Safe solutions for walks on graphs

Nidia Obscura Acosta

Helsinki February 19, 2018

UNIVERSITY OF HELSINKI

Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Studieprogram — Study Programme Computer Science

Tekijä — Författare — Author
Nidia Obscura Acosta

Työn nimi — Arbetets titel — Title
Safe solutions for walks on graphs

Ohjaajat — Handledare — Supervisors
Alexandru I. Tomescu and Veli Mäkinen

Työn laji — Arbetets art — Level Master's Thesis	Aika — Datum — Month and year February 19, 2018	Sivumäärä — Sidoantal — Number of pages 50 pages

Tiivistelmä — Referat — Abstract

In this thesis we study the concept of "safe solutions" in different problems whose solutions are walks on graphs. A safe solution to a problem X can be understood as a partial solution common to all solutions to problem X. In problems whose solutions are walks on graphs, safe solutions refer to walks common to all walks which are solutions to the problem.

In this thesis, we focused on formulating four main graph traversal problems and finding characterizations for those walks contained in all their solutions. We give formulations for these graph traversal problems, we prove some of their combinatorial and structural properties, and we give safe and complete algorithms for finding their safe solutions based on their characterizations. We use the genome assembly problem and its applications as our main motivating example for finding safe solutions in these graph traversal problems.

We begin by motivating and exemplifying the notion of safe solutions through a problem on s-t paths in undirected graphs with at least two non-trivial biconnected components S and T and with $s \in S$, $t \in T$. We continue by reviewing similar and related notions in other fields, especially in combinatorial optimization and previous work on the bioinformatics problem of *genome assembly.*

We then proceed to characterize the safe solutions to the Eulerian cycle problem, where one must find a circular walk in a graph G which traverses each edge exactly once. We suggest a characterization for them by improving on (Nagarajan, Pop, JCB 2009) and a polynomial-time algorithm for finding them.

We then study edge-covering circular walks in a graph G. We look at the characterization from (Tomescu, Medvedev, JCB 2017) for their safe solutions and their suggested polynomial-time algorithm and we show an optimal $O(mn)$-time algorithm that we proposed in (Cairo et al. CPM 2017).

Finally, we generalize this to edge-covering collections of circular walks. We characterize safe solutions in an edge-covering setting and provide a polynomial-time algorithm for computing them. We suggested these originally in (Obscura et al. ALMOB 2018).

ACM Computing Classification System (CCS):
Mathematics of computing → Discrete mathematics → Graph theory → Graph algorithms problems
Mathematics of computing → Discrete mathematics → Graph theory → Paths and connectivity problems

Avainsanat — Nyckelord — Keywords
Genome assembly, Metagenomics, Eulerian cycle, Safe solution, Graph algorithm, Edge-covering walk

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

The aim of this thesis is to describe the concept of *safe solutions* in the context of graph traversals. Informally, a safe solution can be understood as a partial solution to a problem which is promised to appear in all possible solutions to the given problem.

Since many real-world applications are naturally formulated as abstract combinatorial optimization problems, i.e. finding the best solution(s) out of a finite set, finding constant parts in their solutions can ease in their characterization, enumeration or practicality, as well as provide a clearer picture of the solutions in hand. Furthermore, since most combinatorial optimization problems can be formulated naturally in terms of graphs and as (integer) linear programs, it is natural to try to understand the shared properties in all optimal solutions.

For example, for the problem of finding a maximal matching in a bipartite graph, the problem of finding all edges contained in every maximal matching, and all edges never contained in any maximal matching is considered in [Cos94]. One can define different sets of edges, nodes or even combinations of them (walks), as safe solutions, depending on the problem whose solutions we want to further characterize. One could for example, for the problem of finding all Eulerian cycles of a graph G, define as safe solution those walks which are contained (as subwalks) in all the possible Eulerian walks in a given graph G (see Section 3). It is therefore, important to clearly define what is understood as a safe solution for each of the given problems.

The main motivation for this thesis comes from the genome assembly problem in bioinformatics, where an unknown genome must be reconstructed from *reads*, fragments obtained from a sequencing experiment on the genome. While this is one of the oldest, real-world bioinformatics problems, it is considered a difficult problem. This is mainly because it is very difficult to give an accurate model for it and because there are many variables involved in the sequencing process. As we will see in Section 2.2, even simple mathematical formulations of it turn out to be NP-hard. This hardness has motivated practical genome assemblers to split the problem into several heuristic stages. The usual first step in these practical assemblers is the assembly of contigs, namely strings that are promised to appear in the original genome which generated the reads. A common approach to this stage is to use graph theoretical models, by building an adequate graph generated from the reads. In these graphs, one is then interested in identifying certain walks which can model the contigs. In this context, a safe solution contained in every possible solution models the idea that the contigs must be contained in every possible genome that could have generated the reads. The solutions for the problems dealt with in this thesis will be certain types of walks in graphs, and as we will discuss in the following sections, several meaningful formulations for the notion of safe solutions will be possible depending on the problem in question.

The thesis is structured as follows:

In Section 2, we start by introducing related notions on persistency, blockers, traversals, and vital vertices and nodes. In Section 2.2, we look at the concept of safe solutions in a bioinformatics context, especially motivated by the genome assembly problem.

In Section 3 we deal with the problem of finding all walks common to every Eulerian cycle of a given graph; we suggest a characterization for them by improving on [NP09] and a polynomial-time algorithm for finding them.

In Section 4, we look at more general edge-covering walks in strongly connected graphs and we revisit the characterization from [TM17] for their safe solutions in the edge-covering case and we show two algorithms to compute them: an easy polynomial-time algorithm from [TM17] and an optimal $O(mn)$-time algorithm that we proposed in [CMO$^+$17].

Finally, in Section 5, we consider edge-covering collections of circular walks for which we characterize safe solutions and provide a polynomial time algorithm for computing them. Originally, we proposed these in [OMT18].

We conclude this introduction section, first, by providing basic terminology and notions which will be necessary for the rest of the document, in Section 1.1. Second, we show a motivating graph theoretical example of safe solutions in Section 1.2 for the particular problem of finding all the edges contained in every s-t path between two biconnected components S and T. We give a full characterization and a polynomial-time algorithm for finding them.

1.1 Basic notions and notation

Definition 1 (Graph)**.** *A* graph G *consists of a non-empty finite set of* vertices *or* nodes $V(G)$ *and a finite set* $E(G)$ *of ordered pairs of distinct vertices called* edges*. We call* $V(G)$ *the vertex set and* $E(G)$ *the edge set of* G*.*

For a directed edge (u, v), the first vertex u is called its *source* and the second vertex v is called its *target*. We also say that the edge (u, v) *leaves* u and *enters* v. We will denote these source and target vertices as $s(e)$ and $t(e)$ respectively and we call them e's *endpoints*. We call two edges *adjacent* if they share an endpoint.

Definition 2 (Undirected graph)**.** *We call an undirected graph* G *a non-empty finite set of vertices or nodes* $V(G)$ *and a finite set* $E(G)$ *of unordered pairs of distinct vertices called* undirected edges*. We similarly call* $V(G)$ *the vertex set and* $E(G)$ *the edge set of* G*.*

Definition 3 (Loop)**.** *A* loop *is an edge whose endpoints are equal. Parallel edges are edges with the same source and target nodes.*

Definition 4 (Simple graph)**.** *A* simple graph *is a graph having no loops or multiple edges.*

Let G be a graph, possibly with parallel edges and self-loops. The number of nodes and edges in a graph are denoted by n and m, respectively. We use $N^-(v)$ to denote the set of *in-neighbours* and $N^+(v)$ to denote the set of *out-neighbours* of a node v. We call the cardinality of $N^-(v)$ the *in-degree* of node v and the cardinality of $N^+(v)$ the *out-degree* of node v.

Definition 5 (Walk)**.** *A* walk w *is a sequence* $(v_0, e_0, v_1, e_1, \ldots, v_t, e_t, v_{t+1})$ *where* $v_0, \ldots, v_{t+1}$ *are nodes, and each* e_i *is an edge from* v_i *to* v_{i+1}*, and* $t \geqslant -1$*. Its* length *is its number of edges, namely* $t + 1$*. Walks of length at least one are called* proper*.*

Definition 6 (Path)**.** *A* path p *is a walk where the nodes are all distinct, except possibly the first and last nodes may be the same, in which case it will also be called a* cycle*.*

We denote by $s(w)$ the first node of a path (walk) w and by $t(w)$ the last node of w. We call a path (walk) *non-empty* when its number of edges is at least one.

For a path (walk) w, we call *internal* those nodes which are different from $s(w)$ and $t(w)$. A path (walk) with first node u and last node v will be called a *path (walk) from u to v*, and denoted as *u-v path (walk)*.

Definition 7 (Circular walk). *A walk w whose first and last nodes coincide (that is $s(w) = t(w)$) is called* circular *walk.*

A walk is called *node-covering* if it passes through each node of G, and *edge-covering* if it passes through each edge of G.

We call a walk w to be *left-maximal* in a set W, if it is not a proper suffix of a subwalk of another walk $w' \in W$. Similarly, we call it *right-maximal*, if it is not a proper prefix of a subwalk of another walk $w' \in W$. We call a walk w *maximal* if it is both left- and right-maximal.

A walk w is called *closed* if it is non-empty and if $s(w) = t(w)$ and *open* otherwise.

Definition 8 (Subgraph). *A* subgraph *of a graph G is a graph H such that V(H)⊆V(G) and E(H)⊆E(G) and the assignment of endpoints to edges in H is the same as in G.*

Definition 9 (Connected graph). *An undirected graph G is connected if there is a path between each pair of vertices. Otherwise, G is disconnected.*

Definition 10 (Connected component). *A* connected component *H of an undirected graph G is a subgraph of graph G which has the property of being a connected graph. Furthermore, we require that H is maximal, that is, it is not a proper subgraph of any other connected component.*

Definition 11 (Maximal string). *We call a string s maximal in a set S, if and only if it is not entirely contained in any other string of the set S; that is, it cannot be obtained by removing any number of consecutive symbols from the beginning and/or end of any other string $r \in S$.*

1.2 Graph theoretical example on cut edges

Let us begin by presenting a motivating example of safe solutions in a graph theoretical context, namely on graphs with at least two biconnected components. Let us first introduce some relevant definitions.

Definition 12 (Cut edge). *In an undirected graph G, a cut edge is an edge for which its removal increases the number of connected components.*

Analogously we define:

Definition 13 (Cut vertex)**.** *In an undirected graph G, a cut vertex is a vertex for which its removal (and the removal of its incident edges) increases the number of connected components.*

Definition 14 (Biconnected component)**.** *In an undirected graph, a biconnected component refers to a maximal component for which none of its vertices is a cut vertex. This means that the removal of any of its vertices leaves the component still connected.*

Observe that, by definition, it follows that there are two different paths between every pair of vertices. Accordingly, no edge of a biconnected component different than a single edge can be a cut edge. We note that by the above definition, any cut edge is a biconnected component, which we will denote as *trivial*. Any biconnected component which is not a cut edge will be denoted by *non-trivial*. For the rest of this text, by biconnected component we will refer only to a non-trivial biconnected component, unless stated otherwise.

Problem 1. *Given a connected undirected graph G containing at least two non-trivial biconnected components S and T, and given any two vertices $s \in S$ and $t \in T$, find those edges which are contained in all s-t paths.*

Solution 1. *We first note that since both S and T are biconnected components, none of their edges can be contained in all s-t paths, or otherwise their removal would increase the number of connected components, contradicting the fact that they are contained in a biconnected component. Then the only possible edges contained in every s-t path should be edges between S and T.*

We note that all those edges contained in every s-t path must, in particular, be also contained individually in any given s-t path (by definition). If there exists an s-t path, containing a cut edge e, then by definition the removal of this cut edge would increase the number of connected components meaning that there exists no other edge connecting the components S and T. This implies that this cut edge e should be part of every s-t path. If there is a non-cut edge f in an s-t path (apart form the edges of S and T), then by definition of a non-cut edge, its removal does not increase the number of connected components. This implies that the graph G is still connected and in particular that there should exist another s-t path which does not contain f.

This shows us that all edges which are common to all s-t paths must be those cut edges contained in an s-t path. This also shows that for non-cut edges, we can always

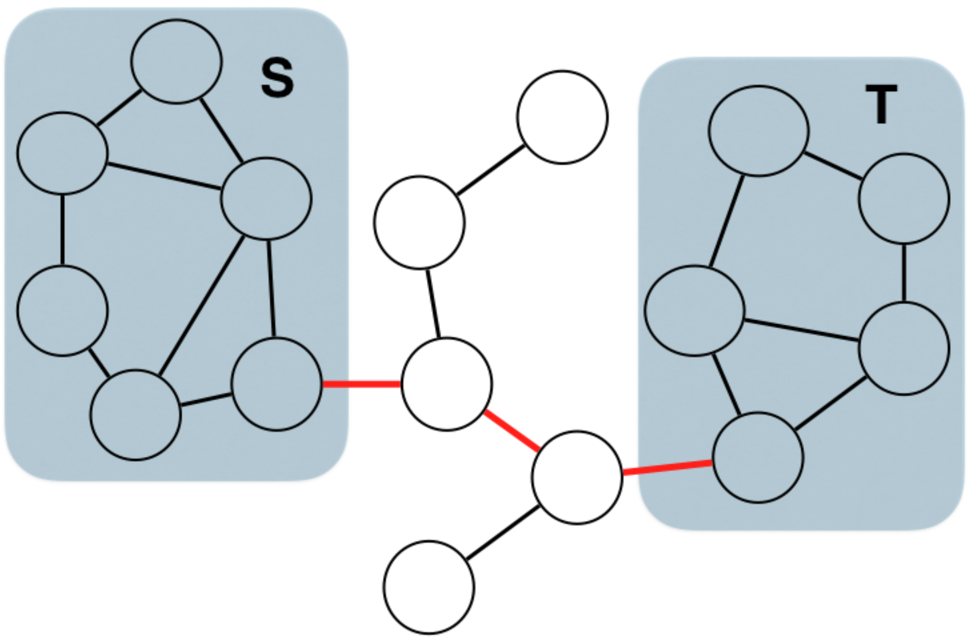

Figure 1: In a graph containing two biconnected components S and T, the red edges contained in all s-t paths with $s \in S$ and $t \in T$ are given by the cut edges of the graph.

find an alternative path that does not pass through them. We can now call these cut edges our safe solutions: edges which are contained in all possible s-t paths.

