cerevisiae*. Of these, $5\,474$, i.e., 90%, are consistently represented in the coordinate system. An additional 272 ORFs, about 5%, have a distance of less then 100 MSA-blocks between them. Only 73, i.e., a bit more than 1% of the ORFs are broken. For *Drosophila melanogaster* are $167\,051$ exons annotated, and part of the alignment $\mathbf{F}$. Due to large and abundant introns the analysis is based on individual exons rather

than complete ORFs for set $\mathbf{F}$. 95% of the exons are consistently represented. Only 779, about 0.5%, are broken. Overall, thus, the supergenome coordinates behave very well for all three data sets.

5.2.4 Yeast Tricaboxylic Acid Cycle

As an example for a comparative genomics analysis the genes of the tricaboxylic acid (TCA) cycle of the yeast species used. The TCA cycle for aerobic respiration is well studied and discussed in yeast (e.g. Krebs, Gurin, and Eggleston (1952)). In *S. cerevisiae*, 20 genes belong to the TCA cycle (Haselbeck and McAlister-Henn, 1993; Oyedotun and Lemire, 1997; Saccharomyces Genome Database Community, n.d.; Yasutake et al., 2003). All of them are part of the initial set of MSA-blocks in the yeast data set ($\mathbf{Y}$). The TCA cycle is at least for *S. cerevisiae* essential. Consequently, it is expected that the genes of TCA cycle are conserved in all yeast species in the yeast data set.

All genes are distributed over at least two MSA-blocks in *S. cerevisiae*. For all genes, evidences is found that they are contained in the initial alignment also for the other species. Note that `BLAST` is used to search the protein sequences in the genome assembly since there is no annotation data available. Many genes of the TCA cycle are conservered in large parts or completely in all species i.e. the sequences from different species belongs to the same MSA-block. Nevertheless, there are several cases with low conservation for one species. For example, SDH1 in *S. kudriavzevii* and *S. cerevisiae* do not share any nucleotide between the annotations (based on BLAST) but in the MSA-blocks for SDH1 in *S. cerevisiae*, a sequence for *S. kudriavzevii* can be found. However, all those issues are likely artifacts from the assemblies and alignment procedure.

After filtering some MSA-blocks are lost. Five genes of all genes loose one MSA-block each due to the length and score filter. However, the information loss is negligible since in each case less than 1% of the gene is lost. The overlap filter discards MSA-blocks of 11 genes such that large parts or almost complete genes are lost. Those cases are investigated more deeply with the result that those genes exhibit a high resemblance to either other genes in the TCA cycle or pseudogenes that result from whole genome duplications (Byrne and Wolfe, 2005). Except *S. cerevisiae*, all genome assemblies in the yeast data set consists of contigs and thus have a rather low quality. Thus, very similar nucleotide sequences might be merged into one sequence. As a result very similar (part of) genes may are not assembled as separate genes but occur only one time in the assembly. A reliable assignment of those misassemblies to genes is therefore difficult.

In the following cases, the overlap filters discards parts or complete genes. Large parts of IDH1 are lost in most of the species due to high similarity to IDH2. Also a fifth of IDH2 is lost due to this similiarity. The same hold for CIT1 and CIT3 where CIT1 is lost in most of the species and a fifth of CIT3. PYC1 and PYC2 are very similar in sequence. After filtering almost all MSA-blocks for those genes are discarded. After filtering, at least 50% of ACO1 is lost due to the similarity to ACO2. Two species loosing ACO1 completely. Large parts of AC02 are retained. SDH1 and its homolog YJL045W is lost almost completely in all species due to

there homology. Also SDH3 has a homolog that is a pseudogene: SHH3. This leads to loss of large parts or the complete loss of SDH3 after filtering. The MDH1 gene is another gene where a part of the gene is filtered. The filtered part overlap with the MDH3 gene. The MDH3 is a very similar gene that has a comparable function as the MDH1 gene in the glyoxylate cycle an other variation of the TCA cycle.

To summarize the results after filtering, the low assembly quality and the alignment method for the whole genome alignment causes the removal of MSA-blocks for eleven genes. For further analysis, only the nine genes which are kept completely after filtering are used. These nine genes are almost complete in the same MSA-blocks for all species. For seven genes, the MSA-blocks of the gene are placed as successive MSA-blocks with successive genomic coordinates in the optimized order. The MSA-block for KGD2 as well as the MSA-block for SDH2 are broken apart by the minimum feedback arc set problem (MFAS) heuristic. Since the information that they should be successive is lost, they are places in different locations in the optimized order. However, at least within the two parts, MSA-blocks and genomic coordinates are successive.

To summarize, the common coordinate system for yeast shows the conservation of nine of the 20 genes in TCA cycle. Filtering statistics indicate that the remaining eleven genes can not be analyzed with respect to conservation. They have homologs or share sequence similarity with other genes such that it is unclear if the sequence is conserved or not. The raw multiple alignments used as input thus suggest conservation which is questionable. For conserved genes, the method mostly creates the correct order of the MSA-blocks allowing to easily access the corresponding alignment parts for comparative analysis.

CHAPTER 6

Discussion and Outlook

Contents

6.1 Superbubbles

In this thesis has re-investigated the mathematical properties of superbubbles and their obvious generalization, the weak superbubbloids. It not only re-derive fundamental results, in particular Proposition 1 (Page 54) and Proposition 2 (Page 66) in a more concise way, it also identify problems with auxiliary graphs propose in Sung et al. (2015) that lead to false positive superbubbles. Although these are not a fatal problem and can be recognized in a post-processing step without affecting the overall time-complexity, the issue can be avoided by using a different, in fact simpler, auxiliary graph that is acyclic. Based on the fact that the superbubbles in a directed acyclic graph (DAG) can be listed in linear time (Brankovic et al., 2016), the problem of listing all superbubbles in an arbitrary digraph can indeed be solved in linear time. For the DAG case a conceptually simpler replacement for the algorithm of Brankovic et al. (2016) is presented that also has linear running time.

Furthermore, the observation that in principle, all superbubbles in G can be identified in linear time in a single depth-first search (DFS), motivate another extension. However, the roots of the individual DFS-trees must be known beforehand. The main result is here that a suitable set of starting points, called quasi-legitimate roots, (1) always exists in every given digraph and (2) can be identified in linear time, using two additional DFSs. In the first pass, a suitable set of cycles is constructed such that every node in G is reachable from a source vertex of one of these cycles. In the second pass, a particular structure of "detours" in a cycle C is used to identify quasi-legitimate roots in a given cycle. To this end, a notion of $\overset{C}{\rightsquigarrow}$-reachability is defined that may also be interesting to characterize (short) cycles.

A comparison of running times of *Directbubble* and previous approaches shows that practically useful performance gains are obtained essentially from two sources: (1) dispensing the construction of auxiliary graphs and (2) avoiding most of the processing for all vertices reachable from a source in G. In practice, a speedup of about a factor of eight on most, but not all, benchmark cases. In all cases, *Directbubble* perform at least as good as all competing algorithms for superbubble detection.

With `LSD`, a reference implementation in `python` is provided, and with `CLSD`,a stand-alone performance-oriented implementation is provided. Thus, the research area of superbubbles is uncovered to a variety of people. Researchers can take the reference implementation as template and implement the detection algorithm quickly in their tool. Alternatively, they can create a pipeline that uses the already implemented optimize solution.

6.2 Parallel Superbubble Detection

Even if this is not the focus of this work, the previous discussion gives also hints about superbubble detecting with a parallel computing model. The literature knows three ways of detecting superbubble: brute force (Onodera, Sadakane, and Shibuya, 2013), vertex merge (Sung et al., 2015), and DFS based (Brankovic et al., 2016;

Gärtner, Müller, and Stadler, 2018; Gärtner and Stadler, 2019). As shown before the DFS based algorithm creates the best linear results. However, the others have large independent parts in the algorithm. Thus, all three are valid candidates for a suitable parallel algorithm.

For the parallel algorithm, the `pregel` model (Malewicz et al., 2009) is used. In short, this model assumes that every vertex is a compute unit which only knows its direct neighborhood. A vertex can send messages to another arbitrary vertex. The algorithm divided into steps where every message sent in one step is received at the beginning of the next step. The number of required steps determines the effectiveness of the algorithm. Thus many parallel computations are possible so long as only local information is used. This model is chosen because it can effectively be implemented in many architectures, including shared-nothing compute clusters. Thus, an effective algorithm with this model could scale to more or less arbitrary large graphs.

A superbubble detection needs to propagate information from the exit t to the entrance s of the superbubble $\langle s, t\rangle$ or vice versa. In `pregel`, this propagation needs $\log(|s \to t|)$ time because only local information is given. Therefore, the lower complexity bound is $\log(|s \to t|)$ where $s \to t$ is the longest path in a superbubble of the graph G. Note that the complete graph could be a path in superbubble; thus, the worst-case lower bound is $\log(|V(G)|)$.

Clearly, for an arbitrary graph, the lower bound cannot hold. A simple contradiction is a graph without any superbubble. On such a graph, the bound would mean that the algorithm terminates after a fixed number of steps. Thus, only a fixed number of local information can be used. However, if the complete graph is a weak superbubble, this is impossible. The weak superbubble can only be discarded in the last step. Thus the detection takes $\log(|V(G)|)$.

First, the DFS approach is checked on parallel compatibility. The first and extensive drawback is that no effective way of doing a DFS in `pregel` is known. The default method is to use a token indicating which vertex is the active vertex at the moment. Thus it needs linear time regarding the number of vertices that the DFS visit.

The algorithm still can be optimized for parallel computation. The essential detail here is that many DFS can be started on legitimate roots at the same time as long as every vertex belongs to precisely one DFS at the end. Note that this does not create DFS-forests, but every superbubble is completely within a subtree of one DFS-tree. Thus, the detection works as before.

If more legitimate roots are used, the approach is more effective. More legitimate roots can be found by adopting Lemma 13 (Page 64): if v is in a (finished) DFS-tree T, and w is a sibling of v but not in T then w is a legitimate root. Thus, when a DFS-tree is finished, it can be scanned for new legitimate roots to start new DFSs.

However, this gives a lower bound for the algorithm in the way that it takes at least linear time concerning the largest connected subgraph of vertices that can only be reached by one DFS started from a legitimate root. This must equal or larger than the largest weak superbubbloid.

Before describing a parallel algorithm based on the vertex merging approach of Sung et al. (2015), the algorithm is recapped. The idea is based on the observation

of Corollary 11 (Page 69). It stated that for every exit a predecessor exists with only one successor. Sung et al. (2015) have proven that when this predecessor is merged in the sink, again at least one predecessor exists that has only one successor (the sink). The merging can be repeated until the superbubble is reduced to a mini superbubble. Therefore, only mini superbubbles must be detected.

Every vertex that has only one successor can be merged with this successor. If this is the only predecessor of this successor is a superbubble is reported prior to merging. This mergings are mostly independent of each other. It is crucial to keep track in which vertex a source is merged. A vertex in which a source is merged cannot be the exit of a superbubble.

However, to report all superbubbles, there is an exception of this independent merging. A path of vertices that all have only one successor. If such a path exists, only the first vertex can be merged because every other vertex in the path could be an exit of a superbubble and can only be merged after this superbubble is reported. Sung et al. (2015) initially handled this problem by using this algorithm only on a DAG and processing in topological order. Note that theoretically, a cycle could exist where every vertex has only one successor. However, if the cycle has two vertices, no superbubble exist, and in the other cases, each vertex pair creates mini superbubbles. They can be reported all at once if such a cycle is detected.

Thus, the algorithm needs as many steps as the length of the longest path where every vertex has only one successor. This bound can be larger or smaller than the largest weak superbubbloid. Thus, compared to the DFS based approach, graphs exists where this approach performed better and also where the other approach performed better.

The last method is the brute force method after Onodera, Sadakane, and Shibuya (2013). This approach can easily be transformed into a parallel algorithm because the structure is already parallel. It assumes every vertex is a superbubble entrance and extends the superbubble until a contradiction or valid exit is found. The only change to the linear algorithm is that this starts every vertex at the same time.

Thererfore, the run time resembles the largest number of vertices that can be included in a superbubble until a contradiction, or a valid exit is found. This largest extended superbubble has at least the size of the largest weak superbubble in the graph. Thus, it is linear concerning the number of vertices in the largest weak superbubble. Note that this bound equals or is less than the bound of the DFS approach in any case. This follows because every added vertex has only already added predecessors. Thus, no legitimate root is added, and every added vertex can only be reached from the start vertex or through the start vertex. However, again, graphs exist that perform better for the merging approach. An example of this is given in Figure 57.

To conclude this section, there exist two different parallel algorithms that have different best and worst cases. None of this parallel algorithms is the best linear solution. The merging approach works best in broad star like graphs, while the brute force graph works best in colinear graphs with only local edges.

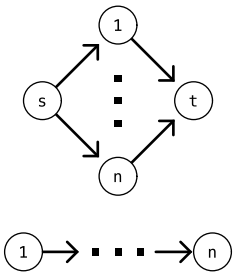


Figure 57: Worst and best cases of parallel superbubble detection. On the top, a graph is shown that is the worst case for the brute force algorithm ($\mathcal{O}(n)$ steps) and the best case for merging algorithm ($\mathcal{O}(1)$ steps). This is due to the fact that the brute force algorithm adds all vertices after each other and the merging algorithm merges all 1 to n vertices in one step in t. On the bottom, a graph is shown where the opposite is true: the brute force algorithm needs $\mathcal{O}(1)$ steps and the merging algorithm $\mathcal{O}(n)$ steps. Here $n-1$ mini superbubbles exist, which the brute force algorithm detects in one step, and a path of n vertices exists that have precisely one successor, so the merging algorithm can only reduce this path by one each step.

6.3 Generalization of Superbubble

The mathematical analysis of superbubbles suggests considering generalizations by relaxing some of the properties.

6.3.1 Cyclebubble

The first generalizations allow possibly restricted sets of cycles within the "bubble" but retain the idea of an induced subgraph that cannot be traversed without passing through the entrance and the exit. For instance, one might relax (S.v) and require only that an interior vertex v cannot be reached from t without passing through s and cannot reach s without passing through t.

Such a generalization should also be detectable in linear time by adopting the *Directbubble* algorithm to handle back edges in more detail. More precisely two new helper functions like $\mathbf{OutChild}(v)$ and $\mathbf{OutParent}(v)$ would be needed that only consider back edges. With these, two new intervals must be considered such that the outgoing back edges end after s and the incoming back edges start before t with respect to the reversed postorder of the DFS. This modification adds only constant effort for every vertex thus does not change that it is a linear time approach.

However, the set of bubble structures that contain cycles detectable in linear time is a small set. Apparently the case that a strongly connected component (SCC) is also a connected component produces worst case scenarios that can not be handled in linear time anymore. As an example, removing the acyclicity condition without substitution, the resulting structures would not be unique anymore. An example of such a graph is shown in Figure 58. Note that such a structure is still a real subclass of the bubbloids.

