Independent Set on P_5-free graphs, an empirical study

Author:
Håvard Haug

Supervisor:
Daniel Lokshtanov

UNIVERSITY OF BERGEN

November 20, 2015

Dedicated to my family who are always supportive of my endeavours, and to my supervisor Daniel for being patient and kind.

Contents

Chapter 1

Introduction

In the field of complexity theory there are several important classes of problems (also known as languages). These classes exist to divide up the problems based on certain properties possessed by the algorithms which can solve the problems. Two of the most well-known classes of problems are the classes P and NP. An easy way to visualize these classes are that P consists of all the problems which are easy to solve for a computer and which are very often used in practice, such as sorting lists of numbers or strings, finding the shortest path between two points and finding out how different two words are (Edit distance). On the other hand NP consists of harder problems, some which are very hard to solve for computers, especially as the problems input size grows.

The classes of problems P and NP are related as we know that $P \subseteq NP$, and the question of whether $P = NP$ or $P \neq NP$ is probably the greatest unsolved mystery in computer science. Most people believe that $P \neq NP$, the problem has been intensively researched for many years and so far nobody has been able to show the opposite. If it turned out that $P = NP$ then this would be a major discovery with large amounts of practical importance, our computer security systems would probably have to be reworked and we would be able to make large amounts of interesting calculations on huge datasets which we are presently unable to do. Until a proof is found for either one we just have to work with what we have, and therefore many methods to solve problems in NP in a more efficient way have been developed.

A set of problems, which is a subset of NP is known as the set of NP-Complete problems. These problems can be said to be the hardest problems in NP and possess a very interesting property: for any other problem in NP, it is possible to transform the input instance to the problem, so that it becomes an instance of an NP-Complete problem and which can then be solved by an algorithm that solves the NP-Complete problem. This transformation can be done in polynomial time.

Definition 1.1. *A problem A is NP-Complete if for any problem B in NP with*

input instance I, it is possible to transform (reduce) I in polynomial time into a new instance I' of problem A. An algorithm which solves problem A can then be applied to I' and yield an answer which can also be used to answer problem B. For A to be NP-Complete it is additionally necessary that A belongs to the class NP.

Throughout this paper the Big-O notation is used to indicate how fast an algorithm is, and for some runtime function f(x), Big-O is written as $O(f(x))$. This is used to hide constant factors and make it more clear to the reader how the functions will grow with size. For instance, an algorithm with runtime $100 \cdot n^{10}$ will be written as $O(n^{10})$.

Algorithms must have some input to run on, and if an algorithm is designed to solve a certain problem A, then the input given to the algorithm is said to be an instance of problem A. The size of the instance is usually denoted by n, and $O(f(n))$ is used to describe the runtime as a function of the instance size. Polynomial Time algorithms are algorithms which run in time polynomial in n, so any polynomial time algorithm will have the Big-O notation $O(n^k)$ where n is the input size and k is some constant associated with the algorithm. Exponential Time algorithms are all the algorithms which require exponential time to execute. They will have their runtime described as $O(k^n)$ where k is some constant and n is the input size. It is easy to see that the difference in runtime between a polynomial time algorithm and an exponential time algorithm is enormous as the size of the input grows, and while most computers can probably manage to solve instances of sizes in the billions in a few seconds for some polynomial time algorithms, they may only be able to solve exponential time algorithms for instances of size less than a few hundred at best.

Utilizing a regular computer (as opposed to a quantum computer or some theoretical computational model), all the problems in the class P can be solved in polynomial time while the problems that are NP-complete will take exponential time. Even though the algorithms we have to solve NP-complete problems are generally slow, they are nonetheless important problems and can have many real life applications as well. This motivates us to find ways to deal with these problems in a better way. The common ways of speeding up an algorithm is to relax the solution requirements of the problem. Some ways of getting polynomial time instead of exponential time algorithms for NP-complete problems are:

Approximation algorithm Instead of requiring the best possible solution, it is acceptable for the solution to be a certain factor away from being optimal. This factor can either be a constant or some function of n.

Restricted input The algorithm will only run on certain instances of the problem, where these instances can be described and grouped together in a concise manner. This leads to large hierarchies and relationship systems for the different classes of inputs, where if you can solve one class in a better way you are also able to solve related problems in a better way.

Parametrized algorithm By using some additional input or property of the input as part of the algorithm, it is possible to edit the way in which the runtime is expressed, giving an algorithm which is not exponential in n.

We focus on the type algorithms which run on restricted input. The problem we consider is restricted to inputs of graphs which are P_5-free (definitions are found in the terminology section). This thesis explores a polynomial time algorithm for Independent Set in P_5-free graphs, described by Lokshtanov *et al.*[12]. Implementing the algorithm and performing empirical tests is the main priority. A large part of the project involves selecting the correct algorithms for smaller sub-problems which do not have well-defined algorithms in the paper which this thesis is based on, but rather just refers to the running-time which is possible. Algorithms we select need to satisfy the overall runtime proven in the paper, even though each sub-routine does not necessarily have to be optimal. A compromise is made between how easy an algorithm is to implement and how fast it is.

A secondary aim of this thesis is to improve the upper bound of the algorithm, either by refining the algorithm or by proving a better upper bound on the same algorithm. Implementation work is done in Java as there is a package of algorithms called Grapher that is already written in Java, and it contains some helper classes that should reduce the time and complexity necessary for the implementation. Java also seems to have equivalent performance to other languages for this type of problem from a general consensus of search results, although for some specific tasks C++ can be faster, these are not very applicable here and the improvement is not significant enough to make large differences in the outcome. Many of the results of interest are sizes of objects generated by the algorithm, and given a correct algorithm, the sizes of these objects will be the same regardless of language or framework or speed of computation. The number of easily accessed libraries in Java also seem to be large enough to cover most needs.

1.1 Independent Set Problem

Finding an independent set of a given size in an input graph $G = (V(G), E(G))$, where V(G) is the vertices of the graph and E(G) are the edges of the graph, is an extensively studied and well-known NP-Complete problem for general graphs. The Independent set problem is equivalent to one of Karp's 21 NP-complete problems[11]. Karp proved NP-Completeness of a problem known as Clique which queries if a graph contains a Clique of size k, where k is given as input. For a graph $G = (V(G), E(G))$ the complement graph D is a graph where for each pair of vertices $u, v \in V(G)$, if there is an edge $(u, v) \in E(G)$ then there is no edge between u and v in D while if the edge $(u, v) \notin E(G)$ then there is an edge between u and v in D. The Clique problem is the Independent Set problem in the complement graph. With this in mind, we know that to answer the question: "Is there an independent set of size k

in the graph G" we just have to answer the question "Is there a clique of size k in the graph D" and the answers to the two questions will be equal. This also shows that Independent set must be an NP-Complete problem, as otherwise we could use it to solve the Clique problem faster, and thus Clique would not be NP-Complete.

Definition 1.2. *An **Independent set** in a graph G is a set of vertices $S \subseteq V(G)$, such that for any pair of vertices $u, v \in S$, G does not contain an edge between u and v.*

There are multiple versions of the Independent set problem, the most commonly used are the decision and the optimization versions.

Definition 1.3 (Independent set decision problem)**.** *Given a graph G and an integer k, does the graph G contain an Independent set $S \subseteq V(G)$ such that $|S| = k$?*

Sometimes the decision problem will ask for an Independent set $S \subseteq V(G)$ such that $|S| \geq k$ instead of asking for equality. As an independent set which is larger than k must contain a subset of size k which is also independent, these two definitions are equivalent.

Definition 1.4 (Independent Set optimization problem)**.** *Given a graph G, pick as large a set S of vertices in G as possible, such that no two vertices in S are neighbours (share an edge) in G.*

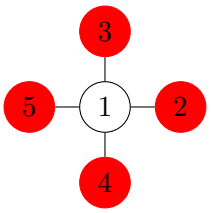


Figure 1.5: A graph with its maximum independent set marked in red. We are not able to add the node with id:1 into the independent set, as it has edges to other nodes already in the independent set, and so we would have to remove those. The resulting independent set would only contain the node with id:1 and be known as a maximal independent set, as we can not increase its size while the set remains independent.

We are mostly interested in the optimization version of the problem. For general graphs the naive way of discovering the largest possible independent set involves simply producing each subset of the set of vertices, test if it is an Independent set and then keep it if it is larger than any other set seen so far. This algorithm would require time $O(n^2 2^n)$ as the power set of a set is of size 2^n, and for each set in the power set one has to test if there are edges between any pair of vertices, and there are n^2 possible pairs of vertices. The best algorithm for Independent Set at present is an $O(1.2108^n)$ algorithm given by Robson[15]. For small graphs, this may be faster

than the polynomial time algorithm discussed in the thesis, but as n grows large, the algorithm in this paper will be much faster. It is also of theoretical interest to produce a polynomial time algorithm as a part of the larger effort to classify algorithmic problems on different graph classes.

The Independent set problem has numerous applications, among them are varying types of scheduling, map colouring (when using Independent set in relation to the graph colouring problem) and analysis of social networks. In general it is used in all sorts of relationship graphs where one wants to find a large group of events that don't depend on or obstruct each other. It is easy to find other applications where it could be used, even in a persons daily life. An example is a student wishing to take as many subjects as possible, with the requirement that the exams of two subjects can not overlap. The student can simply make a graph where each node is a subject and add an edge between two nodes if their exams are on the same day. The student can then find the largest independent set, which will be the largest set of subjects possible to take without exam collisions. Another example could be automatic parallelization of code where one could find parts of code that do not depend on each other and then executing them on different processors.

Given a restricted graph class, it is possible to solve this problem more efficiently. It is possible to solve Independent Set in polynomial time for claw-free[17] and perfect graphs[9] among others. From a recent paper, it is known that the Independent problem on P_5-free graphs is solvable in polynomial time[12]. The class of graphs that are P_5-free are the graphs not containing induced paths of length 5 (See 1.1 for definition of induced path).

1.2 Libraries and technologies

For convenience and to save some time, two larger libraries were used. One was used for most of the implementation work related to the Independent Set algorithm. Another library was used for generating graphs in a procedural way, to run certain tests across all graphs of a certain size, and for some random graph generation (discussed further in the Random neighbourhood subsection).

The first major library is named Grapher (which is strictly speaking an application, not a library). Grapher was created by the informatics department at the University of Bergen and it compiles to an Android App that is available in the Google Play store. This App allows a user to draw graphs by inserting nodes and edges. Afterwards the user can select from a number of algorithms which can be executed on the graph drawn. As this library contains a large number of pre-made algorithms, it seemed convenient to use it for the implementation of the Independent set algorithms of this thesis. As it is written for android, which requires Java code, this was also convenient as it is the language is very flexible. Grapher uses another library called

jgrapht, which is a large collection of graph structures and a few trivial algorithms on graphs. It was therefore necessary to conform to the graph class used across all the algorithms in Grapher. The standard way of representing a graph in the Grapher application is with a class called SimpleGraph, which represents an undirected, unweighted and simple(no multi-edges or self-loops) graph. SimpleGraph is a generic class where any class can represent an edge and any other class can represent a vertex. This is not the most convenient format to work with when this information is not required, so at the start of the Independent Set algorithm any SimpleGraph that is passed as input will be converted into a SimpleGraph with Integer values for vertices and edges, and a mapping is created so that it is possible to retrieve the original values if it should be relevant at any point. The edges in the SimpleGraph are represented by an integer which is created when an edge is added between two vertices, and this number is determined by the Cantor pairing function.

Definition 1.6. *The Cantor pairing function is a bijective function that maps any pair of non-negative integers to a unique number. The formula is* $F(x, y) = \frac{(x+y)(x+y+1)}{2} + y$.

As the graphs are undirected, the numbers x and y are sorted before given as input to the Cantor pairing function and thus adding edge (x,y) and (y,x) gives the same edge. This is very convenient as one can check in constant time in a hashtable if an edge has been added or not. The Cantor pairing function has the restriction that the size of the output will be up to n^2, so if it were given two numbers with the maximum size of a 16 bit integer, it would no longer fit in a 32 bit integer. This is irrelevant for our purposes as the largest integer will be that of the largest graph size used, and since this will never exceed 100, the function output is well within any limitation of the integer class. A few helper functions like finding closed and open neighbourhoods of a node or a set of vertices, and a function to make an induced sub-graph of a given graph together with the inducing vertex set, were also used.

The second library we use is S-Space. This is a very large library, contains all the algorithms necessary for the purpose of building Semantic Spaces. It was created by the Natural Language Processing group at UCLA. We used a single algorithm from this library: Graph Isomorphism, which would have been somewhat tricky to implement. This library was chosen because it was implemented in Java and so it could easily connect with the rest of the code. Isomorphism testing was necessary when building a list of all P_5-free graphs of a given size, as otherwise the number of graphs would have been too large to process. By building a list of non-isomorphic graphs it should be possible to test quickly if a graph is isomorphic to any of the graphs already in the list by creating some data structures to speed the comparisons up, but the algorithm in S-Space does a pairwise test between every graph in the non-isomorphic graphs list and the given graph to test isomorphism for. This puts a big restriction on how fast the algorithm is, and also limits the size of graphs which can be generated if one wants to generate all graphs of a given size. It was not feasible to generate all P_5-free graphs of graphs larger than size 9 due to the

time restrictions of the algorithm. It might have been possible to get some speed-up by either doing a custom implementation or looking at implementations done in C/C++. In addition it is possible that there may be faster graph Isomorphism algorithms for P_5-free graphs than for general graphs. To work within this library it was necessary to use a class of graphs called sparseUndirectedGraph which were simply unweighted and undirected graphs with no multi-edges or self-loops. The graph-class was easy to work with and was used for generating large samples of random P_5-free graphs of select sizes for different tests.

Each of the libraries had their own graph-class, and since a large number of test graphs were generated in the sparseUndirectedGraph graph-class, it was necessary to devise a way to convert between the classes. The solution was to make a binary string representation of the graphs, and from there it was easy to convert between the classes as the logic behind the graph structures was the same. For a more in-depth discussion of the bit representation of the graphs, see the "Generating P_5-free graphs" section.

In retrospect, it was too time consuming to work with all the different aspects of the libraries compared to how much of them were used. If it was possible to restart with current knowledge, a custom graph class with a very simple structure would have been used, and the only external algorithm necessary would have been for Graph Isomorphism, and then the C++ version from the "boost" package would have been used.

1.3 Notation and terminology

A graph G = (V,E) consists of a set of vertices (also referred to as nodes) V, and a set of edges E. An edge connects two vertices and an edge between vertices u and v is denoted as (u,v) or e(u,v). V(G) denotes the set of vertices in G and E(G) denotes edges in G. If v is a vertex in G we say that $v \in V(G)$ and if (u,v) is an edge in G we say that $(u, v) \in E(G)$. The size of the vertex set of G $|V(G)| = n$, and the set of G's edge-set $|E(G)| = m$.

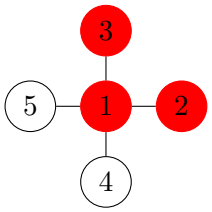
Each node in a given graph G of size n will in this thesis be definable by a unique id such that $id \in [0, n-1]$.

Simple graphs are graphs that disallow self-loops, more specifically $\forall u \in V(G), (u, u) \notin E(G)$.

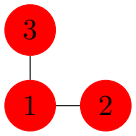
Undirected graphs are graphs where it is possible to travel along (traverse) an edge in both directions so $(u, v) \in E(G) \Rightarrow (v, u) \in E(G)$ and $(v, u) \in E(G) \Rightarrow (u, v) \in E(G)$, while directed graphs can have an edge pointing only one way.

Labelled graphs are graphs where each node is given some label, and two labelled graphs with a seemingly equal structure but different labels are not inter-changeable.

An induced sub-graph G[U] where $U \subseteq V(G)$ is a graph consisting of vertex set U and exactly the edges connecting vertices of U in G. An illustration of the process which creates a sub-graph is seen in figure 1.7. There we create an induced sub-graph as seen in figure 1.7b from an original graph seen in figure 1.7a, where the vertex set we use to induce the sub-graph is marked in red.



(a) Graph G with the vertex set U highlighted in Red.

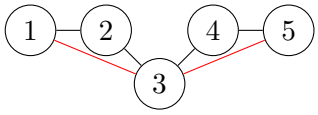


(b) The graph G[U], where only nodes of U and the edges between nodes of U remain

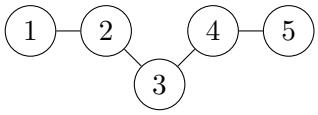
Figure 1.7: The process of creating an induced subgraph. The desired vertex set is marked in Red in the original graph, in the resulting graph, only the marked vertices and the edges which connects them remain.

For some vertex set C, G-C or $G \setminus C$ indicates the graph that remains when the vertices in C, together with any edges with endpoints in C, are removed from G. Let H be any other graph, $G \subseteq H$ if H contains all vertices and edges of G, and H can contain additional edges, we may also so that H is a super-graph of G.

A path in a graph is a set of vertices which can be placed in a sequence such that each vertex in the sequence has an edge to the next one, and the first and last element of the sequence are not connected (If they were then it would be a cycle). If a path is an *induced path* then the only edges connecting the vertices of the path are the ones between the consecutive pairs of vertices. The differences between induced and normal paths are shown in figure 1.8.



(a) A path from node 1 to node 5, with some extra edges within the path highlighted in red



(b) An induced path between nodes 1 and 5

Figure 1.8: In example a) we see a valid path between nodes 1 and 5, which is not an induced path as there are additional edges along the way. Example b) is an induced path of length 5 as there are no other edges than the ones between consecutive nodes in the sequence

Open and closed neighbourhoods for a vertex $v \in G$ are written as $N(v)$ and $N[v]$ respectively. $N(v)$ is the set of vertices which have en edge to v, while $N[v] = N(v) \cup v$. The notation can also be extended to $N_G(v)$ or $N_G[v]$ where one can insert the relevant graph G for which the neighbourhood shall be evaluated. It can also be generalised to a set of vertices C, where $N_G[C] = \bigcup_{v \in C} N_G[v]$, in other words create a new set and add the neighbourhood of each of the nodes in C to the set. If $_G$ is not included, it is assumed that the relevant graph should be assessable from context.

$\delta_G(v)$ denotes the neighbours of vertex v in G which have neighbours outside of the neighbourhood of v. Formally $\delta_G(v) = \{u \in N(v) : N(u) \not\subseteq N[v]\}$.

