# An implementation of a Feedback Vertex Set algorithm.

Arvid Soldal Sivertsen
Supervised by: Yngve Villanger

June 3, 2013

Graphs has been used as fare back as Leonhard Euler, both a mathematician and a physicist, who in the 18th century used graphs to obtain a solution for the touring problem; touring the seven bridges of the city of Kőnigsberg. He showed both a necessary and sufficient criterion for the existence of a tour, hence solving the problem for all problem instances, and not only the seven bridges of Kőnigsberg.

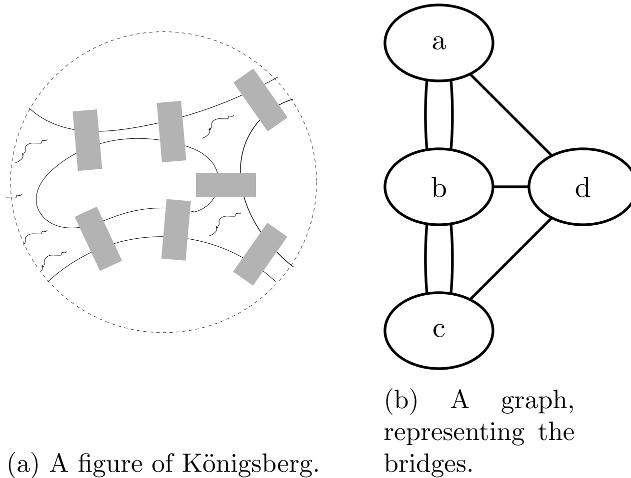
This thesis will define some graph terms and a problem called the Feedback Vertex Set (FVS) problem. Where the basic idea is to cut all cycles. FVS asks to remove a minimum number of ”vertices” to break all cycles. There is always a FVS for any given graph, the difficulty lies in finding a FVS of bounded or optimum size.

Looking at the literature, there has been much interest in the FVS problems, by many people over many decades. Some FVS problems, has even been referred to as a fundamental combinatorial problem. A survey of FVS problems can be seen in [27]. Finding the optimum FVS, is an optimization problem, which is known to be NP-hard, but there are algorithms of the type ”FPT” that solves it.

After introducing the FVS problem that will be the main focus of this thesis, using graph terminology that is also to come, helpful in giving a more precise formulation, an ”FPT” algorithm that solves the problem will be presented. An algorithm that looks to be well suited for paralellization. The algorithm will be presented, followed by a description of conducted experiments and how the experiments perform on a made up problem set comprising of randomly generated graphs, and ending with results that comes from real life problems instances, to get an impression of the problem instances, that can be solved in practice.

The problem of finding a minimum FVS in a graph, i.e. a smallest set of vertices whose deletion makes the graph acyclic, has many applications and in the literature it traces back to the early 60’s [55].

One example of an application for FVS is deadlock recovery in operation systems [53], where a deadlock presents it self as a cycle on the demand for a resource, represented by a resource-allocation graph G. In order to recover from deadlocks, the resource-allocation graph must once again be acyclic. To achieve this, one or more processes in the system are aborted - processes that are part of cycles. Minimizing the number of aborted processes is equivalent to obtaining the minimum FVS. There is also a version of the problem, where the processes has weight and the problem is to find a FVS, minimizing the sum of the weights deleted from G. To parametrize the problem, one look at parameters depending on the graph G e.g., the number processes that has to be removed.




(a) A figure of Königsberg.

(b) A graph, representing the bridges.

Figure 1: The Bridges of Königsberg.

- Euler circuit problem - is there a tour that start on one island u, travels all the bridges exactly once, ending in u again.
- Euler walk problem - is there a tour that starts on one island u, travels all bridges once, ending on island v.

Figure 2: Examples of problems.

As mentioned at the start of this chapter, there are problems that precedes much of the common technology of our time, problems that where well described using graphs. It is imaginable that the bridges where a unique feature of the city Königsberg (see Figure 1), and that exploring the bridges would be interesting to tourists. It would be interesting presenting tourists with a tour that involve crossing every bridge exactly once.

There are two similar Euler tour problem, which are going to be called Euler circuit problem and Euler walk problem, see list of Figure 2. There exists a reduction from Euler walk to the Euler circuit problem. If you construct a bridge between u and v, in the Euler walk problem. Then the problem of whether there was an Euler walk between u and v has the same answer, as whether there now, with the new bridge in place, exist an Euler circuit.

The criterion for whether or not there exist an Euler circuit is well un-

derstood. An Euler circuit exists if and only if every island is incident to an even number of bridges.

For there to exist a Euler circuit, every island must be incident to an even number of bridges. To see that no Euler circuit exist in Figure 1, on this example, it is sufficient to look at one island, and any island will do. Then count the number of bridges that are incident. There is no Euler tour walk either, as the requirement that there be exactly two islands with odd number of incident bridges, is not met.

Euler found how to not only solved this tour problem for the Seven Bridges of Königsberg, but for all cities, by him solving it on graphs. The Bridges of Königsberg are fairly easy to draw on paper. Looking at Figure 1b, you see the bridges drawn as edge and islands as vertices. There are many example where a representation on paper is less than straight forward to draw, and graphs are not an exception. Someone on a road for cars, may have experienced multiple levels, roads on top of roads, often seen close to road intersections. The theory of Euler is however just as valid today.

Euler solved the tour problem for graphs, and not all graphs can be drawn in the plane, without intersecting lines. Figure 1b of the The Bridges of Königsberg is a planar graph, and as you see, the graph was drawn without intersecting lines. Graphs are very useful, however it is important that you represent graphs so that they can not be miss-interpreted.

The existence of a tour in a graphs can easily be checked without knowing where walking "bridges" take you. This is not the case for all problems on graphs. For some problems, finding a solution is not a straight forward process, even if you know how to solve it, and the problem instance is easy to interpret. New results on the solving of problems arise all the time, but there are problems that have seen little progress. To distinguish different problems, many classifications has been suggested. These classifications says something about how problems relate. Researchers are interested in how problems relate, and new insight is of interest to many. Even results that are not immediately applicable to the solving of a problem.

There is also directed graphs, which can change the problem significantly. Directed graphs, would be the representation, if bridges where only passable when crossing the bridge in one of the two directions.

When solving a problem, it can be wise to analyse how much work can be needed to compute it, in addition to implementing a faster algorithm. There is a widely used assumption that P≠NP. Arguments like this, is common when dealing with problems that are not known to be in P, as from empirical experiments, problems not in P tend to be more costly to compute. The hardness of a problem in NP-complete relates to P≠NP. If P=NP, then it

would be natural to try and make a algorithm that runds in polynomial time for NP-complete problems, however any such algorithm would serve as a prove that P=NP. That none has succeeded to proof that P=NP, gives weight to assuming P≠NP.

For planar graphs, finding minimum FVS on directed and undirected graphs, and finding minimum subset FVS on directed and undirected graphs, was already proved to be NP-hard in the 70's [40, 32]. FVS is on Karp's list of NP-complete problems [40].

FVS is an example of a problem, for which there there is much research, some research progress, even progress as if the problem was not known to be NP-hard. There has been several exact algorithms for finding a solution to this problem in the last decade, improving the upper bound on how long it takes to finding a FVS.