The false positives generated by the approach of Sung et al. (2015) may also be considered as the prototype of a broader class of superbubble-like structures. It does not seem obvious, however, to characterize them beyond being induced acyclic subgraphs with a single source and a single sink vertex. A related structure that also generalizes superbubbles are maximal connected convex acyclic induced

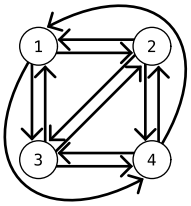


Figure 58: The ambiguous without acyclicity condition. A complete graph G with four vertices (K4) is shown. If considering a structure of a superbubbloid without the acyclicity condition, every vertex could be the entrance and every other vertex the exit of such a structure. The two remaining vertices would be the interior. This follows directly from $[s,t]_{\rightsquigarrow} = V(G) = [t,s]_{\rightsquigarrow}$ for every $s,t \in V(G)$. Thus, such a structure is ambiguous on this graphs. Note that the minimality condition cannot be fulfilled by any of the structures.

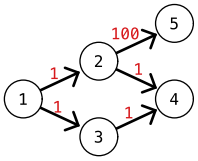


Figure 59: An example of a distance bubble. A graph with five vertices is shown. The red numbers on the edges represent distances. The graph contains no superbubble. However, the edge $(2,5)$ has a much higher distance than the other edges. Thus, it could make sense to create a distance bubble $\langle 1,4 \rangle$ that excludes 5.

subgraphs (Balister et al., 2009). Here, the vertex set U has the property that no two vertices $x, y \in U$ are connected by paths that are not entirely contained in U.

6.3.2 Distance Bubbles

An other generalization of a superbubble could be created in a distance graph. This is a property graph where every edge has a distance property. This property can be used for a modified version of the reach relation. This modification is a specific distance value t as second input and states that a vertex u can be reached from v if a path $v \to u$ exists and the sum of the distances of this path is less or equal than t.

This new reachable relation could be used in the reachability and matching condition of a superbubbloid. The resulting structure depends on the value of t. It could be called a t-distance superbubble. For a constant t the superbubbles can be detected in linear time with the algorithm given here by ignoring too long edges. However, different t values could create different superbubble sets. An example of such a distance superbubble is shown in Figure 59.

This special version could be handy in distance based graphs (compare Section 6.4). For example, in some cases not really existing edges are present in the graph but marked with a high distance. In such a graph a distance superbubble makes more sense then a superbubble. Note that this idea could be further extended to edge types.

6.4 Other Graph Algorithms and Superbubbles

It is already be shown that superbubbles are useful in most graph linearization algorithms. However, they can also be useful in other algorithms. Basically every algorithm that utilize distance or reachability data can profit from superbubbles as a precalculation. As an example the Dijkstra's algorithm is considered.

The Dijkstra algorithm detects from one start vertex the distances to every reachable vertex. A superbubble could be precomputed in this approach if the start vertex is outside of the superbubble. Since the interior and the exit of the superbubble can be only reached through the entrance, the values can be precomputed as reference to the entrance. Then if the minimal distance of the entrance is known the other distances are directly given.

However, the Dijkstra algorithm runs in linear time. Furthermore, the superbubble must be detected in linear time and for every superbubble a Dijkstra run from the entrance must be started. Thus, for a single run of the Dijkstra algorithm the overhead of the precomputation is larger then the reduction of run time. This change if many Dijkstra searches are done. The overhead is the same and the reduction accumulates. For example this is the case in navigation software or in centrality indexes.

This finding can be generalized of every reachable based algorithm. Other examples could be Floyd–Warshall algorithm, flow networks, connected component detection or SCC detection. Beside this, many domain specific algorithms that are based on reachability may profit from superbubbles.

6.5 Supergenome

This thesis proof that the problem of computing a common coordinate system for supergenomes with the *Directed Colored Multigraph Betweenness Problem* is *NP*-hard. It belongs to a class of relatively poorly studied betweenness problems for which few efficient heuristics have been developed so far. Several local simplification rules are introduced that can be applied iteratively to reduce the problem. It is important to note these reduction steps are only heuristics and do not guarantee optimal solutions. In conjunction with a simple serialization approach for the residual graph, they nevertheless yield practically useful results with acceptable computational efforts.

The most immediate application of the supergenome sorting problem is the direct comparison of genome annotations for multiple genomes. Hence it constitutes a prerequisite for comparative genome browsers. The approach is applied to three real-life data sets of different sizes and difficulties. The results indicate that practically useful coordinatizations can be computed. The computational requirement of the method scales favorably so that in principle even the largest genome-wide multiple sequence alignments could be processed.

The present study, however, also highlights the shortcomings of currently available genome-wide multiple sequence alignments (Earl et al., 2014; Ezawa, 2016). The issue is not only the relatively moderate coverage with multiple sequence alignment (MSA)-blocks that contain at least most of the species under consideration, but also the substantial fractions of MSA-blocks that have been removed from the data set due to likely artifactual sequences. No attempts to analyze the UCSC 100-way vertebrate alignments are done, since these data are even more complex than the fly data set due to the very large number of paralogs introduced by genome duplications.

Synteny, i.e., the preservation of relative genome order, is in general a good predictor for homology. This fact suggests to use the common coordinate system to identify likely homologous regions that are not included in the initial MSA-blocks. These could then be (re)aligned at sequence level and included in a revised multiple sequence alignment. This, in turn, could yield an improved common coordinate system. The systematic improvement of genome-wide alignments, albeit an interesting and extremely useful endeavor, is beyond the scope of this thesis.

Possible improvements of the approach taken here are conceivable in at least two directions. First, one may consider a hybrid algorithm that solves subgraphs with a dominant backbone using the method discussed in Haussler et al. (2018). As discussed, it is assumed that large parts of the global graph structure are not amenable to such a solution, but it is also reasonable to assume that gene regions under strong conservation pressures can be solved fairly easily using a local backbone-based approach. A second venue of research is concerned with the determination of the final backbone order. Depending on the phylogenetic range under investigation, the ancestral gene order would provide an useful backbone based on the phylogeny of the species involved in the alignment.

6.6 Parameterized Supergenome

The supergenome creation as shown in this work has only basic parameters to adjust the results. For example the maximal path length in the seriation. However, there exists a simple background model that can be modified. This model follows directly from the definition of the *Directed Colored Multigraph Betweenness Problem*. It tries to maximize the number of valid edges. This background model is used in the three main components: the mini-cycle remover, minimum feedback arc set problem (MFAS), and the seriation. The model decide which connections are kept intact and which are discarded.

However, this model can be modified by defining a weighted version:
Directed Weighted Colored Multigraph Betweenness Problem: Given a directed colored multigraph Γ and a weight function $w : E(\Gamma) \to \mathbb{R}$, find a total order on V such that $E^* \subseteq E(\Gamma)$ maximize $\sum_{e \in E^*} w(e)$ under the condition that $\forall [i \lesssim j \lesssim k] \in \mathscr{C}(V(\Gamma), E^*)$ either $i < j < k$ or $i > j > k$.

The introduction of a weight function opens many possibilities of modifications. A biologically meaningful weight function could be to use one species as backbone. Thus the edges with the color of this species get a much higher weight. This simple solution already creates other orders, i.e., supergenomes. However, one could use even more complex weight functions. The weights could be depending on the taxonomy. Such approach could be used to create results that are more like the ancestral sequence then the normal supergenome.

This alternative supergenomes are called "parameterized supergenomes". They, can also computed with the framework given here. Only three things must be changed. The mini-cycle remover must use the weight function to decide which direction is kept. Furthermore, for MFAS the weighted version is used, which is

well known in the literature (Flood, 1990). The last change is then the adoption of the seriation distance functions to use the weights in the calculation.

The analysis of such parameterized supergenomes and which weighting functions perform best is a topic of further research. However, it must be seen in the contrast to other works in the topic of supergenomes. For example the here presented work of Haussler et al. (2018) and Nguyen, Hickey, Zerbino, et al. (2015), could be represented as parameterized supergenomes with specific designed weight functions.

6.7 Genome Assembly

`Minimap2` (H. Li, 2016, 2018) is the only genome assembler that use superbubbles directly. They use the approach from Onodera, Sadakane, and Shibuya (2013) (as stated independently developed) with a distance limitation to guaranty linear run time. However, there exist several assemblers that uses various similar bubble definitions (Garg, Aach, et al., 2019; Garg, Rautiainen, et al., 2018; Haussler et al., 2018; Minkin and Medvedev, 2019).

Therefore, it is maybe interesting to invest the possibilities to use the novel superbubble detection algorithm in a novel assembler. A superbubble simplifier together with a dead end simplifier creates already a powerful graph simplification tool kit.

Since long error prone reads instead of short high quality reads becomes more default with the new sequencing technologies (Jain et al., 2016; Rhoads and Au, 2015), the de Bruin graph is not well fitting anymore. An alternative for the graph construction could be minimizers (Roberts et al., 2004; Schleimer, Wilkerson, and Aiken, 2003). The idea behind this is using only parts of the reads to create an A-Bruin graph. This graph is used to construct the global structure and solving local errors later in the alignment steps. Note that this idea is similar to H. Li (2016).

If this graph approach is surrounded with powerful filtering, correction, and alignment tools a potent assembly tool can be created. After alignment is mostly used locally to verify the local sequence, a slower but more precise aligner like Kirchner, Retzlaff, and Stadler (2019) could be used. Furthermore, such an exact aligner could use the sequencing quality. Most of the assemblers do not utilize this information. However, considering this in the alignments could dramatically increase the quality of results.

Appendices

APPENDIX A

Supergenome Data Sets

A small data set of different Salmonella strains is created with Cactus (Paten, Earl, et al., 2011). Two larger data sets are downloaded from the UCSC genome browser website. They can be found here: `http://hgdownload.soe.ucsc.edu/downloads.html`

A.1 4way Salmonella

The $\mathbf{B}$ data set is created with Cactus (Paten, Earl, et al., 2011). The genomes of four Salmonella strains are aligned. They are described in Table 2. *Salmonella enterica Newport SL254* is used as reference genome for the alignment.

Table 2: An overview of the sequence data in the Salmonella alignment. In the table the abbreviation "S.e.s." stands "for Salmonella enterica serovar".

Name	Abbreviation	URL	Size	N50
S.e.s. Agona SL483	SenAgo	SenAgo	4836638	4798660
S.e.s. Dublin CT_02021853	SenDub	SenDub	4917459	4842908
S.e.s. Heidelberg SL476	SenHei	SenHei	4983515	4888768
S.e.s. Newport SL254	SenNew	SenNew	5007719	4827641

The Salmonella alignment has $13,416$ blocks. Those blocks contain $50,932$ sequences with a combined total length of $18,047,456$ nucleotides. This corresponds to 91% of the combined genome length of the genomes in the alignment.

The filters described in Section B.1 are applied to the alignment. Statistical results are shown in Table 3. The result are significant different from the other data set due to the alignment method used for this data set. Cactus create no overlap and gives the blocks no score so that this two filters have no effect. On the other hand, cactus creates many small blocks. Thus, the length filter discards 30%

of all blocks. However, the length of the discarded blocks sum up to only 0.4% of the nucletides in the alignment.

Table 3: An overview of the filters for the Salmonella alignment. In the table the abbreviations b for blocks, f for sequence fragments, and n for nucleotides are used.

Filter	b	f	n	b %	f %	n %
none	13416	50932	18047456	100	100	100
length	3986	15080	68141	30	30	0.4
score	0	0	0	0	0	0
overlap	0	0	0	0	0	0
all	3986	15080	68141	30	30	0.4

A.2 7way Yeast

The $\mathbf{Y}$ data set can be found here: `http://hgdownload.soe.ucsc.edu/goldenPath/sacCer3/multiz7way`. It consists of genome assemblies of seven yeast species. They are described in Table 4. *S. cerevisiae* is used as reference genome for the alignment.

Table 4: An overview of the sequence data in the yeast alignment.

Name	Abbreviation	URL	Size	N50
S. cerevisiae	sacCer3	sacCer3	12157105	813184
S. paradoxus	sacPar	sacPar	11872617	46034
S. mikatae	sacMik	sacMik	11470251	19428
S. kudriavzevii	sacKud	sacKud	11132834	11253
S. bayanus	sacBay	sacBay	11477549	24596
S. castelli	sacCas	sacCas	11242286	59644
S. kluyveri	sacKlu	sacKlu	10992590	8690

The yeast alignment has $49,795$ blocks. Those blocks contain $275,484$ sequences with a combined total length of $71,517,259$ nucleotides. This corresponds to 89.01% of the combined genome length of the genomes in the alignment.

The filters described in Section B.1 are applied to the alignment. Statistical results are shown in Table 5.

A.3 27way Insect

The $\mathbf{F}$ data sets can be found here: `http://hgdownload.soe.ucsc.edu/goldenPath/dm6/multiz27way/`. It consists of 27 assemblies of insect species. They are described in Table 6. *D. melanogaster* is used as reference genome for the alignment.

The insect alignment has $2,112,962$ blocks which contain $36,139,620$ sequences. They have a combined length of $2,172,959,429$ nucleotides which correspond to 38.45% of the combined genome length of the genomes in the alignment.

Table 5: An overview of the filters for the yeast alignment. In the table the abbreviations b for blocks, f for sequence fragments, and n for nucleotides are used.

Filter	b	f	n	b %	f %	n %
none	49795	275484	71517259	100	100	100
length	6190	31423	134881	12	11	0
score	110	492	6311	0	0	0
overlap	13458	76141	18446796	27	28	26
all	19758	108056	18587988	40	39	26

The filters described in Section B.1 are applied to the alignments. The statistical result are shown in Table 7.

Table 6: An overview of the sequence data in the Insect alignment. Some of the genomes are in 2bit format for details see here.

Name	Abbreviation	URL	Size	N50
D. melanogaster	dm6	dm6	143726002	25286936
D. simulans	droSim1	droSim1	142420719	22036055
D. sechellia	droSec1	droSec1	166577145	2123299
D. yakuba	droYak3	droYak3	165709965	21770863
D. erecta	droEre2	droEre2	152712140	18748788
D. biarmipes	droBia2	droBia2	169378599	3386121
D. suzukii	droSuz1	droSuz1	232923092	388966
D. ananassae	droAna3	droAna3	230993012	4599533
D. bipectinata	droBip2	droBip2	167263958	663995
D. eugracilis	droEug2	droEug2	156942009	976885
D. elegans	droEle2	droEle2	171267669	1714184
D. kikkawai	droKik2	droKik2	164292578	903682
D. takahashii	droTak2	droTak2	182106768	387676
D. rhopaloa	droRho2	droRho2	197375704	45514
D. ficusphila	droFic2	droFic2	152439475	1050541
D. pseudoobscura	droPse3	droPse3	152696384	12541198
D. persimilis	droPer1	droPer1	188374079	1869541
D. miranda	droMir2	droMir2	136728780	28826359
D. willistoni	droWil2	droWil2	235516348	4511350
D. virilis	droVir3	droVir3	206026697	10161210
D. mojavensis	droMoj3	droMoj3	193826310	24764193
D. albomicans	droAlb1	droAlb1	253560284	23589
D. grimshawi	droGri2	droGri2	200467819	8399593
Musca domestica	musDom2	musDom2	200467819	8399593
Anopheles gambiae	anoGam1	anoGam1	287805703	53272125
Apis mellifera	apiMel4	apiMel4	250287000	13219345
Tribolium castaneum	triCas2	triCas2	199682416	13894384

Table 7: An overview of the filters for the Insect alignment. In the table the abbreviations b for blocks, f for sequence fragments, and n for nucleotides are used.