Observation 1 (Safe solution for the two biconnected components problem)**.** *All those edges which are common to all s-t paths in an undirected graph G with two given biconnected components S and T and $s \in S$ and $t \in T$ are all the cut edges between S and T contained in any s-t path. (See Figure 1.)*

As we will see next and in the following examples throughout this text, our characterization of a safe solution allows us to develop algorithms for their enumeration. As we proved above, it is only necessary to find one s-t path and then find the cut edges contained in it, since every safe solution must itself be contained in every s-t path. This automatically leads to a simple $O(n + m)$-time algorithm to find these safe solutions (Algorithm 7), since finding the cut edges in an undirected graph G can be done in linear time [Tar74].

We would ideally prefer algorithms which are safe (i.e. output only correct solutions) and which are complete (i.e. which output all the correct solutions). However, time complexity and memory space can be a burden for these algorithms, and it will depend on each application which algorithms are preferred.

Algorithm 1: Algorithm to find all and only safe edges in graph G.

Input: A connected undirected graph G with two given biconnected components S and T.

Output: A set of edges S which appear in all s-t paths for $s \in S$ and $t \in T$

```
1 $S := \emptyset$;
2 $B :=$ TarjanAlgoritm(G)            // $B$ is the set of all cut edges of $G$.
3 $P :=$ Any path $s$-$t$ for some $s \in S$ and $t \in T$
4 foreach edge $e \in P$ do
5     if $e \in B$ then
6         $S := S \cup \{e\}$;
7 return $S$, the set of safe edges.
```

Lemma 1. *Algorithm 7 is safe and complete.*

Proof. Algorithm 7 returns only cut edges, since they form a subset of B. Furthermore, it returns only cut edges that appear in an s-t path, which by Solution 1 are safe edges and therefore Algorithm 7 is safe. Similarly, by Solution 1, safe edges can only be edges between S and T. Furthermore, non-cut edges cannot be safe so that only the cut edges between S and T be safe; by definition the removal of cut edges in any s-t path disconnects S and T, so each of them is safe, implying that Algorithm 7 is complete. $\square$

Lemma 2. *Algorithm 7 runs in linear time.*

Proof. Finding all cut edges in a connected graph takes linear time with an algorithm like Tarjan's [Tar74], and finding an s-t path takes linear time. For the *for loop*, we only execute it at most $n - 1$ times, since P is of length at most $n - 1$, while each of its executions takes constant time. $\square$

2 Related notions and previous work

In this section we will introduce notions closely related to ours, starting with that of persistency. We will introduce previous work for characterizing the similarity of solutions in a more general combinatorial setting, and at the end of the section we will present the concept of safe solutions in a bioinformatics context.

2.1 Persistency, blockers, traversals and vital vertices and nodes

In a combinatorial optimization setting, a highly related notion to safe solutions is that of persistency. It was first formally introduced by Costa in [Cos94] for the maximum cardinality matching problem in bipartite graphs. The idea of persistency is that there exist edges, nodes or certain structures which are always contained in an optimal solution, as well as structures which are never contained in an optimal solutions and structures which are contained in only some optimal solutions.

The motivation for this concept in combinatorial optimization is that very often, the optimal (minimum or maximum) solutions for an optimization problem are not unique; in some cases, the number of optimal solutions can be exponential, and listing them may not be efficient. An easy approach to this problem is to add additional constraints to reduce the number of these optimal solutions. This however, can result in increasing substantially the difficulty of our initial problem. It is therefore a more natural approach to try to describe or characterize decision variables with respect to optimal solutions and understand as persistent, those variables which are common to all optimal solutions. In the case of a graph where the feasible solutions are sets of edges, the persistency set is defined as:

Definition 15. *The persistency set is a 3-partition (E_1, E_0, E_w) of the edge set of a graph such that:*

- *E_1 is the set of edges that belong to all optimal solutions (1-persistent edges),*
- *E_0 is the set of edges that belong to no optimal solution (0-persistent edges), and*
- *E_w is the set of edges belonging to at least one optimal solution but not to all (weakly persistent edges).*

For the problem of maximum cardinality matching in bipartite graphs, we want to find a matching M of the edges E (a set of edges from E such that no two edges share an endpoint) and such that M is of maximum size for bipartite graphs. In a persistency setting, E_1 corresponds to the edges belonging to all maximum matchings, E_0 to the edges which do not belong to any maximum matching and E_w to those edges belonging to at least one maximum matching but not to all. Costa [Cos94] studied the problem of finding the persistency set for this problem and found an $O(mn)$ algorithm for it, for a graph with n vertices and m edges. Her results have been generalized and algorithms $O(m^2n)$-time algorithms have been derived for the assignment and transportation problem in [Cec98].

Furthermore, persistency has been studied in many other problems: In [CL01], they look at persistency in combinatorial optimization problems on matroids. For instance, they propose algorithms for finding the maximum weight independent set of a matroid and for finding a maximum cardinality intersection of two matroids. They further extend their analysis to obtain the persistency set.

The problem of finding the vertices belonging to all maximum stable sets of an undirected graph is studied in [BGL02], where a stable set S of a graph is a maximum set where no edge has both its endpoints in S. They improve on previous results from [HHS82] and [LM02] by finding close relationships between the size of a maximum stable set, the size of a maximum matching and the number of vertices belonging to all maximum stable sets on connected graphs. They also prove that recognizing if there is at least one vertex belonging to all maximum stable sets can be done in polynomial time for graphs where the size of a maximum matching is less than $|V(G)|/3$. Furthermore, they prove that in general, determining if there exist more than k vertices belonging to all maximum stable sets is NP-complete for any fixed $k \geqslant 0$

In [Lac00], the persistency set was calculated in polynomial time for the traveling salesman problem, restricted to Halin graphs with three types of cost functions for different versions of the traveling salesman problem. The persistency of the weighted spanning tree and maximum flow problem have also been studied in [Lac98].

In [BNT06] they define the persistency problem for a discrete optimization problem under certain objective as the problem of evaluating the expected optimal value and the marginal distribution of the optimal solution in a discrete maximization problem with a linear objective function and where the coefficients are chosen randomly from a distribution. They solve a special case of this model where the distribution is assumed to belong to the class of distributions defined by given marginal distributions or given marginal moment conditions. They then use this persistency model for problems in discrete choice modeling and in a stochastic knapsack problem.

Two other related notions are those of blockers and traversals, originally defined in [ZRP+09] for an undirected graph G.

Definition 16. *A d-traversal is a subset of V which intersects any optimum solution in at least d elements.*

Definition 17. *A d-blocker is a subset of V whose removal deteriorates the value of an optimum solution by at least d.*

In the problem of maximum matchings, a d-traversal is a set of edges which intersect every maximum matching in at least d edges. From a persistency point of view this can be viewed as a set $S \subset E_1 \cup E_w$ with the requirement that at least d edges are common to every maximum matching. On the other hand, a d-blocker can be seen as a subset of edges such that its removal reduces by at least d the cardinality c of any maximum cardinality matching, that is a set $S \subset E_1$ such that its removal leaves the cardinality of any maximum matching of at most $c - d$.

In [ZRP+09] they looked at the problem of finding the minimum cardinality d-blockers and d-traversals for the maximum matching problem and found relationships between these two notions. They proved that deciding whether there exists a d-blocker or d-traversal of size k in a given undirected graph G for given k and d is NP-complete in bipartite graphs and they further analyse the complexity in complete graphs, regular bipartite graphs, chains and cycles. In [RBP+10], the problem is showed to be solvable efficiently in the special cases where G is a grid graph or a tree.

In [CdWP11] they generalize and investigate the d-traversals and d-blockers of vertex covers or stable sets in split and bipartite graphs. Furthermore, they revisit the problems of matchings studied by [ZRP+09] and the s-t paths and s-t cuts in graphs studied by [KBB+08] and [Wag90].

In [PBP14] blockers are studied for the minimum vertex blocker clique problem, where one needs to remove a subset of vertices of minimum cardinality in a weighted undirected graph, such that the maximum weight of a clique in the remaining graph is bounded above by a given integer r. They prove that even on the special case of unweighted graphs, this problem remains NP-hard.

Finally, closely related to these notions is the concept of most vital vertices/nodes. In [SBNK95], they examine the problem of finding the k most vital edges (or nodes) in a graph, which are those k edges (or nodes) whose removal maximizes the increase in the length of the shortest path from s to t, where each edge e has an associated non-negative length $l(e)$. They prove this to be NP-hard even for edges of unit length.

In [BTT11], they look at the problem of the k most vital nodes independent set, which consists of determining a set of k vertices whose removal results in the greatest decrease in the maximum weight of independent sets, as well as the homologous k most vital nodes vertex cover problem where one must find a set of k vertices whose removal results in the greatest decrease in the minimum weight of vertex covers. They also consider the complementary problems of minimum node blocker independent set and minimum node blocker vertex cover. They show all these problems to be solvable in polynomial time on trees, cographs and graphs of bounded treewidth, as well as on unweighted bipartite graphs, but NP-hard on bipartite graphs.

2.2 Safe solutions in bioinformatics

Illustrative examples of safe solutions come also from bioinformatics applications; in this section, we will discuss the genome assembly problem. In this problem, the genome of an organism must be reconstructed out of sequencing reads. This will be our main motivating example for safe solutions in the gap filling and contig assembly problems.

A common approach to genome assembly is to sequence randomly cut fragments of DNA to produce **reads** (using, for example a whole-genome shotgun sequencing approach [EW14]). These reads however, are not sequenced in any specific order and their orientation is unknown; furthermore, in order to ensure that all the genome is sequenced and represented in the reads, several copies of the same original genome are produced (known as **coverage**). With these reads one would like to find a suitable sequence of nucleotides which contains completely all the reads and which is consistent with the coverage of each read. More formally:

Definition 18 (Genome assembly problem)**.** *Given a set of sequenced reads S coming from a supersequence G, reconstruct back G from the reads in S.*

There exists several mathematical formulations which try to model the problem of reconstructing a string out of reads, for example the **Shortest Common Superstring Problem** (SCSP). In the SCS problem, one is asked to find the minimum superstring of a set of strings S (a string which contains the other strings in S as substrings). When modelling the genome assembly problem, however, the SCS problem can handle wrongly the repeats naturally contained in the genome and collapse them into a shortest superstring; furthermore, the shortest common superstring problem is known to be NP-hard for alphabet sizes bigger than two [GMS80].

Other popular approaches to the genome assembly problem use a graph-theoretical formulation, where the genome corresponds to suitable walks on graphs constructed out of the sequencing reads. In a *string/overlap* graph [Mye05] the nodes of the graph are represented directly by the reads and the edges between them represent valid overlaps between the reads. More formally:

Definition 19 (String graph). *Given a set of reads S and its given overlap graph, a string graph is built out of the overlap graph by applying the transitively inferable edge reduction (if two reads x and z have a valid suffix-prefix overlap, z has a valid suffix-prefix overlap with another read y, and also x and y have a valid suffix-prefix overlap, then the inferable overlap of x to z is removed from the graph).*

In the string graph setting, the genome is then represented by a walk in the graph which uses all reads at least once (covers all nodes at least once).

In the **Minimum s-walk problem** we are given a weighted directed graph G and a classification s of the edges in G into *optional*, *required*, and *exact* and are asked to find a minimum weight cyclical walk of G, which satisfies the constraints in s, or report that one does not exist.

In a De Bruijn graph model [PTW01a], the reads are further divided into k-mers which represent the nodes of the graph, the edges represent valid overlaps among them and the genome is given by a superwalk based on the original reads; more formally:

Definition 20 (Edge-centric de Bruijn graph). *An edge-centric De Bruijn graph obtained from a set S of strings is a graph whose vertices represent length k subsequences (k-mers) of the strings in S. There is an edge from vertex u to vertex v if and only if the $k-1$-length suffix of u overlaps the $k-1$-length prefix of v, and additionally, if the string obtained by traversing u and v consecutively is contained in some read in R. The graph is denoted by $B^k(S)$.*

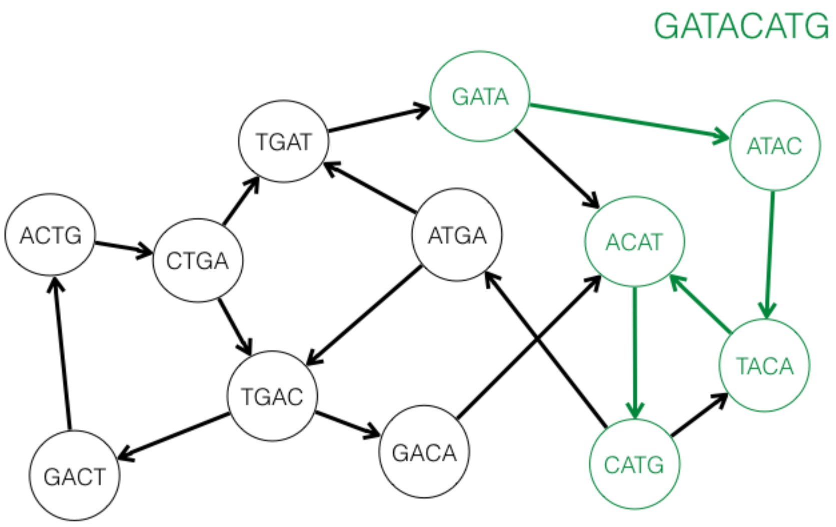

Figure 2: A 4-mer de Bruijn graph and a corresponding sequence **GATACATG** obtained from the walk denoted in green.