Much work has gone into getting away with less work by altering the problem. Either by "easing" the constraint that the found solution is optimum, or considering parameters whose value also depend on input.

An $\alpha$-approximation algorithm for FVS runs in polynomial time, and gives a solution whose size is not more than $\alpha$ times the size an optimum solution. Approximations are interesting as they often give relatively good solutions, and as they have polynomial running time, it may even be worth calculating an approximation, even when the calculation of an exact solution is planed. There are also examples where approximation algorithms can get you very close to an exact solution. Finding an approximation to FVS is however MAX SNP-hard [47], thus it does not have a polynomial time approximation scheme i.e. the ratio cannot go down arbitrarily close to 1, unless P = NP [3] Actually it is so called NP-hard to find a solution with an approximation ratio lower than 7/6 [36], and it was conjectured by Hochbaum [37] that approximating FVS in a constant factor smaller than 2 is itself NP-hard. To be more historically correct [44, 45]; the conjecture among others, where not originally stated for FVS, but for something called a Vertex Cover (VC).

If some instances with specific properties can be shown to be easier to solve in practice, for instance if the solution is small, one may still want to try and find an exact solution. The FVS problem was first shown to be fixed parameter tractable (FPT) by Downey and Fellows [22].

Over the decades, many approaches has been tried on the FVS problem.

The first exact algorithm breaking the trivial $2^n$ barrier is due to I. Razgon [51] with the running time $O(1.8899^n)$, and was improved upon by an exact algorithm with running time $O(1.7548^n)$ [28, 30]. (As an example. the Figure 1b has only one minimum FVS, which is {b}.)

During a study of the number of (vertex) disjoint cycles in a graph, Erdős and Pósa also discovered an approximation with the approximation ratio of $2log_2(n)$ for the minimum FVS on unweighted graphs [25]. There has been much progress since than e.g., constant factor approximations algorithm [4, 8, 6, 5, 26, 42] (There are also applications in [6]). (Valid solutions to a 2-approximation for the minimum FVS problem on the unweighted graph of Figure 1b are; {a,c},{b},{a,b},{b,c} and {b,d})

Examples of various other approaches that has been tried are; linear programming [18], local search [13], polyhedral combinatorics [31, 17], randomized algorithms [48], and parameterized complexity, such as FPT algorithms (see Figure 3 for the list) and randomized parameterized algorithms [7].

The FPT algorithm (see Figure 3 for examples) approach will be dominating in this thesis. There is the size of the input, which is a commonly used variable dependent on input, for bounding the running time of algorithms, and there is parameters. A problem is FPT if there is an algorithm that solves the problem in a running time that is polynomial in the size of the input and potentially exponential in the size of parameter.

It is always a valid question; whether there is an easier way to solve a problem. In the process of trying to make a simpler solution, that is FPT, one might try something like the following, that fails to be FPT. One make an algorithm that on the vertex set, enumerates all subsets of at most k elements, all subsets of at most k vertices, and test each such subset for whether or not it is a FVS, by checking if the graph is acyclic/forest when FVS is deleted. There is however the issue that; the number of subsets of at most k elements is exponential in the number of elements in the vertex set, and vertex set is part of the input. The number of subsets of at most k is exponential in the number of elements in the set one takes the subset of, and still exponential, even when only considering the number of subsets is exactly k in the number of elements in the set one takes subset of. $|\{Y \subseteq X : |Y| \leq k\}| \geq |\{Y \subseteq X : |Y| = k\}| = \binom{|X|}{k}$ is exponential in the number of elements in the set X, as for positive integers

$\binom{n}{k} = \frac{n \cdot (n-1) \cdots (n-k+1)}{k \cdot (k-1) \cdots 1} = (\frac{n}{k}) \cdot (\frac{n-1}{k-1}) \cdot (\frac{n-k+1}{1}) \geq (\frac{n}{k})^k$

Although running such an algorithm would yield a correct answer to the problem, the algorithm would not be FPT, as $(\frac{n}{k})^k$ can not be written in the form; f(k)$n^c$, where c is a constant.

There is also something called kernelization. A kernel is a reduction. In the process of solving a problem for which there is just a yes or no answer; given a problem instance, the instance is a yes-instance if and only if the problem is a yes-instance also when solved on the kernel of the problem

- Bodlaender, Fellows [9, 22] O($17(k^4)!n^{O(1)}$)
- Downey and Fellows [23] O($(2k+1)^k n^2$)
- Raman et al. [49] O($max\{12^k, (4log(k))^k\}n^{2.376}$)
- Kanj et al. [39] O($(2log(k) + 2log(log(k)) + 18)^k n^2$)
- Raman et al. [50] O($(12log(k)/log(log(k)) + 6)^k n^{2.376}$)
- Guo et al. [34] O($(37.7)^k n^2$)
- Dehne et al. [21] O($(10.6)^k n^3$)
- Chen et al. [16] O($5^k kn^2$)
- Cao et al. [15] O($(3.83)^k kn^2$)

Figure 3: The history of FPT algorithms for the unweighted FVS problem. The iterative compression (I.C.) method of [52], is used by the last three on the list.

instance, and further more; the size of the kernel must be bounded by some parameter.

A lot can be learned from understanding and finding kernels. Finding kernels to problems is different than finding algorithm that actually finds a solution to a problem.

The polynomial kernelization of FVS was solved by Burrage et al. [14], producing a kernel of size O($k^{11}$) in polynomial time, improved to a kernel of size O($k^3$) by Bodlaender [10], and further more by Thomassé [54] with a kernel of size at most $4k^2$.

There are also examples of kernelization results that motivate the FVS problem [12].

Some previous results has been presented in this chapter, ending with Figure 3, listing progress on solving the FVS problem, using FPT algorithms on undirected and unweighted graphs.

# Chapter 1

# Introduction

Applications for Feedback Vertex Set (FVS) span from interactive systems like your PC and embedded devices like routers, to some programming languages which is perhaps not very interactive but very expressive.

Creating control systems [24] can be quite costly. It is common to see control systems be developed centrally, by people with a broad idea of what the use is going to be. Ever so often the control system gets distributed as an upgrade or replacement. With efficient solutions to the FVS problems, it becomes more attractive to build systems, that to a greater extent, can reason about them selves. It can play a part in making system that are better adapt, to both the hardware they operate on and environment they serve, with no change in functionality, without user even having to know about it. Preservation of functional behaviour [43]. Usually it goes without saying that one wants systems that claim lesser resources, like wait time and energy, and likewise give more performance and battery uptime.

In some situation, Artificial intelligence (AI) involves modelling of biology, with some reward system. Some AI research has involved languages. AI is often associated with languages such as *lisp* and *prolog*. In some papers[20, 19] concerning performance of some variations of languages, of which prolog belong, FVS is referred to as cycle-cutset.

We will work on an implementation for a FVS problems that will be more precisely specified, on which an optimum solution is sought. Problems, that are not useful in this task, are not going to be specified.

There are papers that seem to use the name “Feedback Vertex Set” less reserved. Like on page 7 of [43].