Filter	b	f	n	b %	f %	n %
none	2112962	36139620	2172959429	100	100	100
length	531676	8362669	30024535	25	23	1
score	25557	384853	4074333	1	1	0
overlap	178882	3134544	285684545	8	9	13
all	736115	11882066	319783413	35	33	15

APPENDIX B

Supergenome Algorithm

B.1 Filter

Noisy data and the alignment procedure make it necessary to curate the input data before calculating the common coordinate system. As part of the curation of input data sets some blocks are filtered out. Here, a more detailed description is given of the three filters already described in the main manuscript. Additionally, some results that explain the chosen parameters is shown.

The filters are used to reduce the noise in the data. The noise has three main sources. Firstly, sequences may align by pure chance on each other. Secondly, `multiz` may produces small blocks that have no or almost no information in it. Thirdly, duplication events lead to sequences which are not different enough to align them unambigous and thus, lead to multiple alignments containing the same sequence for at least one species.

Filters do not discard blocks completely but mark the corresponding blocks and sequences as thrown out. The graph generation algorithms and simplifiers (see below) check the thrown out flag. Blocks and sequences for which the flag is set are skipped.

B.1.1 Length Filter

This filter targets the first and the second source for noisy data. Short blocks created by the aligner can be a result of random sequences. Blocks with a length of at most 10 nucleotides are filtered. This size is used to discard random alignments but no useful information. Since the DNA consists of 4 nucleotides, there are 4^{10} possible sequences of length 10. This are $1.048.576$ possibilities. Even the small yeast data set is 77 times longer than this number. Thus, it is clear that it is very likely that this sequences are the result of random alignments.

Each block has an attribute that shows the length of the corresponding alignment. Each sequence in this block is at most as long as the alignment itself. Sequences might be shorter due to alignment gaps.

B.1.2 Score Filter

`Multiz` may create alignments with bad scores as a side effect of the way how the blocks borders are determined. They likely contain no or not trustworthy information for the common coordinate system. The bad scored blocks are removed with the score filter. In the current implementation, scores are normalized by alignment length and number of aligned sequences. A score of -30 is used as threshold for the filtering out a block.

The alignments scores that are given by the UCSC alignments are calculated on base of the *BlastZ* scoring scheme. So that the score filter is created with this scheme in mind. If another scheme is used, the constants has to adapted or the scores has to recalculated using the BlastZ scoring scheme.

Table 8: The HOXD70 scoring scheme after Chiaromonte, Yap, and W. Miller (2001). In the BlastZ score this used with affine gap scores where the opening penalty are 400 and the extension penalty is 30.

	A	C	G	T
A	91	−114	−31	−123
C	−114	100	−125	−31
G	−31	−125	100	−114
T	−123	−31	−114	91

The `BlastZ` scoring scheme is based on the HOXD70 scoring matrix (Chiaromonte, Yap, and W. Miller, 2001) with affine gap scores. The scoring scheme is shown in Table 8. The gap opening penalty is 400 and gap extension penalty is 30. It is used to compute all pairwise alignment scores. For the scoring of the multiple alignments of the MAF-blocks, the sum-of-pair score is applied, i.e. all pairwise scores are computed and then summed up to form the MAF-Block score.

The pairwise alignment that contains no information is the alignment where both sequences are aligned completely to gaps. The score of such alignment with sequence length n and m would be:

$$\begin{aligned} & -400 - 30 \cdot (n-1) - 400 - 30(m-1) \\ = & -370 - 30 \cdot (n) - 370 - 30(m) \\ = & -740 - 30 \cdot (n+m) \end{aligned} \tag{B.1}$$

The alignment length then is $n+m$. This means that the normalized score of this is:

$$\begin{aligned} & \frac{-740 - 30 \cdot (n+m)}{n+m} \\ = & \frac{-740}{n+m} - 30 \end{aligned} \tag{B.2}$$

It is easy to see that for long sequences such an alignment would have a normalized score of around -30. These is the reason why every block with a score less then -30 most likely contain no useful information. Thus, blocks with a normalized score of at most -30 are discarded.

B.1.3 Overlap Filter

One assumption of the supergenome is that the alignment is injective. However, `multiz` alignments are not injetive. Thus, this feature has to be actively created. This is done by filtering all overlapping sequences. Again, random alignments can be a problem. Especially, the exact start and stop position of an alignment on a genome is often not clear. Therefore, a small number of nucleotides may be used in two alignments. Discarding all overlaps would thus mean a big data loss. In particular, small overlaps are not a problem for the general idea and thus should be kept.

To allow this short overlapping ends, a overlap from up to 20 nucleotides is allowed on both ends. The explanation for this cutoff is again the combiantorical argument. There are 4^{20} possible sequences with 20 nucleotides. This number is larger then the size of the whole insect alignment. Thus, overlaps larger than 20 nucleotides indicate truly duplicated sequences.

Note that sequences in blocks shorter than 20 nucleotides may still overlap completely with other sequences without being filtered out. To avoid this, sequences completly overlapped by other sequences are filtered out. Hereby, sequences may be completely covered by the sequence from the previous block, the next block, or by the previous and next block.

Due to the fact that the sequences on one chromosomes are sorted by their start positions, the calculation of the overlap is a local problem. Overlaps with blocks before the current block and after the current block are calculated independently. The overlap filter marks the sequences as overlapping but not as thrown out to be able to distinguish them from sequences thrown out by previous filtering steps. In a later step, the marked sequences are thrown out.

B.2 Simplifier

B.2.1 Superbubble Simplifier

The superbubble simplifier detect the superbubble as described in Chapter 3. However, the depth-first search (DFS) that is used to create the postorder for the detection is manipulated as stated in Section B.4. This keep a valid DFS postorder but it can directly reused as a topological sorting of the superbubble.

The topological sorting is used to collapse the complete superbubble into one vertex. Where the blocks are ordered after the topological sorting. Therefore, the graph is mutated. Reconsidering that superbubbles can be nested this mutation must be done ordered. With the postorder it is possible to determine the superbubbles that are not nested into an other superbubble. Thus, only these superbubbles are simplified.

B.2.2 Dead End Simplifier

The optimal position of sinks and source are behind and before the predecessor and successor, respectively. Since only sinks and sources with one neighbor considered, the simplifier can determine in constant time if a vertex is a dead end. Thus, simply every vertex is checked.

B.2.3 Mini-Cycle Complex Remover

The mini-cycle complex remover work in four steps. In the first one, all mini-cycles are detected. This is relative simple task because for every neighbor pair, it only must be checked if edges in both directions exist. The second step is then to connect this pairs to complexes. This can be accomplished in linear time by using effective indices. Namely to make it possible to access for every vertex the pairs that contain this vertex in constant time.

The third step is by far the most complex part. The solving of the complex, means to determine for every pair a direction. This is describe in more detail below. The fourth step is then to remove every edge that contradicts the partial order that represents the before determined directions.

Determining the direction is a greedy approach where in every step one pair is solved. If the direction for one pair is decided the orientation of the contradiction parts are reversed. This make sure that the result is as far as possible consistent w.r.t. directions. Thus, one decision influence the other decisions.

Therefore, the starting point is important. The first pair that is used is the pair with the highest support of one direction with respect to the other direction. Namely the pair where the number of edges in one direction is the highest. If more than one pair with the highest number of edges in one direction exist, the one with the higher ratio of edges in the this direction is taken. In more detail, the ratio is calculated as fraction of the more frequent direction and the less frequent direction. The higher the ration, the lower the number of contradicting edges.

From this first pair outgoing the other pairs are decide. In every step the highest supported pair with respect to the before given direction is used. By handling it this way no random breaks are created in the direction. Note that this may use a direction of a pair where the other direction is slightly better supported.

B.3 Minimum Feedback Arc Set problem

The order creation assumes a cycle-free graph. This is done by removing a minimum set of arcs that cause cycles. This *Maximum Acyclic Subgraph* or *Minimum Feedback Arc Set* problem is well-known to be NP-hard (Karp, 1972). It is fixed parameter tractable (FPT) (J. Chen et al., 2008) but APX hard (Kann, 1992). Nevertheless, fast, practicable heuristics have been devised, see e.g. (Eades, X. Lin, and Smyth, 1993; Saab, 2001).

Given the size of our input graphs, it is resort to linear-time heuristics. A modified version of `Algorithm GR` (Eades, X. Lin, and Smyth, 1993) is used because it is known to work particularly well on sparse graphs.

The idea is to create an order π out of the graph then remove all edges that goes from an vertex i to a vertex j if $\pi(j) < \pi(i)$ where $\pi(j)$ is the position of j in the order π. These guaranties an cycle free output graph. The tricky part is to create this order in a way that removes as less edges as possible.

For this, the order is fragmented in two parts, a front π_1 and a back part π_2, where then $\pi = \pi_1\pi_2$. To do so, a vertex removed from the graph and is either append to π_1 or added to the front of π_2. An simple observation is that all sources can be append to π_1 and all sinks can be added to π_2 without losing any edge.

The part where edges are lost is if no sources or sinks are left in the graph. Then a vertex v is removed with a maximal value for $d_{out}(v) - d_{in}(v)$ where d_{out} is the out degree of v and d_{in} is the in degree of v. With other words, it has many outgoing edges and less incoming edges. When v is append to π_1 only few edges are removed (incoming edges) and many edges are saved (outgoing edges).

Only one modification to the algorithm is done. In the original `Algorithm GR`, a random vertex with the highest value is used. Here a vertex with a predecessor that is already is append to π_1 if possible. This minimized the number of sources that are created without influencing the performance of the algorithm. In fact, it optimized the results of the heuristic on our data sets (Table 9).

Table 9: An overview of the heuristics that solve the FAS problem. The columns are the different data sets. The rows are the different algorithm. The first row provides the information of the original graphs of the data sets. The absolut number of edges are shown. The percentage of edges that are kept from the original graph are shown in brackets.

	Bacteria	Yeast	Insect
none	12285	160886	24248434
`Algorithm GR`	10860 (88%)	103076 (64%)	14847253 (61%)
`Eades Serialization`	10871 (88%)	105296 (65%)	14913743 (62%)

It has a time complexity of $O(|E|+|V|)$. The organization of the vertex degrees in an effective way is done by a class "VertexCategorizer". This class has all vertices in lists sorted by there value and special functions that handle the removal of a vertex in an effective way. This removal handling is described also in the original paper (Eades, X. Lin, and Smyth, 1993).

B.4 Topological Sorting

The topological sorting is done with a DFS-topological sorting, i.e., a reversed postorder of a DFS-forest that uses the sources as roots. A DFS-topological sorting is chosen because if it is possible, a successor of the vertex v is placed directly after v in the order. Thus, the DFS approach creates better results as if an unrelated vertex is placed between them.

More specifically, the last possible vertex of the sibling order of the succors of v is placed there. It makes sense that this should be the best-supported successor of v, i.e., the successor u for which the most edges (v, u) exists. Therefore, the sibling

order is on the support information. In the way that the last supported successor is visited first and the best-supported successor last.

B.5 Optimization

The optimization has two main components: the validation of the order and the re-ordering. The validation is done in a simple way of checking the neighborhood of every vertex and count the number of violations of the robinson inequation (see 4). The neighborhood is defined by the number of hops (h). This means that a vertex is seen as neighbor if a path from one to the other exists that has at most length h. Consequently, the maximum neighborhood size is k^h with k being the maximal outdegree. So that this validation has a linear time complexity with respect to the number of vertices.

Besides the validation of an exiting order, the evaluation of potential changes is required for an efficient optimization. Therefore, a function is provided that takes the order and a change object as input and returns the number of violations of the robinson in equation taking the changes in the change object int account. This makes it possible to evaluate a movement before it is applied to the order. The re-ordering is done in loop until no changes that optimize the order are found.

First, a list of candidates are created. These candidates are created only once. A candidate consists of two vertices. It is iterated through the candidates and two changes are tried for each of them. The changes are found with the functions "moveToFront" and "swapSiblings". The best non-overlapping changes is finally applied to the order.

The function "moveToFront" performs a move to front change on the order. This means that a block of vertices after the source position is moved to the target position. The moved block is as large as possible i.e. all vertices after the source position is added to the block as long no predecessor is between the target position and the source position.

The second function, "swapSiblings", performs a swap on the order. This means that a block of vertices after the source position is swapped with a block of vertices after the target position. The blocks are as large as possible but connected. Connected means that, beside the first vertex, every other vertex has a predecessor in the block.

APPENDIX C

Supergenome Results

C.1 Data distribution

As a simple statistical quality assessment, the coverage of the reference species of the alignments with alignment blocks is used. To quantify positional effects, the coverage is assessed for sliding windows along the genome. Furthermore, weight the coverage with number of different species in the respective blocks. These can be done before and after the filtering. Plotting the resulting data into the same coordinate system enables a direct comparison of the coverage by the genome graph before and after filtering. It allows to conclude whether filtering affects specific genome positions more than others.

The plots are generated for data set $\mathbf{Y}$ by using *Saccharomyces cerevisiae* as reference and for $\mathbf{F}$ by using *Drosophila melanogaster* as reference. The results show that for data set $\mathbf{Y}$ the coverage is in general high. The filtering leads to local loss of coverage but change nothing on the general high coverage. In the case of data set $\mathbf{F}$, it is different. Here, six genomic regions aleady have a low coverage before filtering (end of 2L,beginning of 2R, end of 3L, begining of 3R, end of X and complete Y). Those genomic regions overlap with repeat regions of the chromosomes. Since repeat regions are very similar to each other and therfore hard to align, it is not surprising to observe a low coverage for those regions if using an alignment technique as done for the data set. For the remaining genome, the result is the same as by the data set $\mathbf{Y}$. It has a high coverage for the graph before filtering and the filtering only lose local coverage.

After the filtering, the input alignments are transfromed into the supergenome graph and further calculations only make use of the supergenome. This graph is then modified in three steps: first run of simplifer, apply FAS algorithm, and second run of simplifer. Four simplfier are applied to the graph in the fist run: the mini-cycle remover, the dead-end simplifier, and the supperbubble simplifier. In the second run, the mini-cycle remover is replaced by the advance dead-end simplifier.