Picking two independent nodes u and v from G and turning $\delta_G(v)$ and $\delta_G(u)$ into cliques, a new graph G_{uv} is obtained. This is relevant as we can turn the graph G_{uv} into a chordal graph without adding edges between the rest of the graph and u or v.

Chordal graphs are graphs for which every cycle of size at least 4 has a chord within the cycle. These graphs have many special properties (e.g. limitation on the number of maximal cliques) which are exploited in the algorithm implemented in this thesis.

A *triangulation* H of a graph G is a super-graph of G ($G \subseteq H$) in terms of edges, such that H is a chordal graph with the same nodes as G and containing all edges of G, but with an additional set of edges F which we call fill-edges. If we let $G = (V(G), E(G))$, then $H = (V(G), (E(G) \cup F)$, where the addition of F turns G into a chordal graph.

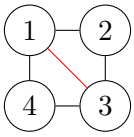


Figure 1.9: A chordal graph which is a triangulation of the graph C_4 where the chord which was added is indicated in red

A triangulation H of G is a *minimal triangulation* when there is no graph H' such that H' is a triangulation of G and a strict sub-graph of H ($\nexists H'$ such that $G \subseteq H' \subset H$).

We call a triangulation for I-good if for a set I of independent vertices in G, the triangulation H does not add any edges between vertices in I and any other vertices. We can call a triangulation good for a some set of I, so a u,v-good triangulation is an I-good triangulation with I = u,v.

A maximal object in a graph G is generally a set of nodes of the graph that has a specific property, and by maximal it is meant that you can not add any more items to the set while still retaining the property. Some specific examples are maximal independent sets, where a maximal independent set S stops being an independent set if one tries to add any other node to it, because all remaining nodes have an edge to a vertex already in S. Another example is a maximal clique Q for which no other node in the graph has an edge to all nodes in Q, so trying to add another node to Q will make it stop being a clique. All maximum(largest) objects are also maximal, but not all maximal objects are maximum.

Potential maximal cliques (referred to as p.m.c.) of a graph G, are cliques in some triangulation H of G which satisfy the criteria of proposition 4.1.

A minimal object is defined in the same way as a maximal object, but rather than not being able to add any more items to the set, in a minimal object you can not remove any items from the set and still retain the desired property.

Let an (a,b)-separator of a graph G be a vertex set S such that removing S from the graph, giving $G \setminus S$ which is a collection of components C, a and b are in different components in C. The separator is minimal if no subset of the separator still separates a from b. For a vertex set Ω, let $\Delta(\Omega)$ denote a function which returns the set of minimal separators contained within Ω.

Elimination ordering is any ordering of the vertices in a given graph.

A *Perfect Elimination Ordering*, commonly referred to as p.e.o. is an elimination ordering of a graph with a special property. For each vertex v in the p.e.o. v will form a clique together with its neighbours that occur later in the p.e.o. A graph is chordal if and only if it has a perfect elimination ordering[8]. Perfect elimination orderings are discussed more thoroughly in the Minimal triangulation chapter.

Connected graphs are graphs where there is a path between any pair of vertices in the graph, in contrast with a disconnected graph which can contain multiple components that are completely independent of each other.

A tree decomposition T of a graph G is a mapping of the graph into a tree. Given a graph G = (V, E), a tree decomposition is a pair T = (X, L) where X is a set of subsets of V which we refer to as bags, and L are edges connecting the bags to form a tree. We say that a vertex $b \in X$ is a bag of T, and for any bag b, we let $\chi(b) = S$, where $S \subseteq V$. For more in-depth discussion on tree decomposition, please consult other resources as it is a common concept, but we only briefly mention it in this thesis so we do not feel more information is needed.

1.4 Overview

By reading this thesis in the given order, each of the algorithms that were implemented during the thesis are explained in some depth. The main algorithm which the thesis is focused on is the polynomial time algorithm for independent set in P_5-free graphs, and the runtime of this algorithm depends on certain structures that are found in the graph. A way to list these structures is therefore necessary, so each algorithm described during this paper was implemented as needed, to be able to construct this list of structures.

P_5-free graphs and their properties that are relevant for our purposes are discussed, together with some methods to generate P_5-free graphs, both a procedural method to generate all possible P_5-free graphs of a certain size and multiple ways to generate random P_5-free graphs of a given size. Towards the end of the thesis we discuss the implementation of the Independent set algorithm itself, and afterwards provide both experimental and theoretical results. The experimental results will both look at the actual runtime of the algorithm when tested on the system available, and it will look at the size of certain objects generated that are a crucial part of the analysis of the algorithms runtime. The conclusion will summarize the findings and look at possible paths for further interesting study.

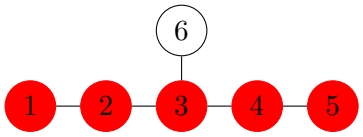
We discovered a dynamic P_5 detection algorithm with runtime $O(nm)$ (where n is the number of nodes and m the number of edges), described under the section with the same name. With this we were able to produce an algorithm which generates size n P_5-free graphs, only requiring time $O(n^3m)$.

Our test results indicated the possibility of a much lower upper-bound on the runtime than the current theoretical bound. This comes mainly from observing that the list of potential maximal cliques generated is much smaller than the bounds would suggest, together with a list of precursors used to generate the potential maxim cliques being smaller than expected as well.

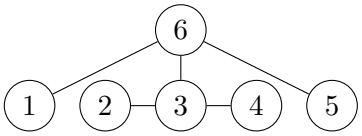
Chapter 2

P_5 free graphs

As mentioned in the introduction, a P_5-free graph is a graph not containing any induced paths of length 5. In figure 2.1 we give two example graphs, where graph a) is a simple graph containing a P_5, while graph b) is an example of a P_5-free graph.



(a) A graph containing an induced P_5, which is highlighted in red. We see that for each pair of nodes in the induced path, there are no other edges connecting the nodes than the edges between consecutive nodes



(b) An example of a P_5-free graph. Any graph which does not contain an induced P_5 is categorized as P_5-free

Figure 2.1: Two example graphs. a) shows a graph which is not P_5-free, as there is an induced path of length 5 highlighted in red, b) shows a graph with no induced path of length 5, the longest path in this graph has length 4.

If two nodes in a connected P_5-free graph are not neighbours, there must be at least one induced path between them, and these paths can contain at most 2 other vertices each. Due to this property, most random P_5-free graphs will have a high number of edges.

To find an independent set in a P_5-free graph, we re-purpose an algorithm originally designed to find the maximum independent set in a general graph. Some background is necessary to understand how this algorithm works, and how it is modified for the

use on P_5-free graphs.

All graphs contain some maximum size independent set I and minimal triangulation H, such that for each maximal clique C of H, the maximal clique contains at most 1 node from I. A formal lemma of this statement, together with a proof is provided in lemma 2.2.

Lemma 2.2. *For any graph G, let I be the maximum independent set in G.* $\exists$ *minimal triangulation H of G such that* $\forall$ *maximal cliques C in H,* $|C \cap I| \leq 1$.

Proof. As there are no edges between nodes in the maximum independent set I in G, they can never be in the same clique in G. For all nodes not in I, form a clique, this is a triangulation K of G. K is a split graph (a graph consisting of a clique and an independent set) and so it must be chordal. Either the triangulation K is a minimal triangulation of G, or there is a graph K' such that $G \subseteq K' \subset K$. K' can easily be obtained by removing edges, so we end up with a minimal triangulation in either case. For each maximal clique C in K', at most 1 node from I can be contained in C as no pair of nodes from I have edges between them and therefore are unable be in the same maximal clique. □

From a paper by Fomin and Villanger[6] we get the following proposition, which we can combine with the properties of P_5-free graphs to obtain a polynomial time algorithm for independent set.

Proposition 2.3. *There is an algorithm that given as input a vertex weighted graph G on n vertices and m edges, together with a list* Π *of potential maximal cliques, outputs in time* $O(|\Pi|n^5m)$ *the weight of the maximum weight independent set I such that there exists a minimal triangulation H of G such that every maximal clique C of H is on the list* Π *and satisfies* $|C \cap I| \leq 1$.

Even though the proposition calls for a weighted graph, it is easy to convert an unweighted graph to the right format. Simply add a weight of 1 to every node, and now the maximum weight independent set will also be the largest independent set in terms of number of vertices. This shows that the proposition also applies to unweighted graphs.

One of the main results of a paper by Lokshtanov *et al.*[12] is to show that for P_5-free graphs, one can compute a list Π of potential maximal cliques in polynomial time, such that for every independent set I there exists a minimal triangulation H of G, such that every maximal clique c of H is on the list Π, and satisfies $|C \cap I| \leq 1$. Particularly this holds for the maximum size independent set I, therefore the algorithm of proposition 2.3 when run on G with the list Π will find the size of the largest independent set of G. The size of Π affects the runtime of the independent set algorithm and in the paper by Lokshtanov *et al.* it is shown that the size of Π is upper-bounded by n^7. The purpose of this thesis is to investigate if the upper-bound is crude or tight.

2.1 Detecting P_5s in general graphs

Being able to detect whether a graph contains a P_5, and doing so in a reasonable time, is highly desirable for multiple purposes. A list of objects we call Π (discussed in the subsection "Generating Π"5.1) requires that a graph is P_5-free to get the required results. Based on the input it is possible to select whether or not it is sensible to execute the algorithm that generates Π, this will be done by checking that the graph is P_5-free. Another important application is in the ability to generate the random P_5-free graphs needed in the empirical tests. The algorithms used to generate random P_5-free graphs are based on taking a graph that is already P_5-free and making modifications to it, with the requirement that the graph retains its P_5-free property at every modification step. To achieve this, we require a fast algorithm to detect induced length 5 paths, otherwise we are unable to generate larger graphs.

The naive algorithm for finding a P_5 in a given graph G is to try all subsets of size 5, which can be counted as $\binom{n}{5} = \frac{n!}{5!(n-5)!} = \frac{n^5}{120}$ giving $O(n^5)$. This is guaranteed to use n^5 execution steps thus for practical purposes it will be rather slow, compared to algorithms which can have better average execution-times. We decided on using a better algorithm, which depends on the number of edges and is practically much faster.

From a recent paper by Hoàng *et al.*[10] we get an algorithm which can determine if a graph G contains an induced P_5 in time $O(m^2)$. The algorithms logic is for each edge (u,v) to make an attempt in extending it into a P_5 of the form uvwxy (shorthand for a sequence $\{u, v, w, x, y\}$ where only the consecutive pairs are connected by an edge) or vuwxy, and this is done by finding a P_4 in either $G \setminus N[u]$ or $G \setminus N[v]$.

An algorithm to find a P_5, in pseudo-code based on the implementation is seen in Algorithm 1. Initially a graph G is passed to the algorithm. It then searches through all possible edges in the orientation uv. Attempts to expand the edge into a path of the form uvwxy are made. On the step where it is initialised with edge (u,v) it lists all components of the graph G induced by $V(G) \setminus (N(v) \cup N(u))$ as $C_1, C_2, ...C_t$. If the graph contains a P_5 then it must also contain a P_4 ending in v, and where the only edge connecting the P_4 to u is uv. This type of P_4 can only exist if we have a vertex $w \in N(v)$ AND $w \notin N(u)$ such that the neighbourhood of w is a strict subset of one of the components listed earlier ($N(w) \subset C_i : C$). If a P_5 is detected, it can be output by selecting the endpoints u and v of the original edge, together with w. Then one must consider which component C_i met the sufficient criteria, and in this pick a vertex x which is a neighbour of w and a vertex y which is a neighbour of x but not of w. These vertices can be chained together form the P_5 uvwxy.

Theorem 2.4. *[10]There is an algorithm which takes as input a graph G and determines if the graph G is P_5-free. It is able to do so in time $O(m^2)$, where m is the number of edges in the graph. Additionally it is able to output a single P_5 in the*

same time-frame.

The idea behind the algorithm is illustrated in the following figure 2.5.

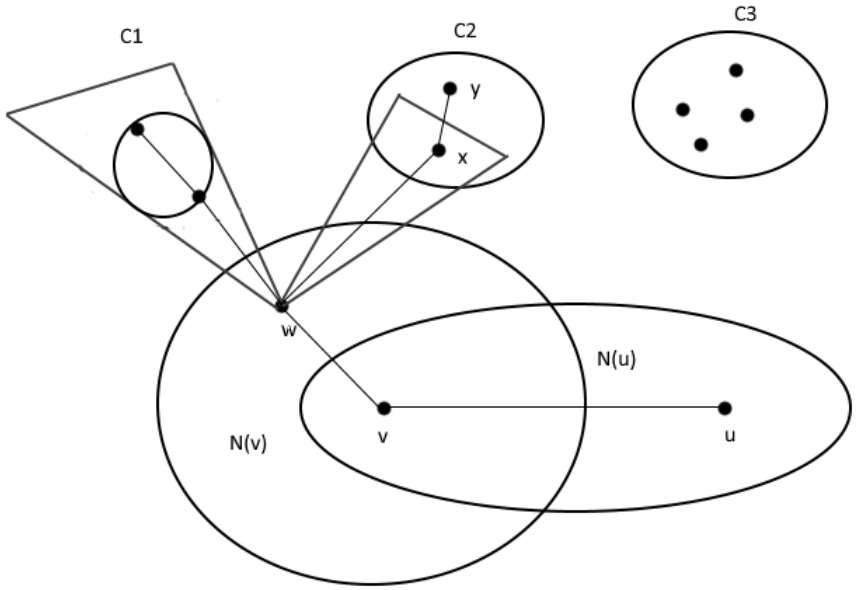


Figure 2.5: Illustration of the P_5 detection algorithm. w contains all of C_1 in its neighbourhood so it is not possible to form a length 3 induced path with vertices from C_1 and w. w contains none of C_3 in its neighbourhood so it is not considered. As w only contains part of C_2 in its neighbourhood we know that we can form a P_5.

In figure 2.5 there is a labelled P_5: uvwxy. The algorithm starts by looking through all edges in the graph, for the iteration where the P_5 shown was found it was considering the edge uv. Next it lists all the components $C_1, C_2, ..., C_n$ that is produced by taking $G \setminus N[u, v]$. It then proceeds to look for a third node w that is a neighbour of v but not of u. Now we have a P_3: uvw. To be able to expand uvw to a P_5, we must find a component C_i where w is a neighbour of some vertices but not all. In our example illustration, we see that w is a neighbour with both vertices in C_1, so we can exclude that, while in C_2 it is only a neighbour of x, so we keep expanding in that direction. Once it has added x, it just needs to find one more vertex in C_2 that is not a neighbour of any of the previous vertices in the path, and that is y. Our P_5 uvwxy is now complete.

If one is only interested in knowing whether there is a P_5 or not, it is possible to skip a few steps: once a w is found, count how many neighbours of w are in each

component. If the number of neighbours of w in component C_i is less than the size of C_i, then we know that it is possible to select one vertex from C_i that is a neighbour of w and one that is not, thus there is a P_5.

Algorithm 1 An $O(m^2)$ algorithm which will output true if the graph contains at least one induced path of length 5. If the algorithm has tested each edge in the edge-set and failed to find a P_5, it will output false

```
1: function FindP5(Graph G)
2:     for all uv : E(G) do
3:         B ← V(G) \ N[u, v]
4:         [C_1, ..., C_n] ← componentsOf(G[B])
5:         c[C.size()] ← 0
6:         Initialize Set L
7:         for all w : {w ∈ N(u) AND w ∉ N(v)} do
8:             for all d ∈ N(w) ∩ B do
9:                 j ← index of component containing d
10:                Count neighbours of w in component j: c[j] ← c[j] + 1
11:                Explore component j later: L ← L ∪ j
12:            end for
13:            for all j : L do
14:                if c[j] ≤ C[j].size() then
15:                    return "YES"
16:                end if
17:                c[j] ← 0
18:                L ← L \ j
19:            end for
20:        end for
21:    end for
22:    return "NO"
23: end function
```

By checking if $c[j] \leq C[j].size()$ we are testing if the component j contains a vertex which is not a neighbour of w. Due to the way they were generated, we know that the components are not neighbours of u and v, so once we have the P_3 uvw, all we need to do is to find a component that contains one x which is a neighbour of w, and one node y which is not a neighbour of w. As x and y are in the same component, there must be a path between them, and so we are guaranteed a P_5 if there is such a component where w is not a neighbour of every node in it. The time analysis results from considering $O(m)$ edges uv, and for each edge we need to list all components which takes time $O(m)$, guaranteeing a runtime of $O(m^2)$. Furthermore each w will visit its neighbourhood in the components, but each edge between a w in $N(u)$ and a component in C is just explored once, so this will also take time $O(m)$. The algorithm for computing the components is run separate of w, so our overall runtime

is $O(m^2)$.

If it is necessary to output the P_5 that the graph contains, then one must replace the line that returns yes with the code in algorithm 2. This continues to explore the component where x and y can be found. It does so by looking at all neighbours x of w in the component j, and then it looks for a vertex $y \in C_j$ which is not a neighbour of w. Once found, return uvwxy as the solution.

Algorithm 2 If a P_5 has been confirmed to exist, use this code-snippet to output full path of length 5

```
1: X ← N(w) ∩ C[j]
2: Y ← C[j] \ X
3: for all x : X do
4:     for all y : N(x) do
5:         if y ∈ Y then
6:             Solution ← v ∪ u ∪ w ∪ x ∪ y
7:             return Solution
8:         end if
9:     end for
10: end for
```

The P_5 detection algorithm must run twice for each edge uv, once in the orientation (u,v) and once in the orientation (v,u). Because the ordering of u and v give different results it is necessary to run both or we could miss a P_5. One can only conclude with the graph being P_5-free at the end of the second run. At first glance, it might as if actually producing the P_5 will exceed the runtime of $O(m^2)$ as algorithm 2 has runtime $O(nm)$ and the for loops that go outside it are of size $O(m^2)$, seemingly giving $O(m^3n)$ but upon more careful consideration one will realize that the code in algorithm 2 will only be executed if a P_5 is found, and this will only happen once, and so the runtime will instead be $O(m^2 + nm) = O(m^2)$.

An even faster algorithm for recognizing P_5s is not unfathomable, and would help in generating even larger P_5-free graphs, even though the final Independent set algorithm would not be affected by it.