An n-length walk in a De Bruijn graph represents a unique sequence of $n+1$ nodes, which in turn represents a unique $k+n$ length sequence, where k is the number of symbols each node represents. Reconstructing back a genome from a set of sequencing reads therefore reduces to finding adequate walks in the graph. (See Figure 2).

In the **de Bruijn Superwalk problem (BSP)** we are given a set of reads S and are asked to find a minimum length superwalk (a walk that contains all the reads as subwalks) in the de Bruijn graph $B^k(S)$ constructed from the reads, or report that one does not exist.

We will refer to the solutions to these problems as genomic reconstructions.

We note, however, that both the **Minimum s-walk problem** and the **de Bruijn Superwalk problem** are proved to be NP-hard: for the **Minimum s-walk problem**, one can prove that finding a minimum length s-walk is NP-hard by reducing from the *Hamiltonian cycle problem for graphs*. For the **de Bruijn Superwalk problem** one can reduce from the *Shortest common superstring problem*, proving that finding such a minimum length walk is NP-hard for $|\Sigma| \geqslant 3$ and for any k-mer length [MGMB07].

The hardness of these problems means that most likely there cannot exist polynomial-time algorithms that can compute them exactly. Because of this, multiple heuristics have been sought for solving them as accurately as possible, namely, by dividing the original problem into separate steps which one tries to solve with different approaches and algorithms and which are then merged together.

In practice, a usual reconstruction of a genome out of sequencing reads consists of a series of heuristic dependent stages, from the extraction of DNA and the obtaining of sequencing reads, the assembly of longer DNA stretches called *contigs*, the generation of *scaffolds*, to the gap filling steps and the joining of scaffolds into linkage groups [EW14]. We briefly explain these steps below:

In the sequencing step, a DNA is typically shreded randomly into small fragments of some given length and the process is repeated a number of times to ensure that there is enough coverage for all areas of the genome. As a next step, these reads are assembled together into longer DNA stretches which can include various types of algorithms and formulations. In the scaffolding phase, these contigs are joined in longer sequences with gaps between them, known as *scaffolds*, which are obtained by looking at mate-pair information or through optical mapping. The gaps between contigs in the scaffolds are then filled with different gap-filling methods based on the sequencing information. Finally, the resulting scaffolds are assigned to linkage groups or chromosomes using genetic maps to order and orient them into longer blocks.

We note that the assembly of a genome executed in this way is the result of a series of assembly heuristics and hence its correctness should be treated accordingly. By ensuring that each of its steps is as correct as possible, one can obtain a sensible final reconstruction. Because of this, we focus on one of the first steps: the contig assembly phase, where one must output strings which are promised to occur in the genome. In our case, we focus on de Bruijn graph algorithms as described above. Ideally, one would like to output those sequences which occur in any possible genomic reconstruction (according to our chosen problem formulation). In graph-theoretic approaches, since different genome reconstructions are possible depending on the nature of the reads, i.e. different node/edge-covering circular walks can be possible on the input graph, we are interested in those sequences (strings) which can be guaranteed to occur in any possible reconstruction, and therefore in the original genome. In the context of genome assembly, we will denote these strings by **safe**.

One of the main motivations for finding safe strings in contig assembly is that we can safely output and use these strings at successive steps of the reconstruction of the genome, namely at the scaffolding and gap-filling phases.

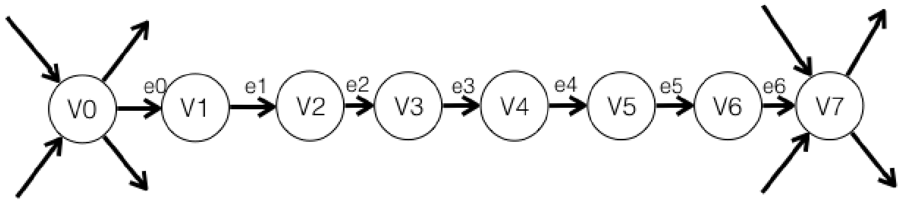

Figure 3: A unitig $U = v_0, e_0, v_1, e_1, v_2, e_2, v_3, e_3, v_4, e_4, v_5, e_5, v_6, e_6, v_7$ contains no internal nodes with in- or out-degree larger than one. A unitig can, however, have starting and ending nodes with in- and out-degrees larger than one.

For the de Bruijn superwalk problem, an easy to prove set of walks inducing safe strings are the so-called **unitigs**. Unitigs are strings spelled by sequences of internal nodes with in- and out- degree one (see Figure 3). For instance, take any internal edge e in a unitig w; in order to traverse e in any edge-covering walk, we need to traverse the whole of w since there is a unique way of reaching e and a unique way of leaving from it (since its previous and following internal nodes in w have in- and out-degrees equal to one). Making use of unitigs, is a common strategy which started in [KM95].

It is worth noting that while unitigs are certainly safe strings, they are not maximal safe solutions. Longer strings may also be safe while not fulfilling the unitig property, as we will demonstrate in Section 4.1.

In [TM17] maximal safe solutions are characterized for the de Bruijn graph model. These safe solutions represent safe walks from edge(node) covering reconstructions on de Bruijn graph but which can be longer in length than the unitigs. We will study these maximal safe solutions in Section 4.

Closely related to the contig assembly phase, gap filling phases can also be translated into the *safe solution* framework.

The problem of gap filling, which emerged as a natural sub-problem of many de novo genome assembly projects (e.g., filling gaps in scaffolds) is studied in [ST18]. They deal with the problem of multiple gap filling solutions, which we define next:

Definition 21. *In the gap filling problem we are given a sequence X containing gaps (e.g. scaffolds produced by a de novo assembler) and a set of sequencing reads. Our task is to fill in the gaps in X using the sequencing reads.*

The gap filling problem is usually formulated by building an assembly graph of the sequencing reads and then filling the gaps by finding suitable s-t paths whose length is consistent with the gap length estimate. [SSMT16]. This problem is proved to be NP-hard [NU02], but various heuristics have been developed, like using the shortest s-t paths whose length is consistent with the gap length estimate [MB09] or by computing a multiple alignment of the paths when there exist only at most a specified number of them [VJR+14]. In [ST18], they use the approach of filling a gap only with those paths of the assembly graph that are subpaths of all possible filling gaps (called *safe*), which applies to the cases when the solutions are not unique and is especially important for guaranteeing reliable results and decreased error rates. They show that in practice, they can retrieve a number of safe solutions by dynamic programming in $O(d^2 m \log m + m \operatorname{polylog}(m, d))$-time where d is the gap length estimate and m is the number of edges of the assembly graph.

Finally, a related bioinformatics problem is the **sequence alignment problem**. In the sequence alignment problem one must compute the similarity score between sequences (for example, of sequences of amino acids in proteins). Since several alignments are possible based on different similarity scores, one would be interested in knowing those aligned sequences which are likely to be correct. In the literature, several methods have been developed in order to retrieve reliable aligned sequences, for example by dynamic programming algorithms like the Needleman-Wunsch algorithm [NW70] or by using aminoacid substitution matrices [Alt91].

3 Safe walks common to all Eulerian cycles

Eulerian walks are a very well studied subject on graph traversals because of their number of applications and polynomial-time computability. An **Eulerian walk** in a graph G is a walk in G which traverses every of its edges exactly once; an **Eulerian cycle** is an Eulerian walk, whose first and last node are the same. The **Eulerian cycle problem** asks to obtain an Eulerian cycle given a graph G. (See Figure 4).

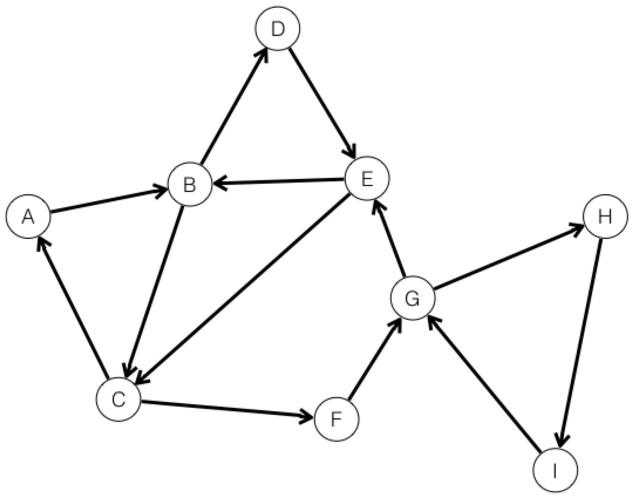

Figure 4: In the above graph, there exist three different Eulerian cycles, namely: **ABDECFGHIGEBCA**, **ABDEBCFGHIGECA** and **ABCFGHIGEBDECA**

Eulerian walks are not only used in bioinformatics applications (such as for algorithms to reconstruct the DNA sequence from its fragments) [PTW01b]; Eulerian walks are also used in for example circuit design to find optimal logic-gate orderings [Roy07].

In this section, we will focus on Eulerian cycles and finding those subwalks which are common to all possible Eulerian cycles of a given graph G. These walks will be our safe solutions for the Eulerian cycle problem, namely:

Definition 22 (Safe solution to the Eulerian cycle problem)**.** *A walk w is called safe for the Eulerian cycle problem, if and only if it is a subwalk of any Eulerian cycle of G.*

In the literature, we find previous work trying to characterize these walks. In [IW95], a characterization of Eulerian graphs which contain a unique Eulerian walk is given, which we cite below.

Theorem 1 (From [IW95])**.** *The intersection graph G_I of simple cycles from G is a tree if and only if there is a unique Eulerian cycle in G.*

Here, G is any graph, which is strongly connected.

An intersection graph G_I of simple cycles of G is constructed as follows: Decompose G into simple cycles $c_1, c_2, \ldots, c_k = c_1$, where no c_i is equal to c_j, except for $c_k = c_1$. An edge can be used by at most one cycle c_j, but vertices can be used arbitrarily many times. Now, we define the intersection graph G_I of the cycles $c_1, c_2, \ldots, c_k = c_1$ by replacing every cycle c_h by a single node C_h. We now have a graph with nodes $C_1, C_2, ..., C_k$, where if cycle c_h and c_j have l vertices in common, we connect C_h and C_j by l edges in G_I. See Figure 5 for an example. It is worth noting that the decomposition of G into simple cycles may not be unique, so that its corresponding intersection graph may not be either.

In order to make our understanding of Eulerian cycles in G and their corresponding intersection graph G_I of simple cycles clearer, we introduce the notion of *spanned edge* by a path p. Given a walk p in G, we say that p *spans* an edge (C_1, C_2) in the intersection graph G_I of simple cycles if p follows the edges in c_1 until its intersection with cycle c_2 and then moves on to traversing edges of the cycle c_2 instead. We note that by this definition, we do not specify the starting or ending point of the path, we only specify that it starts at a node in c_1 (and traverses at least one edge in c_1) and ends at a node in c_2 (traversing at least one edge in c_2). Note also, that there always exists at least one such path if there is an edge (C_1, C_2) in the intersection graph G_I of simple cycles: c_1 and c_2 correspond to different cycles in G and an edge between them corresponds to a node in common, so independently of the starting point when traversing cycle c_1, we will eventually reach its intersection point with c_2.

For an edge e in the intersection graph G_I of simple cycles of G, let us denote by $s(e)$ and $t(e)$ the two cycles for which e is the corresponding intersection, and denote by $v(e)$ the common vertex between $s(e)$ and $t(e)$.

Let us now present a different proof of Theorem 1 from the one presented in [IW95]. For it, we use the idea of spanned edges and how we can formally construct an alternative Eulerian cycle covering the same edges in G.