C.2 Graph edit statistic

For every graph, three edit steps are performed. The first run of simplifier, the remove of all cylces (FAS), and the second run of simplifier. All these steps remove edges from the graph. The mini-cycle remover and FAS remove edges to break cycles in the graph. In the remaining cases, edges are lost as a side effect of the merging of connected vertices and the translation of the edges that connect them. The different steps are repeated by the edit tools several times. To get an idea of the consumption time of each tool in every step, a example consumption time is given. This is the result of an example run on a machine with a Intel(R) Xeon(R) CPU with 2.27GHz and enough RAM to fit the problem size in it. However, the exact numbers are not important. It rather provides an impression how long each step takes compared to the other steps. Moreover, it shows that the simplifier can be calculated even for big data sets in a feasible time on a normal machine.

The results are shown in Table 10, 11, and 12. Here, the three steps are indicated by horizontal lines. For every tool, how many vertices and edges have been removed is listed along with the used time and the number of applications (number distinct position at which the tool could be used). For data set $\mathbf{F}$, it is remarkable that most of the compute time is used in one big mini-cycle complex. The complex has 896,082 vertices and it take 10,144 seconds to decompose it.

Table 10: The overview of the graph editors in data set $\mathbf{B}$. In the table the removed vertices (rv), removed edges (re), the time in seconds (time), and the number of applications ($\#$) are shown.

Edit tool	rv	re	time	$\#$
Mini-Cycle Remover	0	2995	0.17	3
Dead-End Simplifier	4	13	0.03	21
Supperbubble Simplifier	5844	20550	0.50	21
FAS	0	1414	0.05	1
Advance Source/Sink Simplifier	67	197	0.05	14
Dead-End Simplifier	3	6	0.01	14
supperbubble Simplifier	546	1436	1.27	14

C.3 Graph properties

The graph editing steps create different temporary graphs. After each step, some core information of the temporary graphs are collected. This are the number of vertices, edges, and connected components (CC), the median in-degree ($deg_{\leftarrow}$) and median out-degree ($deg_{\rightarrow}$), the mean number of blocks in vertices (mean B), and the maximal number of blocks (max B). As a reference, this values is calculated also for the original graph. The three temproary graphs are: the graph after the first run of simplifier (Simplfier 1), the graph after the FAS-algorithm is applied (FAS), and the graph after the second run of simplifier (Simplfier 2).

Table 11: The overview of the graph modifications in data set $\mathbf{Y}$ In the table the removed vertices (rv), removed edges (re), the time in seconds (time), and the number of applications (#) are shown.

Edit tool	rv	re	time	#
Mini-Cycle Remover	0	62591	1.3	5
Dead-End Simplifier	30	51	0.088	25
supperbubble Simplifier	23087	74977	1.3	25
FAS	0	1888	0.07	1
Advance Dead-End Simplifier	125	465	0.17	9
Dead-End Simplifier	143	346	0.014	9
supperbubble Simplifier	2257	7396	0.63	9

Table 12: The overview of the graph modifications in data set $\mathbf{F}$. In the table the removed vertices (rv), removed edges (re), the time in seconds (time), and the number of applications (#) are shown.

Edit tool	rv	re	time	#
Mini-Cycle Remover	0	10620262	10439.68	10
Dead-End Simplifier	41	99	12.54	48
supperbubble Simplifier	1033047	10282577	397.38	48
FAS	0	55286	6	1
Advance Dead-End Simplifier	3505	25258	17.42	18
Dead-End Simplifier	1641	7243	2.35	18
supperbubble Simplifier	85002	842444	96.91	18

The data is shown in Table 13, 14, and 15. In case of data set $\mathbf{B}$, the second connected component is the third plasmid of *Salmonella enterica Newport SL254* that is with less then 4k nucleotide very small and is aligned with nothing. In case of data set $\mathbf{F}$, the second connected component is created by the FAS algorithm. That can happened since the FAS algorithm is a heuristic. However, it contains only a handful vertices and is simplified to one vertex in the next step.

Table 13: The overview of the graphs in data set $\mathbf{B}$.

Graph	Vertexes	Edges	CC	$deg_{\leftarrow}$	$deg_{\rightarrow}$	mean B	max B
Orignal	9430	35843	2	4	4	1.00	1
Simplfier 1	3582	12285	2	4	4	2.63	133
FAS	3582	10871	2	3	3	2.63	133
Simplfier 2	2966	9232	2	3	3	3.18	133

C.4 Successor statistic

Most sequences have one successor sequence in the data set. These successor sequence is the sequence that follows the sequence on the contig. Note that

Table 14: The overview of the graphs in data set $\mathbf{Y}$.

Graph	Vertexes	Edges	CC	$deg_{\leftarrow}$	$deg_{\rightarrow}$	mean B	max B
Orignal	30580	164114	1	6	6	1.00	1
Simplfier 1	7463	26495	1	4	4	4.10	102
FAS	7463	24607	1	3	3	4.10	102
Simplfier 2	4938	16400	1	3	3	6.19	122

Table 15: The overview of the graphs in data set $\mathbf{F}$.

Graph	Vertexes	Edges	CC	$deg_{\leftarrow}$	$deg_{\rightarrow}$	mean B	max B
Orignal	1383449	24382316	1	18	18	1.00	1
Simplfier 1	350361	3479378	1	10	10	3.95	878
FAS	350361	3424092	2	10	10	3.95	878
Simplfier 2	260213	2549147	2	10	10	5.32	1054

successor depends on the reading direction of the sequence. If it is negative, then the successor sequence is in the contig reading direction direct before the sequence. If the reading direction is positive, the successor is the sequence that is in the contig reading direction direct behind the sequence. These sequences belong to blocks that are contained in the resulting order. So, the distance in blocks between the sequence itself and it's successor can be calculated using the order. The distance is calculated in form that the number of blocks between them is counted. So if they are adjacent they have a distance of zero. The data is disassembled in positive distances where the successor block is behind and in negative distances where the successor is before the sequence. Since the contig reading direction is arbitrary, the reading direction is used to create more positive distances for a contig.

The data is shown in Table 16, 17, and 18. The data is binned in different distance ranges. Different ranges and all successors are evaluted for each direction. The absolute numbers as well as the fraction of all successors in the data set is given.

C.5 ORF statistic

ORFs provide an estimate how much biological entities are retained in the correct order. Thus, how many ORFs are kept together in the final order is calculated. These is reasoned by the idea that ORFs should not be fragmented in the process as long the genes have the same biological functions in the species in the alignment. The ORFs are obtained from the NCBI annotations of the reference species. For data set $\mathbf{F}$, not the complete ORFs but exons are used.

To measure how fragmented a ORF is, the distances between all adjacent blocks in the ORF are add up to the total size of the gaps between the blocks of an ORF. The gap sizes of the ORFs are then binned. Furthermore, for each bin, the number of broken ORFs and the number of ORFs with more than one block are counted. Broken means that the relative order of the blocks is not kept in the genome order.

Table 16: The succesor distances in data set $\mathbf{B}$.

Direction	Distance	Absolute	Fraction
$+$	all	31863	0.8889601874843066
$+$	0	28713	0.8010769187846999
$+$	$1-5$	1074	0.029964009709008733
$+$	$6-20$	331	0.009234718076053902
$+$	$21-100$	341	0.009513712579862176
$+$	$101-1000$	375	0.010462293892810311
$+$	>1000	1029	0.028708534441871495
$-$	all	3980	0.11103981251569343
$-$	0	2570	0.07170158747872667
$-$	$1-5$	306	0.008537231816533214
$-$	$6-20$	68	0.0018971626258962 7
$-$	$21-100$	122	0.0034037329464609548
$-$	$101-1000$	187	0.005217197221214742
$-$	>1000	727	0.02028290042686159

Table 17: The succesor distances in data set $\mathbf{Y}$.

Direction	Distance	Absolute	Fraction
$+$	all	154113	0.9310276082885277
$+$	0	139707	0.843998066815683
$+$	$1-5$	7018	0.04239714855313236
$+$	$6-20$	1551	0.009369902736664049
$+$	$21-100$	981	0.00592641817193258
$+$	$101-1000$	946	0.005714976137256087
$+$	>1000	3910	0.023621095873859722
$-$	all	11417	0.06897239171147224
$-$	0	7009	0.04234277774421555
$-$	$1-5$	1092	0.006596991481906603
$-$	$6-20$	221	0.0013351054189572886
$-$	$21-100$	242	0.0014619706397631849
$-$	$101-1000$	285	0.0017217422823657344
$-$	>1000	2568	0.01551380414426388

Thus, a block exists in the ORF where both adjacent blocks are before it or both are after it in the order.

The results are shown in Table 19, 20, and 21.

C.6 Betweeness statistic

The quality of the result for the *Directed Colored Multigraph Betweenness Problem* is difficult to measure. To solve this, the results of different method and steps are compared. Three orders are compared or, to be more accurate, the graphs that

Table 18: The succesor distances in data set $\mathbf{F}$.

Direction	Distance	Absolute	Fraction
$+$	all	16950668	0.6951258995630639
$+$	0	16046256	0.6580370836488103
$+$	$1-5$	543304	0.022280224103038943
$+$	$6-20$	194421	0.007972964399925151
$+$	$21-100$	44183	0.0018118901048852383
$+$	$101-1000$	21545	$8.835337643381496E-4$
$+$	>1000	100959	0.004140203542066152
$-$	all	7434365	0.304874100436936
$-$	0	6912515	0.28347367830094794
$-$	$1-5$	285028	0.011688645243990443
$-$	$6-20$	117550	0.004820579902434416
$-$	$21-100$	26756	0.0010972304199875391
$-$	$101-1000$	13254	$5.435301235803125E-4$
$-$	>1000	79262	0.0032504364459953776

Table 19: The distribution of the total gap sizes between blocks of annotated ORF in data set $\mathbf{B}$.

Gap size	Absolute	Fraction	Sum of Fractions	Broken	>1 Blocks
0	4806	0.9578	0.9578	0	882
$1-100$	104	0.0207	0.9785	2	104
>100	108	0.0215	1.0000	48	108

Table 20: The distribution of the total gap sizes between blocks of annotated ORF in data set $\mathbf{Y}$.

Gap size	Absolute	Fraction	Sum of Fractions	Broken	>1 Blocks
0	5474	0.9030	0.9030	0	4387
$1-100$	272	0.0449	0.9479	2	272
>100	316	0.0521	1.0000	71	316

with the corresponding order are the solution to the *Directed Colored Multigraph Betweenness Problem*. As reference, the original graph is part of the comparison. The vertex set is thus the same for all graphs and only the number of edges and the number of triples that are created out of the graph with the definition in *Directed Colored Multigraph Betweenness Problem* differs. These are given as a absolute number and as fraction of the original graph. The results are shown in Table 22, 23, and 24.

Table 21: The distribution of the total gap sizes between blocks of annotated exons in data set $\mathbf{F}$.

Gap size	Absolute	Fraction	Sum of Fractions	Broken	> 1 Blocks
0	158956	0.9515	0.9515	0	107594
$1 - 100$	2387	0.0143	0.9658	161	2387
> 100	5708	0.0342	1.0000	618	5708

Table 22: The comparison of different betweenness graphs for data set $\mathbf{B}$. In the table stand "edge" for the number of edges and "BT" for the number of fulfilled betweenness triples. Both are also given as fraction (edge f. and BT f.).

Graph	edge	edge f.	BT	BT f.
Orignal graph	35843	1.0000	35835	1.0000
FAS graph	33438	0.9329	31185	0.8702
Before optimization graph	34123	0.9520	32534	0.9079
Betweenness graph	34146	0.9527	32578	0.9091

Table 23: The comparison of different betweenness graphs for data set $\mathbf{Y}$. In the table stand "edge" for the number of edges and "BT" for the number of fulfilled betweenness triples. Both are also given as fraction (edge f. and BT f.).

Graph	edge	edge f.	BT	BT f.
Orignal graph	164113	1.0000	158098	1.0000
FAS graph	134567	0.8200	103126	0.6523
Before optimization graph	159911	0.9744	150151	0.9497
Betweenness graph	159996	0.9749	150308	0.9507

Table 24: The comparison of different betweenness graphs for data set $\mathbf{F}$. In the table stand "edge" for the number of edges and "BT" for the number of fulfilled betweenness triples. Both are also given as fraction (edge f. and BT f.).

Graph	edge	edge f.	BT	BT f.
Orignal graph	24382316	1.0000	24373551	1.0000
FAS graph	21164222	0.8680	18264760	0.7494
Before optimization graph	24236459	0.9940	24086318	0.9882
Betweenness graph	24237938	0.9941	24089235	0.9883

APPENDIX D

Superbubbles Results

Table 25: The supergenome data set and results. All data sets are downloaded from `http://hgdownload.cse.ucsc.edu/downloads.html`. The used graphs are the "orginal" graphs, i.e., after the filtering but before any simplifier is applied. The superbubbles are presented in normalized form, i.e., the number of superbubbles is divided by the number of vertices in the graph.

Data Set	Name	Superbubbles	Complex Superbubbles
1	canfam1_3	0.415181	0.28936
2	felcat3_4	0.405645	0.345149
3	canfam2_4	0.311271	0.196865
4	mm5_5	0.301006	0.223376
5	geofor1_7	0.16144	0.151035
6	mm9_30	0.0500894	0.0385938
7	tarsyr2_20	0.0185623	0.0154
8	hg38_20	0.00596719	0.00453172

Table 26: The Stanford Large Network Dataset Collection data sets and results. All data sets are downloaded from `http://snap.stanford.edu/data/index.html`. The superbubbles are presented in normalized form, i.e., the number of superbubbles is divided by the number of vertices in the graph.