2.1.1 Dynamic P_5-detection

Due to the way we apply the P_5 detection algorithm, we can modify it to give us a slightly better result in some cases. In particular, we often face the following problem: we have a graph G at our disposal, and we know that G is P_5-free, we add/remove an edge to G to obtain the graph G', and we wish to determine whether G' remains P_5-free. We can often use the knowledge that G was P_5 free, combined with the procedure by which we obtained G' from G. This allows us to check whether

G' is P_5-free in a manner which is faster than in the general setting. This setting is known as dynamic graph algorithms and is quite well studied, see the article by Nikolopoulos *et al.* [14] for a summary of known results for dynamic algorithms for graph class recognition. We now give a fully dynamic algorithm for P_5-free graph recognition, our algorithm supports edge addition and edge deletion in time $O(nm)$. To the best of our knowledge, this is the first algorithm beating order $O(m^2)$ for dynamic P_5-free recognition. We have implemented the algorithm and use it in the section ("Random edge generation"2.2.1). The dynamic P_5 recognition problem is described in definition 2.6.

Definition 2.6 (Dynamic edge P_5 detection)**.** *Given a P_5-free graph $G = (V(G), E(G))$, together with a pair of vertices u and v, determine if $G = (V(G), E(G) \cup (u, v))$ or $G = (V(G), E(G) - (u, v))$ contains a P_5.*

We consider the problems of adding and subtracting edges separately.

Incremental P_5-detection

We define the incremental P_5 detection problem as:

Definition 2.7 (Incremental P_5 detection)**.** *Given a P_5-free graph G, independent vertices u and v, determine if $G = (V(G), E(G) \cup (u, v))$ contains a P_5.*

When a P_5-free graph G gets a new edge (u, v), if $G \cup (u, v)$ is no longer P_5-free then we are always able to find a P_5 containing the edge (u, v). This statement together with a proof is found in lemma 2.8.

Lemma 2.8. *Given a P_5-free graph G where u and v are independent vertices, let $G' = (V(G), E(G) \cup (u, v))$. G' is no longer P_5-free if and only if there is an induced path P of length 5, such that $(u, v) \in E(P)$. Furthermore, any induced path of length 5 in G' must contain (u, v) as an edge.*

Proof. $\Leftarrow$ If G' contains an induced path P of length 5, such that $(u, v) \in E(P)$, then G' by definition contains a P_5.
$\Rightarrow$ Assume G' is not P_5-free but does not contain an induced path P of length 5 with (u, v) as an edge. Let one such P_5 be abcde (we could replace one of the vertices with either u or v, but both u and v can not be in the path by definition), so G induced by abcde produces the edge-set $\{(a, b), (b, c), (c, d), (d, e)\}$. Remove the edge (u, v) from G' to obtain the exact graph G. G should by definition be P_5-free, but G induced by abcde still produces the same edge-set as we have not added or removed edges between any pair of vertices in the path. Thus abcde is still a P_5 in G, contradicting that G was P_5-free. This proves both that if G' is not P_5-free then there must be an induced path P of length 5, such that $(u, v) \in E(P)$ and that for any induced path of length 5 in G' then (u,v) must be an edge in the path. □

Given that the newly added edge uv must be part of some P_5 for the graph to contain a P_5, we need to consider the configurations in which uv can be placed. Any P_5 will have the form p_1, p_2, p_3, p_4, p_5 where there are only edges between consecutive pairs of vertices, this gives us four possible positions for an edge to be. We can say that the edges (p_1, p_2) and (p_4, p_5) are symmetrical, and do the same for (p_2, p_3) and (p_3, p_4), as clearly if we are able to find a P_5 based on an edge in one of the orientations, then we can do the same for the opposite orientation. This leaves us with two options, an "outer-edge" like (p_1, p_2) and an "inner-edge" like (p_2, p_3).

To determine if (u, v) is an "outer-edge", we execute algorithm 1, but instead of looping through all edges in $E(G)$, we simply insert uv into the beginning and execute the algorithm as stated, and make sure to reverse it and test in the vu orientation as well. If uv is an "outer-edge" then it was either added at the start as uvwxy or at the end as yxwvu, but this is irrelevant as the algorithm will find it in either orientation (w has an edge to both x and v, $G \setminus N[u, v]$ will have x and y in the same component as they have an edge between them, and the same situation applies to $G \setminus N[x, y]$). This can clearly be done in time $O(m)$ as all we did was remove the $O(m)$ possible choices for edges in the $O(m^2)$ algorithm.

Determining if (v, w) (we renamed the vertices for convenience) is an "inner-edge" can be done as follows: for a P_5 uvwxy, given vw, find u and xy. We can again insert our vertices into algorithm 1, but this time we look at each u in $G \setminus N[w]$ instead of all edges, and we already know what the value of w is. For each such u, we split the graph into components by taking $G \setminus N[u, v]$ and finding x and y such that x is a neighbour of w and y is not. This can be done in $O(nm)$ as we consider at most n vertices as a possibility for u, and we count $O(m)$ components per n. w is fixed, so instead of $O(m)$ time to explore all possibilities for w as in the usual algorithm, we just explore the neighbourhood of w into the components, and this is bounded by $O(n)$.

So given that G was P_5-free before we added edge (u,v), we simply test if it is "outer-edge" and "inner-edge" P_5-free, with (u,v) as a parameter. If both answer true, then the resulting graph is P_5-free, otherwise a P_5 has formed when (u,v) was added, and (u,v) must be removed to make the graph become P_5-free again.

We end this subsection with lemma 2.9.

Lemma 2.9. *The incremental P_5-detection problem can be solved in time $O(nm)$.*

Proof. The proof is the algorithm described in the previous paragraphs. □

Decremental P_5-detection

Decremental P_5-detection is the problem of finding a P_5 after removing a single edge. The problem is accurately described in definition 2.10.

Definition 2.10 (Decremental P_5-detection)**.** *Given a P_5-free graph G with edge (u,v), determine if the graph $G = (V(G), E(G) \setminus (u, v))$ is P_5-free.*

To create an algorithm for decremental detection, we need to use the result of lemma 2.11.

Lemma 2.11. *Let G be a P_5-free graph with (u,v) an edge in G, let $G' = (V(G), E(G) - (u, v))$. G' contains a P_5 if and only if there is an induced path P on 5 nodes with $u \in P$, $v \in P$, $(u, v) \notin P$.*

Proof. $\Leftarrow$ if there is such an induced path P on 5 nodes, G' necessarily contains a P_5.
$\Rightarrow$ clearly the edge (u, v) is not in the path, as this is not an edge in the graph. Assume there is an induced path L on length 5 such that at most one of $\{u, v\}$ is in the path. Add back the edge (u,v) to obtain G, L must also be an induced path of length 5 in G, as G induced by the path can have no more edges than before, this contradicts the statement that G is P_5-free. Thus any P_5 in G' must contain both u and v. □

Next we need to consider the possible orientations in which u and v can be found in the path. As they do not share an edge, they can not be adjacent in the path. This means that we can have $\{u, p_2, v, p_4, p_5\}$, or $\{u, p_2, p_3, v, p_5\}$, or $\{u, p_2, p_3, p_4, v\}$ or lastly $\{p_1, u, p_3, v, p_5\}$, and any other configuration is just a rotation of one of the states given. With this in mind, we can see that the problem can be reduced to finding a P_5 where either u or v is one of the endpoints of the P_5, or where neither is an endpoint. If one of them is an endpoint, then we can determine if such a P_5 exists by executing algorithm 1 with either u or v pre-determined, and looking at their neighbourhoods. This is done instead of looking over $O(m)$ edges, so we look at the neighbourhood of two vertices, giving us $2n$ possible edges. Components and w is computed in the usual $O(m)$ manner, so an overall runtime of $O(nm)$ is obtained. The second configuration where neither is an endpoint can be determined by letting one of the vertices represent v in the P_5 uvwxy, then looking at each possible u in its neighbourhood. For any w we consider, ensure that u and w are not neighbours. The rest of the algorithm is executed as normal, again giving us a runtime of $O(nm)$

We combine these efforts into making executing the endpoint test for u and v, and subsequently performing the other test for u and v. If none of these provide a P_5, the resulting graph must be P_5-free, otherwise there is a P_5 in the graph.

Lemma 2.12. *The decremental P_5-detection problem can be solved in time $O(nm)$.*

Proof. The proof is the algorithm described in this section. □

Fully dynamic P_5-detection

To solve the problem given in definition 2.6, we combine the two lemmas obtained in the previous sections, that is lemma 2.9 and lemma 2.12. This gives us an overall algorithm to solve the problem together with theorem 2.13.

Theorem 2.13. *The Dynamic P_5-detection problem can be solved in time $O(nm)$.*

Proof. If an edge is added, apply lemma 2.9 which gives an $O(nm)$ time solution. If an edge is deleted, apply lemma 2.12 which gives an $O(nm)$ time solution. □

We can now apply this theorem whenever we need use a relevant algorithm.

2.2 Generating P_5-free graphs

Multiple methods for generating P_5-free graphs for later tests were explored. In an optimal situation, the method should be able to generate any size k graph from the set of all P_5-free graphs of size k, and the probability of it selecting any of the graphs should be equal. An algorithm which generates uniformly random P_5-free graphs for practical purposes has to be able to do the following: given an integer n, in time polynomial in n, output a graph G of size n, such that G is P_5-free and for for any P_5-free graph H from the space of all P_5-free graphs of size n, $P[G = H] = \frac{1}{X_n}$, where X_n denotes the number of P_5-free graphs of size n. An open question of this paper is if there is a way to generate uniformly random P_5-free graphs with a reasonable amount of efficiency. The methods used in this paper can not guarantee $P[G = H] = \frac{1}{X_n}$ (uniform distribution), the first method can not guarantee $P[G = H] \neq 0$ (the sample space contains all possible P_5-free graphs), while the second method of generation can.

Due to the expected runtime of the algorithm, it did not seem necessary to generate very large graphs and so the methods focused on generating small P_5-free graphs. After implementing a larger portion of the algorithm, certain run-times were quite a bit lower than expected, so more experimental results from larger graphs would have been very interesting. Unfortunately the speed of graph generation by the methods provided here are not fast enough to get a good sized set of large graphs.

Several methods for generating the graphs were considered. As the final algorithm can run on separate components independently on each other, we are only interested in connected graphs. The most intuitive way to get P_5-free graphs of a given size with a completely random distribution would be to generate completely random graphs of the given size, then one would keep the graph if it was a connected P_5-free graph and discard it otherwise. This method depends directly on two things, the speed at which one can detect if a graph contains a P_5 and the ratio of the

number of graphs of size n to the number of graphs of size n which are P_5-free and connected. As this ratio probably means a large number of misses and time wasted on each verification, other methods were considered. We know that the probability of picking a P_5-free graph from the set of all random graphs of size n is at best 0.988^n. There are $2^{\binom{n}{2}}$ total labelled graphs of size n. By considering labelled graphs of size 5, we can generate a P_5 in $\frac{5!}{2}$ different ways, as this is the number of ways to order the 5 vertices, and as each ordering is unique except the ones which are equivalent by rotating the graph, we get the previous number. It follows that the probability p of selecting a not P_5-free graph on 5 nodes is $\frac{5!/2}{2^{\binom{5}{2}}} \approx 0.0586$. The probability of a graph being P_5-free is then $(1-p) = 0.941$. For a graph on n nodes we generalize the result by placing the nodes into groups of 5, and we end up with $\frac{n}{5}$ such groups. For each group, the probability of it being P_5-free remains the same as for the graph on 5 nodes, so the probability of all of them being P_5-free will be $(1-p)^{n/5} = (1-p)^{1/5^n}$ and replacing $(1-p)$ with the previously calculated probability and taking the 5th root we end up with 0.988^n as an upper bound on the probability of a graph on at least 5 nodes being P_5-free. As this is exponential we can see that the probability of a large graph being P_5-free is small, e.g. a graph on 100 nodes has a probability of about 0.3 of being P_5-free. But there are many more ways of forming P_5s so although this might be the best case scenario, in reality the chance of a random graph of size 100 being P_5-free is probably close to 0. This method was used to generate a few samples of relatively small graphs (size 10) for some benchmark purposes, but it did not scale as n increased, without setting the edge density of the $G(n,p)$ graphs to a very high amount.

The next method under consideration was to start with a graph containing n nodes and no edges, where n is the desired final graph size. This graph is clearly P_5-free, so one can randomly pick a pair of nodes and add an edge, and if the graph remains P_5-free after this addition, the edge stays, otherwise it is removed. Although this seems like an intuitive method of making random graphs, it is shown in the next section why this will not work. A similar idea based on starting with a clique and removing edges at random was also thought of, but this is also shown to be futile in the next section.

The two methods which were finally decided on are probably not uniformly random, but they seem to give good enough results to perform the tests for which the graphs are needed. The first method described is based on combining the two concepts in the previous paragraph, by both adding and removing edges and as such it should be possible to generate all P_5-free graphs, although this has not been proven. The second method used simply starts with any P_5-free graph and expands it into a graph of the desired size by adding nodes one by one, and picking a random edge-set from the new node into the existing vertices which makes the graph remain P_5-free. As it is possible to pick any P_5-free extension of a given graph, one can see that this is able to generate absolutely all P_5-free graphs.

2.2.1 Random edge generation

With this method, one starts by specifying how large the desired graph should be. The algorithm then creates a set of vertices with no edges between them. Next the algorithm will pick a pair of vertices at random and it will either try to add an edge or remove an edge, but it will only edit the graph if it remains P_5-free after the edit has happened. The pseudo-code is seen in algorithm 3.

Algorithm 3 Random Edge P_5-free graph generation, which generates a P_5-free graph by a dynamic algorithm so attempts to both add an remove edges are made while preserving the P_5-free property

```
1: function randP5free(graphsize, iterations)
2:     G ← graphsize many nodes
3:     for i = 0; i < iterations; i ← i + 1 do
4:         j ← Random[0, n)
5:         k ← Random[0, n)
6:         if  j = k then
7:             continue
8:         end if
9:         if FindP5(G ± edge(j,k)) = false then
10:            G ← G ± edge(j, k)
11:        end if
12:    end for
13: end function
```

For readability the algorithm is compressed to one evaluation on line 10 where it originally tries to add an edge if there is none, and tries to remove it if it already exists. The randomness comes from selecting a pair of vertices to consider with a probability function.

However this algorithm does not yet have a proof of its ability to generate all possible graphs of size n, as this would require a proof of connectivity in the graph of P_5-free graphs with edges between graphs differing in 1 edge only. This connectivity was tested for graphs up to and including 9 vertices, and it did not break connectivity for those sizes.

Both adding and deleting vertices seem to be necessary to get all graphs as there are counter-examples showing that only adding or only deleting can not produce all P_5-free graphs from any starting points. For deleting, if you start with an arbitrary graph and at some point end up with a cycle on 5 nodes, no matter which edge you delete you will end up with a P_5, even though it's possible to get P_5-free graphs of size 5, for instance by deleting one edge from a P_5. This is illustrated in the figure below.

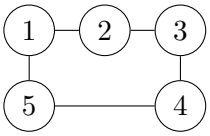


Figure 2.14: C_5, a graph on 5 nodes where no matter which edge is deleted, a P_5 is produced.

Lemma 2.15. *The graph C_5 is edge minimal with respect to remaining P_5-free.*

Proof. Assume the graph G = C_5 is not edge minimal. There must exist an edge $e \in E(G)$ such that $G-e$ is P_5-free. There are only 5 edges in G and we can see that the graph obtained from removing a single edge is isomorphic to all other graphs obtained by removing a single edge. Remove a single edge. The obtained graph is exactly a P_5, contradicting our original assumption, and thus proving G to be edge minimal. □

The counter-example for only adding edges consists of a large amount of P_4s that are connected in such a manner so that no matter which edge is added, a P_5 is produced. This graph is illustrated below.

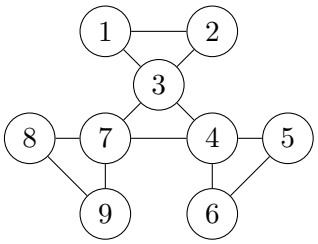


Figure 2.16: A graph on 9 nodes where no matter which edge is added, a P_5 is produced. This graph will be referred to as the QuadTri graph (quadruple triangle)

Lemma 2.17. *The graph QuadTri is edge maximal with respect to remaining P_5-free.*

Proof. It is easy to see that the graph is quite symmetrical, and the only graphs that can be obtained from adding a single edge will either be equivalent to the graph obtained by adding the edge (1,7) or the edge (1,8). By adding (1,7), the P_5s (2,1,7,4,5) and (2,1,7,4,6) are produced. By adding (1,8) the P_5s (2,1,8,7,4), (1,8,7,4,5), (1,8,7,4,6) and others are obtained. This proves that the QuadTri graph is edge maximal. □

A test was run on all graphs of size 9 to determine if each P_5 graph could be reached from any other P_5-free graph by adding or deleting edges. This test was performed by first generating all possible P_5-free graphs of size 9 up to isomorphism (using

an external library for isomorphism tests). The test then went on to create a new graph where all non-isomorphic P_5-free graphs of size 9 were represented as vertices in the graph. In this connectivity graph, an edge was added between two graphs A and B if it was possible to add an edge to A and get a graph isomorphic to B, or vice-versa. After doing this for all graphs, a connectivity test was performed. The test produced only a single component consisting of all graphs, thus all graphs of size 9 can be transformed into one another by adding or subtracting one edge at a time.

When we insert the result for Dynamic P_5-detection, we get an algorithm with runtime $O(n^3m)$, and get

Lemma 2.18. *It is possible to generate a random P_5-free graph of size n in time $O(n^3m)$ by adding or subtracting edges at random, as long as the graph remains P_5-free at every step.*

Proof. For each random pair of vertices u,v either add or subtract an edge based on what the current situation is. For each such action, perform the dynamic P_5-free detection algorithm from theorem 2.13, if a P_5 appears, reverse the action. This gives us $O(n^2)$ pairs and a $O(nm)$ test for each pair, yielding an overall $O(n^3m)$ algorithm. □

Starting with an independent set and attempting to connect the nodes with edges, it is highly likely that the result will be a relatively sparse graph. It would also be possible to start with a clique and perform the same algorithm, but this graph would likely be very dense. As highly dense graphs are unlikely to have the properties we are looking for (and a few sample runs were performed with dense graphs which supported this hypothesis), it was decided that only the graphs which started as independent sets would be created.