We wish to improve the empirical performance of solving the FVS problem. Performance, that saves time and allow for bigger problem instances

to be solved. We have not been able to find relevant experiments in the literature, that could be used for comparison. We will not be to able infer any success. We will not be able to show improvement on previous results, but our own.

In this chapter 1, we will introduce the basics and give a shallow overview of relevant subjects in complexity theory. It consists of this overview, important terminology, mostly graph terminology. Though some programming experience and maybe a complexity course will likely do better, some representations and complexity theory will be briefly talked about in this chapter. It should not be thought of as more than a taste, and at best an overview. Ending with some comments on programming languages and architectures. More specific information on tools and about the problem is in chapter 2. Chapter 3 describes the algorithm. After reading chapter 3, one should be able to comprehend, precisely what our problem is, and perhaps something about the algorithm, we are going to implement. A FPT algorithm for finding minimum FVS in undirected graphs. There is a version that works for undirected graphs with weighted vertices, found alongside with a simpler algorithm that still works for unweighted version [16]. The two algorithms share upper bound on time. We will be focusing solely on the unweighted version of those two. Chapter 4 presents an implementation written using C++, and it's C++ Standard Library (std) and for paralellization the OpenMP API, and will explain experiments that has been conducted. How one can avoid some performance issues not directly relate to the parameter, identifying other bottlenecks, and trying to improve the performance of those. Experiments benchmarked based on randomly generated graphs, and performance on graphs found in real world applications are presented in chapter 5, along a summary of the results, followed by chapter 6, with Concluding remarks.

## 1.1 Terminology

A graph G is a pair (V,E) of vertices V and edges. Every element of the set E is an unordered pairs of vertices u and v, the pair is a set and denoted; {u,v}.

That there existing an edge in E, an {u,v}∈E, is equivalent to u and v being adjacent. One might say that adjacent vertices u and v are neighbours. Edge {u,v} is incident to vertex u, as $u \in \{u, v\}$. We say that the two incident vertices of an edge are endpoints.

We will only consider finite data. From an engineering perspective; this is very enabling. We can measure cardinality. We will start of with various measurements; $|V| = n$, $|E| = m$.

We will only talk about undirected graphs in this thesis. Undirected graphs will simply be referred to as graphs.

Let N(v), be the set of vertices all of which are adjacent to v.

The degree of a vertex is the number of edges that it is incident to. $deg_G(u) = |\{\{u, v\} \in E\}|$, where graph G=(V,E). Remember that an edge is an unordered pair of vertices, so $\{u, v\} = \{v, u\}$. The following is true; $\{v, u\} \in \{\{u, v\}\}$.

A set X⊆V induced on graph G, denoted G[X], is a graph where all vertices but X are removed, and also edges that does not have both endpoints in X.

Assume no parallel/multi - edges, and no self-loops. Example of a parallel/multi-edge, with a cycle on 2 vertices can be seen in figures 1 and 1.1a.

A {u,v}-walk is a sequence of vertices, of the form $v_1, v_2, v_3, \ldots, v_l$ where every vertex $v_i$ is in V, and every edge $\{v_i, v_{i+1}\}$ ∈E (in addition to $\{u, v_1\}$ ∈E and $\{v_l, v\}$ ∈E.

When the sequence constituting a walk, do not contain a vertex twice, no repeated occurrence of any vertex, it is called a path. One can check whether or not a walk is a path, by crossing of vertices in V as you read the sequence. If at any time, there are no uncrossed vertex of the vertex that you are reading, it is not a path. Where as if you read until there is no more to read, every time crossing something that was not previously crossed, then it is a path.

A {u,u}-path with no repeated edges is a cycle.

Graphs that has no cycles can be referred to as acyclic graphs. An acyclic undirected graph is also referred to as a forest.

The graph is connected if there is for every pair of vertices u and v, there is a {u,v}-walk.

A graph that is both acyclic and connected is called a tree.

Every maximal set I∈V, s.t. induced graph G[I] is connected is called a connected component (number of vertices is also maximum). A connected graph only has one connected component.

A FVS F of graph G=(V,E), is a set F∈V, s.t. for ever cycle C in G, $F \cap C \neq \emptyset$ , thus a FVS must intersect every cycle.

Given a graph G=(V,E), and a vertex set F, then F is called a FVS of G, if and only if G$[V \backslash F]$ is a forest.

The number of elements in a given set, is the cardinality of the set.

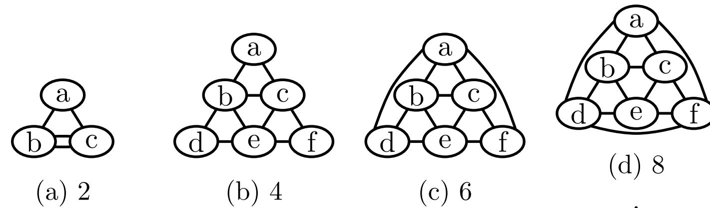



Figure 1.1: Graphs and the number of triangles, where triangle is the length of the shortest cycle. Subfigure a), has a multi/parallel - edge. Those edges, can not be drawn with straight line segments. The other graphs do not have multi/parallel-edges, hence, do not have a cycle on less than 3 vertices.

A "proper" (or "strict") subset X of Y, denoted X⊂Y, means that Y contains an element not in X.

Bipartition is a graph defined by 2 sets A and B, such that $V = A \cup B$ and $A \cap B = \emptyset$, on graph G=(V,E). It is supplied together with some property that must hold for A and/or B.

For instance, a bipartite graph is a graph with 2 sets A and B, such that $V = A \cup B$ and $A \cap B = \emptyset$, on graph G=(V,E), such that u∈A and u∈B for each edge {u,v}∈E. A graph is bipartite if and only if it is a bipartition where both graph G[A] and G[B] have empty edges set's.

**Using the terminology in an example** The problems in list of Figure 2, where assumed to be on connected graphs. If the graph is not connected, there can not be a tour. Let's state the criterion of Euler, with regard to tours, more precisely. Given a graph G=(V,E). An Euler circuit exists if and only if G is connected, and for every vertex v in V, $deg(v)$ is even.

## 1.2 Optimization

The topic of optimization is vast. The FVS problem that is going to be solved, can be formulated as an optimization problem, but we will come to that in Chapter 3. This section on optimization, is just some examples of optimization problems, that give an idea of how graphs can be a useful tool in solving problems. We also get to practice, using the words, we introduced.

There are many examples of optimization problems that take polynomial time to solve. Many involve picking edges. Problems such as finding shortest {u,v}-path for given vertices u and v in a graph, where edges typically has weights. The shortest {a,e}-paths in Figure 1.1b, are {b} and

{c}. The shortest {a,e}-paths in Figure 1.1c, are {b}, {c}, {d} and {f}. You could equivalently formulate it as finding a {u,v}-path which minimizes the travelling, or even as finding a minimum {u,v}-path. If there are only positive weights on edges, then a minimum {u,u}-walk would be the same as a minimum {u,v}-path.

Minimization problem, refers to a problem where it is natural to seek the lesser solution. A maximization for such a problem could be trivial to find. A Minimal solution X is a solution for which no proper subset is a (feasible) solution. Minimum means that there are no (globally) solution that does better. A minimum solution is also a minimal solution. Maximization problem, refers to a problem where it is natural to seek the greater solution. A minimization for such a problem could be trivial to find. A Maximal solution X is a solution that is not a proper subset of any (feasible) solution. Maximum means that there are no (globally) solution that does better. A maximum solution is also a maximal solution.