Proof. of Theorem 1. Let us prove the forward implication by induction. Let G_I be any intersection graph of simple cycles of G, and let it be a tree with n vertices. If G_I contains one vertex, then G has one cycle and the theorem is true. Assume the

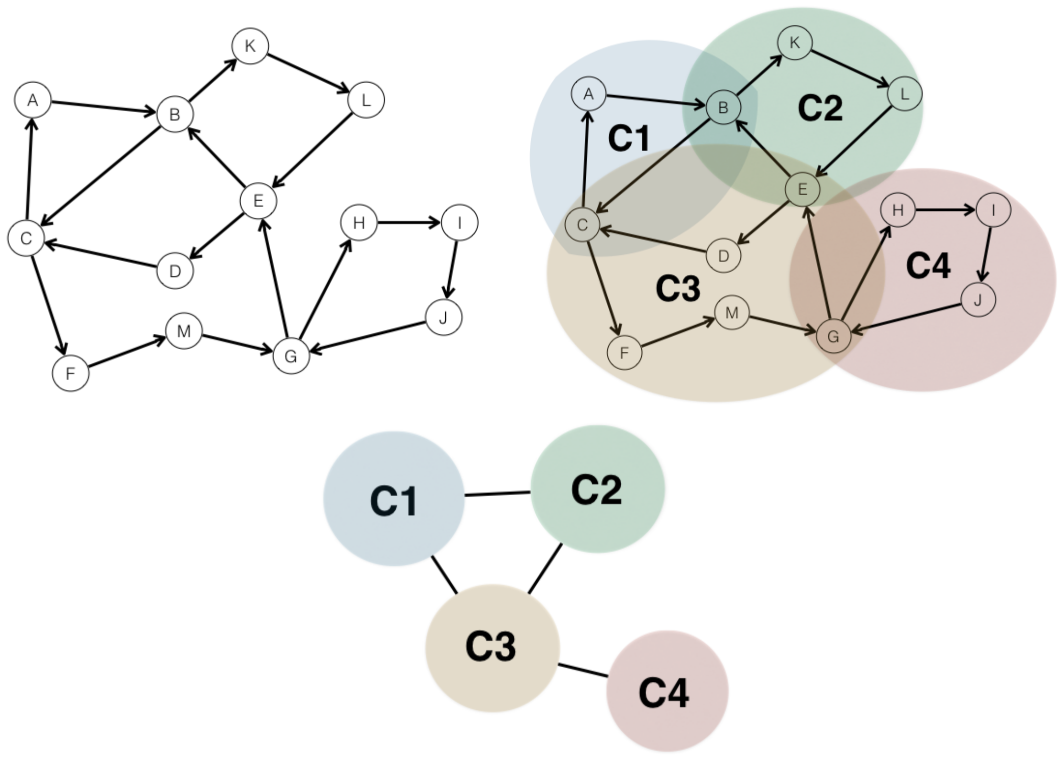

Figure 5: Construction of an intersection graph of simple cycles of a given graph G. The figure on the top-left is the given graph G. The figure on the top-right is one possible decomposition of G into simple cycles $C1, C2, C3$ and $C4$. The figure in the bottom is obtained from the second by making each cycle a single node and adding edges between them corresponding to their nodes in common. Namely, edge **(C1, C2)** corresponds to node B which is common to both $C1$ and $C2$, edge **(C1, C3)** corresponds to node C which is common to both $C1$ and $C3$, edge **(C3, C2)** corresponds to node E which is common to both $C3$ and $C2$ and edge **(C3, C4)** corresponds to node G which is common to both $C3$ and $C4$.

statement is true for all trees up to $n-1$ vertices. Consider a leaf corresponding to a cycle C in the n-vertex tree G_I. We note that C has only one vertex v in common with the graph G' obtained from G by removing the cycle C. Now, the graph G' is a tree with $n-1$ vertices, which by hypothesis has a unique Eulerian cycle E passing through v. Then, the unique Eulerian cycle in G starts in v, passes through E, back to v, through C and ends in v. In our notion of spanned edge, we note that the unique Eulerian cycle precisely corresponds to the concatenation of walks spanning only the edges in G_I and it is obtained by traversing the edges in G_I in a depth first manner.

Now for the reverse direction, we suppose that G has a unique Eulerian cycle and we will prove G_I is a tree. Assume for a contradiction that this is not the case and that G_I contains at least one cycle. Consider such a cycle X; for every cycle X_i in G_I (including X) iteratively remove one of its edges until both G_I and X remain trees; call these G_t and t_1 respectively. By the forward implication of this theorem (which we proved above), there exists a unique Eulerian cycle E_1 which passes through all the edges of G. We show how to construct another Eulerian cycle, using a different tree t_2 obtained by removing edges from G_I. Let f_1 be the last edge to be deleted from X before ending up with t_1. Since t_1 is a tree, we know that $t_1 \cup \{f_1\}$ contains a simple cycle c' where the deletion of any of its edges leaves the graph a tree. Delete from t_1 any edge f_2 from c' different from f_1 (one exists since a cycle has strictly more than one single edge) and add f_1 to t_1. By construction, $t_1 \cup \{f_1\} \setminus \{f_2\}$ is a tree and by the forward implication of this theorem, also contains a unique Eulerian cycle E_2 which passes through all the vertices of G.

In order to see that E_1 and E_2 are different, consider the cycles in G spanning the edges f_1 and f_2 in t_1: f_1 corresponds to traversing cycle $s(f_1)$ and then moving on to cycle $t(f_1)$ in E_2, passing through $v(f_1)$. This implies that E_2 contains a subwalk of $s(f_1)$ and a subwalk of $t(f_1)$ joined by their common vertex $v(f_1)$. In E_1, however, since f_1 is not contained in $t_1 \cup \{f_2\} \setminus \{f_1\}$, there cannot exist any intersection between cycles $s(f_1)$ and $t(f_1)$ (there would be an edge otherwise), so that no concatenation of edges in $s(f_1)$ and $t(f_1)$ with a common edge between them can exist in the Eulerian cycle, and therefore, E_1 and E_2 must be different. Finally, since both E_1 and E_2 cover all the edges in G, both are valid Eulerian cycles and this contradicts the fact that G contained a single Eulerian cycle. □

3.1 Characterization of safe walks

In [NP09] an algorithm for finding walks which are subwalks of all Eulerian walks is suggested based on the characterization in [IW95] of Eulerian graphs which contain a unique Eulerian Walk. We illustrate it below:

> "*In the case of Eulerian tours, reconstructing sub-tours that are part of every Eulerian tour is feasible in polynomial time (see Thm 7.5 in [Waterman, 1995]) based on finding acyclic subgraphs in the cycle-graph decomposition of the original graph.*"

We note that the *cycle-graph decomposition* which they mention refers to the intersection graph G_I of simple cycles of G. In [NP09] this claim is given without a proof. Below, we sketch a more complete characterization of safe solutions. It turns out that we indeed need to use the intersection graph of simple cycles, but the key notion for it ends up being the cut edges of G_I. We first introduce some notation and definitions and then prove some properties about these safe solutions.

We first note that the intersection graph of simple cycles from the original graph is made up of both nodes contained in subgraphs which are biconnected components, and nodes which are contained in subgraphs which are trees; keeping in mind this distinction, we can always create Eulerian walks in the original graph by *appending* subtrees to biconnected components and viceversa. Since the graph G is connected, the intersection graph of simple cycles is connected as well; assume that we are computing an Eulerian cycle passing through a given biconnected component, then, for a given tree t connected to it at node v, we can extend the Eulerian walk by introducing the unique Eulerian cycle passing though t (because of Theorem 1) which starts at v and one can then continue traversing the original Eulerian cycle in the biconnected component. Similarly, if we were traversing an Eulerian cycle in a tree t in the intersection graph of simple cycles, we could extend the Eulerian cycle in a similar fashion when t intersects with a biconnected component at node u by traversing an Eulerian cycle in the connected component first and then moving back to t.

With this notion of extension, we can restrict our attention to separate such components in the intersection graph of simple cycles, when proving safeness of certain paths since we can always merge them together as described above.

For instance, it is enough to prove that for each non-safe walk w contained in a biconnected component B and an Eulerian cycle spanning it, we can construct an alternative Eulerian cycle that does not span w, by using a different set of walks which span edges in B. We could then append this new sequence of nodes to the original Eulerian cycle at the correct position as mentioned above. We illustrate this idea in the following:

Lemma 3. *Given a walk w containing edges from two cycles c_1 and c_2 whose corresponding edge (C_1, C_2) in an intersection graph G_I of simple cycles of G is part of a non-trivial biconnected component, we can find an Eulerian cycle of G which does not contain w.*

Proof. Let B be a biconnected component in the intersection graph G_I of simple cycles of G. We show that we can construct an Eulerian cycle which does not use the path spanning edge (C_1, C_2) and which covers all edges of G. First, we note that we can always delete edges from B in order to form a tree. Furthermore, we can always delete the edge (C_1, C_2) (and possibly other edges) such that the remaining graph is a tree (this holds since B is a biconnected component and removing one edge does not leave it disconnected). Because of Theorem 1, once we have this tree t where we deleted edges from B, we can use the unique Eulerian cycle E that spans its edges and that covers all of B; furthermore, E only contains as subwalks, walks which span an edge in t or which are completely contained in one of the cycles of G. This implies that no walk w containing edges from two different cycles c_1 and c_2 and which spans the edge (C_1, C_2) (which we explicitly deleted from G_I) can be contained as a subwalk of E. Hence E is an Eulerian cycle covering all of G and not spanning edge (C_1, C_2). □

We note that Lemma 3 implies that for every edge (C_1, C_2) which is contained in a biconnected component in G_I, we can always find an Eulerian cycle that covers the entire graph and which does not span (C_1, C_2). This means that no edge (C_1, C_2) contained in a biconnected component G_I can be contained in all Eulerian cycles, and thus cannot be safe.

Claim 1. *Those walks that are subwalks of every Eulerian cycle in G are given by the connected components of the subgraph induced by the cut edges of the intersection graph G_I of simple cycles of G and by those paths in G which do not contain any internal node of in-degree or out-degree larger than one.*

Proof sketch:

Because of Lemma 3, no subwalk which contains consecutive edges from two cycles c_1 and c_2 and whose corresponding edge (C_1, C_2) in G_I is contained in a cycle can be safe. This means that only subwalks which span edges (C_1, C_2) that are not part of a biconnected component in G_I can both contain edges from different cycles and be safe. One can prove that an edge (C_1, C_2) not contained in a biconnected component (an edge which is a cut edge) is spanned by a safe walk in G. Furthermore, we claim that the concatenation of two adjacent edges (C_1, C_2) and (C_2, C_3) which are both not contained in a biconnected component in G_I is also spanned by a safe walk, so that the concatenation of the edges in the connected subgraph of G_I induced by its cut-edges is also spanned by safe subwalks. Finally, those paths in G which do not

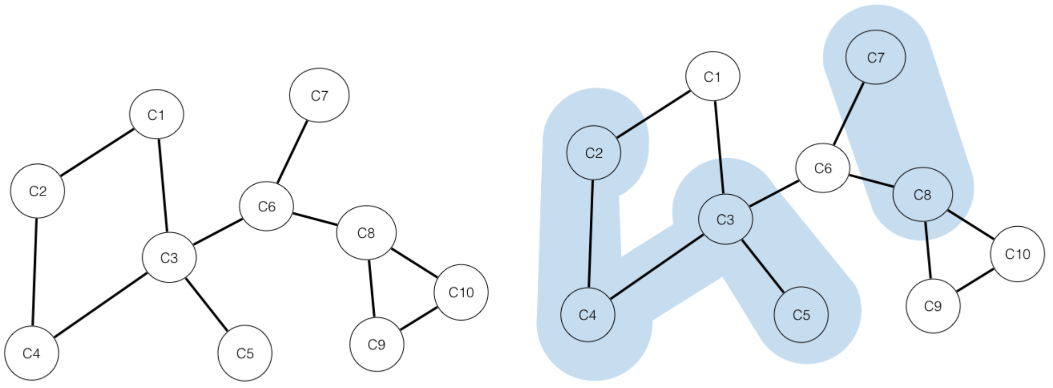

Figure 6: A graph obtained from the cycle-graph decomposition where its acyclic subgraphs don't necessarily correspond to safe walks. The graph on the left is the original graph, while in the one on the right the blue shade corresponds to the counterexample subgraphs which do not directly span a safe walk.

contain any internal node of in-degree or out-degree one (and are hence, contained in a single cycle) are precisely the *unitigs* defined previously and which we prove to be safe for any edge-covering walk of G (in particular an Eulerian cycle) in Section 4.1.

3.2 A polynomial-time algorithm

As we discussed above, [NP09] proposes a polynomial-time algorithm based on finding acyclic subgraphs of the cycle-graph decomposition. This however, lacks detail and a rigorous proof, and as we show next, is an incomplete characterization that can lead to erroneous safe walks.

In Figure 6 we show a graph where the acyclic subgraph obtained by taking the nodes $\{C_7, C_8\}$ results in two connected components from where it is not clear how to construct a safe walk; furthermore, the acyclic subgraph obtained by taking the nodes $\{C_2, C_4, C_3, C_5\}$, itself contains edges spanned by a non-safe walk, namely, edges (C_2, C_4) and (C_4, C_3) which are part of a biconnected component in the intersection graph G_I of simple cycles of G and which we showed above that cannot be safe.

Based on the lemmas presented in the previous subsection, we suggest a polynomial-time algorithm based on the subgraph induced by all cut edges of G. This works as follows:

- Given a strongly connected graph G and an intersection graph G_I of simple cycles, compute the set B of the cut edges of G_I and the corresponding subgraph G_B induced by B.
- Compute the set of connected components Q of G_B.
- For each connected component Q_i, compute their corresponding unique Eulerian cycle (by Theorem 1) since by definition Q_i is a tree.
- Compute all the *unitigs*, namely those subwalks of G which do not contain any internal node with in or out-degree bigger than one.

The algorithm is clearly polynomial since each of its steps takes polynomial time, given that there can exist at most n connected components in a graph H with n nodes and since G_B can contain at most n nodes (each node in G_B corresponds to a cycle in G and G contains n nodes).

4 Safe solutions for edge-covering circular walks

A natural generalization to Eulerian cycles where one must cover every edge exactly once are **general circular edge-covering walks** of graph G, where one must cover all the edges of G, without a constraint on the number of times any single edge can be covered.

In this section, we will be showing the characterization given in [TM17] for walks contained in every edge-covering circular walk on a strongly connected graph based originally on the genome assembly edge-centric de Bruijn graph model. We also illustrate the Unitig algorithm and the safe and complete Omnitig algorithm also described in [TM17]. This Omniting algorithm is based on (an exhaustive search) launching graph traversals from every edge of the graph. At the end of this section, we describe some of our results from [CMO$^+$17]. We present some combinatorial properties of these walks contained in every edge-covering circular walk which lead to a faster $O(mn)$ safe and complete algorithm for finding these maximal walks. We also give an upper bound on their number as well as a bound on their total length.

We will begin by formally defining the problems in the de Bruijn graph model which we had briefly described in Section 2.2.