2.2.2 Random neighbourhood generation

Another method for generating P_5-free graphs was discovered while generating all possible P_5-free graphs up to a certain size. Starting with any P_5-free graph of size n, a P_5-free graph of size n+1 can be generated in the following manner: create a new vertex v, and pick a random selection of vertices from the original graph. After adding edges between v and the randomly selected vertices, test if the resulting graph is P_5-free, and if it is then return it, otherwise keep looping and trying other options. Save configurations already attempted to avoid wasting a large amount of time retesting configurations. The algorithm used is seen in Algorithm 4.

Algorithm 4 Random expansion P_5-free graph generation algorithm, which when given a seed graph G is able to expand the graph-size by a single node, and do so in a way which can generate absolutely any P_5-free graph of the relevant size. An additional function which expands a graph to any size desired by using the first function as a subroutine is also provided

```
1: function randExtendP5(seedGraph G)
2:     G ← G ∪ u
3:     upperBound ← 2^|G−1|
4:     Tried
5:     while |Tried| ≤ upperBound do
6:         neigh = rand[0, upperBound]
7:         if neigh ∈ Tried then
8:             continue
9:         end if
10:        Tried ← Tried ∪ neigh
11:        position i in the binary representation of neigh represents vertex i in G,
       add edge (u,i) to G if i = 1, don't add edge otherwise.
12:        if isP5free(G) then
13:            return G
14:        else
15:            remove the edges that were added.
16:        end if
17:    end while
18: end function
19: function randExtendP5(targetSize, seedGraph G)
20:    while |G| ≤ targetSize do
21:        G ← randExtendP5(G)
22:    end while
23:    return G
24: end function
```

Representing the nodes in the graph as a binary sequence, where the index of each position in the binary sequence corresponds to the node with id=index, gives a very useful and practical algorithm for looping through the possible neighbourhoods of the new node u. In practice, the speed of the algorithm is improved, as checking if a configuration has already been attempted becomes much quicker, simply by doing a lookup in a hash-table for the relevant binary sequence. This reveals if it already has been tried in constant time. This is done rather than storing a set of integers representing the vertices, and then for each new trial one would have to make a lookup which would take time $O(n \cdot k)$ where n is the number of potential neighbourhoods already tested and k is the size of the neighbourhood which needs to be tested. If one insisted on using this method (in case of special requirements of the

class representing nodes), it would be possible to improve it's speed significantly by using a prefix-tree(Trie) to find out if the neighbourhood had already been tested. Using a prefix-tree would yield an algorithm for lookup which runs in time $O(k)$ if it is implemented correctly. The problem of checking if a sequence has already been attempted could be eliminated completely. This could be done by taking the set of all possible neighbourhoods and giving it some random ordering, then the neighbourhoods could be tested one by one, and this avoids the chance of testing the same neighbourhood twice completely. However, problems arise with this idea as the number of possible neighbourhoods is $O(2^n)$ and even though the graphs in general are somewhat small, this would still be guaranteed to add a large amount of time to the algorithms pre-compute step. If we were willing to sacrifice randomness, it is also possible to create a function which generates every value in the given range by starting on a random number and then generating the remainder in some deterministic order.

A way to completely represent a graph of size ≤ 11 as a long was also used for some tests. The format was simply to save the lower triangle of the graphs adjacency matrix as a binary sequence, and for graphs smaller than 12 nodes, this fit exactly inside a long. This was later extended using something capable of storing larger values, known as BigInteger. This format lends itself naturally to many applications, and one which was especially convenient was to use this encoding as a way to store large sets of generated graphs. This allowed printing the generated graphs to file and reading them back in a very efficient manner. Operations involving comparisons of 2 graphs can easily get more speed in practice due to how the processor is normally able to manipulate 64 bits at a time within the registry, and therefore the run-time on many normally linear functions can be divided by 64. This was originally used in practice to test connectivity (for the set of all graphs, not just non-isomorphic) in the graph of graphs where two nodes shared an edge if the graphs they represented only differed by a single edge. Using the binary representation of the graphs and considering that the graphs were of the same size, one could simply do an XOR operation on the bit-representations and then count the number of bits set to 1 in the output bit-string. If exactly one bit was set to 1, then the two graphs differed by exactly 1 edge and an edge was added between the nodes representing the graphs.

It would also be possible to generate larger graphs by the generate all graphs method, but the available tool for testing graph isomorphism did not run at a speed that would allow it, even though there might be better algorithms there did not seem to be any other known isomorphism testers for Java. The isomorphism tester seemed to work by checking isomorphism against every graph in the set of non-isomorphic graphs in some low polynomial time. Even for size 9 graphs, this took several hours to generate the complete list, and for size 10 graphs it might take weeks. If it was possible to check graph-isomorphisms against a set in a time not bounded by the size of the set (which might be possible, given the right data-structures), each check would probably be bounded in time by checking for P_5s instead of the isomorphism

test.

Some interesting conclusions can be drawn by investigating the set of graphs generated by the algorithm that generates all non-isomorphic graphs for different sizes of i. The result of generating all non-isomorphic graphs of sizes i=5 through i=9 yielded the following result for the sizes of the different sets:

i	Size
5	33
6	136
7	685
8	4550
9	38123

Figure 2.19: A table showing the relationship between the size of graphs i and the number of non-isomorphic P_5-free graphs of that size

The general trend seems to be multiplying the previous size with approximately 10 to get the number of graphs in the next set. If we keep multiplying by 10 we get the following predictions for values 10 to 12:

i	Size
10	381230
11	3812300
12	38123000

Figure 2.20: A table showing the predicted relationship between the size of graphs i and the number of non-isomorphic P_5-free graphs of that size

Given that the P_5-free test takes $O(m^2) \leq O(n^4)$ we can predict that to generate all graphs of size 12, it would take $12^4 \cdot 38123000 = 790,518,528,000$ operations (assuming no overhead, which is probably far from the truth) which approaches what is feasible to do on a computer in a reasonable amount of time. So even if graph isomorphism was much faster, runtime of the P_5-free test, combined with the large size of results implies that it is not reasonable to generate much larger graphs than 9 either way.

An important consideration for the randomized extender algorithm is the uniformity of the results, that is, will it pick P_5-free graphs of size n from the pool of all P_5-free graphs of size n with an even distribution. By considering a single expansion iteration of the algorithm this problem is easier to study. Assume the algorithm is given the set of all possible P_5-free graphs of size k. We know that the set of P_5-free graphs of size k+1 it generates is equal to the set generated by starting with graphs of size k, such that the graphs are not restricted to being P_5-free. This is quite easy to see as a graph that contains a P_5 can not have it removed by adding a new node

to the graph, because inducing the graph by the nodes in the P_5 will give exactly the same edges independently of how the rest of the graph is modified. This shows that we can generate all possible P_5-free graphs of size k+1, so next we will consider the chances of getting a graph of size k+1 with the same probability as just selecting one at random from a precomputed set of all possible size k+1 P_5-free graphs. To do this, we will consider the set of labelled graphs on k+1 nodes. If the algorithm was truly random, our chance of getting any one of the labelled graphs on k+1 nodes would be $\frac{1}{X_{k+1}}$, but by considering a branching tree from all size k graphs into k+1 graphs, we realize that the different degrees of the nodes representing size k graphs will make a uniform distribution impossible. This tree is illustrated in figure 2.21.

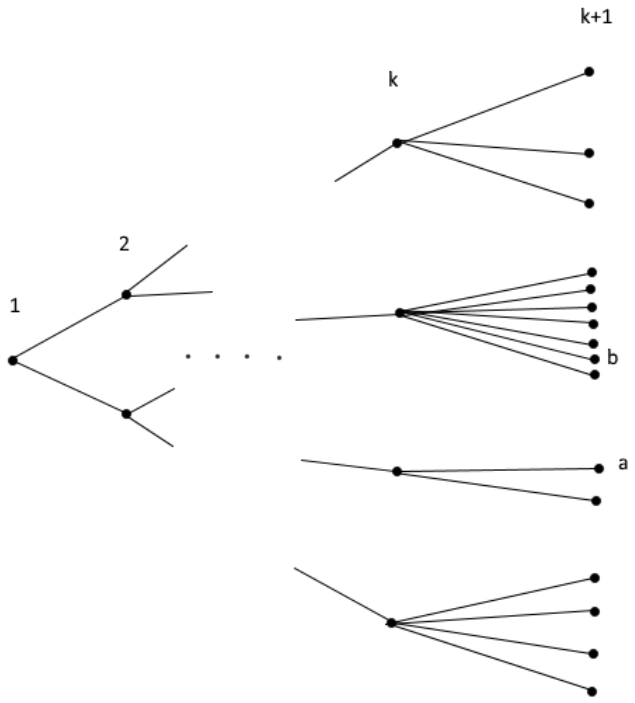


Figure 2.21: Illustration of the branching tree formed by generating larger graphs

We see that some graphs of size k can only branch into very few size k+1 graphs, while others can branch into many more. We are always guaranteed to be able to branch into 2 new graphs, one where the added node is universal (has an edge to every existing node) or where it is independent of the existing graph. We have a $\frac{1}{4} \cdot \frac{1}{2} = \frac{1}{8}$ probability of picking node a, while node b is only selected with a probability $\frac{1}{4} \cdot \frac{1}{7} = \frac{1}{28}$. It is hard to analyse exact probabilities, but we are at least able to say that the distribution will not be uniform.

The runtime for this algorithm is potentially exponential as it may only have the

two options of a universal or an independent node for expanding the graph while keeping it P_5-free, but as this is unlikely and we randomly try nodes, it seems to be able to expand the graph size relatively fast, at least up until n exceeds size 30. It would be possible to make this algorithm practically faster by finding a dynamic P_5 detection algorithm for adding a single node with runtime $O(nm)$. We almost have this, but we are missing an $O(nm)$ time algorithm to determine if a given node w is in a P_5 of the form uvwxy, so this is stated as an open problem.

Chapter 3

Minimal Triangulations

Triangulation is the process of making any given graph G into a chordal graph, which is defined as a graph in which every cycle of length ≥ 4 has a chord (an edge within the cycle). Turning any graph into a chordal graph by the triangulation method, involves adding a set of edges called a fill-set (will be referred to as **F**), filling the gaps in too long cycles with chords. The triangulation of G = (V(G),E(G)) yields a new graph H = (V(H),E(H)) where $V(G) = V(H)$ and $E(H) = E(G) \cup F$. A chordal graph H, $G \subseteq H$ is a called a triangulation of G, and H is a minimal triangulation if no proper sub-graph of H is a triangulation of G. A formal definition is given in definition 3.1.

Definition 3.1. *A minimal triangulation H of a graph G is a chordal graph with the same set of nodes as G, but if G is not chordal, H also contains a fill-set of edges which turn G into a chordal graph. For H to be minimal, no chordal graph H' exist such that $G \subseteq H' \subset H$.*

From a paper by Rose *et al*[16], we get an another useful property of a minimal triangulation. A triangulation H of G is said to be a *minimal triangulation* if there is no $e \in E(H)$ such that $H - e$ is still a chordal.

Chordal graphs always have a *perfect elimination ordering* (referred to as p.e.o.), which is a sequence of vertices of the graph, such that any vertex in the sequence forms a clique with its neighbours that come later in the sequence.

3.1 Maximum Cardinality Search

Calculating a minimal triangulation of a given graph G can be done in $O(nm)$ time by an algorithm known as MCS-M [1]. MCS-M is based on MCS (Maximum cardinality search) which is a commonly used $O(n + m)$ algorithm for computing a p.e.o. for a given chordal graph H. MCS computes a p.e.o. by first assigning all

vertices weight 0. At each step it picks the vertex v with highest weight (which has not been picked before), and if multiple nodes have the same weight our version of the algorithm will proceed to pick the one with lowest id, although in general the algorithm can be executed on an unlabelled graph where any node of maximum weight is selected. It then increases the weight of all unmarked (nodes which have not been previously picked) neighbours of v by 1, marks v and repeats the process. The p.e.o. is constructed in backwards order, so the first vertex selected by the algorithm becomes the last one in the p.e.o. and the last selected comes first. An ordering δ of the vertices in graph G is known as an MCS ordering, if there is a way of assigning id's to the vertices of G, such that the MCS algorithm executed on G outputs p.e.o. δ.

MCS-M is an algorithm based on MCS which runs on non-chordal graphs. When MCS-M is executed on graph G, in addition to computing a perfect elimination ordering of G, the MCS-M algorithm will also compute a minimal fill-set. The minimal fill-set can then be used to create a minimal triangulation. The p.e.o. computed by MCS-M will only be a perfect elimination ordering in the minimal triangulation created by MCS-M. It runs in the same way as MCS, but at the step where MCS increases the weight of N(v) by 1, MCS-M also considers all unmarked nodes u for which there exists a path $(v, x_1, ..., x_i, u)$ and the weight of u is greater than the weight of any other node x_i along the path. For each such u, the edge (u,v) is added to F and the weight of u is increased by 1. At completion of execution the algorithm has produced a fill-set F which is used to create a minimal triangulation, and the order in which the vertices were picked will form a p.e.o. of H, exactly like in MCS.

The algorithm for MCS-M in pseudo-code is given in Algorithm 5, and it uses a sub-function called Minimum bottleneck paths, described in the next section.

Algorithm 5 MCS-M, an algorithm which given as input a graph G computes a minimal fill-set which when added to G yields a minimal triangulation H of G. If G is already a chordal graph then the fill-set is empty, but a perfect elimination ordering is still computed.

```
1: function MCS-M(Graph G)
2:     initialize peo
3:     initialize weightof to 0 for all vertices
4:     initialize unNumbered with all vertices of G
5:     for i = |V(G)|; i ≥ 0; i ← i − 1 do
6:         initialize S
7:         Vertex k ← maximum weight vertex in unNumbered
8:         for all u : (N(k) ∩ unNumbered) do
9:             S ← S ∪ u
10:        end for
11:        for all u : Mbp(k) do
12:            S ← S ∪ u
13:        end for
14:        for all u : S do
15:            weightof(u) ← weightof(u) + 1
16:            unNumbered ← unNumbered \ u
17:        end for
18:        peo ← peo ∪ k
19:    end for
20: end function
```

3.1.1 Minimum bottleneck paths

An important part of the algorithm for MCS-M is to find paths between vertices such that at every step of the path from v to u, $(v, x_1, ..., x_i, u)$ the weight of u is always strictly greater than any x_i on the path. To find these paths an algorithm called Minimum bottleneck path is used, and will henceforth be referred to as m.b.p. The algorithm m.b.p. starts by selecting any vertex v, then it proceeds to explore the remainder of the graph in a breadth-first manner, and at each node it keeps two entries, the first entry keeps track of the minimum weight path which allows one to reach the node from v and the second entry is the maximum of the nodes own weight and the minimum weight path (the first entry). Upon completion, the algorithm proceeds to search through all the nodes to retrieve nodes whose own weight exceeds the weight of the minimum weight path which reaches them, and those nodes are added to the set S which is used in MCS-M. A pseudo-code version of the m.b.p. algorithm is seen in algorithm 6.

Algorithm 6 Minimum bottleneck paths

```
1: function MBP(List unNumbered, vertex v, vertex-weights w)
2:     initialize set S
3:     initialize Queue q
4:     initialize map weightTo
5:     initialize map weightIncluding
6:     for all Vertice u : N(v) do
7:         S.add(u)
8:     end for
9:     q.add(v, 0)                          ▷ place v with weight 0 into queue
10:    while !q.isEmpty() do
11:        (vertex k, weight kW) ← q.pop()
12:        if kW ≥ weightTo(k) OR k ∉ unNumbered then
13:            continue
14:        end if
15:        weightTo(k) ← kW
16:        weightIncluding(k) ← Max(kW, w(k))
17:        for all vertex u : N(k) do
18:            q.add(u, weightIncluding(k)
19:        end for
20:    end while
21:    for all vertex u : w(u) ≥ weightTo(u) do
22:        S ← S ∪ u
23:    end for
24:    return S
25: end function
```

Minimum bottleneck paths, also known as widest paths problem is a common problem, found in many real life scenarios. The most common version of the problem uses weighted edges, and finds the path between two vertices which maximises the size of the smallest edge. A common illustration of the problem is to find the set of pipes that allow the largest amount of water to go between two points at once. Our version of the problem is node-weighted instead of edge-weighted, and for our application (minimal triangulation) we start with a graph where all nodes have weight 0. The version given here will use time $O(n \log n + m)$ for finding all the values from a single source, but it is possible to get an $O(n + m)$ algorithm [18] as we are only interested in integer weighted and undirected graphs but as this will not have any impact on the overall runtime of the independent set algorithm, it is not implemented.

Necessary for our method of finding maximal cliques is the following result, found in theorem 3.2. The result is quite apparent from the algorithm descriptions, but we were unable to find a proof in the literature, so one is given here.

Theorem 3.2. *MCS-M executed on graph G outputs a minimal triangulation H and a p.e.o. δ_1, MCS executed on graph H outputs p.e.o. δ_2, $\delta_1 = \delta_2$.*

Proof. As both MCS-M and MCS at each stage adds the highest weight vertex to the p.e.o.(lowest id if several vertices have same weight), it is enough to show that at all nodes in V(G) have the same weight at the same step of execution of MCS-M and MCS.

Step 0: At start of execution all vertices have the same weight. Both MCS-M and MCS will choose node with id=0, then increase weight of the neighbourhood of the vertex by 1.

Step k: MCS-M and MCS will have selected the highest weight vertex v_k at this step. MCS-M will increase the weight of the neighbourhood of v_k, and $\forall u \in V(G), u \notin N(v_k)$ such that $mbp(v_k, u) \leq w(u)$, it will increase their weight by 1 and then add edge (v_k, u) to E(H). When MCS is run on H from v_k, the neighbourhood of v_k is the same as $N_G(v_k)$ + the fill edges that was added by MCS-M at this step, as MCS-M only adds edges between unlabelled vertices, no additional edges are added between v_k and any other node after it has been processed. MCS will therefore increase the weights of exactly the same set of vertices that MCS-M increased weight on. So at every step of execution MCS and MCS-M will pick the same node and give other nodes the same weight, thus they create the same p.e.o. □

3.2 Listing Maximal Cliques in a Minimal Triangulation

After producing a minimal triangulation of the given graph, we are interested in finding all the maximal cliques in this minimal triangulation. We use these to fill up the list of potential maximal cliques of the original graph (discussed in the "Potential Maximal Cliques" chapter). Luckily we can exploit some useful properties of the way we create the minimal triangulations of graphs to obtain a fast algorithm which lists all the maximal cliques.

Using Theorem 3.2 together with lemma 8 from a paper by Blair and Peyton, restated here as proposition 3.3, there is a simple method for producing all maximal cliques of a chordal graph in linear time.