Data Set	Name	Superbubbles	Complex Superbubbles
1	email-EuAll	0.100183	0.0
2	soc-sign-epinions	0.0897078	0.000204812
3	sx-askubuntu	0.0271159	1.88305e-05
4	soc-Epinions1	0.021587	3.95366e-05
5	p2p-Gnutella04	0.0183891	0.0
6	sx-superuser	0.0171523	0.0
7	p2p-Gnutella09	0.0158984	0.0
8	web-Google	0.015472	0.00253964
9	p2p-Gnutella05	0.0140176	0.0
10	p2p-Gnutella08	0.013966	0.0
11	p2p-Gnutella06	0.011816	0.0
12	web-NotreDame	0.00959693	0.000273233
13	p2p-Gnutella24	0.00874877	0.0
14	p2p-Gnutella31	0.00829259	0.0
15	cit-HepPh	0.00787356	0.000636832
16	p2p-Gnutella25	0.00749328	0.0
17	wiki-Vote	0.00646521	0.0
18	web-Stanford	0.00615105	0.000205745
19	p2p-Gnutella30	0.00588845	0.0
20	cit-Patents	0.00499368	3.5234e-05
21	wiki-Talk	0.00395049	1.25293e-06
22	sx-mathoverflow	0.00330405	0.0
23	CollegeMsg	0.00315956	0.0
24	soc-sign-bitcoinalpha	0.0026434	0.0
25	soc-LiveJournal1	0.00223225	2.08352e-05
26	wiki-talk-temporal	0.00219533	0.0
27	email-Eu-core-temporal	0.0020284	0.0
28	soc-pokec-relationships	0.00179385	3.67466e-06
29	soc-sign-bitcoinotc	0.00136031	0.0
30	amazon0312	0.000114791	0.0
31	amazon0302	9.91946e-05	0.0
32	amazon0505	8.77544e-05	0.0
33	amazon0601	1.48738e-05	0.0
34	soc-Slashdot0902	0.0	0.0
35	twitter_combined	0.0	0.0
36	soc-Slashdot0811	0.0	0.0
37	gplus_combined	0.0	0.0

Table 27: The LDBC Graphalytics data sets and results. All data sets are downloaded from `https://graphalytics.org/datasets`. The superbubbles are presented in normalized form, i.e., the number of superbubbles is divided by the number of vertices in the graph.

Data Set	Name	Superbubbles	Complex Superbubbles
1	datagen-8_3-zf	0.0603148	0.000101354
2	datagen-7_7-zf	0.0571091	9.20298e-05
3	datagen-7_8-zf	0.0566572	8.34045e-05
4	datagen-8_2-zf	0.0558878	8.48301e-05
5	graph500-22	0.00373145	0.0
6	graph500-23	0.00356425	0.0
7	graph500-24	0.00344597	0.0
8	datagen-8_1-fb	3.86079e-06	0.0
9	datagen-8_4-fb	3.15036e-06	0.0
10	datagen-8_0-fb	1.17195e-06	0.0
11	datagen-7_5-fb	0.0	0.0
12	datagen-7_6-fb	0.0	0.0
13	datagen-7_9-fb	0.0	0.0

Table 28: The simulated data sets and results. All data sets are simulated directed graphs. "BA" graphs are created with the Barabasi–Albert model. "ER" graphs are created with the Erdoes–Renyi model. The number gives the percent of the edge probability. "WS" graphs are created with the Watts–Strogatz model. The number given the percent of rewiring probability. The superbubbles are presented in normalized form, i.e., the number of superbubbles is divided by the number of vertices in the graph.

Data Set	Name	Superbubbles	Complex Superbubbles
1	BA	0.31112	0.0
1	BA	0.31063	0.0
1	BA	0.30892	0.0
1	ER_001	0.0	0.0
1	ER_001	0.0	0.0
1	ER_001	0.0	0.0
1	ER_005	0.0	0.0
1	ER_005	0.0	0.0
1	ER_005	0.0	0.0
1	ER_010	0.0	0.0
1	ER_010	0.0	0.0
1	ER_010	0.0	0.0
1	WS_001	0.0	0.0
1	WS_001	0.0	0.0
1	WS_001	0.0	0.0
1	WS_005	0.0	0.0
1	WS_005	0.0	0.0
1	WS_005	0.0	0.0
1	WS_010	0.0	0.0
1	WS_010	0.0	0.0
1	WS_010	0.0	0.0

List of Symbols

Bubbles

$\tilde{G}_C$ The augmented graph of a component C that contains the same weak superbubbles as G that are contained in C.

$\hat{G}_C$ The augmented directed acyclic graph (DAG) of a component C that contains the same weak superbubbles as $\tilde{G}_C$.

$\prec s, t \succ$ A bubble with the entrance s and the exit t

$\mathbf{OutChild}(v)$ The maximal position of a child of v in a inverse postorder of a DFS-tree.

$\mathbf{OutParent}(v)$ The minimal position of a parent of v in a inverse postorder of a DFS-tree.

$\langle s, t \rangle$ A superbubble with the entrance s and the exit t

Cycles

$d_C(c_i, c_j)$ The cycle distance from c_i to c_j on the cycle C

$\max_c(v)$ The vertex $u \in C$ that is $v \overset{a}{\rightsquigarrow} u$nd $d_C(c, u)$ is maximal.

$\min_c(v)$ The vertex $u \in C$ that is $v \overset{a}{\rightsquigarrow} u$nd $d_C(c, u)$ is minimal.

$Q(C)$ The set of all $\overset{C}{\rightsquigarrow}$-covered vertices of the cycle C.

$K(C)$ The set of all $\overset{C}{\rightsquigarrow}$-cut vertices of the cycle C.

$\mathfrak{Q}(C)$ The set of all $\overset{C}{\rightsquigarrow}$-covered intervals of the cycle C.

$\ell(v)$ The lowlink property of Tarjan's strongly connected component (SCC) algorithm.

$\mathfrak{L}(C)$ The set of all $\overset{C}{\rightsquigarrow}$-covered intervals of C for which there is no larger $\overset{C}{\rightsquigarrow}$-covered interval with the same starting point.

$\mathfrak{B}$ The cover of a cycle.

C A cycle.

Graph

$\overline{A}_{/symG}$ The acyclic component of the graph G.

$\mathfrak{P}_{/symG}$ A set of every non singleton SCC and the acyclic component of the graph G.

$v \sim u$ The vertex v is connected with the vertex u

$[v,u]_{\rightsquigarrow}$ A set of every vertex that can be reach to v whiteout passing through u.

$[v,u]_{\rightsquigarrow}$ A set of every vertex that can be reached from v whiteout passing through u.

$v \overset{x}{\rightsquigarrow} u$ The constrained reach relation. This means that v can reach u without passing through x, i.e., a path $p = v \to u$ exists with $p \cap \{x\} = \emptyset$

(v,u) An edge from v to u

$E(G)$ The vertex edge of G

$V(e)$ The vertices that are connected by the edge e

$\mathrm{head}(e)$ The head vertex of the edge e, i.e., the vertex at which the edge points.

$\mathrm{indeg}(v)$ The in-degree of the vertex v

$G[U]$ Creates induced subgraph of G with vertex set U

$\mathrm{neighbor}(v)$ The neighbors of the vertex v

$v \not\overset{2}{\rightsquigarrow} u$ The negated 2-reach relation. This means that v has at most one vertex-independent path to u

$\mathrm{outdeg}(v)$ The out-degree of the vertex v

$v \to u$ A path from v to u, i.e., a order set $\{v, \ldots, u\}$

$\mathrm{pre}(v)$ The predecessor of the vertex v

$\mathrm{property}(v,k)$ Access the information of a property k on vertex v

$v \rightsquigarrow u$ The vertex v can reach the vertex u

$[v]_{\rightsquigarrow}$ A set of every vertex that can be reached from v and v itself.

$\mathfrak{S}_G$ A set of every non singleton SCC of the graph G.

$H \subseteq G$ H is a subgraph of G

$\mathrm{suc}(v)$ The successor of the vertex v

$\mathrm{tail}(e)$ The tail vertex of the edge e, i.e., the vertex at which the edge starts.

$v \twoheadrightarrow u$ The 2-reach relation. This means that v has at least two vertex-independent paths to u

$V(G)$ The vertex set of G

$\mathbf{A}$ The incidence matrix of a digraph G. The rows are the vertices and the columns are the edges. In the column of edge e is in the row of $\mathrm{head}(e)$ a -1 and in the row of $\mathrm{tail}(e)$ a 1.

G A graph.

S An element of $\mathfrak{P}_G$.

Complexity

$\mathcal{O}(n)$ The complexity of a function. The complexity is linear dependent of n.

NP The complexity class "nondeterministic polynomial time".

Order

$[a \lesssim b \lesssim c]$ The betweenness order relation. This means that b is between a and c.

$[a \leq b \leq c]$ The cycle order relation. This means that relative to a, b comes for c.

$\circlearrowleft c_1, \ldots, c_k \circlearrowright$ The shorthand notion for a cyclic set

$[n]$ The set of the first n natural numbers.

$\{x \mid x \in \mathbb{N} \wedge 2 < x < 6\}$ The set with all natural numbers that are greater two and smaller 6, i.e., $3, 4, 5$.

$\varpi[3\!:\!10]$ The interval from the third to the tens element of the ordered set (X, ϖ), i.e. $\{x \mid x \in X \wedge 3 \leq \varpi(x) \leq 10\}$. When ϖ is the identity function it can be skipped. Thus $[3\!:\!5]$ are the natural number $3, 4, 5$.

ϖ^{-1} The inverse function of the total order ϖ. Thus, it gets for a position the corresponding element.

$\varpi(3\!:\!10)$ The open interval from the third to the tens element of the ordered set (X, ϖ), i.e. $\{x \mid x \in X \wedge 3 < \varpi(x) < 10\}$.

$a \leq b$ The order relation. This means that a come for b.

(X, ϖ) The ordered set with X as set and $\varpi\colon X \to \mathbb{N}$ as order defining bijection

$\overline{\varpi}$ The reverse total order of the total order ϖ.

$(1, 2, \ldots, 100)$ The sequence that contains $[1\!:\!100]$

$\{3, 4, 5\}$ The set with $3, 4, 5$ as elements. This can also used as shorthand notion for a ordered set

ϖ A bijection that represents an order.

Supergenome

$B[k]$ The column k of genome-wide multiple sequence alignment (gMSA) block B.

$\mathrm{columns}(B)$ All columns of gMSA block B.

$\tau_{\mathcal{G}}(B)$ The projection that extracts from an alignment block B the sequence intervals belonging to assembly $\mathcal{G}$

$(\mathcal{G}, c, i, j, \sigma)$ A sequence interval from positions i to j on contig c of genome assembly $\mathcal{G}$ with reading direction σ.

$\mathscr{C}(\mathfrak{A})$ The ternary relation $(A, C, B) \in \mathscr{C}(()\mathfrak{A})$ whenever C is between A and B for some assembly $\mathcal{G}$

Γ The supergenome graph.

γ The total order of the supergenome graph.

$\mathbf{B}$ The bacteria dataset.

$\mathbf{F}$ The fly dataset.

$\mathbf{Y}$ The yeast dataset.

$\mathcal{G}$ A genome assembly of a species, often referenced as genome.

$\mathfrak{A}$ A gMSA.

$\mathfrak{C}$ A mini-cycle complex.

σ The reading direction of an interval.

d A Robinsonian dissimilarity measure.

Tree

$v \prec u$ The ancestor partial order of a tree.

$\mathrm{lca}(v, u)$ The least common ancestor of the tree vertices v and u.

$\mathrm{parent}(v)$ The parent of the tree vertex v, i.e., the vertex that have v as successor.

$\mathrm{root}(T)$ The root vertex of the tree T.

$v \lhd u$ The sibling partial order of a tree.

$T(v)$ The subtree of T rooted at v

$p_T(v)$ The path from $\mathrm{root}(T)$ to the tree vertex v in T.

π The postorder of a tree.

ρ The preorder of a tree.

F A forest.

T A tree.

List of Abbreviations

BFS breadth-first search.

C1S simultaneous consecutive ones property.

DAG directed acyclic graph.

DFS depth-first search.

DNA deoxyribonucleic acid.

gMSA genome-wide multiple sequence alignment.

MFAS minimum feedback arc set problem.

MSA multiple sequence alignment.

ORF open reading frame.

RNA ribonucleic acid.

SCC strongly connected component.

SSP supergenome sorting problem.

TCA tricaboxylic acid.

Definition Index

Bibliography

1000 Genomes Project Consortium (Sept. 2015). "A global reference for human genetic variation". In: *Nature* 526.7571, pp. 68–74. *DOI*: 10.1038/nature15393.

Acuña, V., R. Grossi, G. F. Italiano, L. Lima, R. Rizzi, G. Sacomoto, M.-F. Sagot, and B. Sinaimeri (2017). "On Bubble Generators in Directed Graphs". In: *Graph-Theoretic Concepts in Computer Science, 43rd WG*. Ed. by H. L. Bodlaender and G. J. Woeginer. Vol. 10520. Lecture Notes Comp. Sci. Heidelberg: Springer, pp. 18–31. *DOI*: 10.1007/978-3-319-68705-6_2.

Altschul, S. F., W. Gish, W. Miller, E. W. Myers, and D. J. Lipman (Oct. 1990). "Basic local alignment search tool". In: *Journal of Molecular Biology* 215.3, pp. 403–410. *DOI*: 10.1016/s0022-2836(05)80360-2.

Arnold, C., F. Externbrink, J. Hackermüller, and K. Reiche (Nov. 2014). "CEM-Designer: Design of custom expression microarrays in the post-ENCODE Era". In: *Journal of Biotechnology* 189, pp. 154–156. *DOI*: 10.1016/j.jbiotec.2014.09.012.

Balister, P., S. Gerke, G. Gutin, A. Johnstone, J. Reddington, E. Scott, A. Soleimanfallah, and A. Yeo (Dec. 2009). "Algorithms for generating convex sets in acyclic digraphs". In: *Journal of Discrete Algorithms* 7.4, pp. 509–518. *DOI*: 10.1016/j.jda.2008.07.008.

Barabási, A.-L. and R. Albert (1999). "Emergence of Scaling in Random Networks". In: *Science* 286.5439, pp. 509–512. *ISSN*: 0036-8075. *DOI*: 10.1126/science.286.5439.509. eprint: https://science.sciencemag.org/content/286/5439/509.full.pdf.

Barth, D., F. Pellegrini, A. Raspaud, and J. Roman (1995). "On bandwidth, cutwidth, and quotient graphs". In: *RAIRO - Theoretical Informatics and Applications* 29.6, pp. 487–508. *DOI*: 10.1051/ita/1995290604871.

Belda, E., A. Moya, and F. J. Silva (Mar. 2005). "Genome Rearrangement Distances and Gene Order Phylogeny in γ-Proteobacteria". In: *Molecular Biology and Evolution* 22.6, pp. 1456–1467. *DOI*: 10.1093/molbev/msi134.

Bernt, M., A. Donath, F. Jühling, F. Externbrink, C. Florentz, G. Fritzsch, J. Pütz, M. Middendorf, and P. F. Stadler (Nov. 2013). "MITOS: Improved de novo metazoan mitochondrial genome annotation". In: *Molecular Phylogenetics and Evolution* 69.2, pp. 313–319. *DOI*: 10.1016/j.ympev.2012.08.023.

Bertrand, D., M. Blanchette, and N. El-Mabrouk (Oct. 2009). "Genetic Map Refinement Using a Comparative Genomic Approach". In: *Journal of Computational Biology* 16.10, pp. 1475–1486. *DOI*: 10.1089/cmb.2009.0094.

Bertrand, P. (Apr. 2008). "Systems of sets such that each set properly intersects at most one other set—Application to cluster analysis". In: *Discrete Applied Mathematics* 156.8, pp. 1220–1236. *DOI*: 10.1016/j.dam.2007.05.023.