Definition 23 (The safe and complete contig assembly problem (de Bruijn model))**.** *Given a set of reads R and a de Bruijn graph $B^k(R)$, output* all *the safe walks for $B^k(R)$.*

Definition 24 (Safe walk for $B^k(R)$ in the genome assembly problem)**.** *Given a set of reads R and a de Bruijn graph $B^k(R)$, a walk s is said to be a* safe walk for the de Bruijn model *if for every edge-covering walk, s is a subwalk of w.*

It is important to note that such model contains important assumptions based on the nature of the genomes sequenced, namely:

1. The genome sequenced is circular.
2. There is a good coverage of the genome sequenced; that is, there are no fractions of the genome which are left unsequenced and not represented in the reads.
3. There are no errors in the reads in the sequencing experiment; that is, no fragment of the genome is sequenced incorrectly.

From a theoretical perspective, however, these assumptions can be interpreted into a graph model: 1) can be interpreted as G being a strongly connected graph and the solution walks being circular, 2) is necessary for ensuring that all fragments of the genome sequenced are present in the graph and 3) is necessary for showing that one of the valid circular edge-covering walks in the graph corresponds to the true sequenced genome.

Using the de Bruijn graph model, we are able to characterize the safe solutions for the contig assembly problem. We note, however, that not all safe solutions are maximal, i.e. they can be contained in another safe solution, as we will see in Section 4.1.

4.1 Easy example: unitigs

As we have been mentioning in earlier sections, there exist walks which are contained in every circular edge-covering walks of the de Bruijn graph G of reads from a sequencing experiment. That is, there exist walks which are common to all possible edge-covering circular walks in a strongly connected graph. We note that each edge in G is contained en every edge-covering walk in G and hence is safe. There can exist, however, longer walks in G which are safe. We will use the *unitigs* as our first illustrative example of such walks, because of their simplicity and due to their importance in genome assembly. Let us recall their formal definition.

Definition 25 (Unitig). *Let $u = u_0, u_1, \dots, u_k$ be a path in a graph G. We call u an unitig if and only if the in-degree u_i equals 1 and the out-degree u_i equals 1 for all $i = 1, \dots, k-1$.*

Lemma 4. *Let $u = u_0, u_1, \dots, u_k$ be a unitig in G. Then u is a safe walk in G.*

Proof. First note that we have to traverse node u_1 at some point in the edge-covering walk (otherwise it would not cover all the nodes and hence the edges). Note also that while u_0 may have in or out-degree bigger than one, the only way of reaching node u_1 is by passing through u_0. Similarly, once we accessed node n_1 we need to continue traversing $u_2, u_3, \dots, u_{k-1}$ in order (since there is just one way of entering those nodes and one way of exiting them), then since u_{k-1} has out-degree 1, we will need to traverse u_k regardless of its out- or in-degree. □

We now note that even though *unitigs* are safe solutions, they do not characterize all safe solutions in a graph G. Furthermore, there may exist longer safe solutions which

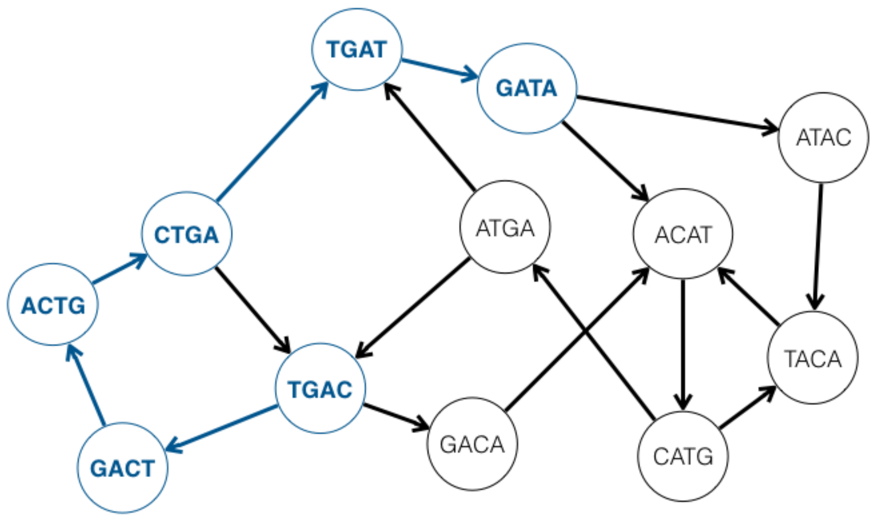

Figure 7: A walk in a de Bruijn graph G which is safe and which contains internal nodes with in- and out-degree larger than one, contradicting the unitig property. Indeed the walk (shown in blue) given by nodes $TGAC, GACT, ACTG, CTGA, TGAT, GATA$ is contained in all edge-covering walks in G and the in-degree of node $TGAT$ and out-degree of node $CTGA$ are both larger than one.

contain completely a *unitig* and which do not fulfil themselves the *unitig* property. We depict one such case in Figure 7.

Being able to characterize all safe solutions of a graph G in the de Bruijn model would allow us to obtain longer safe solutions for the initial steps of genome assembly and hence an easier subsequent steps in the whole genome reconstruction.

4.2 Characterization of safe strings: omnitigs

In this section, we provide a characterization of walks that spell safe strings taken from [TM17]. This characterization will be the basis of the safe and complete *omnitig* algorithm which we will present in the next section.

Definition 26 (Omnitig, edge-centric model)**.** *Let G be a graph and let $w = (v_0, e_0, v_1, e_1, \ldots, v_t, e_t, v_{t+1})$ be a walk in G. We say that w is a* omnitig *if and only if for all $1 \leqslant i \leqslant j \leqslant t$, there is no proper v_j-v_i path with first edge different from e_j, and last edge different from e_{i-1}. (See Figure 8.)*

The following theorem gives an equivalence between being safe (a subwalk of all edge-covering walks) and that of being an omnitig. See [TM17] for a proof.

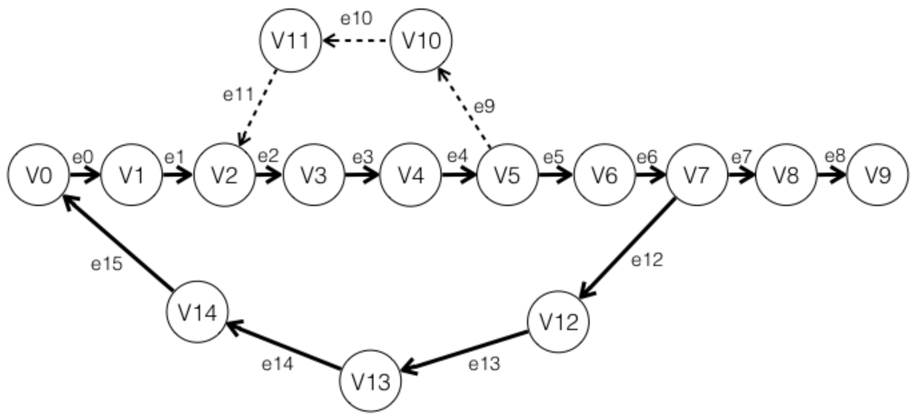

Figure 8: An omnitig $w = (v_0, e_0, v_1, ..., e_8, v_9)$ admits the path $p = (v_7, e_{12}, v_{12}, e_{13}, v_{13}, e_{14}, v_{14}, e_{15}, v_0)$ because its last edge is equal to e_{15}, but it does not admit the path $p_2 = (v_5, e_9, v_{10}, e_{10}, v_{11}, e_{11}, v_2)$ shown in dotted lines since it uses edge $e_9 \neq e_5$ as first edge and $e_{11} \neq e_1$ as last edge.

Theorem 2 (From [TM17]). *Let G be a graph. Then a walk w is contained as a subwalk in all edge-covering walks of G if and only if it is an omnitig.*

A very useful consequence from Theorem 2 is that one can check that a walk $w = (v_0, e_0, v_1, e_1, \ldots, v_{t-1}, e_{t-1}, v_t)$ is an omnitig by simply checking that there exists no path from any internal node v_i to any other previous internal node v_j in w which uses as first and last edges any edges different from e_i and e_{j-1}, respectively.

A very useful observation using this fact is that one can extend an omnitig w by adding an extra edge at its end and only checking for paths connecting the second-last vertex of the extended omnitig to other internal nodes in w. This observation is also exploited in the Omnitig algorithm and we illustrate it below.

Observation 2 (From [TM17]). *Consider a walk $w' = (v_0, e_0, \ldots, e_{t-1}, v_t, e_t, v_{t+1})$ of length at least two, and consider its subwalk $w = (v_0, e_0, \ldots, e_{t-1}, v_t)$. Then w' is an omnitig if and only if (i) w is an omnitig and (ii) for all $0 \leqslant i \leqslant t-1$, there is no proper v_t-v_i path with first edge different from e_t and last edge different from e_{i-1}.*

4.3 Omnitig algorithm

In this section, we depict and explain the omnitig algorithm (Algorithm 2) from [TM17], which is based on Theorem 2, and give a sketch proof that it runs in

Algorithm 2: Omnitig algorithm to find all safe strings of a graph G. Taken from [TM17].

```
1  extend($w$)
2      Denote $w = (v_0, e_0, v_1, e_1, \dots, v_{t-1}, e_{t-1}, v_t)$;
3      foreach edge $e = (v_t, y)$ out-going from $v_t$ do
4          $X := (N^-(v_1) \cup \cdots \cup N^-(v_t)) \setminus \{v_0, \dots, v_t\}$;
5          let $G'$ equal $G$ minus the edge $e$;
6          if there is no path in $G'$ from $v_t$ to a node of $X$ then
7              extend($(v_0, e_0, v_1, e_1, \dots, v_{t-1}, e_{t-1}, v_t, e, y)$);
8      if $w$ was never extended then
9          $W := W \cup \{w\}$;
10 $W := \emptyset$;
11 foreach edge $e = (u, v)$ of $G$ do
12     extend($(u, e, v)$);
13 remove from $W$ any walk that is a subwalk of another walk in $W$;
14 return $\{\text{spell}(w)\ :\ w \in W\}$;
```

polynomial time. The algorithm exploits the fact that one can easily extend an omnitig one edge at a time by Observation 2 and that since subwalks of omnitigs are also omnitigs, whenever an extension we of an omnitig w with an edge e ceases to be an omnitig, we can forget about any further extension using this extension we as prefix.

The algorithm starts at an edge (which is an omnitig) and starts extending it recursively in a depth-first fashion while it remains an omnitig. The algorithm tries extending the current omnitig $w = (v_0, e_0, v_1, e_1, \dots, v_{t-1}, e_{t-1}, v_t)$ at all edges outgoing from v_t. Once an omnitig cannot be extended, it is added to the set of right-maximal omnitigs. As a final step, the algorithm removes those omnitigs which are not maximal from the set W of maximal omnitigs and returns W.

In [TM17] they prove that the number of right maximal omnitigs obtained by the algorithm (before deleting non-maximal ones) is polynomial, and that furthermore, the length of each omnitig is at most mn.

For the polynomial-time complexity, we first note that the conditions needed to be checked inside the **foreach** loop, can be done in polynomial time by hiding edges from the graph and launching polynomial-time graph traversals. Using the fact that the length of each omnitig is at most mn, they prove that the algorithm reports each omnitig in polynomial time and that there are only polynomially many omnitigs reported.

We note, that obtaining maximal safe solutions makes sense not only on a genome assembly perspective, where longer initial safe solutions at the initial steps of assembly can lead to better heuristic solutions to the genome assembly problem. From a graph theoretical perspective, maximal safe solutions also contain non-maximal solutions and hence can be used to retrieve them, being a more compact representation of all safe solutions.

We now present an $O(mn)$-time algorithm to compute the set of all maximal omnitigs of a given graph G.

4.4 An optimal $O(mn)$-time algorithm

In [CMO+17] we studied further properties of the omnitigs, leading to an optimal $O(mn)$-time algorithm for finding all maximal omnitigs. In this section, we will state some of the main lemmas related to these results and we will then give the algorithm described in [CMO+17] along with a note on its complexity. In order to state these lemmas, we first need to introduce some relevant notation and definitions.

Definition 27. *A node u is called* branching*, if its out-degree is more than one. We call each edge e with $s(e) = u$ a* branch *and we call two different edges e and e' with $s(e) = u = s(e')$* siblings.

Definition 28. *We denote by G^R the reverse graph of G. This is the graph with same vertex set as G, $V(G) = V(G^R)$, and interchanged endpoints in its edges. In particular, for every edge $e \in E(G)$ and starting and ending points $s(e)$ and $t(e)$, we have an edge $e' \in E(G^R)$ with $s(e') = t(e)$ and $t(e') = s(e)$.*

We denote by *R-branch* an edge which is a branch in G^R.

Definition 29. *A walk is called univocal, if none of its edges is a branch, and R-univocal if none of its edges is an R-branch.*

We denote by wq the concatenation of two walks w and q, where $t(w) = s(q)$.

Definition 30. *On a graph G, an edge is called a* strong bridge *if its removal increases the number of strongly connected components in G.*

The algorithm presented in [CMO+17] is achieved by looking into three main properties of the maximal omnitigs, namely:

- There exists at most a single left-maximal omnitig which ends with a given branch in G.
- There is an acyclic partial order between all of the branches of the graph.
- There are at most $O(n)$ omnitigs not starting with a strong bridge. For the rest of the cases, these omnitigs are of the form fq where f is a strong bridge and q is an univical path.