If a minimal solution happens to be an empty solution, or a minimal solution is not a proper subset of anything, then it is also a minimum solution. Likewise for maximization problems.

## 1.3 Data structures

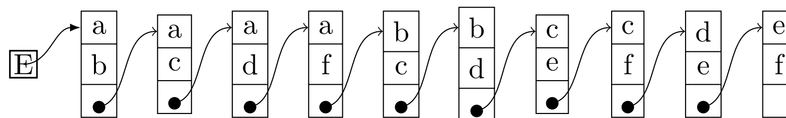
Here, we will mention a few ways to represent graphs, and ways to organize the data in memory. Usually there is abstraction that facilitate the placement of elements in memory, so one need not worry about whether, and where, there is room for data. Programming languages helps with keeping track of what is what and where.

We had that a graph is a pair; vertices and edges. You could have vertices, labelled trough a natural numbering With edges in a list (see Figure 1.2a).

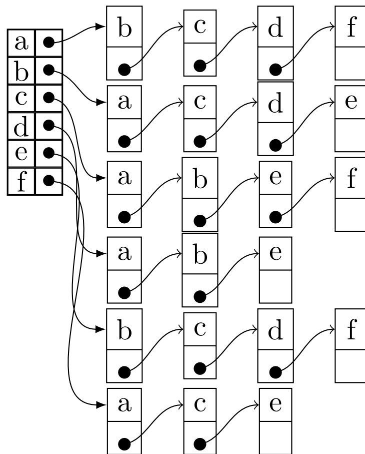
When determining if a give edge is in E, one might look through the whole list. There are other structures as well, ones where you can binary search (std::binary_search), and balanced binary search trees (std::set). That perform better for a wide variety of cases.

Which structure you should use can for instance effect the time it takes to add and removing vertices and edges. Thought much of the difference of performance among the representations of graphs is on the operation of adding and removing vertex.

Figures 1.2a, 1.2b and 1.2c, respectivly show an edge list, an adjacency list and an adjacency matrix.




(a) Shows an list of the edges. Did not draw vertices, as it would just be, a table. No arrows till or from.




(b) The adjacency list

.

$$M = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

(c) An adjacency matrix.

.

Figure 1.2: Common representations of graphs, in memory. Representing the drawn graph of Figure 1.1c.

**Adjacency List** Perhaps you want to remove a whole vertex. With degree of vertex u being $deg(u)$, we would perhaps like to remove these incident edges without looking every one up. To speed up the removal of these edges, one can for each vertex, store adjacent vertices and/or edges.

Location of edges is a bit different. Looking at the adjacencies of a given vertex involves iterating a list. A linked list is common.

When working with undirected graphs, and you remove a vertex u, you also have to remove node from list of adjacencies in each vertex v adjacent to u, the node that indicate u is adjacent to v. Vertex removal, can be a single lookup for each of the incident edges removed.

There are problems that involve having information on edges (e.g. weight). Perhaps you do not wish to have the information, lie more than one place in memory. In such cases it might be practical to have edge be objects arbitrarily located in memory also. With the structure that helps to remove incident edges mentioned in last paragraph, having an even more fragmented placement of data is a minor change. However, when implementing something it is practical to avoid trying to generalize the parts that are going to be building blocks, beyond any foreseeable application.

Adjacency list, is also an useful structure when searching in graph. Graph searches such as Breath-First Search (BFS), and Depth-First Search (DFS).

**Adjacency Matrix** Adjacency matrix is a structure that might claim some memory, but looking up becomes a single accesses. However, removing or adding vertices to such a graph structure, is largely, the same as making a whole new matrix. Essentially duplication.

When algorithm involve many operations that makes few changes to a graph matrices are often not used. Except for lookup in special circumstances, and their simplicity is attractive.

If however one wish to reorder the vertices, for what ever reason. Swapping the place of one vertex with an other vertex, is doing $2 \cdot n$ accesses. And it is possible to revert any number of swapping, by swapping $n$ times, if one just keep track of which vertex is where.

It is also difficult to represent some graphs. Representing multi/parallel-edges on a weighted graph, can be difficult in matrices.

Looking at the adjacency matrix, one sees a $M \in R^{nxn}$ matrix. Let $M_{i,j}$ be the number of edges incident to i and j.

$A \in R^{nxn}$

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}$$

$A_{i,j} = a_{i,j}$

If there are no {u,u} edge, than there are no self-loop, Thus the diagonal $M_{u,u}$ are all zeroes.

For undirected graphs the adjacency matrix is symmetrical; $M_{u,v} = M_{v,u}$.

Looking up whether or not two vertices are adjacent is one access when you have an available adjacency matrix.

Following, is an adjacency matrix representing Kőnigsberg seen in Figure 1a.

$$M = \begin{pmatrix} 0\ 2\ 0\ 1 \\ 2\ 0\ 2\ 1 \\ 0\ 2\ 0\ 1 \\ 1\ 1\ 1\ 0 \end{pmatrix}$$

One can see that $M_{i,j}$ is the number of edges incident to both i and j, the number of {i,j} edges.

Letting $R = \mathbb{R}$ and use the definition of matrix multiplication; $[A \cdot B = C]$ $[C_{i,j} = \sum_{k=0}^{n} A_{i,k} \cdot B_{k,j}]$ As we are working on a graph, with no loops, vertices of a walk constitute walking on edges. One can interpret the number $C_{i,j}$, as counting the number of ways one can do a {i,j}-walk, being the number of valid walks one can take, starting in vertex i, walking a walk counted by A, followed by a walk counted by B, and finally ending in vertex j.

And when all walks in A are on a number of vertices and all walks in B is on b number of vertices, than all walks in C are on (a+b) number of vertices.

Using matrix multiplication, can be an efficient way of solving some problems. Like finding how many triangles there are in a given graph, where a triangle is a cycle on 3 vertices. Se Figure 1.1.

## 1.4 Complexity

In earlier parts of the 20th century, the Church-Turing thesis addressed, what was computable. Algorithms where shown among other things, to be equivalent to Turing-machines. Non-deterministic Turing machine and

deterministic Turing machine. This is a powerful machine model for doing computation, which is still in use today.

An important concepts in complexity is; recognizable.

If we have both recognizable and co-recognizable; we have decidable.

There are many interesting problems that feels natural to formulated as optimization problems. To formulate an optimization problem as a decision problem, they are formulated as seeking a solution that is not worse than some constant k. For the optimization problem; "find shortest {u,v}-path", one could be formulated as the decision problem; "is there a {u,v}-path with length ≤k ?" .

For practical reasons, nothing but decidable problems will be treated in this thesis.

Analysis of the resources needed, by an algorithm, for an input. Resource like time and space. Generally the analysis is based on the size of the input.

The upper bounds on resources one establishes through analysis of an algorithm can be quite verbose. One would like to have short and meaningful expressions for resource usage. We use the asymptotic notation O(f(n))={$g(n)| \underset{n_0,c \in N}{\exists} \forall_{n_0 \leq n}\, g(n) \leq c \cdot f(n)$}. In many cases it allows for a simpler expression f(n). Presenting; "solvable in O(f(n))", is to mean, that the established upper bound on resource, is in the set O(f(n)).