Bertrand, P. and J. Diatta (May 2017). "Multilevel clustering models and interval convexities". In: *Discrete Applied Mathematics* 222, pp. 54–66. *DOI*: 10.1016/j.dam.2016.12.019.

Blanchette, M. et al. (2004). "Aligning Multiple Genomic Sequences With the Threaded Blockset Aligner". In: *Genome Research* 14.4, pp. 708–715. *DOI*: 10.1101/gr.1933104.

Bodlaender, H. L., F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos (Jan. 2011). "A Note on Exact Algorithms for Vertex Ordering Problems on Graphs". In: *Theory of Computing Systems* 50.3, pp. 420–432. *DOI*: 10.1007/s00224-011-9312-0.

Bonfield, J. K., K. F. Smith, and R. Staden (1995). "A new DNA sequence assembly program". In: *Nucleic Acids Research* 23.24, pp. 4992–4999. *DOI*: 10.1093/nar/23.24.4992.

Booth, K. S. and G. S. Lueker (Dec. 1976). "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms". In: *Journal of Computer and System Sciences* 13.3, pp. 335–379. *DOI*: 10.1016/s0022-0000(76)80045-1.

Brankovic, L., C. S. Iliopoulos, R. Kundu, M. Mohamed, S. P. Pissis, and F. Vayani (Jan. 2016). "Linear-time superbubble identification algorithm for genome assembly". In: *Theoretical Computer Science* 609, pp. 374–383. *DOI*: 10.1016/j.tcs.2015.10.021.

Bray, N. and L. Pachter (Apr. 2004). "MAVID: Constrained Ancestral Alignment of Multiple Sequences". In: *Genome Research* 14.4, pp. 693–699. *DOI*: 10.1101/gr.1960404.

Brudno, M., C. B. Do, G. M. Cooper, M. F. Kim, E. Davydov, N. C. S. Program, E. D. Green, A. Sidow, and S. Batzoglou (Mar. 2003). "LAGAN and Multi-LAGAN: Efficient Tools for Large-Scale Multiple Alignment of Genomic DNA". In: *Genome Research* 13, pp. 721–731. *DOI*: 10.1101/gr.926603.

Burr, S. A. (1984). "Some undecidable problems involving the edge-coloring and vertex-coloring of graphs". In: *Discrete Mathematics* 50, pp. 171–177. *DOI*: 10.1016/0012-365x(84)90046-3.

Byrne, K. P. and K. H. Wolfe (Sept. 2005). "The Yeast Gene Order Browser: Combining curated homology and syntenic context reveals gene fate in polyploid species". In: *Genome Research* 15.10, pp. 1456–1461. *DOI*: 10.1101/gr.3672305.

Capotă, M., T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz (2015). "Graphalytics". In: *Proceedings of the GRADES'15 on - GRADES'15*. ACM Press. *DOI*: 10.1145/2764947.2764954.

Chen, J., Y. Liu, S. Lu, B. O'sullivan, and I. Razgon (Oct. 2008). "A fixed-parameter algorithm for the directed feedback vertex set problem". In: *Journal of the ACM* 55.5, pp. 1–19. *DOI*: 10.1145/1411509.1411511.

Chen, X. and M. Tompa (May 2010). "Comparative assessment of methods for aligning multiple genome sequences". In: *Nature Biotechnology* 28.6, pp. 567–572. *DOI*: 10.1038/nbt.1637.

Chiaromonte, F., V. B. Yap, and W. Miller (Dec. 2001). "SCORING PAIRWISE GENOMIC SEQUENCE ALIGNMENTS". In: *Biocomputing 2002*. WORLD SCIENTIFIC, pp. 115–126. *DOI*: 10.1142/9789812799623_0012.

Chor, B. and M. Sudan (Nov. 1998). "A Geometric Approach to Betweenness". In: *SIAM Journal on Discrete Mathematics* 11.4, pp. 511–523. *DOI*: 10.1137/s0895480195296221.

Christof, T., M. Oswald, and G. Reinelt (1998). "Consecutive Ones and a Betweenness Problem in Computational Biology". In: *Integer Programming and Combinatorial Optimization*. Ed. by R. E. Bixby, E. A. Boyd, and R. Z. Ríos-Mercado. Springer Berlin Heidelberg, pp. 213–228. *DOI*: 10.1007/3-540-69346-7_17.

Chvátal, V. and B. Wu (Sept. 2011). "On Reichenbach's Causal Betweenness". In: *Erkenntnis* 76.1, pp. 41–48. *DOI*: 10.1007/s10670-011-9321-z.

Ciccarelli, F. D., T. Doerks, C. von Mering, C. J. Creevey, B. Snel, and P. Bork (Mar. 2006). "Toward Automatic Reconstruction of a Highly Resolved Tree of Life". In: *Science* 311.5765, pp. 1283–1287. *DOI*: 10.1126/science.1123061.

Collier, J. H. and A. S. Konagurthu (Oct. 2014). "An Information Measure for Comparing Top k Lists". In: *2014 IEEE 10th International Conference on e-Science*. IEEE. *DOI*: 10.1109/escience.2014.39.

Cuthill, E. and J. McKee (1969). "Reducing the Bandwidth of Sparse Symmetric Matrices". In: *Proceedings of the 1969 24th National Conference*. ACM '69. New York, NY, USA: ACM, pp. 157–172. *DOI*: 10.1145/800195.805928.

Darling, A. E., B. Mau, and N. T. Perna (June 2010). "progressiveMauve: Multiple Genome Alignment with Gene Gain, Loss and Rearrangement". In: *PLoS ONE* 5.6. Ed. by J. E. Stajich, e11147. *DOI*: 10.1371/journal.pone.0011147.

Dayhoff, M., R. Schwartz, and B. Orcutt (1978). "22 a model of evolutionary change in proteins". In: *Atlas of protein sequence and structure*. Vol. 5. National Biomedical Research Foundation Silver Spring, pp. 345–352.

De Bruijn, N. G. (1946). "A combinatorial problem". In: *Koninklijke Nederlandse Akademie v. Wetenschappen* 49, pp. 758–764. *URL*: http://www.dwc.knaw.nl/DL/publications/PU00018235.pdf.

Dehal, P. and J. L. Boore (Sept. 2005). "Two Rounds of Whole Genome Duplication in the Ancestral Vertebrate". In: *PLoS Biology* 3.10. Ed. by P. Holland, e314. *DOI*: 10.1371/journal.pbio.0030314.

Dodd, M. S., D. Papineau, T. Grenne, J. F. Slack, M. Rittner, F. Pirajno, J. O'Neil, and C. T. S. Little (Mar. 2017). "Evidence for early life in Earth's oldest hydrothermal vent precipitates". In: *Nature* 543.7643, pp. 60–64. *DOI*: 10.1038/nature21377.

Drillon, G. and G. Fischer (Aug. 2011). "Comparative study on synteny between yeasts and vertebrates". In: *Comptes Rendus Biologies* 334.8-9, pp. 629–638. *DOI*: 10.1016/j.crvi.2011.05.011.

Dugar, G., A. Herbig, K. U. Förstner, N. Heidrich, R. Reinhardt, K. Nieselt, and C. M. Sharma (May 2013). "High-Resolution Transcriptome Maps Reveal Strain-Specific Regulatory Features of Multiple Campylobacter jejuni Isolates". In: *PLoS*

Genetics 9.5. Ed. by D. Hughes, e1003495. *DOI*: 10.1371/journal.pgen.1003495.

Eades, P., X. Lin, and W. F. Smyth (Oct. 1993). "A fast and effective heuristic for the feedback arc set problem". In: *Information Processing Letters* 47.6, pp. 319–323. *DOI*: 10.1016/0020-0190(93)90079-o.

Earl, D. et al. (Oct. 2014). "Alignathon: a competitive assessment of whole-genome alignment methods". In: *Genome Research* 24.12, pp. 2077–2089. *DOI*: 10.1101/gr.174920.114.

Erdös, P. and A. Rényi (1959). "On Random Graphs I". In: *Publicationes Mathematicae Debrecen* 6, p. 290.

Euler, L. (1741). "Solutio problematis ad geometriam situs pertinentis". In: *Commentarii academiae scientiarum Petropolitanae*, pp. 128–140.

Ezawa, K. (Mar. 2016). "Characterization of multiple sequence alignment errors using complete-likelihood score and position-shift map". In: *BMC Bioinformatics* 17.1. *DOI*: 10.1186/s12859-016-0945-5.

Fagin, R., R. Kumar, and D. Sivakumar (Jan. 2003). "Comparing Top k Lists". In: *SIAM Journal on Discrete Mathematics* 17.1, pp. 134–160. *DOI*: 10.1137/s0895480102412856.

Feige, U. (2000). "Coping with the NP-Hardness of the Graph Bandwidth Problem". In: *Algorithm Theory - SWAT 2000*. Springer Berlin Heidelberg, pp. 10–19. *DOI*: 10.1007/3-540-44985-x_2.

Fellows, M. R., D. Hermelin, F. Rosamond, and H. Shachnai (May 2016). "Tractable Parameterizations for the Minimum Linear Arrangement Problem". In: *ACM Transactions on Computation Theory* 8.2, pp. 1–12. *DOI*: 10.1145/2898352.

Fischer, G., E. P. C. Rocha, F. Brunet, M. Vergassola, and B. Dujon (2006). "Highly Variable Rates of Genome Rearrangements between Hemiascomycetous Yeast Lineages". In: *PLoS Genetics* 2.3, e32. *DOI*: 10.1371/journal.pgen.0020032.

Fishburn, P. C. (July 1971). "Betweenness, orders and interval graphs". In: *Journal of Pure and Applied Algebra* 1.2, pp. 159–178. *DOI*: 10.1016/0022-4049(71)90016-8.

Flood, M. M. (Jan. 1990). "Exact and heuristic algorithms for the weighted feedback arc set problem: A special case of the skew-symmetric quadratic assignment problem". In: *Networks* 20.1, pp. 1–23. *DOI*: 10.1002/net.3230200102.

Fried, C., W. Hordijk, S. J. Prohaska, C. R. Stadler, and P. F. Stadler (2004). "The Footprint Sorting Problem". In: *J. Chem. Inf. Comput. Sci.* 44, pp. 332–338. *DOI*: 10.1021/ci030411+.

Friedberg, R., A. E. Darling, and S. Yancopoulos (2008). "Genome Rearrangement by the Double Cut and Join Operation". In: *Bioinformatics*. Vol. 452. Humana Press, pp. 385–416. *DOI*: 10.1007/978-1-60327-159-2_18.

Garg, S., J. Aach, H. Li, R. Durbin, and G. Church (Mar. 2019). "A haplotype-aware de novo assembly of related individuals using pedigree graph". In: *DOI*: 10.1101/580159.

Garg, S., M. Rautiainen, A. M. Novak, E. Garrison, R. Durbin, and T. Marschall (June 2018). "A graph-based approach to diploid genome assembly". In: *Bioinformatics* 34.13, pp. i105–i114. *DOI*: 10.1093/bioinformatics/bty279.

Gärtner, F., C. Höner zu Siederdissen, L. Müller, and P. F. Stadler (Sept. 2018). "Coordinate Systems for Supergenomes". In: *Algorithms for Molecular Biology* 13.1, p. 15. *DOI*: 10.1186/s13015-018-0133-4.

Gärtner, F., L. Müller, and P. F. Stadler (Dec. 2018). "Superbubbles revisited". In: *Algorithms for Molecular Biology* 13.1, p. 16. *DOI*: 10.1186/s13015-018-0134-3.

Gärtner, F. and P. F. Stadler (Apr. 2019). "Direct Superbubble Detection". In: *Algorithms* 12.4. *ISSN*: 1999-4893. *DOI*: 10.3390/a12040081.

Gavril, F. (1977). "Some NP-complete problems on graphs". In: *Proceedings of the 11th conference on Information Sciences and Systems*. Baltimore: Johns Hopkins Univ., pp. 91–95.

Gawad, C., W. Koh, and S. R. Quake (Jan. 2016). "Single-cell genome sequencing: current state of the science". In: *Nature Reviews Genetics* 17.3, pp. 175–188. *DOI*: 10.1038/nrg.2015.16.

Gerstein, M. B., C. Bruce, J. S. Rozowsky, D. Zheng, J. Du, J. O. Korbel, O. Emanuelsson, Z. D. Zhang, S. Weissman, and M. Snyder (June 2007). "What is a gene, post-ENCODE? History and updated definition". In: *Genome Research* 17.6, pp. 669–681. *DOI*: 10.1101/gr.6339607.

Gibbs, N. E., W. G. Poole, and P. K. Stockmeyer (1976). "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix". In: *SIAM Journal on Numerical Analysis* 13.2, pp. 236–250. *ISSN*: 00361429. *URL*: http://www.jstor.org/stable/2156090.

Giegerich, R. (2000). "Explaining and Controlling Ambiguity in Dynamic Programming". In: *Combinatorial Pattern Matching*. Springer Berlin Heidelberg, pp. 46–59. *DOI*: 10.1007/3-540-45123-4_6.

Gogarten, J. P. and J. P. Townsend (Aug. 2005). "Horizontal gene transfer, genome innovation and evolution". In: *Nature Reviews Microbiology* 3.9, pp. 679–687. *DOI*: 10.1038/nrmicro1204.

Goryunov, D. V., B. E. Nagaev, M. Y. Nikolaev, A. V. Alexeevski, and A. V. Troitsky (Nov. 2015). "Moss phylogeny reconstruction using nucleotide pangenome of complete Mitogenome sequences". In: *Biochemistry (Moscow)* 80.11, pp. 1522–1527. *DOI*: 10.1134/s0006297915110152.

Gotoh, O. (Dec. 1982). "An improved algorithm for matching biological sequences". In: *Journal of Molecular Biology* 162.3, pp. 705–708. *DOI*: 10.1016/0022-2836(82)90398-9.

Grötschel, M., M. Jünger, and G. Reinelt (Dec. 1984). "A Cutting Plane Algorithm for the Linear Ordering Problem". In: *Operations Research* 32.6, pp. 1195–1220. *DOI*: 10.1287/opre.32.6.1195.

Hagberg, A., D. A. Schult, and P. Swart (2008). "Exploring network structure, dynamics, and function using NetworkX". In: *Proceedings of the 7th Python in Science Conference (SciPy 2008)*. Ed. by G. Varoquaux, T. Vaught, and J. Millman. scipy.org, pp. 11–16. *URL*: https://conference.scipy.org/proceedings/scipy2008/paper_2/full_text.pdf.

Hahsler, M., K. Hornik, and C. Buchta (2008). "Getting Things in Order: An Introduction to the R Package seriation". In: *Journal of Statistical Software* 25.3. *DOI*: 10.18637/jss.v025.i03.