We will justify these properties with the following lemmas taken from [CMO+17]:

Lemma 5. *If G contains at least a branch, then every univocal walk is an open path.*

Proof. Take for a minimal counterexample a univocal closed path p. We know that G is strongly connected, and since every path from $s(p)$ is a prefix of p, p has to contain every node in G, leaving no branches. □

Lemma 6. *If w is an omnitig and q is a univocal path from $t(w)$, then wq is an omnitig.*

Proof. Let p be a path which shows that wq is not an omnitig according to Definition 26. We now see that if $s(p)$ is a node of q, since q is univocal, a whole suffix of q has to be a prefix of p, such that the property that the first edge of p differs from e_j is not satisfied. Therefore $s(p)$ has to be an internal node of w, but this implies that p is a path which contradicts the fact that w is an omnitig. □

Lemma 7. *Every left-maximal omnitig contains a branch.*

Proof. Assume for a contradiction that there exists a left-maximal omnitig w which is univocal and (since G is strongly connected) let e be any edge with $t(e) = s(w)$. Since edge e is an omnitig, we can apply Lemma 6 to e and w to obtain that ew is an omnitig, contradicting w's left-maximality. □

Lemma 8. *Let fqe be an omnitig where q is an open path and e is a branch. Take any sibling e' of e and a closed path $e'p$ starting with e'. Then, fq is a suffix of $e'p$.*

Proof. Take for a contradiction, a minimum omnitig fqe such that fq is not a suffix of $e'p$. Since fqe is an omnitig, qe is also an omnitig and q is a suffix of $e'p$ because of minimality. We know that $q \neq e'p$ since q is an open path and so q should be a suffix of p. Then, we can rewrite $e'p$ as rq, where r is non-empty. Then we note that r is a path certifying that fqe is not an omnitig since it starts with and edge $e' \neq e$ and ends with an edge $f' \neq f$ (this is guaranteed by the fact that fq is not a suffix of $e'p = rq$), which contradicts the fact that fqe is an omnitig. $\square$

Lemma 9. *Let $e'pe$ be a walk where e and e' are siblings and $e'p$ is a closed path. Then, $e'pe$ is an omnitig if and only if p is univocal and e' is the only sibling of e.*

Proof. ($\Longleftarrow$) It is easy to see that $e'pe$ satisfies the conditions of an omnitig, since a path certifying that it is not an omnitig could not start at a node in p (p is univocal) and starting from $s(e)$ would imply that either is uses $t(e)$ or $t(e')$, where $e'pe$ would still maintain its omnitig property.

($\Longrightarrow$). First we show that e' is the only sibling of e. Take any sibling e'' of e and any closed path $e''p'$. Then, we can apply Lemma 8 so that $e'p$ is a suffix of $e''p'$; since both paths are closed, they must be equal and $e'p = e''p'$ with $e'' = e'$, so that e' can be the only sibling of e.

In order to show that p is univocal assume for a contradiction that it is not and write $p = qfr$, with f a branch. Take a sibling f' of f and a closed path $f'p$; since $s(e) \neq s(f) = s(f')$, f' cannot appear in the closed path $e'p = e'qfr$. Now take the shortest prefix q' of p' such that $t(q')$ is a node in p (one exists since $t(p') = s(f') = s(f)$ is in p). Note that the last edge of q' does not appear in $e'p$ and that $t(q')$ should be either a node in q or a node in r. In the first case, the path $f'q'$ is a path certifying that $e'qf$ is not an omnitig; in the second, the path $e'qf'q'$ is a path certifying that fre is not an omnitig. This shows that in either case, the walk $e'qfre = e'pe$ is not an omnitig, which is a contradiction. $\square$

Let us now prove the main theorem of this section using the previous lemmas.

Theorem 3 (From [CMO+17]). *There exists a unique left-maximal omnitig we, ending with a given branch e.*

Moreover, for any sibling e' of e and a closed path $e'p$, either:

- *$we = p'e'pe$, where p' is the longest R-univocal path to $s(e)$,* or
- *we is a suffix of pe,*

where the first case occurs iff e' is the only sibling of e and p is univocal.

Proof. In order to see that there is a unique left-maximal omnitig we for a given branch e we show that either we is a suffix of pe or of the form $we = p''e'pe$, where p'' is an R-univocal path. This also shows that at least one of the cases in the theorem must occur.

We first note that by Lemma 8, if w is an open path, then we must be a suffix of pe; if it is not, by taking the shortest suffix of w denoted by $e''p$ which is not an open path we get that $e' = e''$ again by Lemma 8, since p is an open path. Hence a counterexample is an omnitig $we = fqe'pe$ with q an R-univocal open path (by Lemma 5) and f an R-branch. We note that since $t(q) = t(p)$ and q is R-univocal, then q has to be a suffix of $e'p$; furthermore we can write $e'p = rq$ and $we = fqrqe$ where r is non empty, since q is also a suffix of p because it is open.

Next we show that there can exist no omnitig of the form $fqrqe$, where qr is a closed path, r is non-empty, e is a branch, and f is an R-branch. Assume it exists and let e' be the first edge of r, let $r = e'r'$, $e'' \neq e$ be a sibling of e and let $e''p$ be a closed path. Now, since $r'q$ is an open path, by Lemma 8 $e'r'q$ is a suffix of $e''p$. Furthermore, since both $e'r'q$ and $e''p$ are closed paths, $e'r'q = e''p$ and hence $e' = e'' \neq e$. We now note that we can apply the same argument to the reverse graph G^R, since the notion of omnitig is symmetrical. This implies that for the last edge f' of r, it holds that $f' \neq f$. Hence, we get a contradiction, since r is a non-empty path with first edge $e' \neq e$ and last edge $f' \neq f$ contradicting the fact that $we = fqrqe$ is an omnitig.

Finally, for the first case to happen we only need to satisfy the conditions in Lemma 9, since the fact that by applying Lemma 6 in the reverse graph gives us that $p'e'pe$ is an omnitig if and only if $e'pe$ is an omnitig.

□

Corollary 1. *There are at most m maximal omnitigs.*

Proof. We follow the proof given in [CMO+17]. We begin by noting that because of Lemma 7, every maximal omnitig w contains a branch; we can then write w as $w'er$, where r is an univocal path and e is its last branch. Because of Lemma 6, r is the longest univocal path form $t(e)$, which is uniquely determined by branch e. Similarly, because of Theorem 3, w' is uniquely determined by branch e. Therefore, every omnitig has to contain a last branch, and every branch is the last branch of

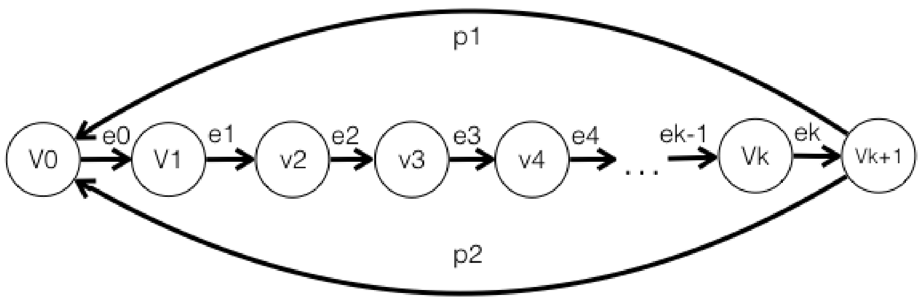

Figure 9: A graph with two maximal omnitigs $w_1 = lp_1lp_2l$ and $w_2 = lp_2lp_1l$ where $l = (v_0, e_0, v_1, e_1, v_2, e_2, v_3, e_3, v_4, e_4, \ldots, e_{k-1}, v_k, e_k, v_{k+1})$ and where every node is used exactly three times in both omnitigs.

at most one maximal omnitig. □

Let us also note that there exist families of graphs where the bound in Corollary 1 is tight, namely, in a complete graph on n nodes where each of its edges is a maximal omnitig.

Corollary 2. *Every maximal omnitig traverses any node at most three times, and thus has length at most* $3n - 1$*.*

Proof. We can rewrite any maximal omnitig w as $w = w'er$ where e is its last branch. Using Theorem 3, we note that either w' should be an open path or $w = p'e'per$ where p', p, r are univocal and open paths (by Lemma 5). Since open paths visit each node at most once, our claim follows. □

Corollary 3. *The total length of maximal omnitigs is* $O(nm)$*.*

In Figure 9 we can observe that there are families of graphs where the maximal omnitigs are of length $3n - 1$ and in Figure 10, graphs where the total length of omnitigs is $\Omega(nm)$, hence Corollaries 2 and 3 are also tight.

To conclude this section on the properties of omnitigs, we note that exploiting the fact that each left-maximal omnitig can be extended is crucial to the implementation of a faster optimal algorithm.

4.4.1 The algorithm

Using the previous properties of omnitigs, an $O(mn)$ algorithm has been described by [CMO+17]. We illustrate it below and we give a note on its complexity and

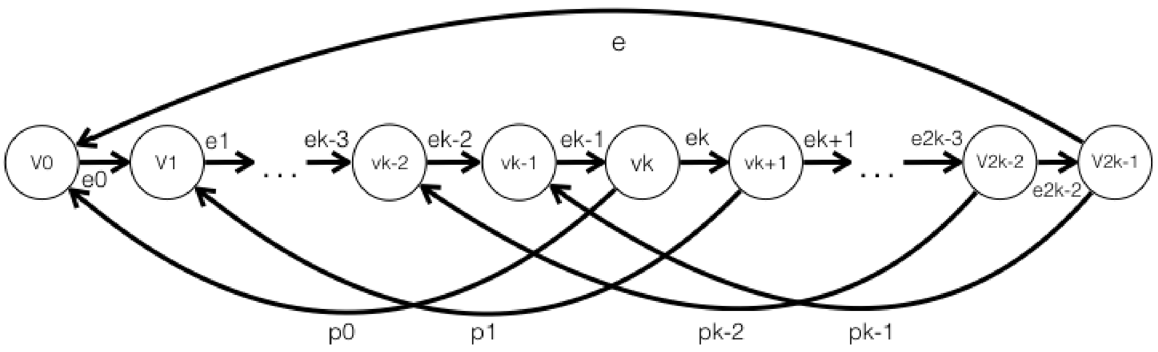

Figure 10: A graph where the total length of maximal omnitgs is $\Omega(mn)$. The walk $w_i = (v_i, e_i, v_{i+1}, e_{i+1}..., v_{i+k-1}, e_{i+k-1}, v_{i+k}, e_{i+k}, v_{i+k+1})$ is a maximal omnitig, for $0 \leqslant i \leqslant k-2$, and has length k + 1.

Algorithm 3: Algorithm to compute all the maximal omnitigs in G. Taken from [CMO+17]

```
1 $W := \emptyset$;
2 $B :=$ set of branches in $G$;
3 foreach $e \in B$ do
4     $w =$ OmnitigEndingWith$(e)$;
5     Let $p$ be the longest univocal path from $t(e)$;
6     $W = W \cup \{wp\}$
7 Remove those walks in $W$ which are not right-maximal;
8 return $W$, the set of maximal omnitigs.
```

correctness.

The main idea in Algorithm 8 is that because of Lemma 7 and Theorem 3, each maximal omnitig contains a branch e, and furthermore, we can find a unique left-maximal omnitig ending with a given branch e. This allows us to look at each branch in G, compute the unique left-maximal omnitig that ends with e and then append the longest univocal path starting at $t(e)$, which is right-maximal and by Lemma 6 is also an omnitig.

Algorithm 8 uses a subroutine OmnitigEndingWith(e) which finds the left-maximal omnitig ending with a given branch e. For e' a sibling of e, $e'p$ any closed path, f the last branch of $e'p$ and fq the suffix of $e'p$ starting with f, and p' the longest R-univocal path to $s(e)$, the left-maximal omnitig we ending with e is computed by one of four cases:

- $we = p'e'pe$ when e has only one sibling e' and p is univocal (by Theorem 3).
- $we = p'e$ when e is not a strong bridge and therefore e' could not be an internal edge.
- $we = \mathsf{LongestSuffix}(fqe)$, when $\mathsf{LongestSuffix}(fqe) \neq fqe$ and fqe is not an omnitig.
- $we = \mathsf{LongestSuffix}(w''qe)$, when fqe is an omnitig and where $w'' = \mathsf{OmnitigEndingWith}(f)$.

Here the function LongestSuffix(w) returns the longest suffix of w which is an omnitig.

For a correctness proof the reader can see [CMO+17]. However, we note that one of the previous cases must occur and therefore the algorithm always returns a left-maximal omnitig ending with branch e.

In order to see that Algorithm 8 can be implemented in $O(mn)$ time, we first note that the function LongestSuffix(w) can be implemented in $O(m)$ time by finding an acyclic partial order between the branches which allows to reuse computation and pay linear time per-branch.

For the OmnitigEndingWith(e) function, we observe that in the first two cases, the algorithm takes only $O(n)$ time, since the lengths of p and p' are linear. By using memoization, one then achieves an $O(mn)$ time bound for the executions of OmnitigEndingWith(e) where the first two cases occur.

Since there are $O(n)$ strong bridges in G, the number of executions for the fourth case is also linear, which together with the fact that LongestSuffix(w) can be implemented in linear time and *memoization* gives an $O(mn)$ bound. For the third case, an $O(mn)$ bound can also be shown by noting that the total length of the walks in W is $O(mn)$ and then using an auxiliary data structure to remove non-right-maximal ones.

5 Safe solutions for edge-covering collections of circular walks

Similarly as in the previous section, we are now interested in generalizing the notion of safe walks of a single edge-covering circular walks to safe walks of edge-covering **collections** of circular walks. In a graph theorectical perspective, one would be interested in finding those walks which are subwalks of at least one walk in the collection of circular walks for every possible edge-covering collection of circular walks.

The main motivation for this characterization in terms of *collections* comes from metagenomic assembly, the problem of reconstructing back the genomes of a series of organisms whose sequenced reads are all contained in the same set of reads.

In a biological setting, reads from multiple organisms could all have been sequenced together and one is then interested in reconstructing back each particular genome. This scenario could happen when for example, sequencing samples of the gut bacteria [SMT+13] or ocean environments [VRC+12] where the number of organisms to be reconstructed is unknown.