Proposition 3.3. *[2] Let $v_1, v_2, ..., v_n$ be a perfect elimination ordering obtained by applying the maximum cardinality search algorithm to a connected chordal graph G. Then the list of maximal cliques in G contains precisely the following sets: $\{v_1\} \cup madj(v_1)$, together with any set $\{v_{i+1}\} \cup madj(v_{i+1}), 1 \leq i \leq n-1$, for which $|N(v_i) \cap L_{i+1}| \leq |N(v_{i+1}) \cap L_{i+2}|$*

In the above proposition the term **madj(v)** is the set of neighbours of v that are also placed later in the perfect elimination ordering. L_i are the vertices from i up

to n in the p.e.o. , more specifically $L_i = v_i, v_i + 1, ..., v_n$. The vertex set produced by $madj(v_i)$ is equal to the set produced by taking $N(v_i) \cap L_{i+1}$ as both produce vertices that are neighbours of v_i and which occur after v_i in the p.e.o. and both sets exclude v_i.

Given that the p.e.o. produced by MCS-M is equal to the one produced by MCS, to get all maximal cliques of G in triangulation H, one can simply pick the sets specified by Proposition 3.3. Start by selecting the first vertex in the p.e.o. together with its neighbours, as they by definition have to occur later in the ordering. Proceed to select $\{v_{i+1}\} \cup madj(v_{i+1})$ for any vertex v which does not produce a smaller madj(v) than the previous vertex in the ordering. This is to ensure that each clique is maximal. This procedure will yield an exhaustive list of maximal cliques.

Chapter 4

Potential Maximal Cliques

Referring back to proposition 2.3 we see that potential maximal cliques are a crucial part of the polynomial time algorithm for independent set in P_5-free graphs which was implemented in this thesis. The main result comes from restricting the number of potential maximal cliques in a P_5-free graphs to some size polynomial in n, and showing that this list of potential maximal cliques satisfy the requirements of the algorithm. It is therefore important to have efficient methods for working with potential maximal cliques, both for recognizing and generating them. In this section the methods used to verify if a set of vertices is a potential maximal clique in a specific graph are discussed.

A potential maximal clique (often shortened to p.m.c.) of graph G is a set S of vertices such that there exists a minimal triangulation H of G, in which S is a maximal clique.

From a paper by Bouchitté and Todinca[4] we get the proposition 4.1 regarding the structure of a p.m.c.

Proposition 4.1. *Let $\Omega \subseteq V$ be a set of vertices of the graph G. Then Ω is a potential maximal clique of G if and only if:*

1. $G \setminus \Omega$ has no full component associated to Ω, i.e. for every $S \in \Delta(\Omega)$ we have $S \subset \Omega$, and

2. the graph on the vertex set Ω obtained from $G[\Omega]$ by completing each $S \in \Delta(\Omega)$ into a clique, is a complete graph. In other words, every pair of non-adjacent vertices of Ω is in some $S \in \Delta(\Omega)$.

Note that $\Delta(\Omega)$ denotes the set of minimal separators of G contained in Ω. The set of minimal separators can be generated by the logic of the following lemma.

Lemma 4.2. *Let G be a graph, Ω a p.m.c. The minimal separators of Ω in G are:* $\Delta(\Omega) = \{N(C) : C \in ConnectedComponents(G \setminus \Omega)\}$

Proof. For each connected component C, the neighbourhood of C: N(C), is exactly the subset S of vertices in Ω that must be removed to separate C from the rest of the graph. If a single node is removed from S, C has an edge to this vertex and so C is not separated from the rest of the graph, thus S is minimal. □

A full component with regards to some vertex set Ω is a component for which the neighbourhood of the component is Ω, in other words:

Definition 4.3. *Vertex set C is a full component to Ω if $N(C) = \Omega$.*

These properties naturally lead to an algorithm which can test whether or not a vertex set V is a p.m.c. in a graph G.

4.1 Verify P.M.C.

From our definitions, we create an algorithm which receives a graph and a vertex set, and either outputs true if the vertex set is a p.m.c. in the graph, or false if it is not.

Proposition 4.4. *There exists an algorithm to determine if a vertex set $\Omega \subseteq V(G)$ is a potential maximal clique of a graph G, and it runs in time $O(nm)$*

The p.m.c. verification algorithm can be seen in Algorithm 7.

Algorithm 7 An algorithm to test whether or not vertex set V is a potential maximal clique in the given graph G.

```
function verifyPMC(Graph G, Vertex-set V)
    if !isMinimal(G, V) AND isCompleteNM(G, V) then
        return true;
    else
        return false;
    end if
end function
function isMinimal(Graph G, Vertex-set V)
    comps ← connected components of G \ V
    for all c : comps do
        neighbours ← N(c), c ∈ V
        if |neighbours| = |V| then
            return true
        end if
    end for
    return false
end function
function isCompleteNM(Graph G, Vertex-set V)
    initialize adjacency matrix Vmat of size |V|
    comps ← connected components of G \ V
    minSeps ← N(c) for each c : comps
    for all seps : minSeps do
        add edges between each pair in s
    end for
    if Vmat yields complete graph then
        return true
    else
        return false
    end if
end function
```

In complexity theory there is a plethora of different models which represents computers. Some of these models are closely related to real computers used today, while others are much more unlikely to ever exist. By considering an algorithm in the different models of computation, the analysis will yield a variation of run-times. This can help to focus on different parts of the algorithm that could be improved by future computers, or which are not as relevant to the runtime as others.

Turing machines, which are models of computation based on a simple tape which can be read and written to, together with some extra functionality, provide a framework to analyse algorithms. By expressing an algorithm in a format that is natural for a

human, it is possible to create a Turing machine from the given algorithm, and the two formats are said to be equivalent. Using this common framework, we reach the following lemma.

Lemma 4.5. *There exists an algorithm to determine if a vertex set $\Omega \subseteq V(G)$ is a potential maximal clique of a graph G, and it runs in time $n^2 \frac{c}{k}$ and in Big-O $O(n^2 c)$ but since c is bounded by n, the actual upper bound is $O(n^3)$.*

In Lemma 4.5, the runtime uses symbols c and k. c is the number of minimal separators in Ω while k is the size of the CPU registry. It is necessary that $\frac{c}{k} \geq 1$ as it does not make sense for an operation to take less than a single instruction cycle, so if the result is smaller than 1, it must be replaced by exactly 1. This version of the algorithm should run faster in practice than the $O(nm)$ algorithm, and so it is used instead. As c is bounded by n, where n is the number of nodes in the graph, and these nodes are represented by a binary string which has length c (one bit in the binary string of a node represents one minimal separator), if the graph is somewhat small the algorithm is very efficient. As most of the graphs dealt with in this thesis are smaller than 100, the $\frac{c}{k}$ component of the runtime can mostly be ignored, giving us an $O(n^2)$ time algorithm for graphs that fit within the registry.

The improved runtime is due to an improvement in the speed of the isComplete function. The changed method looks as follows.

Algorithm 8 A method to test whether completing minimal separators of V in G will make V a complete

```
function isCompleteCK(Graph G, Vertex-set V)
    for all i : V do
        s_i ← 0
    end for
    minSepList ← minimal separators of V in G
    for all minSep : minSepList do
        for all j : minSep do
            set position k in s_j to 0 if v_j is not in minimal separator k, 1 otherwise.
        end for
    end for
    for all i : V do
        for all j : V do
            if i ≠ j AND i and j not neighbours in G AND s_i, s_j share no minimal
separators then
                return false
            end if
        end for
    end for
end function
```

Algorithm 7 with isComplete method from Algorithm 8 runs in $O(n^2 \frac{c}{k})$ by the following reasoning: For every pair of vertices $O(n^2)$ perform a binary AND operation of the associated binary strings. Lengthwise the binary string is at most the number of minimal separators, which again is equal to the number of connected components in the graph $G \setminus V$. An AND operation of two numbers is normally a constant time operation. By using Java and representing the binary string with BigInteger, taking an AND of two BigIntegers simply splits the string into chunks of size long, which in modern processors are 64-bit, and then executes the AND operation on each pair. Thus the maximum number of operations necessary to take an AND of two BigIntegers on a 64-bit processor is $\frac{c}{64}$ and generally on a processor with size k registry we use $O(\frac{c}{k})$ operations. As the AND operation is done for each pair of vertices, the total runtime comes to $O(n^2 \frac{c}{k})$.

By considering the RAM model of computation, more specifically the Transdichotomous model proposed by Fredman and Willard[7], and even more specifically the Word RAM model, some interesting results can be obtained. The relevant parts of this ram model for the isComplete check is that the word size (register size) is related to the problem input, so the registers are of size $\geq log_2 n$. Common operations are also considered to be executed in constant time, which is relevant for the AND function used in the algorithm, in real computers this should also hold true for AND operations between two numbers that each fit inside a register.

Definition 4.6. *Transdichotomous RAM model is a model of computation which is a variant of the random access machine where the word size is assumed to match the problem size. Usually word size* $\geq log_2 n$.

Definition 4.7. *Word RAM model is an extension of the Transdichotomous RAM model which adds O(1) time operations that are common in "C-style" languages. Examples of common operations are* $+, -, \cdot, |, \&$.

When the isComplete algorithm from Algorithm 8 is reconsidered in the Word RAM model, some different results can be obtained.

Lemma 4.8. *There exists an algorithm to determine if a vertex set* $\Omega \subseteq V(G)$ *is a potential maximal clique of a graph G, and it runs in time* $n^2 \frac{c}{log_2 n}$ *(c is assumed to be larger than* $log_2 n$ *and if it's not,* $\frac{c}{log_2 n}$ *should be replaced by 1) and in Big-O notation* $O(\frac{n^3}{log_2 n})$.

Proof. Given a graph G and a vertex set V, to test if V is a p.m.c. in G do the following: List minimal separators of $G \setminus V$ in time $O(n + m)$. A binary string associating each vertex with the minimal separators they are found in is constructed in time $O(n^2)$, and has length at most n. For each pair of vertices (n^2) perform an AND operation of their binary strings. In the Word RAM model, each binary string will take at most $\frac{n}{log_2 n}$ registers, and the AND operation can be performed in constant time for each pair of registers. $\frac{n}{log_2 n} \geq 1$ as otherwise it would be possible

to perform a computer operation in less than 1 instruction cycle, which does not make sense on a regular computer (maybe quantum computers?). So for each pair of vertices, run the and operation, and this yields a total runtime of O($\frac{n^3}{log_2 n}$). □

We briefly mention here that in the analysis of the method to generate the list Π, the paper provides an algorithm with a total runtime of $O(|\Delta|n^6m)$, as first a list of size $O(|\Delta|n^4)$ is produced, and for each element on the list, an algorithm with runtime $O(n^2m)$ is performed. The $O(n^2m)$ algorithm gets its bound by performing the p.m.c. verification algorithm $O(n)$ times. If we analyse the algorithm within the bounds of the Word RAM model, together with our new version of the verify p.m.c. algorithm, we get an $O(\frac{n^4}{\log_2(n)})$ algorithm which we perform the same number of times as before, giving us a total of $O(\frac{|\Delta|n^8}{\log_2(n)})$. In practice this should be faster, as the Word RAM model holds true for any graph of size less than $O(2^{64})$ on a modern computer. And so we get a slight improvement in our overall runtime for generating Π.

Chapter 5

Algorithms for computing Independent set

The algorithm discussed in this chapter is the same algorithm created and explained in the paper by *Lokshtanov et al.*[12].

The main algorithm which this paper considers is an algorithm which receives a graph G, a list Π of potential maximal cliques of G, and through the use of dynamic programming is able to find the largest independent set in G. It is able to do so for both the weighted and unweighted versions of the problem. The weighted optimization version of the problem is defined as follows.

Definition 5.1 (Weighted independent set problem)**.** *Given a node-weighted graph G, pick the set S of vertices which maximizes the sum of weights of the vertices in S, with the restriction that no pair of vertices in S have an edge between them.*

The basis of the independent set algorithm is a list of potential maximal cliques of the graph G. We call this list Π, and it is a subset of all potential maximal cliques of G. Π is filled with potential maximal cliques which are entirely within the closed neighbourhood of independent vertices u and v. It also contains the potential maximal cliques which only contain vertices outside $N_G[u, v]$. The procedure to generate this list is described in the following section.

5.1 Generating Π

Π is made by taking the union of two smaller lists, known as Π_1 and Π_2, which both contain sets of p.m.c.'s. The final list that the paper is aiming for is therefore defined as:

Definition 5.2. $\Pi = \Pi_1 \cup \Pi_2$

5.1.1 creating Π_1

The first of the two lists that we are creating, Π_1, is a list of maximal cliques of some particular triangulations of the graph G. The definition is given below.

Definition 5.3. *For each pair of non-adjacent vertices (u,v) in G, let G_{uv} be defined as the graph obtained by turning $\delta_G(u)$ and $\delta_G(v)$ into cliques. Let H_{uv} be a triangulation of G_{uv} for which no edges are added to u or v (also known as u,v-good).*

$$\Pi_1 = \bigcup_{u,v \in G} MaximalCliques(H_{uv}[N_G[u,v]])$$

So the minimal triangulation H_{uv} can be generated by simply running MCS-M on G_{uv} and we can guarantee that H_{uv} is u,v-good by the simple property that no node can form a cycle of length ≥ 4 with only its immediate neighbourhood, so therefore any such cycle must go through neighbours which have outside neighbours, but in our case we already made these nodes into a clique, and so any cycle that goes through u or v and 2 of their neighbours, already has a chord. All maximal cliques of the triangulated graph are then generated by the method described earlier in this thesis. All maximal cliques are evaluated, and they are discarded if they contain vertices from outside $N_G[u,v]$. This leaves the final list Π_1.

Lemma 5.4. $|\Pi_1| \leq n^3$ *and* Π_1 *can be computed in time* $O(n^3 m)$

Proof. The size bound comes from the fact that there are n^2 possible choices for u and v, and for each chordal graph H_{uv}, the number of maximal cliques is bounded by n [5]. The runtime is derived from having the selection of n^2 possible pairs u,v, together with the runtime of $O(nm)$ to calculate the minimal triangulation, yielding an algorithm with runtime $O(n^3 m)$. □

The runtime is a slight improvement over the $O(n^6)$ algorithm given in the paper by Loksthanov *et al.* Considering that the triangulation has to stay u,v-good, they assumed that an algorithm with runtime $O(n^4)$ was necessary for computing such a minimal triangulation. As the graph to be triangulated is of the form G_{uv}, any minimal triangulation will necessarily be u,v-good and so the faster minimal triangulation algorithm with runtime $O(nm)$ can be used instead.

A triangulation H of G is said to be I-good if, given some independent set I in G, the triangulation does not add edges between vertices in I.

The result in lemma 5.5 was produced during the writing of this thesis and was used to achieve some speed-up for the algorithm which generates Π_1, compared to the runtime in the paper. However this does not affect the size of Π_1 or the overall runtime of the independent set algorithm.

Lemma 5.5. *Given a graph G_{uv} where u and v are independent vertices and $\delta_G(u), \delta_G(v)$ are cliques, any minimal triangulation H_{uv} of G_{uv} is (u,v)-good.*

Proof. Assume H_{uv} is a minimal triangulation of G_{uv} which is not (u,v)-good, so either an edge was added to u or to v. It is enough to look at one of the vertices and generalize the result. Assume that an edge (u,x) was added between u and some vertex x. As $\delta_G(u)$ is a clique and any cycle involving u must go through vertices in $\delta_G(u)$, for any such cycle there is already a chord between the vertices in the neighbourhood of u, and so removing the edge that was added to u lets the graph remain chordal. As $H_{uv} - (u, x)$ is still a chordal graph and a triangulation of G_{uv}, we contradict the original assumption that H_{uv} was a minimal triangulation. Thus showing that no minimal triangulations add an edge to u or v. □

The pseudo-code for this algorithm is given in algorithm 9.

Algorithm 9 A method which generates the list Π_1, which is the list of all potential maximal cliques of u,v-good triangulations of some graph G, such that the p.m.c.s are subsets of $N[u, v]$.

```
function GENERATEPI1(Graph G)
    Solution S
    for i = 0; i < n; i = i + 1 do
        for j = i + 1; j < n; j = j + 1 do
            if i = j OR j ∈ neigh(i) then
                continue
            end if
            H_ij ← MinimalTriangulationOf(G_ij)
            for c : maximalCliquesOf(H_ij) do
                if c ⊆ N_G[i, j] then
                    S ← S ∪ c
                end if
            end for
        end for
    end for
    return S
end function
```

Π_1 will generate all maximal cliques of I-good minimal triangulations which can be covered by the neighbourhood of 2 independent vertices u and v. As such, to get a complete list of maximal cliques that can be generated from I-good minimal triangulations, the next step is to generate all maximal cliques which are NOT covered by the neighbourhood of 2 independent vertices. This list will be called Π_2.

5.1.2 creating Δ_2

To generate Π_2, a precursor set of vertex sets called Δ_2 is needed. Δ_2 is defined so that for any maximal clique Ω of an I-good minimal triangulation H, such that Ω is not a subset of $N_G[u, v]$ for any choice of $u, v \in I$, it will satisfy $\Delta(\Omega) \subseteq \Delta_2$, where $\Delta(\Omega)$ is the list of minimal separators in Ω.

Δ_2 is defined by definition 5.6, from the paper by Lokshstanov *et al.*

Definition 5.6. Δ_2 *is a list of sets of vertices such that for each ordered triple (u,v,w) of vertices from G*

- *(u,v,w) are independent vertices in G*
- C_w *is the connected component of* $G \setminus N_G[u, v]$ *containing w*
- $\hat{C}_u$ *is the connected component of* $G \setminus N_G[C_w]$ *containing u*

we put $N_G(\hat{C}_u) \in \Delta_2$.

An illustration of how an object in Δ_2 looks is seen in figure 5.7.