Hamilton, W. R. (Dec. 1856). "LVI. Memorandum respecting a new system of roots of unity". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 12.81, pp. 446–446. *DOI*: 10.1080/14786445608642212.

Haselbeck, R. J. and L. McAlister-Henn (1993). "Function and expression of yeast mitochondrial NAD- and NADP-specific isocitrate dehydrogenases." In: *Journal of Biological Chemistry* 268.16, pp. 12116–12122. *URL*: http://www.jbc.org/content/268/16/12116.abstract.

Haussler, D., M. Smuga-Otto, J. M. Eizenga, B. Paten, A. M. Novak, S. Nikitin, M. Zueva, and D. Miagkov (July 2018). "A Flow Procedure for Linearization of Genome Sequence Graphs". In: *Journal of Computational Biology* 25.7, pp. 664–676. *DOI*: 10.1089/cmb.2017.0248.

Henikoff, S. and J. G. Henikoff (Nov. 1992). "Amino acid substitution matrices from protein blocks." In: *Proceedings of the National Academy of Sciences* 89.22, pp. 10915–10919. *DOI*: 10.1073/pnas.89.22.10915.

Herbig, A., G. Jager, F. Battke, and K. Nieselt (June 2012). "GenomeRing: alignment visualization based on SuperGenome coordinates". In: *Bioinformatics* 28.12, pp. i7–i15. *DOI*: 10.1093/bioinformatics/bts217.

Hezroni, H., D. Koppstein, M. Schwartz, A. Avrutin, D. Bartel, and I. Ulitsky (May 2015). "Principles of Long Noncoding RNA Evolution Derived from Direct Comparison of Transcriptomes in 17 Species". In: *Cell Reports* 11.7, pp. 1110–1122. *DOI*: 10.1016/j.celrep.2015.04.023.

Hierholzer, C. and C. Wiener (Mar. 1873). "Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren". In: *Mathematische Annalen* 6.1, pp. 30–32. *DOI*: 10.1007/bf01442866.

Higgins, D. G. and P. M. Sharp (Dec. 1988). "CLUSTAL: a package for performing multiple sequence alignment on a microcomputer". In: *Gene* 73.1, pp. 237–244. *DOI*: 10.1016/0378-1119(88)90330-7.

Hopcroft, J. and R. Tarjan (June 1973). "Algorithm 447: efficient algorithms for graph manipulation". In: *Communications of the ACM* 16.6, pp. 372–378. *DOI*: 10.1145/362248.362272.

Idury, R. M. and M. S. Waterman (Jan. 1995). "A New Algorithm for DNA Sequence Assembly". In: *Journal of Computational Biology* 2.2, pp. 291–306. *DOI*: 10.1089/cmb.1995.2.291.

Jain, M., H. E. Olsen, B. Paten, and M. Akeson (Nov. 2016). "The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community". In: *Genome Biology* 17.1. *DOI*: 10.1186/s13059-016-1103-0.

Johannsen, W. (Dec. 1914). "Elemente der exakten Erblichkeitslehre. Mit Grundzügen der biologischen Variationsstatistik". In: *Zeitschrift für Induktive Abstammungs- und Vererbungslehre* 11.1, pp. 200–200. *DOI*: 10.1007/bf01704312.

Kahn, A. B. (Nov. 1962). "Topological sorting of large networks". In: *Communications of the ACM* 5.11, pp. 558–562. *DOI*: 10.1145/368996.369025.

Kann, V. (1992). "On the approximability of NP-complete optimization problems". PhD thesis. Stockholm: Royal Institute of Technology. *URL*: http://www.nada.kth.se/~viggo/papers/phdthesis.pdf.

Karp, R. M. (1972). "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations*. Springer US, pp. 85–103. *DOI*: 10.1007/978-1-4684-2001-2_9.

Kececioglu, J. (1993). "The maximum weight trace problem in multiple sequence alignment". In: *Combinatorial Pattern Matching*. Springer-Verlag, pp. 106–119. *DOI*: 10.1007/bfb0029800.

Kehr, B., K. Trappe, M. Holtgrewe, and K. Reinert (Apr. 2014). "Genome alignment with graph data structures: a comparison". In: *BMC Bioinformatics* 15.1. *DOI*: 10.1186/1471-2105-15-99.

Kendall, M. G. (June 1938). "A New Measure of Rank Correlation". In: *Biometrika* 30.1/2, p. 81. *DOI*: 10.2307/2332226.

Kent, W. J., C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, Haussler, and David (2002). "The Human Genome Browser at UCSC". In: *Genome Research* 12.6, pp. 996–1006. *DOI*: 10.1101/gr.229102.

Kirchner, F., N. Retzlaff, and P. F. Stadler (2019). "A General Framework for Exact Partially Local Alignments". In: *Proceedings of the 12th International Joint Conference on Biomedical Engineering Systems and Technologies*. SCITEPRESS - Science and Technology Publications. *DOI*: 10.5220/0007380001940200.

Kirkman, T. P. and A. Cayley (Jan. 1856). "XVIII. On the representation of polyedra". In: *Philosophical Transactions of the Royal Society of London* 146, pp. 413–418. *DOI*: 10.1098/rstl.1856.0019.

Krebs, H. A., S. Gurin, and L. V. Eggleston (1952). "The pathway of oxidation of acetate in baker's yeast". In: *Biochemical Journal* 51.5, pp. 614–628. *ISSN*: 0264-6021. *DOI*: 10.1042/bj0510614. eprint: http://www.biochemj.org/content/51/5/614.full.pdf.

Lande, R. and M. Kirkpatrick (July 1988). "Ecological speciation by sexual selection". In: *Journal of Theoretical Biology* 133.1, pp. 85–98. *DOI*: 10.1016/s0022-5193(88)80026-2.

Leskovec, J. and A. Krevl (2014). *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data.

Li, H. (Mar. 2016). "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences". In: *Bioinformatics* 32.14, pp. 2103–2110. *DOI*: 10.1093/bioinformatics/btw152.

Li, H. (May 2018). "Minimap2: pairwise alignment for nucleotide sequences". In: *Bioinformatics* 34.18. Ed. by I. Birol, pp. 3094–3100. *DOI*: 10.1093/bioinformatics/bty191.

Li, K., X. Tang, B. Veeravalli, and K. Li (Jan. 2015). "Scheduling Precedence Constrained Stochastic Tasks on Heterogeneous Cluster Systems". In: *IEEE Transactions on Computers* 64.1, pp. 191–204. *DOI*: 10.1109/tc.2013.205.

Liiv, I. (2010). "Seriation and matrix reordering methods: An historical overview". In: *Statistical Analysis and Data Mining* 3, pp. 70–91. *DOI*: 10.1002/sam.10071.

Lin, S. et al. (Nov. 2014). "Comparison of the transcriptional landscapes between human and mouse tissues". In: *Proceedings of the National Academy of Sciences* 111.48, pp. 17224–17229. *DOI*: 10.1073/pnas.1413624111.

Linné, C. von (1767). *Systema naturae per regna tria naturae, secundum classes, ordines, genera, species, cum characteribus, differentiis, synonymis, locis*. Typis Ioannis Thomae von Trattner, *DOI*: 10.5962/bhl.title.156772.

Lorenz, R., S. H. Bernhart, F. Externbrink, J. Qin, C. H. zu Siederdissen, F. Amman, I. L. Hofacker, and P. F. Stadler (2012). "RNA Folding Algorithms with G-Quadruplexes". In: *Advances in Bioinformatics and Computational Biology*. Springer Berlin Heidelberg, pp. 49–60. *DOI*: 10.1007/978-3-642-31927-3_5.

Lorenz, R., S. H. Bernhart, C. H. zu Siederdissen, H. Tafer, C. Flamm, P. F. Stadler, and I. L. Hofacker (Nov. 2011). "ViennaRNA Package 2.0". In: *Algorithms for Molecular Biology* 6.1. *DOI*: 10.1186/1748-7188-6-26.

Lutz, B., H. C. Lu, G. Eichele, D. Miller, and T. C. Kaufman (Jan. 1996). "Rescue of Drosophila labial null mutant by the chicken ortholog Hoxb-1 demonstrates that the function of Hox genes is phylogenetically conserved." In: *Genes & Development* 10.2, pp. 176–184. *DOI*: 10.1101/gad.10.2.176.

Makedon, F. S., C. H. Papadimitriou, and I. H. Sudborough (July 1985). "Topological Bandwidth". In: *SIAM Journal on Algebraic Discrete Methods* 6.3, pp. 418–444. *DOI*: 10.1137/0606044.

Malewicz, G., M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski (2009). "Pregel: a system for large-scale graph processing". In: *Proceedings of the 28th ACM symposium on Principles of distributed computing - PODC '09*. ACM Press. *DOI*: 10.1145/1582716.1582723.

Martí, R., J. J. Pantrigo, A. Duarte, and E. G. Pardo (Jan. 2013). "Branch and bound for the cutwidth minimization problem". In: *Computers & Operations Research* 40.1, pp. 137–149. *DOI*: 10.1016/j.cor.2012.05.016.

Martí, R. and G. Reinelt (2011). *The Linear Ordering Problem*. Vol. 175. Springer Berlin Heidelberg. *DOI*: 10.1007/978-3-642-16729-4.

McGrath, C. L. and L. A. Katz (Jan. 2004). "Genome diversity in microbial eukaryotes". In: *Trends in Ecology & Evolution* 19.1, pp. 32–38. *DOI*: 10.1016/j.tree.2003.10.007.

Medini, D., C. Donati, H. Tettelin, V. Masignani, and R. Rappuoli (Dec. 2005). "The microbial pan-genome". In: *Current Opinion in Genetics & Development* 15.6, pp. 589–594. *DOI*: 10.1016/j.gde.2005.09.006.

Meidanis, J., O. Porto, and G. P. Telles (Nov. 1998). "On the consecutive ones property". In: *Discrete Applied Mathematics* 88.1-3, pp. 325–354. *DOI*: 10.1016/s0166-218x(98)00078-x.

Menger, K. (1927). "Zur allgemeinen Kurventheorie". In: *Fundamenta Mathematicae* 10, pp. 96–115. *DOI*: 10.4064/fm-10-1-96-115.

Metcalf, L. and W. Casey (2016). "Graph theory". In: *Cybersecurity and Applied Mathematics*. Elsevier, pp. 67–94. *DOI*: 10.1016/b978-0-12-804452-0.00005-1.

El-Metwally, S., T. Hamza, M. Zakaria, and M. Helmy (Dec. 2013). "Next-Generation Sequence Assembly: Four Stages of Data Processing and Computational Challenges". In: *PLOS Computational Biology* 9.12, pp. 1–19. *DOI*: 10.1371/journal.pcbi.1003345.

Miller, J. R., S. Koren, and G. Sutton (June 2010). "Assembly algorithms for next-generation sequencing data". In: *Genomics* 95.6, pp. 315–327. *DOI*: 10.1016/j.ygeno.2010.03.001.

Minkin, I. and P. Medvedev (Feb. 2019). "Scalable multiple whole-genome alignment and locally collinear block construction with SibeliaZ". In: *DOI*: 10.1101/548123.

Murphy, R. C., K. B. Wheeler, B. W. Barrett, and J. A. Ang (2010). "Introducing the graph 500". In: *Cray Users Group (CUG)* 19, pp. 45–74.

Nagarajan, N. and M. Pop (July 2009). "Parametric Complexity of Sequence Assembly: Theory and Applications to Next Generation Sequencing". In: *Journal of Computational Biology* 16.7, pp. 897–908. *DOI*: 10.1089/cmb.2009.0005.

Nawrocki, E. P. and S. R. Eddy (Sept. 2013). "Infernal 1.1: 100-fold faster RNA homology searches". In: *Bioinformatics* 29.22, pp. 2933–2935. *DOI*: 10.1093/bioinformatics/btt509.

Necsulea, A. and H. Kaessmann (Oct. 2014). "Evolutionary dynamics of coding and non-coding transcriptomes". In: *Nature Reviews Genetics* 15.11, pp. 734–748. *DOI*: 10.1038/nrg3802.

Neme, R. and D. Tautz (Feb. 2016). "Fast turnover of genome transcription across evolutionary time exposes entire non-coding DNA to de novo gene emergence". In: *eLife* 5. *DOI*: 10.7554/elife.09977.

Nguyen, N., G. Hickey, B. J. Raney, J. Armstrong, H. Clawson, A. Zweig, D. Karolchik, W. J. Kent, D. Haussler, and B. Paten (Aug. 2014a). "Comparative assembly hubs: Web-accessible browsers for comparative genomics". In: *Bioinformatics* 30.23, pp. 3293–3301. *DOI*: 10.1093/bioinformatics/btu534.

Nguyen, N., G. Hickey, B. J. Raney, J. Armstrong, H. Clawson, A. Zweig, D. Karolchik, W. J. Kent, D. Haussler, and B. Paten (Aug. 2014b). "Comparative assembly hubs: Web-accessible browsers for comparative genomics". In: *Bioinformatics* 30.23, pp. 3293–3301. *DOI*: 10.1093/bioinformatics/btu534.

Nguyen, N., G. Hickey, D. R. Zerbino, B. Raney, D. Earl, J. Armstrong, W. J. Kent, D. Haussler, and B. Paten (May 2015). "Building a Pan-Genome Reference for a Population". In: *Journal of Computational Biology* 22.5, pp. 387–401. *DOI*: 10.1089/cmb.2014.0146.

Nitsche, A., D. Rose, M. Fasold, K. Reiche, and P. F. Stadler (Mar. 2015). "Comparison of splice sites reveals that long noncoding RNAs are evolutionarily well conserved". In: *RNA* 21.5, pp. 801–812. *DOI*: 10.1261/rna.046342.114.

Nuutila, E. and E. Soisalon-Soininen (Jan. 1994). "On finding the strongly connected components in a directed graph". In: *Information Processing Letters* 49.1, pp. 9–14. *DOI*: 10.1016/0020-0190(94)90047-7.

Onodera, T., K. Sadakane, and T. Shibuya (2013). "Detecting superbubbles in assembly graphs". In: *International Workshop on Algorithms in Bioinformatics*. Ed. by A. Darling and J. Stoye. Vol. 8126. Berlin, Heidelberg: Springer Verlag, pp. 338–348. *DOI*: 10.1007/978-3-642-40453-5_26.

Opatrny, J. (Feb. 1979). "Total Ordering Problem". In: *SIAM Journal on Computing* 8.1, pp. 111–114. *DOI*: 10.1137/0208008.

Oswald, M. and G. Reinelt (May 2009). "The simultaneous consecutive ones problem". In: *Theoretical Computer Science* 410.21-23, pp. 1986–1992. *DOI*: 10.1016/j.tcs.2008.12.039.