In this section we give characterizations for these walks contained in all collections of circular edge-covering walks which we originally proposed in [OMT18]. We adapt the proofs for node-covering walks (walks covering all nodes of a graph) for their edge-covering counterparts. We also show a safe and complete algorithm for finding all maximal such walks which runs in $O(m^2 + n^3 \log n)$ adapted from our results in [OMT18].

We first start with definitions for the **metagenomic assembly problem**, for the **de Bruijn metagenomic superwalk problem** and the **safe solutions** to it in a strongly connected graph G.

Definition 31 (Metagenomic assembly problem)**.** *Given a set of subsequences (reads) R obtained from a set $\mathcal{C}$ of (possibly) multiple different sequences (organisms), reconstruct back the set $\mathcal{C}$ from R.*

One can then define, similarly as for the **de Bruijn superwalk problem**, a **de Bruijn metagenomic superwalk problem**, where one is interested in finding a minimum cardinality set C of circular walks (genomes), which are of minimum total length and where each of the reads is contained in at least one of the circular walks. More formally:

Definition 32 (de Bruijn metagenomic superwalk problem (BMSP))**.** *Given a set of reads R and a de Bruijn graph $B^k(R)$ constructed from them, find a minimum cardinality set $\mathcal{C}$ of circular walks which are of minimum total length and such that every read in R is contained as a subwalk in some circular walk $c \in \mathcal{C}$.*

Let us call the solution to the **de Bruijn metagenomic superwalk problem**, that is, the set $\mathcal{C}$ of edge-covering circular walks which contains as subwalks each of the reads, a **metagenomic reconstruction**.

Note that the **genome assembly problem** is a special case of the metagenomic assembly problem for $|\mathcal{C}| = 1$ and hence, its NP-hardness follows. This motivates similar heuristics as the ones used in the single genome assembly problem; in particular, the use of contigs as an initial step of the metagenomic reconstruction.

Following the lines of Definitions 23 and 24 in the previous section one can define:

Definition 33 (The safe and complete metagenomic contig assembly problem (de Bruijn model))**.** *Given a set of reads R and a de Bruijn graph $B^k(R)$, output* all *the safe walks for $B^k(R)$.*

Definition 34 (Safe string for $B^k(R)$ in the metagenomic assembly problem)**.** *Given a set of reads R and a de Bruijn graph $B^k(R)$, a walk s is said to be a* safe walk for the metagenomic de Bruijn model *if for every metagenomic reconstruction $\mathcal{C}$ (collection of circular edge-covering walks), s is a subwalk of at least one $c \in \mathcal{C}$.*

5.1 Characterizations of safe walks

In this section we give characterizations of safe walks. We present the equivalent of Theorem 2 for the single edge-covering walk, namely that the property of being an omnitig, together with a simple additional property characterize the walks of a strongly connected graph G that are subwalks of all edge-covering collections of circular walks of G.

Theorem 4 (Adapted from [OMT18])**.** *Let G be a strongly connected graph. A walk $w = (v_0, e_0, v_1, e_1, \ldots, v_t, e_t, v_{t+1})$ in G is a safe walk in G if and only if the following conditions hold:*

(a) w is an omnitig, and

(b) there exists $e \in E(G)$ such that w is a subwalk of all cycles passing through e.

Proof. ($\Rightarrow$) Assume that w is safe. Suppose first that (a) does not hold, namely that w is not an omnitig. This implies that there exists a proper v_j-v_i path p with $1 \leqslant i \leqslant j \leqslant t$ with first edge different from e_j, last edge different from e_{i-1}, then, for any edge-covering reconstruction $\mathcal{C}$ of G, and any circular walk $c \in R$ such that w is a subwalk of c, we replace c in $\mathcal{C}$ by the circular walk c', not containing w as subwalk, obtained as follows. Whenever c visits w until node v_j, c' continues with the v_j-v_i path p, then it follows $(v_i, e_i, \dots, e_{j-1}, v_j)$, and finally continues as c. Since p is proper, and its first edge is different from e_j and its last edge is different from e_{i-1}, the only way that w can appear in c' is as a subwalk of p. However, this implies that both v_j and v_i appear twice on p, contradicting the fact that p is a v_j-v_i path. Since each such circular walk c' covers the same edges as c, the collection $\mathcal{C}'$ of circular walks obtained by performing all such replacements is also an edge-covering reconstruction G. This contradicts the safety of w.

Suppose now that (b) does not hold, namely, that for every $e \in E(G)$, there exists a cycle c_e passing through e such that w is not a subwalk of c_e. The set $R = \{c_e : e \in E(G)\}$ is an edge-covering reconstruction of G such that w is not a subwalk of any of its elements. This contradicts the safety of w.

($\Leftarrow$) Let $\mathcal{C}$ be an edge-covering reconstruction of G, and let $c \in \mathcal{C}$ be a circular walk covering the edge e. If c is a cycle, then (b) implies that w is a subwalk of c, from which the safety of w follows.

Otherwise, let $G[c]$ be the subgraph of G induced by the edges of c. Clearly, c is an edge-covering circular walk of $G[c]$, and thus $G[c]$ is strongly connected. Moreover, we can argue that w is an omnitig in $G[c]$, as follows. By taking the shortest proper circular subwalk of c passing through e we obtain a cycle $\widetilde{c}$ passing through e. From (b), we get that w is a subwalk of $\widetilde{c}$. Since all edges of $\widetilde{c}$ appear in $G[c]$, then also all edges of w appear in $G[c]$ and thus w is a walk in $G[c]$. The condition from the definition of omnitigs is preserved under removing edges from G, thus w is an omnitig also in $G[c]$. By applying Theorem 2 from the single edge-covering walk case to $G[c]$ we obtain that w is a subwalk of all edge-covering circular walks of $G[c]$, and in particular, also of c. We have thus shown that for every edge-covering reconstruction $\mathcal{C}$ of G, there exists $c \in \mathcal{C}$ such that w is a subwalk of c. Therefore, w is a safe walk for G. $\square$

The following statement is a simple corollary of condition (b) from Theorem 4.

Corollary 4. *Let G be a strongly connected graph, and let w be a safe walk in G.*

Then w is either a path or a cycle.

5.2 The algorithm for finding all safe walks

In this section we give an algorithm for finding all safe walks of a strongly connected graph and some necessary lemmas to prove its correctness adapted from [OMT18].

We begin with a lemma stating a simple condition when a maximum overlap of two omnitigs is an omnitig.

Lemma 10. *Let G be a graph, and let $w = (v_0, e_0, v_1, \dots, v_t, e_t, v_{t+1})$ be a walk of length at least 2 in G. We have that w is an omnitig if and only if $w_1 = (v_0, e_0, v_1, \dots, v_t)$ and $w_2 = (v_1, e_1, v_2, \dots, v_t, e_t, v_{t+1})$ are omnitigs and there is no v_t-v_1 path with first edge different than e_t and last edge different than e_0.*

Proof. The forward implication is trivial, as by definition subwalks of omnitigs are omnitigs. For the backward implication, since w_1 is an omnitig, then for all $1 \leqslant i \leqslant j \leqslant t-1$, there is no proper v_j-v_i path with first edge different from e_j, and last edge different from e_{i-1}. If there is no v_t-v_1 path with first edge different than e_t and last edge different than e_0, we obtain that w is an omnitig. $\square$

The following definition captures condition (b) from Theorem 4.

Definition 35 (Certificate)**.** *Let G be a graph and let w be a walk in G. An edge $e \in E(G)$ such that w is a subwalk of all cycles passing through e is called a* certificate *of w. The set of all certificates of w will be denoted $\mathsf{Cert}(w)$.*

By Theorem 4, safe walks are those omnitigs with at least one certificate. In the following Lemma we relate the certificates of an omnitig with the certificates of its edges.

Lemma 11. *Let G be a graph and let $w = (v_0, e_0, v_1, \dots, v_t, e_t, v_{t+1})$ be a proper omnitig in G. Then $\mathsf{Cert}(w) = \mathsf{Cert}(e_0) \cap \mathsf{Cert}(e_1) \cap \cdots \cap \mathsf{Cert}(e_t)$.*

Proof. We prove the claim by double-inclusion. The inclusion $\mathsf{Cert}(w) \subseteq \mathsf{Cert}(e_0) \cap \mathsf{Cert}(e_1) \cap \cdots \cap \mathsf{Cert}(e_t)$ is trivial, since all cycles passing through an edge $e \in \mathsf{Cert}(w)$ also contain each of $e_0, \dots, e_t$.

We now prove the reverse inclusion by induction on the length of w. The case when w is of length one is trivial, so we check the base case when w has length two. Assume for a contradiction that there is a cycle C passing through $e \in \mathsf{Cert}(e_0) \cap \mathsf{Cert}(e_1)$ and not having $w = (v_0, e_0, v_1, e_1, v_2)$ as subpath. Then, after visiting e, (i) C first traverses e_0 and then reaches e_1 with a path different than v_1, or (ii) C first traverses e_1 and then e_0. The case (i) cannot happen since admitting a proper path from the end of e_0 to the start of e_1 would imply that there is a proper v_j-v_j path, contradicting the fact that w is an omnitig. If (ii) holds, then C has to traverse the node v_1 twice: first at the start of e_1 and second at the end of e_0, contradicting the fact that C is a cycle.

We now use the inductive hypothesis to show if $e \in \mathsf{Cert}(e_0) \cap \mathsf{Cert}(e_1) \cap \cdots \cap \mathsf{Cert}(e_t)$, then $e \in \mathsf{Cert}(w)$. We partition w into the two walks $w_0 = (v_0, e_0, v_1, \ldots, v_t)$ and $w_t = (v_t, e_t, v_{t+1})$. By induction, since $e \in \mathsf{Cert}(e_0) \cap \mathsf{Cert}(e_1) \cap \cdots \cap \mathsf{Cert}(e_t)$ we have $e \in \mathsf{Cert}(w_0)$. Analogously, since $e \in \mathsf{Cert}(e_t)$, we have $x \in \mathsf{Cert}(w_t)$. Since v_t is a node in both w_0 and w_t, then any cycle passing through e, once it passes through w_0 it must continue passing through w_t. Therefore, any cycle passing through e passes also through w, and hence $e \in \mathsf{Cert}(w)$. □

Given a circular walk $c = (v_0, e_0, v_1, \ldots, v_{d-1}, e_{d-1}, v_d = v_0)$, $i \in \{0, \ldots, d-1\}$ and $k \in \{0, \ldots, d\}$, we denote by $c(i,k)$ the subwalk of c starting at v_i and of length k, that is, $c(i,k) = (v_i, e_i, v_{i+1 \bmod d}, \ldots, v_{(i+k) \bmod d})$.

Algorithm 4 finds all safe walks of a strongly connected gsaferaph G (possibly with duplicates), but does not return each safe walk explicitly. Instead, it returns an edge-covering circular walk c of G and the set of pairs (i,k) such that $c(i,k)$ is a safe walk. The algorithm works by scanning c and checking whether each subwalk of c of length k is an omnitig and has at least one certificate.

Theorem 5 (Adapted from [OMT18]). *Given a strongly connected graph G, Algorithm 4 correctly computes all the safe walks of G, possibly with duplicates.*

Proof. We will first prove by induction on k that the set S_k contains all those indices i for which $c(i,k)$ is a safe walk of length k. In the base case $k = 1$ (Line 6), we know that each $c(i,1)$ is an omnitig because all edges in G are omnitigs. We also check if $c(i,1)$ has at least one certificate, by checking (due to Lemma 11) whether $\mathsf{Cert}(e_i) \neq \emptyset$ (Line 7). Thus, for each i we checked whether $c(i,1)$ is a safe walk (due to Theorem 4), and the claim follows for S_1.

We assume now that the claim is true for S_{k-1}. For each i, by Lemma 10, $c(i,k)$ is an omnitig if and only if $c(i,k-1)$ and $c(i+1 \bmod d, k-1)$ are omnitigs, and there is no $v_{i+k-1 \bmod d}$-$v_{i+1 \bmod d}$ path with first edge different than $e_{i+k-1 \bmod d}$ and

Algorithm 4: Computing the safe walks of a strongly connected graph G. Adapted from [OMT18]

Input: A strongly connected graph G; $n = |V(G)|$.

Output: One edge-covering circular walk C in G, and all pairs (i, k) such that $c(i, k)$ is safe.

```
1  compute $c = (v_0, e_0, v_1, \dots, v_{d-1}, e_{d-1}, v_d = v_0)$ an edge-covering circular walk of $G$;
2  compute $\mathsf{Cert}(e)$, for every $e \in E(G)$;
3  for $k := 1$ to $n$ do
4      $S_k := \emptyset$;        // stores those indices $i$ for which $c(i, k)$ is safe
5      for $i := 0$ to $d - 1$ do
6          if $k = 1$ then
7              if $\mathsf{Cert}(e_i) \neq \emptyset$ then                    // $c(i, 1)$ is safe
8                  $S_k := S_k \cup \{i\}$;
9          else
10             if $i \in S_{k-1}$ and $i + 1 \bmod d \in S_{k-1}$ and there is no
               $v_{i+k-1 \bmod d}$-$v_{i+1 \bmod d}$ path with first edge different than $e_{i+k-1 \bmod d}$ and
               last edge different than $e_i$ then
11                 if $\mathsf{Cert}(e_i) \cap \cdots \cap \mathsf{Cert}(e_{i+k \bmod d}) \neq \emptyset$ then     // $C(i, k)$ is safe
12                     $S_k := S_k \cup \{i\}$;
13 return $c$ and $\{(i, k) : k \in \{0, \dots, n\}, i \in S_k\}$.
```

last edge different than e_i. This is verified in Line 10. In Line 11 we check whether $\mathsf{Cert}(C(i,k)) \neq \emptyset$ by checking whether $\mathsf{Cert}(v_i) \cap \cdots \cap \mathsf{Cert}(v_{i+k \bmod d}) \neq \emptyset$ (due to Lemma 11). Thus the claim is true for all S_k.