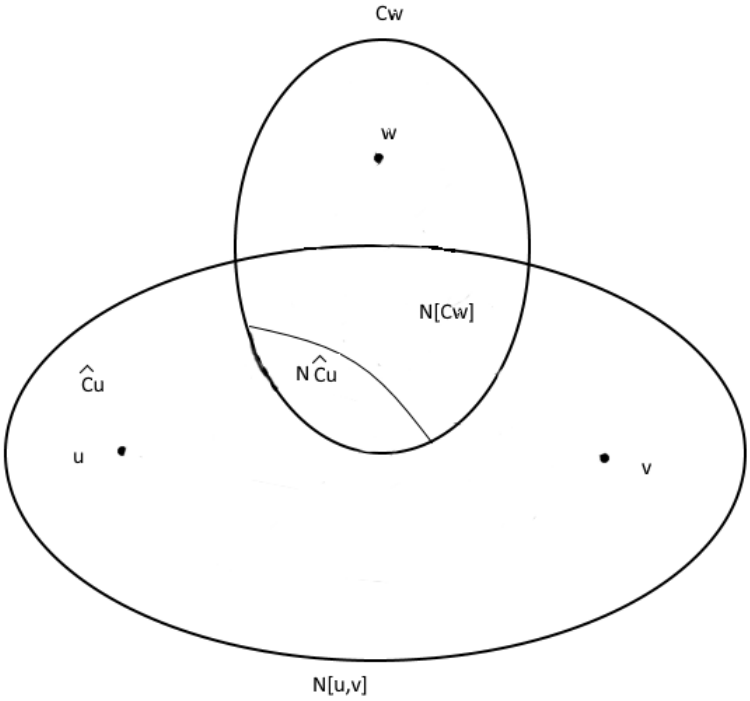


Figure 5.7: Graph split into components used in finding vertex set for Δ_2. In this example u and v are in the same component when the graph is divided up by $N_G[C_w]$. The output is $N_G(\hat{C}_u)$ which is a subset in both this example and in general of $N_G[C_w]$.

In the illustration figure 5.7, u and v are found in the same component so $\hat{C}_u$ is equal to $N_G[u, v] \setminus N_G(\hat{C}_u)$. It is also possible for them to be in separate components as seen in figure 5.8.

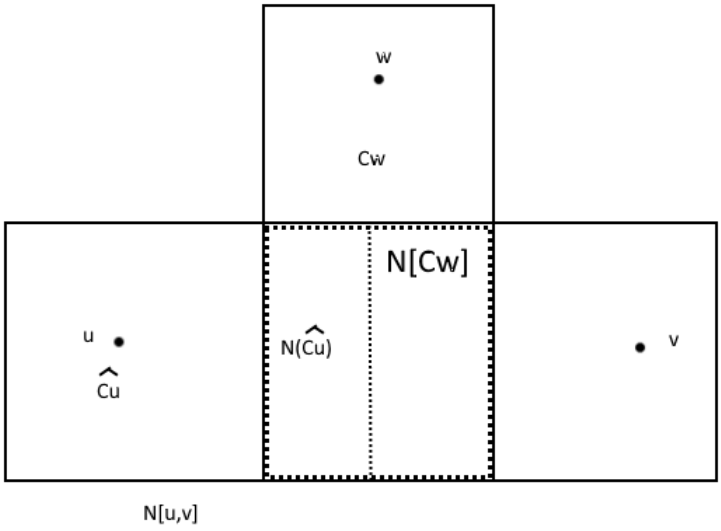


Figure 5.8: Graph split into components used in finding vertex set for Δ_2. In this example u and v are in different components when the graph is divided up by $N_G[C_w]$. The output is $N_G(\hat{C}_u)$.

Creating an object in Δ_2 consists of picking any two independent vertices u and v in the graph G, splitting up the graph into at most n components by removing $N_G[u, v]$ from G, then removing components one by one and inspecting the component containing u in the resulting graph. Finally the neighbourhood of the component containing u in the original graph is selected.

So the definition and the derived algorithm to calculate Δ_2 is very simple. The algorithm can be executed in time $O(n^4m)$, and the size of the output has the following property.

Lemma 5.9. $|\Delta_2| \leq n^3$.

Proof. Each set in Δ_2 is uniquely defined by three vertices u, v and w and so it can not possibly be larger than n^3. □

The size of Δ_2 is carried on into the size of the list Π_2 and so it is a good place to

focus efforts for a possibility of reducing the final runtime. Results regarding this will be discussed in the results chapter.

5.1.3 creating Π_2

Finally we need to consider Π_2 which is the theoretically largest list of p.m.c.s and also the most complicated to generate. Π_2 is defined so that it contains all p.m.c.s in G that are not a subset of $N_G[u, v]$ where u and v are independent vertices (it can also contain p.m.c.s within some $N_G[u, v]$, but for any p.m.c. which do not satisfy this property, the p.m.c. will be found in Π_2). As these were already generated in Π_1, we know that Π will cover all p.m.c.s from I-good triangulations of G.

From the paper by Lokshtanov *et al.* we obtain lemma 5.10.

Lemma 5.10. *There is an algorithm that given as input a P_5-free graph G and a family Δ_2 of vertex sets in G, outputs in time $O(|\Delta_2|n^6m)$ the family $\{\Omega \in \Omega(G) : \Delta(\Omega) \subseteq \Delta\}$. The size of the family output by the algorithm is no more than $O(2|\Delta_2|n^4)$.*

In the lemma $\{\Omega \in \Omega(G) : \Delta(\Omega) \subseteq \Delta_2\}$ by $\Omega(G)$ we mean the list of p.m.c.s of I-good triangulations of G and by $\Delta(\Omega)$ we mean the list of minimal separators within the p.m.c. Ω. In other words, we are looking for p.m.c.s where every minimal separator in the p.m.c. is found on the list Δ_2. For the proof this this lemma you are referred to the paper [12].

To generate the list Π_2 we need to generate some some additional lists Π_A^i and Π_B^i. We can then construct each p.m.c. $\Omega \in \Pi_2$ by using $\Omega_i \in \Pi_A^i \cup \Pi_B^i$ as a base and expanding it through an iterative process. Finally we test if $\Delta(\Omega) \subseteq \Delta_2$ and if this is true we can add Ω to the list Π_2.

We will now consider each of the steps necessary to generate Π_2 individually.

As a pre-computation step, a sub-graph of G is produced for each i from 0 to n. The sub graph G_i is produced by the following definition.

Definition 5.11. *G_i is the sub-graph of G which is induced by the vertices with ids from 0 up to and including n. In other words $G_i = G[0, ..., i]$.*

As this is only done once and is relatively quick, the naive method is used and we do not need to consider the runtime as long as it is sensible.

Our first step is to compute a list of lists which we call Δ^i, that is used in the generation of the other lists we produce as part of making Π_2. For each $S \in \Delta_2$ and for each $i \leq n$ compute all components of $G_i \setminus S$, then put the neighbourhood of each of those components into Δ^i. This will give the minimal separators of S in the graph G_i that we will later use to generate Π_2. The formal definition we use is found in definition 5.12;

Definition 5.12. *$\forall i \leq n \forall S \in \Delta_2$ let $CC = C_1, ..., C_j$ be all the components of $G_i \setminus S$, then $\{X \in \Delta^i : X = N_{G_i}(C_k \in CC)\}$.*

Now that we have done all the necessary set-up, we can move on to produce lists Π^i_A and Π^i_B. We start by defining Π^i_A.

Definition 5.13. $\forall i \leq n : \Pi^i_A = \{(S_i \cup \{v\}) \in \Pi^i : S_i \in \Delta^i, v \in V_i\}$

In the above definition V_i is the set of vertices with indices from 0 up to and including i, in other words $v \in V_i \Leftrightarrow v \in [0, ..., i]$. So to generate Π_A we must do it for one value of $i, 0 \leq i \leq n$ at a time. Start with the lowest value of i, proceed through the list Δ^i in any order and at each step pick some set $S \in \Delta^i$. For each $v \in V_i, v \notin S$ proceed create a new set for each combination of S and v and add this new set $S \cup v$ to Π^i_A. Once this has been done exhaustively, the list Π_A will contain a sub-list for each value of i which will again contain all of the sets generated which are associated with that value of i, and Π_A is then completed.

We move on to the next list which we name Π_B and which consists of vertex sets which attempts to expand the vertex sets in Δ^i in a different manner than Π_A. We define Π_B as follows:

Definition 5.14. *$\forall i \leq n$ loop through $S_i \in \Delta^i, v \in S_i, C_i \in CC(G_i \setminus S_i)$ to get triplets of the form (S_i, v, C_i). Then $S_i \cup (N_{G_i}(v) \cap C_i) \in \Pi^i_B$.*

$CC(G \setminus S)$ is shorthand for the list of connected components obtained by removing vertex set S from graph G. Π_B consists of sets S of Δ^i which are expanded by splitting the graph into components $C_0, ..., C_i$ by taking $G \setminus S$, then proceeds to loop through vertices in S and testing the overlap of the neighbourhood of each vertex with each of the components, ending up with $N_G(v) \cap C_i$ where C_i is some component obtained as mentioned previously. This is then added to S, and the result is added to the Π_B. Π_B contains a sub-list for each i, which is related to the Δ^i from which S was obtained. Pseudo-code which explains how to generate Π_B is given in algorithm 10.

Algorithm 10 A method for generating Π_B, which is a list used as part of generating Π_2

```
function generatePiB(Graph G, List Delta)
    Solution Π_B
    for i = 0; i < n; i = i + 1 do
        for all S_i ∈ Δ^i do
            for all C_i ∈ CC(S_i) do
                for all v : S_i do
                    Π^i_B ← Π^i_B ∪ (S_i ∪ (N_{G_i}(v) ∩ C_i))
                end for
            end for
        end for
    end for
    return Π_B
end function
```

For some value i, we call the vertex sets in Π_A^i and Π_B^i for seed potential maximal cliques of G_i as these are what we will use to produce proper potential maximal cliques of G. Let the set of seed potential maximal cliques of G_i be found in the list Ω_i, and define $\Omega_i = \Pi_A^i \cup \Pi_B^i$. We can now proceed to use the seed potential maximal cliques to create the potential maximal cliques using lemma 5.15 from a paper by Bouchitté and Todinca.

Lemma 5.15 ([3])**.** *$\forall i \leq n, \Omega_i \in G_i$ there is exactly one p.m.c. of G which satisfies $\Omega_i = \Omega \cap V_i$. Given Ω_i, Ω can be computed in time $O(n^2m)$.*

The algorithm used to reconstruct the potential maximal clique from a seed potential maximal clique is seen in algorithm 11.

Algorithm 11 Function which constructs a valid p.m.c. in G from a seed p.m.c., which is a vertex set that is a valid p.m.c. in some sub-graph of G

```
function reconstructPMC(set seedPMC, int start)
    Solution S ← seedPMC
    for i = start + 1; i < n; i++ do
        if !isPMC(G_i, S) then
            S ← S ∪ v_i
        end if
    end for
    return S
end function
```

The algorithm is given a set of vertices seedPMC which is a p.m.c. in G_{start}. At each step of execution we may or may not alter seedPMC. If we add a new vertex v_{i+1} to G_i there are two possible outcomes, seedPMC will remain a p.m.c. in $G_i \cup v_{i+1}$

or it does not. If it is still a p.m.c. then we can keep expanding the graph while keeping seedPMC the same, while if it is no longer a p.m.c. then by the definition of a p.m.c. the new node must be a full component to seedPMC, and so our only option is to add the new node to seedPMC to obtain a p.m.c. in $G_i \cup v_{i+1}$. This process is repeated until $i = (n-1)$ (for a zero-indexed graph) and we have the full graph. Once we have the full graph, seedPMC after the modification steps described will be a p.m.c. in G.

We can now move on to the final step, which is to remove the potential maximal cliques generated so far which do not satisfy the requirement of lemma 5.10. This implies that for each p.m.c. $\Omega \in \Pi_2$ it is necessary that $\Delta(\Omega) \subseteq \Delta_2$. In other words: each minimal separator of a p.m.c. Ω such that $\Omega \in \Pi_2$ must be found in the list Δ_2. The number of p.m.c.s which needs to go through the filtering process will be $O(|\Delta_2|n^4)$ as for each value of i, $|\Pi_A^i \cup \Pi_B^i| \leq |\Delta_2|n^3$ and there are n possible values of i. For each p.m.c. Ω of the $|\Delta_2|n^4$ p.m.c.s we need to consider n possible components from $G \setminus \Omega$, this means that we need to perform the test $s \in \Delta_2$ for each set $s \in \Delta(\Omega)$ and in total there are $|\Delta_2|n^5$ such sets. The naive algorithm to perform this check would compare set s with each set in Δ_2 and only accept if it finds a $q \in \Delta_2$ such that $s = q$ (irrespective of the order of the elements in the set). We would end up with an algorithm with runtime $|\Delta_2|^2 n^5$ if we did it this way, and this has the potential to increase the overall runtime of the independent set algorithm. Therefore we need a better algorithm, and the solution is to use a Trie.

Definition 5.16. *A* Trie*, also known as a prefix-tree, is a data-structure with the shape of a tree, which is used for fast lookups of membership. The root node r branches into one node for each possible value which an element in the Trie can have in the relevant position, and the branching continues in the same manner all the way to the leafs. By following some path from the root to a leaf, the sequence obtained is a member of the set of elements used to construct the Trie.*

Depending on how the Trie is constructed we get different amounts of speed-up over the naive version. The Trie has to consist of every element in Δ_2 and if one sequence in Δ_2 is a subset of another sequence, the Trie must indicate where each sequence ends, as we need to test for exact equality. So essentially what we need is a data-structure with ability to add a new sequence to the tree, stop-mark where the sequence ends if the last element in the sequence does not end up as a leaf in the tree, we must also be able to test whether the Trie contains a given sequence, where the sequence must end on either a stop-mark or on a leaf.

To actually construct a Trie we need two classes, a Trie class and a Trienode class. A Trienode will contain the symbol of the node, together with an array of Trienodes of the same size as the number of symbols possible. Each entry in the Trienode array will point to null until a child is added to the Trienode. The Trie class will contain one root node which will point to its children, and thus allow traversing of

the tree. Add sequence and lookup sequence will also be performed from the root node by methods in the Trie class. For the Trie constructed from Δ_2, the symbols are the set of values which members of Δ_2 can contain, and this is the same as the integers from 0 to n as Δ_2 contains sets of vertices of the graph of size n.

Before we add a set of Δ_2 to the Trie, it is necessary to sort the set so we get an increasing sequence. As Δ_2 only contains integers in the range from 0 to n, we represent Δ_2 as a 2-dimensional matrix of size $|\Delta_2|$ rows by n columns. We need to sort the set found in each row, but do not need to consider the rows in comparison to each other. This is done with algorithm 12.

Algorithm 12 Function which sorts each row in a 2D matrix of m entries, where each entry has value at most n

```
function SORT2D(List<List<Integer» matrix)
    List<List<Integer» tmpS
    for i = 0; i < m; i = i + 1 do
        for j = 0; j < matrix[m].size(); j = j + 1 do
            tmpS[matrix[i][j]] ← tmpS[matrix[i][j]] ∪ i
        end for
    end for
    List<List<Integer» S
    for i = 0; i < n; i = i + 1 do
        for j = 0; j < tmpS[i].size(); j = j + 1 do
            S[tmpS[i][j]] ← S[matrix[i][j]] ∪ i
        end for
    end for
    return S
end function
```

The sorting algorithm creates a new 2D matrix tmpS, which is actually a list of lists, so it is scalable. We proceed to loop through the input matrix, one row at a time. For each element x in row i of the original matrix, we remember that x occurred in row i by adding i to row x of tmpS. So tmpS ends up with n rows, and in each row x the rows of the original matrix containing x are listed. Finally we create matrix S which is a recreation of the original matrix but in a sorted form. This is done by starting at row 0 of tmpS, then for each row in the original matrix which contained 0 we add 0 as the first element in the corresponding row of the sorted matrix S. This is repeated until the whole array is sorted, and this takes time $O(mn)$ with m being the number of rows in the matrix and n is the largest element in the input matrix. So for our usage this leads to a runtime of $|\Delta_2|n$.

Finally we add each sorted sequence of Δ_2 to the Trie, and we end up with a tree in which we can make fast lookups. To test if a sequence $x_0, x_1, ..., x_n$ is contained in the Trie, we ask the root node of the Trie if it has a child x_0, as the list of children

is of size n and x_0 is an integer, we can make an instant lookup. In the code we would perform the following routine if(root.children[x_0] $\neq$ null), and if this is true we retrieve the child and continue this procedure for each element in the sequence. If we reach a leaf before the whole sequence has been found then the sequence is not a member of Δ_2, and if the sequence ended in a non-leaf node which does not have a stop-mark then it is not in Δ_2, or if the the query for a child yields null then the sequence is not an element of Δ_2. As lookups are done in constant time, and the maximum length of a sequence is n (as Δ_2 contains subsets of the vertex set of the graph) we can see that each lookup is done in time $O(n)$.

It might be possible to do faster lookups (constant time) by creating some hash-function which takes a sorted set of integers and produces some unique integer for each set, but this hash-table would require n^n space and as n grows larger this is quite infeasible.

After filtering the p.m.c.s which do not contain minimal separators in Δ_2, we end up with the list we wanted, Π_2. The last list, Π is made by combining Π_1 and Π_2, so $\Pi = \Pi_1 \cup \Pi_2$.

5.2 Polynomial time algorithm for Independent Set on P_5-free graphs

Proposition 2.3 mentions the algorithm which forms the basis of this thesis. In the previous chapters, all the components that are necessary for making an algorithm to generate a list Π of potential maximal cliques have been explained and analysed to a necessary depth.

We can now proceed to discuss the actual algorithm which is used in producing a maximum independent set from the given graph together with the list Π.

From the input graph G, we now have a list Π of p.m.c.s of G. From lemma 16 in the paper[12], we know that this list has the following property. For every inclusion maximal independent set I it is possible to build a tree decomposition of G where each bag is an element in Π and every bag of the tree decomposition contains at most one element from I. Our aim now becomes making an algorithm which constructs such a tree decomposition of G for a maximum size independent set I. This will allow us to find the maximum size independent set of G.

We have two possible algorithms which give the same solution. One slow algorithm with a total runtime of $O(|\Pi|^2 n^4 m)$, and a fast algorithm which which is the one described in 2.3, giving us runtime $O(|\Pi| n^5 m)$. Both papers are described in the paper by Fomin and Villanger, but only the slow one has been proved to work for P_5-free graphs in the intended way, although a proof for the fast one follows very closely from a proof to the slow algorithm, and we did not find any discrepancy in

the the size of the maximum independent set the two algorithms output. We will now give a high level sketch of the two algorithms.

Slow algorithm

To solve the independent set problem given the list Π of p.m.c.s, we pick a p.m.c. and find the largest independent set in a tree-decomposition in which the given p.m.c. is a bag. To do this we need to create a lookup table M, together with an algorithm used to create the lookup table.