**Polynomial (P)** Means that the upper bound can be formulated as a polynomial. Solvable in O($n^{O(1)}$) time.

**Fixed-parameter tractable (FPT)** It is natural to base the analysis of an algorithm based on it's input size, as it is usually accessed by the algorithm. In man cases there is something about the input, that plays a role in the performance, something that can be determined, but is not known. As all P problems has an algorithm that can be written O($n^{O(1)}$), all FPT problems has an algorithm that can be written O(f(k)·$n^{O(1)}$), where k is defined alongside a FPT problem. If k is fixed, as in a constant, than O(f(k)·$n^{O(1)}$) time can be simplified to O($n^{O(1)}$) time.

**Approximable** Means there exists an approximation algorithm for the problem. Approximations often makes sense for problems that are not in P. Approximations algorithm use polynomial time. When making an approximations algorithm, one try to make an algorithm, where there is a bound on how bad the solution can be, compared to opt. Some function is available to evaluate the value w of a solution S, namely w(S). w(S) is assumed to

always be non-negative. An $\alpha$-approximation for a problem, is an algorithm which returns a solution X, where n·w(X) is not worse than w(B), where B is the best solution.

If $\alpha$ is a constant (e.g. a 2-approximation), the approximation is often referred to as a; "constant factor approximation".

**Preprocessing** A notable preprocessing is the kernelization. A kernelization is an algorithm in P. A kernelization is an algorithm that reduce problem X to problem X, to reduce a problem to it self, in such a way that new input to X, takes O(f(k)) space. k is defined alongside a kernelization for the problem.

Seeing as we bound the resource usage on input size. The existence of a kernel implies that the problem is FPT. So saying that a problem can be kernelized is at least as stronger statement, as saying problem is FPT.

When such an reduction is made, any algorithm for actually solving, how ever crude, is a FPT algorithm.

Noteworthy kernelizations are polynomial kernelizations and a subset there of; linear kernelizations.

Important to remember, that even if a problem is kernelizable (or FPT), it can still be highly impractical. Practically speaking; arbitrarily impractical.

**Reduction** [40] One can imagine that an issue of solving some problem can be reduced to solving another problem, by only alteration of the input. The problem one reduce to is in a sense more powerful. More correctly; the problem on reduces to is at least as powerful as the one reduced from.

**NP-hard** Unlike P and NP, that says something about how easy a problem is. Showing that a problem is NP-hard, is done by showing that it is harder than some problem that is already known to be NP-hard.

If a problem is in P, one can trivially claim that it is NP. There are also many NP problem that one can trivially see is NP, yet are not believed to be in P. In literature one seldom sees NP-complete problems for which seeing that the problem is NP, is not trivial. Exploring NP-hard problems, with NP-hard in mind, is not old, as an field. There are still frequently results on problems in P.

A problem being known to be NP-hard, suggest that one should use an algorithms that not necessarily give the globally best answer. While at the same time, being FPT suggest that it is feasible. With FPT, one can

even expect estimates of the real running time, to be analogous to what most people are exposed to when using a computer, if certain aspects of the problem instance, is known.

**Non-deterministic polynomial time (NP).** Trying to explain what NP means using the word; "mistake". With mistake being the opposite of taking the best course of action. Imagine an omniscient entity. This entity would (or at leas could choice to) never do a mistake. If this entity where to solve a problem, the solution would come immediately (and likely quite unfulfilling). If someone not thrusting or unaware of this entities potential, requested in addition, information to help acknowledge that the solution was indeed correct, one should not be surprised at some very clever constructive answers. If the entity where performing the steps of an algorithm, it could proceed very fast to the solution. Every time seeing more than one course of actions, the entity could just pick the one that would lead to a solution, or lead to a solution the earliest. NP problems would be solved in P time by this entity. But more useful, simulating/pretending the solving of a NP problem, would be possible in P time, if you already know the best choices. Many problems one can identify as NP, by easier means.

In practical terms NP means that some information can be supplied which allow the return to be recognized as a correct answer to the problem in question using no more than P time, which in addition means, that the information can not take more than P space.

Sometimes, there are several, just as good, answers. Sometimes there is some kind of equivalence in the input (and sometimes even in the problem). Isomorphie. A graph isomorphism in the case of undirected unweighted graphs, exist, when there is a function f: vertces$\mapsto$vertces, that is a bijection/permutation, other than the identity. S.t. $E = \{\{f(u), f(v)\}|\{u, v\} \in E\}$. It is often far easier to see that two graphs are isomorphic, when drawn as points and lines, rather than viewing two adjacency matrices.

Figure 2.3 shows two identical graphs, and hence also isomorphic. If the labelling of vertices where mixed up, it is imaginable that recognizing that they are isomorphic, would be more difficult.

An example of problem that is NP, but perhaps not trivially so is; graph isomorphism. Which is the problem of whether; two graphs are isomorphic. Given the permutation/bijection between the vertices of two graphs, recognizing that the two graphs are isomorphic is straight forward. Recognizing that two graphs are not isomorphic (coNP) could be more difficult.

An example of problem that is NP and coNP is integer factorization.

The problem of deciding if; a given integer is the product of multiplying two integer (will be ignoring the fact that $1 \cdot x = x$ and $-1 \cdot (-x) = x$). Whether the problem of integer factorization can be trivially be seen to be NP, depends on whether or not on view integer multiplication as trivial. Recognizing that; an integer X is not the product of multiplying two integer, is known to be in P, so one can trivially infer it to be coNP (coP⊆coNP). Yet the algorithm in P for doing that recognition, is not so trivial.

## 1.5 Practicals

**Parallelization** Empirical data, can be very useful. Analysis of how well algorithms parallelize, is still often done by experiments on real machines. When evaluating how well the algorithm is parallelizable it is common to use speedup and efficiency. Speedup $S_p$ is defined as, $S_p = T_1/T_p$, where $T_1$ is the serial time used and $T_p$ is the parallel time used. And efficiency is $E_p = \frac{T_1 \cdot 1}{T_p \cdot p} = S_p/p$. If we where only timing the time of a serial implementation, taking the best out of several run would be reasonable. Due to the nature of the algorithm we are testing, it is not natural to take the best, we will take the average. OpenMP is a language extension, who's specification is maintained a group of firms. A crude distinction of parallel general purpose machines is; distributed and shared memory machine. It is designed for programming on system with shared memory. Shared memory machines usually have lower memory latency, but also lower bandwidth than distributed memory machines. Programming for systems with shared memory is relatively straightforward. It is the pervasive architecture on home computers. If you have machines that cooperate to solve a problem, communicating over for instance the internet, it is a distributed memory architecture. The total bandwidth typically does not surpass shared memory, before adding a significant number of nodes. Getting familiar with OpenMP can be done in very little time.

As the problem that we are solving is NP-hard, it not to expect that we can solve for all graphs that take significant space. As the algorithm is FPT, input may take significant space. Making random graph is one problem, making random graph with specific characteristics can be much bigger problem. There is also the question about preprocessing.