By Corollary 4, all safe walks of G are paths or cycles, thus of length at most n. By the definition of safe, they are also subwalks of c. Thus for each safe walk w of G of length $k \leqslant n$, there exists $i \in \{0, \dots, d-1\}$ such that $w = c(i,k)$ and $i \in S_k$. $\square$

Each of the steps in our algorithm takes polynomial time and it can be implemented in time $O(m^2 n)$ as we show in [OMT18]. Furthermore, we also show that it can be further optimized to output only maximal safe strings.

6 Conclusions

In this thesis, we focused on exemplifying and motivating the use of safe solutions as an extra tool in problems on walks on graphs. Furthermore, we justified the characterization of safe solutions for their own importance and structural properties which have proved to be of relevance in applications like bioinformatics.

We studied four main graph traversal problems, we presented full safe solution characterizations and safe and complete polynomial-time algorithms for three of them and we suggested a full characterization and safe and complete algorithm for the Eulerian walk problem. In particular, we studied the problem of finding all edges contained in all s-t paths between two biconnected components S and T in an undirected graph U (**Problem 1**), the problem of finding an Eulerian cycle in a graph G (the **Eulerian cycle problem**), the problem of finding a minimum length circular edge-covering walk in a graph G, and the problem of finding a minimum cardinality set of collections of circular edge-covering walks of minimum length.

We see this work, together with [TM17] as starting points and motivating examples for the characterization of safe solutions on broader combinatorial ambits: from related graph problems to problems on sets and functions.

6.1 Future work

We leave as future work an optimal polynomial-time algorithm for computing safe solutions to the Eulerian cycle problem and a rigorous proof of the given characterization of safe solutions. Similarly, we leave as an open problem an optimization to our polynomial-time algorithm for finding safe solutions in the metagenomic case which currently lacks a tight bound and which we believe could be improved by using similar optimizations as in Section 4.4.

A close variant to the problems studied in this thesis is presented in [TM17], where one considers instead node-covering circular walks (circular walks covering every node). One is then interested in finding those walks which are subwalks of every node-covering walk of a graph G. In [TM17], they present a characterization for these walks and a polynomial-time algorithm for computing them. However, their algorithm is not bounded and whether an optimal algorithm like Algorithm 8 which we presented in Section 4.4.1 can be achieved is still open. Similarly, in [OMT18] we studied the node-covering variant of the *safe solution to metagenomic assembly* *problem*, namely, we are interested in finding those walks which are subwalks of every node-covering collection of circular walks. We showed an analogous characterization to the one in Theorem 4 and a polynomial-time algorithm for computing them. It is, however, also open whether a similar optimization to the algorithms as in Section 4.4 which we originally presented in [CMO$^+$17] is possible.

Close to our problems, one would like to consider as a genomic reconstruction an edge-covering (node-covering) walk which is not necessarily circular, allowing us to model also non-circular genomes.

Following the line of multi-assembly, one would like to consider as a genomic reconstruction of a graph G, a set of s-t paths (where s and t are given source and sink nodes) that altogether cover all the nodes of G. This problem is known as the *minimum path cover problem* which for acyclic graphs can be solved in polynomial time using an algorithm based on Dilworth's theorem for partially ordered sets [Ful56]. One could translate this problem into a safe solutions setting by looking at the subwalks contained in all s-t paths, or those subwalks contained in at least one s-t path for every minimum path cover.

The safe solutions setting could be translated into other particular problems dealing with edge (or node) traversals on graphs; for instance a problem where one must find a walk satisfying a classification of the edges (nodes) into *optional, required, exact* or a classification of the edges (nodes) based on the number of times they must be traversed.

In a general combinatorial setting, safe solutions could be characterized as such in problems which involve sets, like in games and their sets of moves, or in social choice problems and sets of allocations. For example, one could look at those sets of items which are contained in every *complete envy-free allocation* for the problem of allocating indivisible items among two players with the allocation having the property of being envy-free, as defined in [BKK14].

References

Alt91 Altschul, S. F., Amino acid substitution matrices from an information theoretic perspective. *Journal of Molecular Biology*, 219,3(1991), pages 555 – 565. URL http://www.sciencedirect.com/science/article/pii/002228369190193A.

BGL02 Boros, E., Golumbic, M. C. and Levit, V. E., On the number of vertices belonging to all maximum stable sets of a graph. *Discrete Applied Mathematics*, 124,1-3(2002), pages 17–25. URL https://doi.org/10.1016/S0166-218X(01)00327-4.

BKK14 Brams, S. J., Kilgour, D. M. and Klamler, C., Two-person fair division of indivisible items: An efficient, envy-free algorithm. *Notices of the AMS*. URL http://www.ams.org/notices/201402/rnoti-p130.pdf.

BNT06 Bertsimas, D., Natarajan, K. and Teo, C., Persistence in discrete optimization under data uncertainty. *Math. Program.*, 108,2-3(2006), pages 251–274. URL https://doi.org/10.1007/s10107-006-0710-z.

BTT11 Bazgan, C., Toubaline, S. and Tuza, Z., The most vital nodes with respect to independent set and vertex cover. *Discrete Applied Mathematics*, 159,17(2011), pages 1933–1946. URL https://doi.org/10.1016/j.dam.2011.06.023.

CdWP11 Costa, M., de Werra, D. and Picouleau, C., Minimum d-blockers and d-transversals in graphs. *J. Comb. Optim.*, 22,4(2011), pages 857–872. URL https://doi.org/10.1007/s10878-010-9334-6.

Cec98 Cechlárová, K., Persistency in the assignment and transportation problems. *Math. Meth. of OR*, 47,2(1998), pages 243–254. URL https://doi.org/10.1007/BF01194399.

CL01 Cechlárová, K. and Lacko, V., Persistency in combinatorial optimization problems on matroids. *Discrete Applied Mathematics*, 110,2-3(2001), pages 121–132. URL https://doi.org/10.1016/S0166-218X(00)00279-1.

CMO$^+$17 Cairo, M., Medvedev, P., Obscura Acosta, N., Rizzi, R. and Tomescu, A. I., Optimal omnitig listing for safe and complete contig assembly. *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland*, 2017, pages 29:1–29:12, URL https://doi.org/10.4230/LIPIcs.CPM.2017.29.

Cos94 Costa, M., Persistency in maximum cardinality bipartite matchings. *Oper. Res. Lett.*, 15,3(1994), pages 143–149. URL `https://doi.org/10.1016/0167-6377(94)90049-3`.

EW14 Ekblom, R. and Wolf, J. B. W., A field guide to whole-genome sequencing, assembly and annotation. *Evolutionary Applications*, 7,9(2014), pages 1026–1042.

Ful56 Fulkerson, D. R., Note on dilworths decomposition theorem for partially ordered sets. *Proc. Amer. Math. Soc*, 7,4(1956).

GMS80 Gallant, J., Maier, D. and Storer, J. A., On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20,1(1980), pages 50–58.

HHS82 Hammer, P. L., Hansen, P. and Simeone, B., Vertices belonging to all or to no maximal stable sets of a graph. *SIAM Journal on algebraic and discrete methods*, 3,4(1982), pages 511–522.

IW95 Idury, R. M. and Waterman, M. S., A new algorithm for DNA sequence assembly. *Journal of Computational Biology*, 2,2(1995), pages 291–306. URL `https://doi.org/10.1089/cmb.1995.2.291`.

KBB$^+$08 Khachiyan, L., Boros, E., Borys, K., Elbassioni, K. M., Gurvich, V., Rudolf, G. and Zhao, J., On short paths interdiction problems: Total and node-wise limited interdiction. *Theory Comput. Syst.*, 43,2(2008), pages 204–233.

KM95 Kececioglu, J. D. and Myers, E. W., Combinatiorial algorithms for DNA sequence assembly. *Algorithmica*, 13,1/2(1995), pages 7–51.

Lac98 Lacko, V., Persistency in optimization problems on grpahs and matroids. Master's thesis, 1998.

Lac00 Lacko, V., Persistency in the traveling salesman problem on halin graphs. *Discussiones Mathematicae Graph Theory*, 20,2(2000), pages 231–242. URL `https://doi.org/10.7151/dmgt.1122`.

LM02 Levit, V. E. and Mandrescu, E., Combinatorial properties of the family of maximum stable sets of a graph. *Discrete Applied Mathematics*, 117,1-3(2002), pages 149–161. URL `https://doi.org/10.1016/S0166-218X(01)00183-4`.

MB09 Medvedev, P. and Brudno, M., Maximum likelihood genome assembly. *Journal of Computational Biology*, 16,8(2009), pages 1101–1116.

MGMB07 Medvedev, P., Georgiou, K., Myers, G. and Brudno, M., Computability of models for sequence assembly. *Algorithms in Bioinformatics, 7th International Workshop, WABI 2007, Philadelphia, PA, USA, September 8-9, 2007, Proceedings*, 2007, pages 289–301, URL `https://doi.org/10.1007/978-3-540-74126-8_27`.

Mye05 Myers, E. W., The fragment assembly string graph. *ECCB/JBI'05 Proceedings, Fourth European Conference on Computational Biology/Sixth Meeting of the Spanish Bioinformatics Network (Jornadas de BioInformática), Palacio de Congresos, Madrid, Spain, September 28 - October 1, 2005*, 2005, page 85, URL `https://doi.org/10.1093/bioinformatics/bti1114`.

NP09 Nagarajan, N. and Pop, M., Parametric complexity of sequence assembly: Theory and applications to next generation sequencing. *Journal of Computational Biology*, 16,7(2009), pages 897–908. URL `https://doi.org/10.1089/cmb.2009.0005`.

NU02 Nykänen, M. and Ukkonen, E., The exact path length problem. *J. Algorithms*, 42,1(2002), pages 41–53.

NW70 Needleman, S. B. and Wunsch, C. D., A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48,3(1970), pages 443 – 453. URL `http://www.sciencedirect.com/science/article/pii/0022283670900574`.

OMT18 Obscura Acosta, N., Mäkinen, V. and Tomescu, A. I., A safe and complete algorithm for metagenomic assembly. *Algorithms for Molecular Biology*, 13,3(2018).

PBP14 Pajouh, F. M., Boginski, V. and Pasiliao, E. L., Minimum vertex blocker clique problem. *Networks*, 64,1(2014), pages 48–64.

PTW01a Pevzner, P. A., Tang, H. and Waterman, M. S., A new approach to fragment assembly in DNA sequencing. *Proceedings of the Fifth Annual International Conference on Computational Biology, RECOMB 2001, Montréal, Québec, Canada, April 22-25, 2001*, 2001, pages 256–267, URL `http://doi.acm.org/10.1145/369133.369230`.

PTW01b Pevzner, P. A., Tang, H. and Waterman, M. S., A new approach to fragment assembly in dna sequencing. *Proceedings of the Fifth Annual International Conference on Computational Biology*, RECOMB '01, New York, NY, USA, 2001, ACM, pages 256–267, URL `http://doi.acm.org/10.1145/369133.369230`.

RBP$^+$10 Ries, B., Bentz, C., Picouleau, C., de Werra, D., Costa, M. and Zenklusen, R., Blockers and transversals in some subclasses of bipartite graphs: When caterpillars are dancing on a grid. *Discrete Mathematics*, 310,1(2010), pages 132–146. URL `https://doi.org/10.1016/j.disc.2009.08.009`.

Roy07 Roy, K., Optimum gate ordering of CMOS logic gates using euler path approach: Some insights and explanations. *CIT*, 15,1(2007), pages 85–92.

SBNK95 Schieber, B., Bar-Noy, A. and Khuller, S., The complexity of finding most vital arcs and nodes. Technical Report, College Park, MD, USA, 1995.

SMT$^+$13 Sharon, I., Morowitz, M. J., Thomas, B. C., Costello, E. K., Relman, D. A. and Banfield, J. F., Time series community genomics analysis reveals rapid shifts in bacterial species, strains, and phage during infant gut colonization. *Genome Research*, 23,1(2013), pages 111–120.

SSMT16 Salmela, L., Sahlin, K., Mäkinen, V. and Tomescu, A. I., Gap filling as exact path length problem. *Journal of Computational Biology*, 23,5(2016), pages 347–361.

ST18 Salmela, L. and Tomescu, A. I., Safely filling gaps with partial solutions common to all solutions. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, PP,99(2018), pages 1–1.

Tar74 Tarjan, R. E., A note on finding the bridges of a graph. *Inf. Process. Lett.*, 2,6(1974), pages 160–161.

TM17 Tomescu, A. I. and Medvedev, P., Safe and complete contig assembly through omnitigs. *Journal of Computational Biology*, 24,6(2017), pages 590–602. URL `https://doi.org/10.1089/cmb.2016.0141`.

VJR$^+$14 Vandervalk, B. P., Jackman, S. D., Raymond, A., Mohamadi, H., Yang, C., Attali, D. A., Chu, J., Warren, R. L. and Birol, I., Konnector: Connecting paired-end reads using a bloom filter de bruijn graph. *Bioinformatics and Biomedicine*. IEEE Computer Society, 2014, pages 51–58.

VRC$^+$12 V, I., RM, M., CD, F., CT, B., RL, M. and EV., A., Untangling genomes from metagenomes: revealing an uncultured class of marine euryarchaeota. *Science*, 335,6068(2012), pages 587–90.

Wag90 Wagner, D. K., Disjoint (s, t)-cuts in a network. *Networks*, 20,4(1990), pages 361–371.

ZRP^{+}09 Zenklusen, R., Ries, B., Picouleau, C., de Werra, D., Costa, M. and Bentz, C., Blockers and transversals. *Discrete Mathematics*, 309,13(2009), pages 4306–4314. URL `https://doi.org/10.1016/j.disc.2009.01.006`.