Our function M needs to be a recursion which accepts three parameters, p.m.c. $\Omega \in \Pi$, a vertex $v \in \Omega \cup e$ where e signifies the empty element (in practice, this can be denoted by giving v a special value like -1), and the component $C \in CC(G \setminus \Omega)$ which is a component obtained by removing Ω from the graph G. The function M will return the size of the maximum independent set I in $\Omega \cup C$ where Ω contains v, and so $\Omega \cap I = v$. The solution has a few more requirements related to the tree-decomposition, but we refer the reader to the paper [12] for details. The unweighted version of the algorithm which we end up with is seen in algorithm 13.

In the following algorithm, let $|x| = 0$ if $x = e$, 1 otherwise.

Algorithm 13 Function which returns the largest independent set by selecting vertex sets which together form a tree-decomposition of the graph G, where each bag is a p.m.c. from the list Π, and I contains at most one vertex from each bag

```
function M(Ω, x, C)
    max
    if M[Ω,x,C] ≠ null then
        return M[Ω,x,C]
    end if
    for all Ω' ∈ Π : Ω' ⊆ Ω ∪ C, N(C) ⊆ Ω', Ω' ∩ C ≠ 0 do
        for all x' ∈ Ω' : Ω' ∩ x = Ω ∩ x', x' ∉ N_G(x) do
            sum ← maximum(|x'| − |x|, 0)
            for all C' ∈ CC(G \ Ω' : C' ⊆ C do
                sum ← sum + (M(Ω', x', C') − |x'|)
            end for
            if sum > max then
                max ← sum
            end if
        end for
    end for
    max ← max + |x|
end function
```

Dynamic programming requires a table structure for lookup, so we use memoization. The table M can have its values accessed by specifying x, the set Ω and the set C. For each set, we need to be able to do efficient lookup by e.g. having a hash-table.

As each bag $\Omega \in \Pi$ of the tree decomposition is a separator of the graph, it is sufficient to maximize across each section of the tree at a time and then sum together the value obtained. To obtain the maximum independent set in G, we run algorithm 13 for each $\Omega \in \Pi$ and $x \in \Omega \cup e$, and maximize across the runs. The result will give us the size of the largest independent set in G, and to obtain the actual independent set some standard back-tracking is used.

An illustration of the algorithm can be seen in figure 5.17.

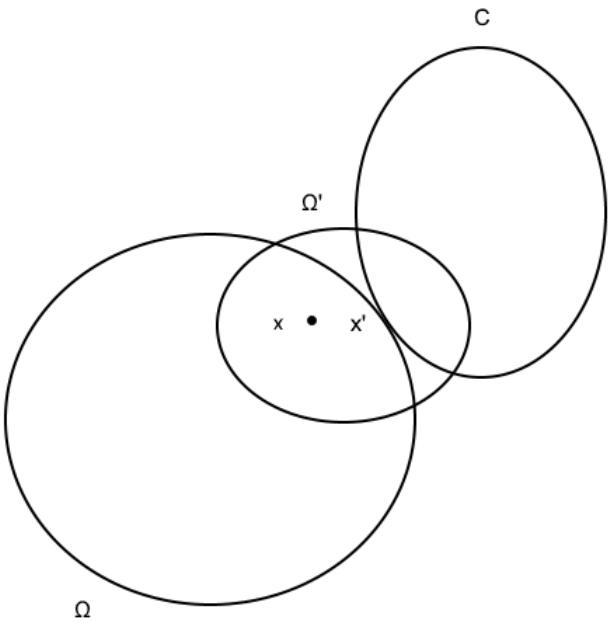


Figure 5.17: The algorithm iterates over potential maximal cliques, and potential subset x of I. In the original graph, two maximal cliques can overlap in either 0 or 1 members of the maximum Independent set I. In this example, the two potential maximal cliques Ω and Ω' overlap in the vertex x.

As each potential maximal clique Ω is a bag in the tree-decomposition of G, by definition Ω will separate the graph. We therefore look at its components C and

try to chain together p.m.c.s, with either 1 or 0 vertices x in each p.m.c. which will form the basis of I. Eventually we will have explored every possibility for selecting p.m.c.s from Π, and end up with the optimal result.

For more details see the paper [12].

Fast algorithm

We also implement a faster algorithm, which was briefly mentioned in the paper. This algorithm has a theoretical runtime of $O(|\Pi|n^5m)$, which is a big improvement over the slow algorithm. The key to this algorithm is to consider the minimal separators Δ within each Ω.

For each $\Omega \in \Pi$, compute the list of minimal separators in Ω denoted as $\Delta(\Omega)$, and take the union over all Ω to form the list S. Then create a graph R, where each node represents a vertex set. Each node will be equal to either Ω or a set $s \in S$. Add an edge between vertex a and b if $a \in \Pi, b \in S, b \in \Delta(a)$. We now need two DP tables, T1 and T2.

The method T1 takes the same parameters as M: vertex set $\Omega \in \Pi$, node $x \in \Omega \cup e$ and set $C \in CC(G \backslash \Omega)$. It proceeds to calculate a minimal separator Q in Ω which is the neighbourhood of C. Next it calls T2 with parameters Q, C, x, where everything but Q retains the value it had at time of input. The method T2 retrieves the node q associated with Q in the graph R (which connects minimal separators and p.m.c.s). Each neighbour of q which is also a subset of $Q \cup C$ are added to a new list QuC. Each element in QuC is a p.m.c., and so we look at them one at a time, calculate the components of the graph when the p.m.c. is removed, and call T1 with the triplet of values, where the first value is an element of QuC, the second value is one of the components, and the third value is a vertex in the p.m.c. The sum functions are the same as in algorithm 13.

To maximize across the graph, pre-compute the list S, then maximize over all values of $\Omega \in \Pi$ together with the potential values of x and C, as previously described.

For more details see the paper of Fomin and Villanger [6].

Chapter 6

Results

In this chapter we aim to give a thorough overview of the experimental results, together with any theoretical conclusions that we were able to draw from the experiments, or that were found otherwise and which are relevant to the runtime of the algorithm. We use the symbols Δ_2 and Δ interchangeably in this section, and both refer to the set Δ_2, described earlier.

To ensure correctness of the independent set algorithms result, the size of output for each random test-graph was compared with the output of another Independent set algorithm which was implemented by another person as part of the Grapher package. As there were no discrepancies in results, it is reasonably assumed that the algorithm is correct.

The run-times calculated are the sum of times for the actual minimum independent set algorithm to perform its task, summed with the time to pre-compute the list Π, according to the specifications discussed in this thesis. We use two different independent-set algorithms, the "slow" one with time bound $O(|\Pi|^2 n^4 m) \leq O(n^{18} m)$ and a "fast" algorithm with time bound $O(|\Pi| n^5 m) \leq O(n^{12} m)$.

For each data-set we consider the average data trend, and the maximum. The results may be skewed based on the methods used to generate the graphs, but by having different methods and comparing them, it is possible to have some overview of how far from average the results are. If there is a family of graphs which pushes the sizes of the objects close to the theoretical limit, then their random appearance may also skew the maximum results from what another more average, random sample would contain.

Three sets of graphs were generated, the first set is a set of random uniformly distributed graphs of size 10 over the given edge probability (P_5-free G(n,p) graphs), with a sample of 1000 of each graph and each of the probabilities 0.2 - 0.9 in 0.1 increments. The second set consists of graphs generated by the random edge method, but one of the edges were added as a universal vertex in the start, as otherwise it

would be nearly impossible to get a connected graph of a reasonable size. Graphs in the third set are random neighbourhood generated graphs. The random neighbourhood graphs of graph-size 9 include all non-isomorphic, connected graphs of size 9, while the remainder are simply randomly generated graphs, potentially isomorphic to other graphs in the set of graphs with the given size (as the sample size is small compared to the total number of graphs, it is not very likely that many of the graphs are isomorphic).

6.1 Effects of edge density

Based on the method of generation and the size of the graph, the density of edges in the graph will change. To get some idea of how large the effects of edge density on the size of the desired objects are, a small experiment was performed. For small graphs (size ≤ 10), it is possible to generate completely random graphs and filter out the P_5-free graphs in a reasonable amount of time. Equipped with this knowledge, a test-set was generated by creating a completely random graph with a given edge probability, and accepting it into the test-set if the graph was a single component and P_5-free. A collection of graphs were generated for each probability increment of 0.1, from 0.2 to 0.9. The plot of results of the algorithm executed on these graphs can be seen in figure 6.1.

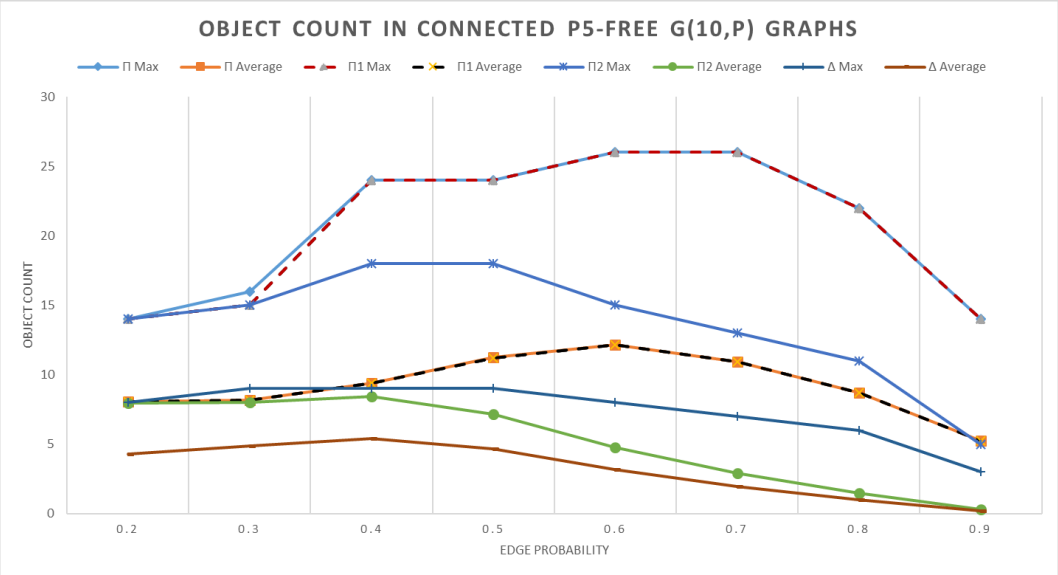


Figure 6.1: A plot displaying the relationship between the edge probability of the random graphs and the size of the objects generated. For each edge probability, we provide the average size of the objects, and the maximum object size across all graphs of the given edge probability. The striped plots overlap another plot, so to determine which plot it is, look in the legend for which dataset a striped plot is used.

The figure was generated from a sample size of 1000 randomly generated graphs of each probability value. We see that Δ_2 and Π_2 (which is highly dependent on the size of Δ_2) decreases quickly as the density of the graph grows larger, this is likely due to that fewer unique components of $G \setminus N[u, v]$ can be generated in a dense graph as the neighbourhood grows large. Average size of Π follows Π_1 very closely, and even though Π_2 which is strongly dependent on Δ_2 starts to decrease at approximately 0.5, Π keeps growing. As Π_1 is the number of p.m.c.s which can be found in the neighbourhood of two vertices, this number naturally increases as the average neighbourhood size grows larger.

Π_2 is much smaller than its theoretical bound in this case, and we will attribute this to the small size of the graphs. As Π_2 is dependent on Δ_2, and Δ_2 requires triplets of independent nodes, it is hard to create a P_5-free graph of a small size where there are many such triplets, and the basic cases where you can find many triplets (like star graphs) will generally only be able to create a few unique components based on the specification of Δ_2. Due to the time taken for generating these completely random graphs, it was not possible (in a reasonable amount of time) to make this experiment for graphs larger than size 10, and so we consider these results to be more useful as an indicator of Π and Π_1 size, than Π_2 and Δ.

We also studied the effect of the edge density on the run-times of the maximum

independent set algorithms. The results can be seen in figure 6.2.

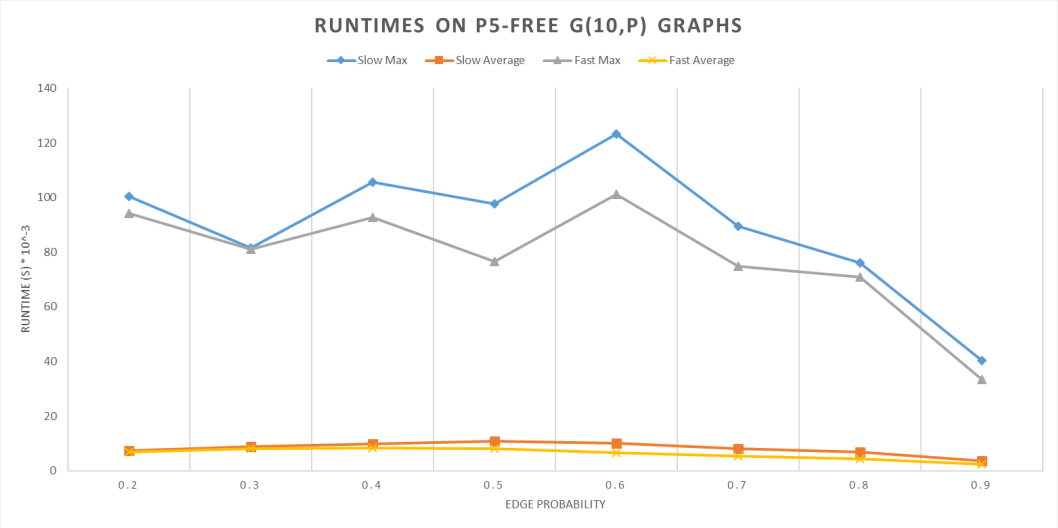


Figure 6.2: A graph displaying the relationship between the edge probability of the random graphs and the runtimes of the independent set algorithm. We plot the result of executing the two algorithms described earlier, the "slow"5.2 algorithm and the "fast"5.2 algorithm. For each of the algorithms we give the average runtime at each edge probability, and the maximum runtime of any graph of the given edge probability.

The run-times are the total run-times including both generating the list Π and performing the maximum independent set algorithm on the resulting list and graph. We can see that the slow algorithm is more dependent on the size of Π than the fast algorithm, as can be expected from their theoretical run-times. The general size trend seems to follow the same trend as the size of Π for both the slow and fast algorithm, but as the time used for generating Π increases with the number of size 2 or more independent sets, we see that the less dense graphs require more time. The runtimes are given in $seconds \cdot 10^{-3}$ (milliseconds) for easier readability.

Using these results, we may be able to compare our findings for the other graphs with these benchmarks to have some insight into which edge density the graphs skew towards.

6.2 Size of Π and Δ_2-lists

In this section we will present results regarding the size of list Π produced by the algorithm previously described. We will discuss the results and attempt to gain

some insight into limitations of the algorithm, together with some properties which can improve the algorithm.

In figure 6.3 we see a list of data produced by the algorithm on sample inputs. The sample sizes are large enough and likely random enough to give a good impression of the output on an average P_5-free graph of the given size.

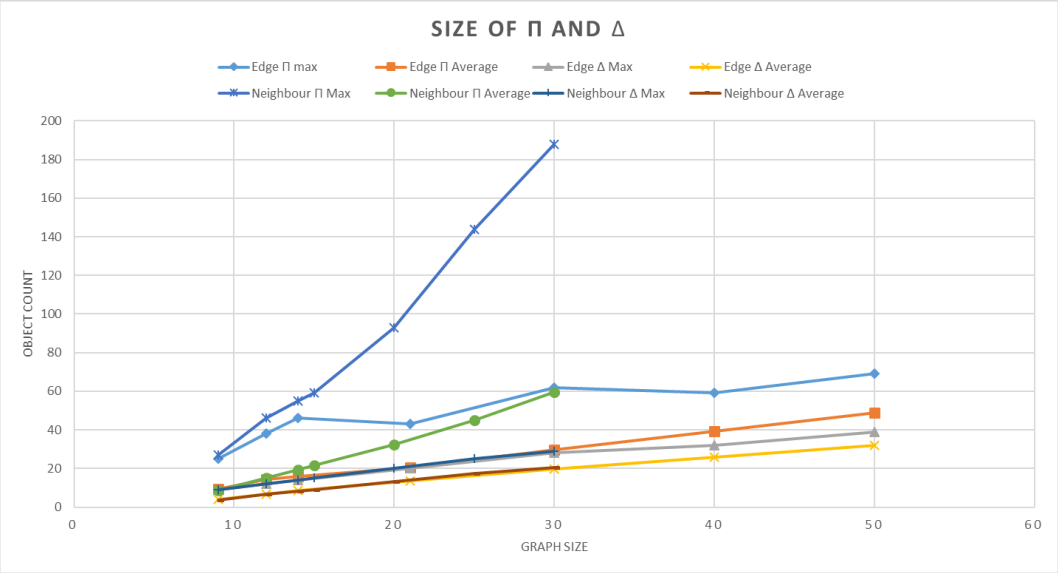


Figure 6.3: A plot displaying the relationship between the graph size and the size of the lists Π and Δ. We have two different methods of generating graphs, the "Edge" method (random edge generation) and the "Neighbour" method (random neighbourhood generation). For each method, we plot the average value of Π and Δ, and we plot the maximum value across all our sample graphs of the given size of the same two objects.

In this plot we see the relationship between graph sizes and the sizes of the lists Π and Δ, output by the algorithm. Plots include data from executing the algorithm on graphs generated by the random edge generator and the random neighbourhood generator. The max size of the list Δ_2 follows the size of the graph very closely, generally being equal to or 1 below the graph size. Δ_2s average size seems to grow at the same rate as graph size, although it's value is somewhat lower. Comparing the plots for Π average between edge and neighbourhood generated graphs, it seems to both have a consistently higher value as well as a higher growth rate in the neighbourhood graphs. If we refer back to figure 6.1 which shows object counts of size 10 graphs for different edge probabilities, we can assume that the random edge graphs are closer to the extreme values (very high of very low density) than the random neighbourhood graphs. Considering the size of Π_2 average in figure 6.5 we see that the random edge generated graphs has a lower value than random

neighbourhood generated graphs, connecting this with the other information it seems likely that random edge generation produces high density graphs, while random neighbourhood are medium to high density. To investigate this further we generated the average and maximum edge densities for each set of graphs we use and we list these results in figure 6.4.