Oyedotun, K. S. and B. D. Lemire (1997). "The Carboxyl Terminus of the Saccharomyces cerevisiaeSuccinate Dehydrogenase Membrane Subunit, SDH4p, Is Necessary for Ubiquinone Reduction and Enzyme Stability". In: *Journal of Biological Chemistry* 272.50, pp. 31382–31388. *DOI*: 10.1074/jbc.272.50.31382.

Pardo, E. G., R. Martí, and A. Duarte (2018). "Linear Layout Problems". In: *Handbook of Heuristics*. Springer International Publishing, pp. 1025–1049. *DOI*: 10.1007/978-3-319-07124-4_45.

Paten, B., D. Earl, N. Nguyen, M. Diekhans, D. Zerbino, and D. Haussler (June 2011). "Cactus: Algorithms for genome multiple sequence alignment". In: *Genome Research* 21.9, pp. 1512–1528. *DOI*: 10.1101/gr.123356.111.

Paten, B., J. M. Eizenga, Y. M. Rosen, A. M. Novak, E. Garrison, and G. Hickey (July 2018). "Superbubbles, Ultrabubbles, and Cacti". In: *Journal of Computational Biology* 25.7, pp. 649–663. *DOI*: 10.1089/cmb.2017.0251.

Paten, B., J. Herrero, K. Beal, S. Fitzgerald, and E. Birney (Nov. 2008). "Enredo and Pecan: Genome-wide mammalian consistency-based multiple alignment with paralogs". In: *Genome Research* 18.11, pp. 1814–1828. *DOI*: 10.1101/gr.076554.108.

Pearce, D. J. (Jan. 2016). "A space-efficient algorithm for finding strongly connected components". In: *Information Processing Letters* 116.1, pp. 47–52. *DOI*: 10.1016/j.ipl.2015.08.010.

Peer, Y. V. de, S. Maere, and A. Meyer (Aug. 2009). "The evolutionary significance of ancient genome duplications". In: *Nature Reviews Genetics* 10.10, pp. 725–732. *DOI*: 10.1038/nrg2600.

Pevzner, P. A., H. Tang, and M. S. Waterman (Aug. 2001). "An Eulerian path approach to DNA fragment assembly". In: *Proceedings of the National Academy of Sciences* 98.17, pp. 9748–9753. *DOI*: 10.1073/pnas.171285098.

Pevzner, P. A., H. Tang, and G. Tesler (Sept. 2004). "De Novo Repeat Classification and Fragment Assembly". In: *Genome Research* 14.9, pp. 1786–1796. *DOI*: 10.1101/gr.2395204.

Prohaska, S. J., S. J. Berkemer, F. Gärtner, T. Gatter, N. Retzlaff, C. H. zu Siederdissen, and P. F. Stadler (Dec. 2017). "Expansion of gene clusters, circular orders, and the shortest Hamiltonian path problem". In: *Journal of Mathematical Biology* 77.2, pp. 313–341. *DOI*: 10.1007/s00285-017-1197-3.

Rahm, E., W. E. Nagel, E. Peukert, R. Jäkel, F. Gärtner, P. F. Stadler, D. Wiegreffe, D. Zeckzer, and W. Lehner (Dec. 2018). "Big Data Competence Center ScaDS Dresden/Leipzig: Overview and selected research activities". In: *Datenbank-Spektrum* 19.1, pp. 5–16. *DOI*: 10.1007/s13222-018-00303-6.

Reid, J. K. and J. A. Scott (Jan. 2006). "Reducing the Total Bandwidth of a Sparse Unsymmetric Matrix". In: *SIAM Journal on Matrix Analysis and Applications* 28.3, pp. 805–821. *DOI*: 10.1137/050629938.

Rhoads, A. and K. F. Au (Oct. 2015). "PacBio Sequencing and Its Applications". In: *Genomics, Proteomics & Bioinformatics* 13.5, pp. 278–289. *DOI*: 10.1016/j.gpb.2015.08.002.

Risi, C., B. Belknap, E. Forgacs-Lonart, S. P. Harris, G. F. Schröder, H. D. White, and V. E. Galkin (Dec. 2018). "N-Terminal Domains of Cardiac Myosin Binding Protein C Cooperatively Activate the Thin Filament". In: *Structure* 26.12, 1604–1611.e4. *DOI*: 10.1016/j.str.2018.08.007.

Roberts, M., W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke (July 2004). "Reducing storage requirements for biological sequence comparison". In: *Bioinformatics* 20.18, pp. 3363–3369. *DOI*: 10.1093/bioinformatics/bth408.

Robinson, W. S. (Apr. 1951). "A Method for Chronologically Ordering Archaeological Deposits". In: *American Antiquity* 16.4, pp. 293–301. *DOI*: 10.2307/276978.

Ronse, C. (Oct. 2014). "Axiomatics for oriented connectivity". In: *Pattern Recognition Letters* 47, pp. 120–128. *DOI*: 10.1016/j.patrec.2014.03.020.

Rosen, Y., J. Eizenga, and B. Paten (2017). "Describing the Local Structure of Sequence Graphs". In: *Algorithms for Computational Biology – 4th AlCoB*. Ed. by D. Figueiredo, C. Martín-Vide, D. Pratas, and M. A. Vega-Rodríguez. Vol. 10252. Lecture Notes Comp. Sci. Heidelberg: Springer, pp. 24–46. *DOI*: 10.1007/978-3-319-58163-7_2.

Saab, Y. (May 2001). "A Fast and Effective Algorithm for the Feedback Arc Set Problem". In: *Journal of Heuristics* 7.3, pp. 235–250. *ISSN*: 1572-9397. *DOI*: 10.1023/A:1011315014322.

Saccharomyces Genome Database Community (n.d.). *SGD Yeast Pathway: Saccharomyces cerevisiae TCA cycle, aerobic respiration*. http://pathway.yeastgenome.org/YEAST/NEW-IMAGE?object=TCA-EUK-PWY. Accessed: 2017-05-18.

Sankoff, D. (1983). "Time warps, string edits, and macromolecules". In: *The Theory and Practice of Sequence Comparison, Reading*.

Schleimer, S., D. S. Wilkerson, and A. Aiken (2003). "Winnowing". In: *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD '03*. ACM Press. *DOI*: 10.1145/872757.872770.

Schwartz, S., W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller (2003). "Human–Mouse Alignments with BLASTZ". In: *Genome Research* 13.1, pp. 103–107. *DOI*: 10.1101/gr.809403.

Smith, T. and M. Waterman (Mar. 1981). "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147.1, pp. 195–197. *DOI*: 10.1016/0022-2836(81)90087-5.

Spearman, C. (Jan. 1904). "The Proof and Measurement of Association between Two Things". In: *The American Journal of Psychology* 15.1, p. 72. *DOI*: 10.2307/1412159.

Spingola, M., L. Grate, D. Haussler, and M. Ares Jr. (Feb. 1999). "Genome-wide bioinformatic and molecular analysis of introns in *Saccharomyces cerevisiae*". In: *RNA* 5.2, pp. 221–234. *DOI*: 10.1017/s1355838299981682.

Sung, W.-K., K. Sadakane, T. Shibuya, A. Belorkar, and I. Pyrogova (July 2015). "An $O(mlogm)$-time algorithm for detecting superbubbles". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 12, pp. 770–777. *DOI*: 10.1109/TCBB.2014.2385696.

Sylvester, J. J. (1878). "On an Application of the New Atomic Theory to the Graphical Representation of the Invariants and Covariants of Binary Quantics,

with Three Appendices". In: *American Journal of Mathematics* 1.1, pp. 64–104. *DOI*: 10.2307/2369436.

Tankyevych, O., H. Talbot, and N. Passat (2013). "Semi-connections and Hierarchies". In: *Mathematical Morphology and Its Applications to Signal and Image Processing*. Ed. by C. L. Luengo Hendriks, G. Borgefors, and R. Strand. Vol. 7883. Lecture Notes in Computer Science. Berlin: Springer, pp. 159–170. *DOI*: 10.1007/978-3-642-38294-9_14.

Tarjan, R. (June 1972). "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2, pp. 146–160. *DOI*: 10.1137/0201010.

Tarjan, R. E. (1976). "Edge-disjoint spanning trees and depth-first search". In: *Acta Informatica* 6.2, pp. 171–185. *DOI*: 10.1007/bf00268499.

Trifonov, E. N. (Oct. 2011). "Vocabulary of Definitions of Life Suggests a Definition". In: *Journal of Biomolecular Structure and Dynamics* 29.2, pp. 259–266. *DOI*: 10.1080/073911011010524992.

Tucker, A. (1972). "A structure theorem for the consecutive 1's property". In: *Journal of Combinatorial Theory, Series B* 12.2, pp. 153–162. *ISSN*: 0095-8956. *DOI*: https://doi.org/10.1016/0095-8956(72)90019-6.

Viro, O., O. Ivanov, N. Netsvetaev, and V. Kharlamov (Sept. 2008). *Elementary Topology*. American Mathematical Society. *DOI*: 10.1090/mbk/054.

Wang, L. and T. Jiang (Jan. 1994). "On the Complexity of Multiple Sequence Alignment". In: *Journal of Computational Biology* 1.4, pp. 337–348. *DOI*: 10.1089/cmb.1994.1.337.

Washietl, S., M. Kellis, and M. Garber (2014). "Evolutionary dynamics and tissue specificity of human long noncoding RNAs in six mammals". In: *Genome Research* 24.4, pp. 616–628. *DOI*: 10.1101/gr.165035.113.

Watson, J. D. and F. H. C. Crick (Apr. 1953). "Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid". In: *Nature* 171.4356, pp. 737–738. *DOI*: 10.1038/171737a0.

Watts, D. J. and S. H. Strogatz (June 1998). "Collective dynamics of 'small-world' networks". In: *Nature* 393.6684, pp. 440–442. *DOI*: 10.1038/30918.

Xiao, S., X. Cao, and S. Zhong (July 2014). "Comparative epigenomics: defining and utilizing epigenomic variations across species, time-course, and individuals". In: *Wiley Interdisciplinary Reviews: Systems Biology and Medicine* 6.5, pp. 345–352. *DOI*: 10.1002/wsbm.1274.

Xue, Y. et al. (Sept. 2009). "Human Y Chromosome Base-Substitution Mutation Rate Measured by Direct Sequencing in a Deep-Rooting Pedigree". In: *Current Biology* 19.17, pp. 1453–1457. *DOI*: 10.1016/j.cub.2009.07.032.

Yasutake, Y., S. Watanabe, M. Yao, Y. Takada, N. Fukunaga, and I. Tanaka (2003). "Crystal Structure of the Monomeric Isocitrate Dehydrogenase in the Presence of NADP+: INSIGHT INTO THE COFACTOR RECOGNITION, CATALYSIS, AND EVOLUTION". In: *Journal of Biological Chemistry* 278.38, pp. 36897–36904. *DOI*: 10.1074/jbc.M304091200.

Yue, J.-X. and G. Liti (May 2019). "simuG: a general-purpose genome simulator". In: *Bioinformatics*. Ed. by J. Hancock. *DOI*: 10.1093/bioinformatics/btz424.

Zerbino, D. R. and E. Birney (Feb. 2008). "Velvet: algorithms for de novo short read assembly using de Bruijn graphs". In: *Genome Research* 18.5, pp. 821–829. *DOI*: 10.1101/gr.074492.107.

Publications

R. Lorenz, S. H. Bernhart, F. Externbrink, J. Qin, C. H. zu Siederdissen, F. Amman, I. L. Hofacker, and P. F. Stadler (2012). "RNA Folding Algorithms with G-Quadruplexes". In: *Advances in Bioinformatics and Computational Biology*. Springer Berlin Heidelberg, pp. 49–60. *DOI*: 10.1007/978-3-642-31927-3_5.

M. Bernt, A. Donath, F. Jühling, F. Externbrink, C. Florentz, G. Fritzsch, J. Pütz, M. Middendorf, and P. F. Stadler (Nov. 2013). "MITOS: Improved de novo metazoan mitochondrial genome annotation". In: *Molecular Phylogenetics and Evolution* 69.2, pp. 313–319. *DOI*: 10.1016/j.ympev.2012.08.023.

C. Arnold, F. Externbrink, J. Hackermüller, and K. Reiche (Nov. 2014). "CEM-Designer: Design of custom expression microarrays in the post-ENCODE Era". In: *Journal of Biotechnology* 189, pp. 154–156. *DOI*: 10.1016/j.jbiotec.2014.09.012.

S. J. Prohaska, S. J. Berkemer, F. Gärtner, T. Gatter, N. Retzlaff, C. H. zu Siederdissen, and P. F. Stadler (Dec. 2017). "Expansion of gene clusters, circular orders, and the shortest Hamiltonian path problem". In: *Journal of Mathematical Biology* 77.2, pp. 313–341. *DOI*: 10.1007/s00285-017-1197-3.

F. Gärtner, C. Höner zu Siederdissen, L. Müller, and P. F. Stadler (Sept. 2018). "Coordinate Systems for Supergenomes". In: *Algorithms for Molecular Biology* 13.1, p. 15. *DOI*: 10.1186/s13015-018-0133-4.

F. Gärtner, L. Müller, and P. F. Stadler (Dec. 2018). "Superbubbles revisited". In: *Algorithms for Molecular Biology* 13.1, p. 16. *DOI*: 10.1186/s13015-018-0134-3.

E. Rahm, W. E. Nagel, E. Peukert, R. Jäkel, F. Gärtner, P. F. Stadler, D. Wiegreffe, D. Zeckzer, and W. Lehner (Dec. 2018). "Big Data Competence Center ScaDS Dresden/Leipzig: Overview and selected research activities". In: *Datenbank-Spektrum* 19.1, pp. 5–16. *DOI*: 10.1007/s13222-018-00303-6.

F. Gärtner and P. F. Stadler (Apr. 2019). "Direct Superbubble Detection". In: *Algorithms* 12.4. *ISSN*: 1999-4893. *DOI*: 10.3390/a12040081.

Presentations

07.10.2014 **Oriented graph grammars and stereochemistry**
12. Herbstseminar der Bioinformatik

29.09.2015 **Cilk Plus - The power of multicore and vector processing**
13. Herbstseminar der Bioinformatik

18.02.2016 **Golden Genome**
31nd TBI Winterseminar in Bled

15.02.2017 **From Genomes to Supergenomes - How to deal with betweenness**
32nd TBI Winterseminar in Bled

15.02.2018 **The Magic of Graph Databases**
33rd TBI Winterseminar in Bled

02.10.2018 **Superbubbles**
16. Herbstseminar der Bioinformatik

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Dissertation selbständig und ohne unzulässige fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angeführten Quellen und Hilfsmittel benutzt und sämtliche Textstellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, und alle Angaben, die auf mündlichen Auskünften beruhen, als solche kenntlich gemacht. Ebenfalls sind alle von anderen Personen bereitgestellten Materialen oder erbrachten Dienstleistungen als solche gekennzeichnet.

(Ort, Datum)

(Unterschrift)