Thought some problems one invest space for a more flattering time analysis, such a trade of is not known for this problem.

**Machine programming language** It is usually extremely time consuming to directly write instructions for a machine, also called machine code. There is assembly language, and there is the higher level programming languages.

There are many higher level programming languages, each facilitating on or more programming paradigm. Each programming paradigm having things that are easy to do, and if you try to go against the paradigm, you may have a harder time. A compromise between doing things your self, and automatic translation doing things.

In our experiments, we test implementations of the algorithm presented in chapter 2, made in the programming language C++. Using Standard Template Library (STL), a part of the C++ Standard Library (std). SGI provides very good documentation/reference of STL, on their web-server, that has not changed since 2000. C++ has been around for some time, and is still evolving. There are very good tools for translating C++ into machine code.

# Chapter 2

# Tools and practical information.

Will here present information, useful for the algorithm that is to be presented and in the implementation of the algorithm.

An acyclic graph is a forest. For graphs, opposed to directed graph, it is common to use union-find to test whether or not it is acyclic.

## 2.1 union-find

Union-find, also known as disjoint set's, is a usefull data structure as we work on undirected graphs. Given $f : V \mapsto V$. Union find does the following and often more; For $v \in V$, let $find(v) = \begin{cases} v & \text{if } v = f_v \\ find(f_v) & otherwhise \end{cases}$

That $f_v = v$ for all v∈V, where G=(V,E), is representative of a graph without edges, or that the data structure has just been initialized, and no edges has been looked at as of yet.

In Figure 1, an algorithm for the union operation is defined. Running time of union is based on the time find takes. Find takes O($\alpha(\dots)$) time, where $\alpha(\dots) \leq V$.

The time O($\alpha(\dots)$) that find takes, can be reduced to $\alpha(\dots) \leq$

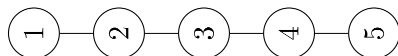



Figure 2.1: This graph is a tree.

**Algorithm 1** union({u,v}), where {u,v} is an edge in E, and let operation; y←x, value assignment, represent y taking on the value of x also

1 $i \leftarrow find(u)$
2 $j \leftarrow find(v)$
3 **if** $i = j$ **then**
4 return "NO" # There is not only the path ∅ between u and v, due to this {u,v}∈E , but also some other path.
5 **else**
6 $f_i \leftarrow j$ or $f_j \leftarrow i$ # which ever you prefer, choice does not even have to be consistent. Either way, i and j is linked. There is a path between u and v; $\{u, v\}$.
7 **end if**
8 return "YES"

$\lfloor ln(n)/ln(2) \rfloor$, by using something called rank.

On line 6 of Algorithm 1, there was an arbitrary choice made. Ranked union-find, involves making a less arbitrary choice. Every vertex $v \in V$ is assigned rank. Initialized to 0. If $i = j$, rank of vertex i or j is arbitrarily incremented. Of i and j, the one with higher rank is set as parent of the other. Find there for is $\alpha(\dots) \leq$ vertex with max rank. By induction, a vertex r and it's descendens, has at least $2^{rank(r)}$ vertices, where $rank(r)$ is rank of r. There for the vertex with max rank is $\leq \lfloor ln(n)/ln(2) \rfloor$.

This can be improved further, by amortizing. Said; that using rank, one get expression $2^{\alpha}(\dots) \leq n$. By altering the find, one can get $2^{2^{2^{\cdots}}} \leq n$ , where the amortizing time $\alpha(n)$ is the number of 2's in expression.

Usually you do not write the 1 in expression; "$x^1$", but when $\alpha(\dots) = 0$, there is no 2's. There is a 1.

## 2.2 forest

Algorithm 2 determines whether a graph is a forest or not. Note that with regards to data structure, it is important that one does not consider an edge twice. This can in practice, be achieved by in a consistent fashion; only considering adjacencies where $u < v$ or $u > v$.

The term component has to do with how a graph is connected, the connectivity. An example of a graphs with little connectivity, are graphs withouth edges. When two vertices u and v are connected, means that there is a {u,v}-walk or path between the vertices. Sometimes this is referred to

**Algorithm 2** Forest(G), G=(V,E), and let operation; y←x, value assignment, represent y taking on the value of x also

```
1  if V < |E| − 1 then
2    return "NO"
3  end if
4  for v ∈V do
5    f_v ← v
6  end for
7  for {u, v} ∈E do
8    if not union({u,v}) then
9      return "NO" # There is not only the path {u, v} between u and v,
       but also some other path.
10   end if
11 end for
12 return "YES"
```

as vertex u and v, being in the same connected component. Union find can be thought of as a function find(v), which takes as input a vertex u, and gives an id or label, corresponding to the connected component u is in. If for vertices u and v, find(u)=find(v), u and v are in the same component.

Let $\alpha(\dots)$, be the same function seen, when describing, union-find.

**Theorem 1.** *Forest(G) takes O($|V|\alpha(\dots)$) time.*

*Proof.* We described an algorithm that does this. See Algorithm 2. After initial assigning. The algorithm iterates over the edges E of G (it does not always iterate all of E, it may halt earlier). In each iteration step, besides from the find operations (on both endpoints $u$ and $v$ of $\{u, v\}$), everything takes O(1) time. There are at most $|V|$ steppes of the iteration, before it halts, as tree has $|V| = |E| - 1$, and any added edge to a tree is part of a cycle.

The time complexity of Forest(G), boils down to the time of doing the $|V|$ union-find find operations. O($|V|\alpha(\dots)$) time. □

The find operation of union-find does not take constant time. There are many strategies to minimize the amortized running-time of the find operation of union-find. When it comes to halting early, only the order of which E is iterated can have any effect. Note that halting early can only occur if there is a cycle, and that there is a cycle that does not involve all the vertices of the graph.

## 2.3 Making a forest by deleting vertices

Given a graph G=(V,E) FVS is just a set $F \in$V, s.t. $G[V \backslash F]$ is a forest.

In it's self it is not a problem finding a FVS. For instance V is a FVS,

Feedback Vertex Set is NP-complete on undirected graphs [33], but they must assume something more, as just return-ing V is not NP-Hard, returning V would only take O($|V|$) time.

From a pragmatic point of view you would typically assume that you want the return FVS F, to be minimum. $|F|$ as small as possible.

Examples of problems based on finding FVS $F$ of graph G=(V,E).

**List 2.1**

(i) $|F|$ as small as possible. Optimization problem. (What we will do).

(ii) $|F|$ as small as we can get given resources (time, space, energy, ...), and how much of the resources, is available.

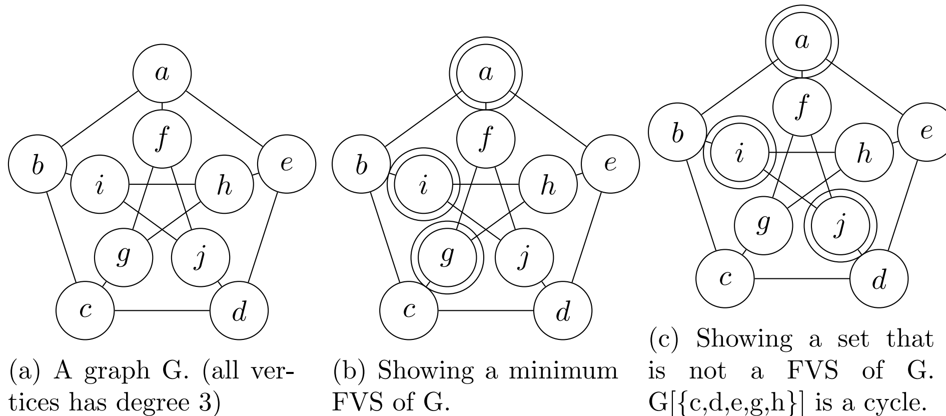
(iii) $|F| \leq k$, where k is some integer supplied alongside G.