Graph size	Edge Average	Edge Max	Neighbour Average	Neighbour Max
9	0.52	0.75	0.52	0.9
12	0.55	0.78	0.53	0.72
14	0.52	0.76	0.55	0.74
15	•	•	0.56	0.73
20	•	•	0.6	0.75
21	0.36	0.67	•	•
25	•	•	0.63	0.76
30	0.26	0.48	0.66	0.78
40	0.2	0.32	•	•
50	0.16	0.21	•	•

Figure 6.4: A table of edge densities across the different graphs used for our tests. The density is related to the number of edges in the graph, and the probability for there to be an edge between two randomly selected nodes. For the graph sizes for which we only generated graphs with one of the methods, a black circle is placed as a filler for the other method.

In this table we see the average and maximum edge densities for the graphs generated by the random edge method, denoted "Edge" and the graphs generated by the random neighbourhood method, denoted "Neighbour". Counter to our assumption, we see that the random Edge generated graphs are actually of lower densities, and as the graph size grows larger, the density tends towards a lower value. This can be explained by the large influence adding one edge has in a large graph which already has many other edges, as the graph grows larger adding a single edge has the potential to create more P_5s. The reason for our assumption being erroneous is likely due to Π_2 not being able to express its full size on the size 10 graphs used in our comparison figure. As discussed earlier, this figure was applied due to the difficulty of generating fully random P_5-free graphs of any larger size. The random neighbourhood generation method produces graphs of medium density, but as the graph size increases, the trends seems to tend towards more dense graphs. It would be interesting to see how this trend continues, but the method for generating graphs of this form is too slow to create a reasonable sample size of larger graphs. The random Edge generator method starts from an independent set and fills in edges, therefore it is also likely to become more sparse as graph size increases, but as we assume the set of allP_5-free graphs to be connected, we could also have started with cliques and executed the same algorithm. By starting with cliques, the output graph

would be connected with a high likelihood, and the edge density would likely tend towards 1, instead of 0 as the current situation is.

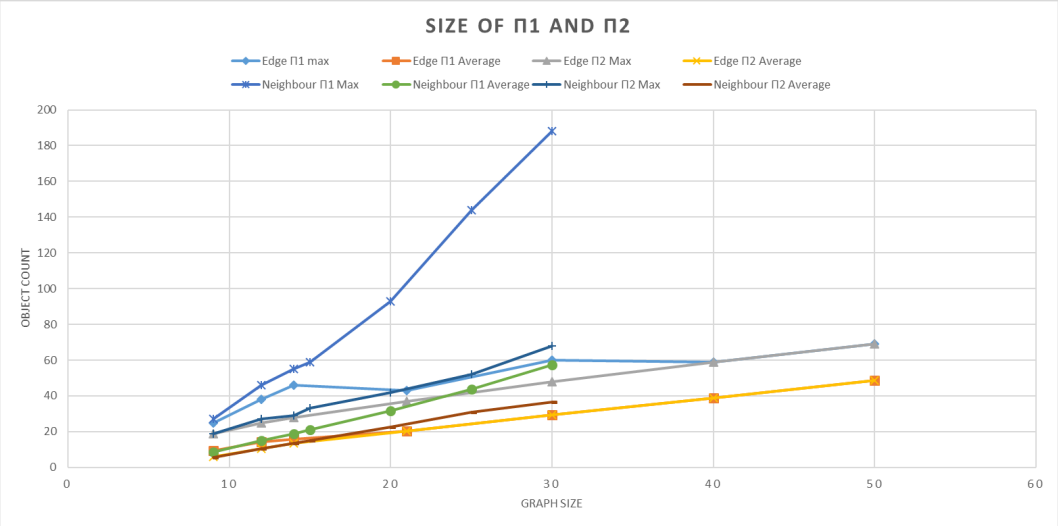


Figure 6.5: plots displaying the relationship between graph size and the size of the lists Π_1 and Π_2 for the two methods of generating graphs, together with plots for the average and maximum object size at each graph size.

In figure 6.5 the neighbourhood method produces graphs with potentially much larger sizes than the edge method. However, the sizes of the lists are far below the theoretical bound. The size of Π_1 and Π_2 are closely correlated, and for the edge method we see that their averages overlap. This is because Π_2 generates a large amount of the same p.m.c.s as Π_1, and the number of unique p.m.c.s in Π_2 is very low. The largest difference between Π_1 and Π_2 size can be found for the size 30 random neighbourhood graphs, and the average is only approximately 2, the max difference is 13.

In conclusion, we can say that the list Π is far below the theoretical bound, even at its largest. The maximum value found was 188 for size 30 graphs, while the bound is $O(|\Delta_2|n^4) = O(n^7)$ which for a size 30 graph would be 21,870,000,000, but as we never found a Δ_2 larger than n, we could try to bound it to $O(n^5)$ which would equal 24,300,000 but this is still far beyond the size of the output.

6.3 Poly-time Independent set algorithm

Running the two algorithms on some sample graphs the results obtained are found in the table in the following figure.

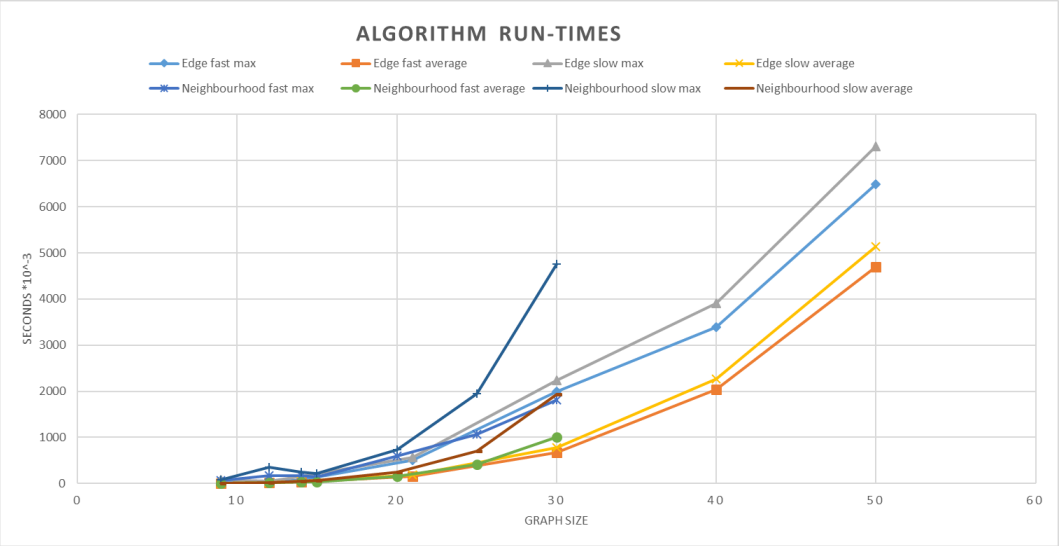


Figure 6.6: Plot displaying the relationship between graph size and runtime for the two methods of generating graphs and the average and highest possible time taken for a given n-value

The figure contains plots for both the fast and the slow polynomial time maximum independent set algorithms. The slow algorithm is more sensitive to the size of the list Π, while the fast one is slightly more dependent on the graph size. Considering the plot for Neighbourhood slow max, the runtime grows very large for size 30 graphs, which likely correlates with the max size Π list, of size 188. Considering that the time needed to generate the list Π is relatively high compared to the time taken for executing the minimal independent set algorithm on the finished list, the difference in runtime between the slow and the fast algorithm seems small even though the fast algorithm would be much faster if given Π as part of the input. A quick comparison was done with the exponential algorithm which was used as a benchmark for the size of the maximum independent set, as expected it had a lower runtime until n started approaching 25-30, in which case the polynomial time algorithms were faster, the runtime data from this does not add much value to the rest of the discussion so they are not included.

If we want to relate the practical run-time of the algorithm to the theoretical runtime, we can relate the runtime to the number of operations performed. We do this by measuring the computers Instructions Per Second (IPS), and then relating this back to the time taken, to get an approximate number of execution steps. We measure IPS by performing some very simple task (like summing numbers) a large number of times and measuring the time taken. For the testing-rig a sum operation was performed 10^{11} times, with a number so that the final result fit into a long (64-bit). This took time 45027477813 ns (nano-seconds), dividing number of instructions by

time taken in seconds (45.027...) we get the number of instructions per second. In our case this number is $2.22 \cdot 10^9$, which matches closely to the 2.3GHz from the i7-3610QM CPU used. To obtain the number of instructions which the algorithm uses, we take the runtime in seconds and multiply it by the number of instructions. Finally we want to relate this back to some n^k, and as we know n, we just solve for $n^k = runtime$, giving us $k = \log_n(runtime)$.

n	k fast	runtime fast ($s \cdot 10^{-3}$)	k slow	runtime slow ($s \cdot 10^{-3}$)
9	8.55	64	8.63	78
12	7.97	179	8.25	356
14	7.49	174	7.63	254
15	7.24	148	7.38	215
20	7.00	591	7.08	727
25	6.70	1068	6.89	1951
30	6.50	1807	6.79	4763

Figure 6.7: A presentation of the relationship between k in n^k and the maximum runtime, for all random neighbourhood graphs. The table contains values for both the slow and the fast independent set algorithm

In the table, we notice that as n grows larger, k seems to grow smaller. For the graphs of size 9, we are considering absolutely all non-isomorphic connected graphs of the given size, thus the given answer is the absolutely worst possible case among all P_5-free graphs of size 9. The reason for k to be decreasing may be related to the smaller sample size compared to the total number of graphs as n grows larger, as without a very powerful computer it would be hard to compute results for much larger sample sizes. Even though we have samples for larger graphs for the random Edge graphs, k is lower than for the random neighbourhood graphs, which corresponds well with the results for object count that we have observed in the graphs so far.

There are three theoretical run-times which are meaningful for comparing the results. There is the runtime for generating the list Π which is given as $O(|\Delta_2|n^6m)$, the runtime of the "slow" maximum independent set algorithm which is $O(|\Pi|^2n^4m)$ and the "fast" algorithm with time bound $O(|\Pi|n^5m)$. As we sum the runtime for generating Π together with the runtime of the algorithm, we end up with an $O(|\Delta_2|n^6m + |\Pi|^2n^4m)$ algorithm for the slow algorithm and $O(|\Delta_2|n^6m + |\Pi|n^5m)$ for the fast version. For the slow algorithm, the worst case scenario obtained is a $O(n^{8.63}) \approx O(n^9)$ algorithm, which is far better than the $O(n^{18}m)$ we get by replacing Π and Δ_2 by their upper-bounded values. Based on our data, we have yet to find a Δ_2 larger than $O(n)$ and we have not seen any Π which seemed larger than $O(n^2)$ (for 30 we have $max(|\Pi|) = 188$ which could be $\frac{30^2}{c}$ where c is some constant). Inserting for these results which are based on our data, we get $O(n^7m)$ to generate Π, $O(n^8m)$ for the slow algorithm and $O(n^7m+n^6m) = O(n^7m)$ for the fast algorithm. This fits our data very well, and it is possible that this is a lower

bound of the current algorithm. The fast maximum independent set algorithm may be able to give a total $O(n^6 m)$ algorithm if a better algorithm for generating Π is found, even though $|\Pi|$ remains unchanged.

Chapter 7

Conclusion

In this thesis we have explained each part necessary for producing a maximum independent set of a P_5-free graph in polynomial time. We have provided descriptions of how an actual implementation is done, and we included means of generating P_5-free graphs for any future testing.

The graph which maximizes the runtime may not be in the set of graphs generated, perhaps it is in its own family of graphs. We were however able to provide a method which has the possibility of generating absolutely any P_5-free graph of a given size. If such a family of graphs exists, it is likely to be very small as none of our results have provided any data supporting the existence of such a graph family. It is difficult to analyse whether the method by which we generate graphs impose some structure on the graphs, which make them less likely to maximise the runtime of the algorithm. We have not been able to expose any such structure from our investigations of the data.

We implemented an algorithm to detect whether a graph contains a P_5. Because we generate graphs by performing single modification steps at a time, we also devised a new dynamic P_5-free graph recognition algorithm with better runtime for each modification step than the runtime of the static recognition algorithm of Hoàng *et al.*[10]. To the best of our knowledge, this is the first such algorithm. We implemented the dynamic P_5-free recognition algorithm and used it in our tools for generating P_5-free graphs.

We discovered a new algorithm to determine whether a vertex set is a potential maximal clique in a given graph. This algorithm has runtime $O(\frac{n^3}{\log_2(n)})$ in the RAM model of computation was discovered. For practical purposes on graphs of the sizes we considered (up to 50), our implementation of this algorithm performed better than the previously known $O(nm)$ algorithm.

We provided a large set of results, both in terms of objects generated and in terms of runtime of the algorithms. Even though we were not able to provide a better

theoretical upper bound of $|\Pi|$, it now seems highly likely that $|\Pi|$ is much smaller than the upper bound of $O(n^7)$ by Lokshtanov *et al.* Moreover the list Δ_2 has yet to exceed size $O(n)$, and it might be possible to bound it to $O(n)$. Assuming that $|\Delta_2| \leq O(n^2)$, we also end up with $|\Pi| \leq n^6$, and inserting this into the algorithm we have runtime $O(n^{11}m)$ as a much more likely upper bound, even though it is probably even lower based on the experimental data.

From our experimental results, the list Π has yet to exceed size $O(n^2)$, and of the two sub-lists of Π, Π_1 seems to be larger, as this approaches size $O(n^2)$ while Π_2 does not appear to be larger than some constant multiplied by n, in other words from the data it is possible that $|\Pi_2| \leq O(n)$.

We also performed experimental tests to determine the upper bound on the runtime, and we concluded with us being unable to get a runtime higher than $O(n^9)$ for the whole algorithm (generating Π and finding the maximum independent set).

The maximum independent set algorithm seems fast enough for it to be viably executed on most P_5-free graphs of size at least up to 50.

All code produced during this thesis can be found at the relevant links in the appendix, together with some test-files used and the code used to generate test graphs.

7.1 Open problems

Some interesting questions that arose during the production of this thesis have been mentioned in the sections in which they are relevant, but for a quick overview they are reproduced here.

- Is there a better algorithm than $O(m^2)$ for detecting if a graph contains a P_5?
- For the dynamic P_5-recognition problem, can we do better than $O(nm)$ for edge addition and deletion? Can we obtain a $O(nm)$ algorithm for the node addition problem (almost possible at present, only missing a way to determine if G contains a P_5 in which the node w is given and w is in the middle of the P_5 of the form uvwxy, in time $O(nm)$)?
- Can one generate random P_5-free graphs of a given size n, such that the graphs are uniformly distributed over the space of all possible P_5-free graphs of size n, and that is reasonably fast? (must be much better than just generating a random graph and discarding if not P_5-free).
- Can we prove NP-Completeness for independent set on some P_k-free graph? If so, what is the lowest P_k for which independent set is NP-Complete? From a more recent paper by Lokshtanov *et al.*[13], it is known that Independent set can be solved on P_6-free graphs in time $O(n^{\log^2(n)})$ so it is likely that k must be at least 7.

- What is the lowest upper bound on $|\Pi|$ using the methods described?
- How many unique components can be generated in a size n P_5-free graph by splitting the graph using two independent vertices u,v, and their closed neighbourhood $N[u, v]$. What effect will this have on $|\Delta_2|$?

7.2 Other applications

For any graph class that has a restricted number of potential maximal cliques, it is possible to use the same algorithm to produce the maximum independent set size. Some graph classes with very small lists of potential maximal cliques are chordal graphs and trees which have at most $O(n)$ maximal cliques. Further Circle graphs, Circular arc graphs and weakly chordal graphs have $O(n^{O(1)})$ potential maximal cliques, for these the algorithms which we implemented run in polynomial time.

Appendix A

The code produced during this project can be found on Github. A push will be made at the same time as this thesis is delivered so it is possible to revisit the code, but generally the code might become edited over time, even after delivering this thesis. Code for random neighbourhood graph generation, together with other tools to analyse the structures considered can be found at

https://github.com/mitresthen/P5freeGraphGen

The main algorithm, together with any helper classes added are found at

https://github.com/mitresthen/PolyP5freeIndependentSet

Please contact Håvard Haug for any further information regarding the code.

Bibliography

[1] Anne Berry, Jean RS Blair, and Pinar Heggernes. Maximum cardinality search for computing minimal triangulations. In *Graph-Theoretic Concepts in Computer Science*, pages 1–12. Springer, 2002.

[2] Jean RS Blair and Barry Peyton. An introduction to chordal graphs and clique trees. In *Graph theory and sparse matrix computation*, pages 1–29. Springer, 1993.

[3] Vincent Bouchitté and Ioan Todinca. Listing all potential maximal cliques of a graph. *Theoretical Computer Science*, 276(1–2):17 – 32, 2002.

[4] Vincent Bouchitté and Ioan Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM J. Comput.*, 31(1):212–232, 2001.

[5] G.A. Dirac. On rigid circuit graphs. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 25(1-2):71–76, 1961.

[6] Fedor V. Fomin and Yngve Villanger. Finding induced subgraphs via minimal triangulations. In *27th International Symposium on Theoretical Aspects of Computer Science, STACS 2010, March 4-6, 2010, Nancy, France*, pages 383–394, 2010.

[7] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424 – 436, 1993.

[8] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15(3):835–855, 1965.

[9] Martin Grötschel, László Lovász, and Alexander Schrijver. Geometric algorithms and combinatorial optimization. 1988.

[10] Chính T Hoàng, Marcin Kamiński, Joe Sawada, and R Sritharan. Finding and listing induced paths and cycles. *Discrete applied mathematics*, 161(4):633–641, 2013.

[11] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, pages 85–103, 1972.

[12] Daniel Lokshantov, Martin Vatshelle, and Yngve Villanger. Independent set in p 5-free graphs in polynomial time. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 570–581. SIAM, 2014.

[13] Daniel Lokshtanov, Marcin Pilipczuk, and Erik Jan van Leeuwen. Independence and efficient domination on p_6-free graph. *CoRR*, abs/1507.02163, 2015.

[14] Stavros D. Nikolopoulos, Leonidas Palios, and Charis Papadopoulos. A fully dynamic algorithm for the recognition of -sparse graphs. *Theoretical Computer Science*, 439:41 – 57, 2012.

[15] J.M Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425 – 440, 1986.

[16] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.

[17] Najiba Sbihi. Algorithme de recherche d'un stable de cardinalite maximum dans un graphe sans etoile. *Discrete Mathematics*, 29(1):53 – 76, 1980.

[18] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, May 1999.