Although we are working with undirected graphs, it is worth nothing; the last problem on this list, was described for directed graphs in the paper "Reducibility Among Combinatorial Problems"[41] from 1972. Alongside many others. Most if not all problems are of the form; "does there exist ...", which would have a boolean answer. Most if not all problems being NP. It is naturally to think that solving such problems could yield more information than a simple Yes or No. One may wonder what implications, finding the solution in full, would have on the difficulty of problems. In many problems it appears to only be tedious.

There are many problems on the web, that one can do as an exercise. For instance dynamic programming problems. Which may help one, with regards to formulating problems, as well as be exercise.

Later in this paper you will be confronted with something opposing (in addition to not being DP), an algorithm that solves a problem and naturally returns the answer, albeit an algorithm with far from the best performance if directly implemented. And a reformulation of this algorithm will be made. A reformulation that makes the algorithm return, a boolean answer. This might be a bit tedious.

Saying the last paragraph again; you will see an algorithm that solves Problem 2.1 (i), get reformulated into an algorithm that solves Problem 2.1 (iii). If the reason for doing so does not become immediately apparent,




(a) A graph G. (all vertices has degree 3)

(b) Showing a minimum FVS of G.

(c) Showing a set that is not a FVS of G. G[{c,d,e,g,h}] is a cycle.

Figure 2.2: A graph with minimum FVS of cardinality 3.

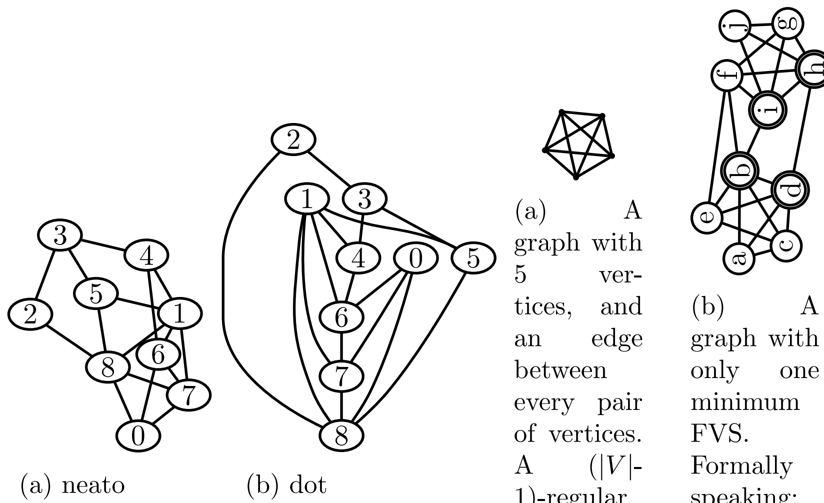
it will become apparent once we, proceed with how one can solve Problem 2.1 (i), using Problem 2.1 (iii).

Figure 2.2 shows one minimum FVS. It is easy to see that there are more, since one can for example mirror the graph about the ”Y-axis”. The ”xy-plane“ has 5 rotations. You can easily see there are $2 \cdot 5$ minimum FVS's. Even more minimum FVS's can be found in this G, by looking at symmetries/ graph isomorphisms. Symmetries/ graph isomorphisms is just ”relabelling of vertices in picture“ without ”cheating”. Erasing labels and writing any permutation of the labels, would be ”cheating”. We will not consider graph isomorphism. We will not be enumerating the solution of our problem. Focus is finding a solution. Focus is on optimization.

Paper [29] describes a graph class (graphs of arbitrary size) with many minimum FVS's.

Figures 2.3 and 2.4a, also has minimum FVS of cardinality 3.

For every (not necessarily minimum) FVS $F$ of the graph in Figure 2.4a, G[F] is not a forest. Can you find a minimum FVS $F$ in the graph of Figure 2.3, such that G[F] is not a forest, and one such that G[F] is a forest? The number of minimum FVS's in the graph of Figure 2.4a is 10, as the number of minimum FVS's in a ”complete graph” with $|V| = n$, is $\binom{n}{n-2} = \binom{n}{2}$ (which is also the number of edges).




(a) neato

(b) dot

Figure 2.3: Two drawings of the same graph G. As for all graphs, it has a minimum FVS F. In this graph $|F|$=3

(a) A graph with 5 vertices, and an edge between every pair of vertices. A ($|V|$-1)-regular graph. A "complete graph".

(b) A graph with only one minimum FVS. Formally speaking; an unique minimum FVS.

Figure 2.4

# Chapter 3

# The algorithm

Problems that are in NP, e.g. finding a FVS of upper bounded size, would be in P, if P=NP. These terms are applied to problems that have yes no answers. While optimization has to do with getting better solutions. A common approach in many fields, is to work with a feasible solution and improve on it. There are many examples real life problems that are so involved, it becomes difficult as you work with the problem, to keep all aspects of the problem in minds. If no better solution than what is sought should exists, if finding the best solution possible is the problem, one might question whether trying to improve on the best feasible solution known, is a worth while approach. Computing a 2-approximation, a FVS with approximation ratio 2, would be looking for finding a descent solution, from which one could try and improve, however it does not just give a feasible solution and upper bound on the size of the best solution, but it also gives a lower bound on the size of the best solution. A lower bound may indication whether finding an optimum solution is practical. Computing approximation is also motivated by what was earlier mentioned, that approximation algorithms run in polynomial time. Still the question remains, is it worth while trying to iteratively improve on previous known best solution.

Many new results in computer science are algorithms, that provably in some regard, has a better time complexity than what is already present in the literature. Unfortunately, some are impractical, and are only of theoretical value. Many algorithms performs better, on a large range of problem instances, than analysis reveals. There are many examples of new results, that is solely an analysis of previously published algorithms. As in just an improvement on some number in an expression.

In this thesis, we will focus on empirical evidence, from the implemen-

tation of a promising algorithm. Empirical evidence can be just as useful as an analysis, when it comes to improving on an algorithm. There has been published quite a few FPT algorithms for solving FVS recently. FVS has been known to be FPT for quite some time, by somewhat general theory involving monadic second-order logic (MSO). FVS was found to be FPT where parameter was the size of the FVS in [11], but no practical FPT algorithm that specifically solved the FVS problem, has appeared before. Recently, with many papers, such as [35], that uses compression, there might be some hope of practical use.

This chapter is largely self-contained, building on on terms, defined early, in previous chapter. We define the problem as follows.

FIND MINIMUM FVS
Input: A graph $G = (V, E)$.
Output: $F \subseteq$V, s.t. $|F|$ is minimum and $G[V \backslash F]$ is a forest.

This is an optimization problem. F is a minimal FVS of G=(V,E) if and only if a induced forest G$[V \backslash F]$ is maximal. F is a minimum FVS of G=(V,E) if and only if a induced forest G$[V \backslash F]$ is maximum.

In this chapter we describe the unweighted algorithm in [16], which solves the stated problem. We will prove why the algorithm solves our, problem, and the correctness of the algorithm. Followed by a simple analysis of the algorithm. Later as we concern our self with empirical evidence. We are content with showing that algorithm is FPT.

This chapter has been divided into 3 sections; 3.1 with an optimization FVS algorithm, and the decision FVS algorithm, 3.2 with the optimization algorithm. and **??** with an analysis.

Roughly speaking. An optimization algorithm solves an optimization problem. And a decision algorithm solves a decision problem.

In previous chapter, when talking about NP, decision problems, where mentioned, but there where no examples where decision formulation of problems can be of direct use, in solving a problem. Now this is not a history of how, thee FVS algorithm came to be, but it could very well be. As common practice is, one first show that the algorithm is true to the name, that it actually solves the problem. This is what Section 1. is about. Then we follow up with, Section 2, where the optimization algorithm of Section 1 is formulating as a decition algorithm. It is quite analogous to formulating, the optimization problem as a decision problem, albeit somewhat more tedious.

This decision algorithm will, be written out in full as well. Figure 4 in this thesis and Figure 2 in the paper of Chen et al. [16], showing an algorithm, are the same. At least for all intents and purposes, they are the same in this thesis.

Showing a FPT algorithm exists is one way of showing that a problem is FPT.

Anyway, the main point of this section is to show that the decision algorithm is FPT. We already mentioned that FVS, is known to be FPT. We will show that it is FPT, where the parameter k is k=$|F|$, where F is FVS F.

The problem that we are going to solve, is the decision problem; "is there a FVS-bipartition s.t. $|F| \leq k$ ?", can be solved in $O(f(k)n^c)$ time, for some constant c. FVS-bipartition us a constrained set much like the FVS.

Will show how to solve the decision problem (see Figure 3), and how it is made even more tedious, by it also yielding the set F (see Figure 4). As this paper is about empirical evidence. For more precise information on the time complexity, see [16].

Section 3 will contain how the decision problem of Section 1 can be used to solve thee optimization problem, of finding minimum FVS.

By the way this chapter has been presented so far, one might be laid to think that Section 3 is a lot simpler than the previous two. That may very well be true, but keep in mind that the two first sections in on a different FVS problem, the FVS-bipartition problem.

In Section 3, we show how one can solve in $O(f(k)n^c)$ time, the decision problem; "is there a FVS F s.t. $|F| \leq k$ ?“, using the decision problem of Section 2. With this, just a little more effort is needed to show that minimum FVS, can be solved with same bound on time used.

## 3.1 An optimization FVS algorithm

This chapter introduces one new problem, the FVS-bipartition problem. We will, as mentioned, show that an optimization algorithm works, in this section. An algorithm that solves minimum FVS-bipartition. (To that end we will first introduce several rules.) The minimum FVS-bipartition algorithm, as later seen in Figure 4, be used later in this chapter. It is shorter, and it is perhaps easier to see that it works. Which is why it has the word; "pedagogical“, in it's name.

Given a graph G=(V,E) and a partitioning {A,B} of V, such that G[A] and G[B] are forests. A vertex set F⊆A is a FVS-bipartition if and only if

$G[V\backslash F]$ is a forest.

**Definition 1.** *Given a FVS-bipartition of a graph G=(V,E), where both* $G[A]$ *and* $G[B]$ *are forests, is a set F, s.t.* $F \subseteq A$*, and* $G[V\backslash F]$ *is a forest.*

The set A is always a FVS-bipartition of a graph G, where G meets the requirements.

FIND MINIMUM FVS-BIPARTITION
Input: Graph $G = (A \cup B, E)$ that is a bipartition, where both $G[A]$ and $G[B]$ are forests.
Output: $F \subseteq$A, s.t. $|F|$ is minimum and $G[V\backslash F]$ is a forest.

We will elaborate on the solving of FVS-bipartition is this and next section, and show how to use FVS-bipartition, in Section 3.

We need introduce observations and reduction rules.

A tree has n-1 edges. If you add an edge to a tree you connect two vertices that are already connected, you create a cycle. Adding an edge to a tree, gives you something that is not neither a tree nor a forest. A forest has at most n-1 edges.

**Observation 1.** *Every forest has a vertex with degree* $\leq 1$*.*

*Proof.* Assume the opposite, that we have a graph G, where every vertex has degree $\geq 2$. Then there must be at leas n, edges. As a forest had at most n-1 edge, G can not be a forest. □

**Definition 2.** *Let (G,k) be the decision problem of finding a FVS F of G, where* $|F| \leq k$*.*

FVS-bipartition is also a FVS. Stating the following could, just as well be done using FVS-bipartition.

**Reduction 1.** *If there is a vertex w with degree* $\leq 1$ *in G, then (G,k) is a YES instance if and only if (G[V*$\backslash\{w\}$*],k) is a YES instance .*

*Proof.* We are reducing problem instance (G,k) to a problem instance (G′,k), where G′ ← G[V \ {w}].
⇐ Given a solution F′ of (G′,k). Which means that G′[V′\F′] is a forest.

⇒ F is a FVS for (G,k). Whether or not $w \in F$. $F' = F\backslash\{w\}$. Given a solution F of (G,k). G[V\F] is a forest. G[V\F]=G′[V′\F′] which means G′[V′\F′] is a forests, $|F'| \leq k$ be a solution of (G′,k).

□

Reduction 1 directly leads to Rule 1.

**Definition 3.** *Let (G,A,B,k) be the decision problem of finding a FVS-bipartition F of G, where* $|F| \leq k$.

**Rule 1.** *If there is a vertex w* $\in$*A with degree* $\leq$ *1 in G, then reduce (G,A,B,k) to (G[V*$\setminus\{w\}$*],A*$\setminus\{w\}$*,B,k).*

The following reduction on FVS-bipartition is similar to a reduction on FVS, where the graph has one fewer vertex after the reduction. However, next reduction does not alter the number of vertices.

**Reduction 2.** *if there is a vertex w with degree* $\leq$ *1 in both G[A*$\cup\{w\}$*] and G[B*$\cup\{w\}$*], of graph G, then (G,A*$\cup\{w\}$*,B*$\setminus\{w\}$*,k) is reducible to (G,A*$\setminus\{w\}$*,B*$\cup\{w\}$*,k).*

*Proof.* N(w)={u,v}, where u∈A and v∈B.

⇒ all cycles that involve w, also involves u and v.

⇒ Let F be a FVS-bipartition. Then also F′=(F\{w}∪{u}) is a FVS-bipartition. □

Reduction 2 leads to Rule 2.

**Rule 2.** *If there is a vertex w both with degree* $\leq$ *1 in G[A] and in G[B], then reduce (G,A,B,k) to (G,A*$\setminus\{w\}$*,B*$\cup\{w\}$*,k).*

**Observation 2.** *If w∈A and there is a cycle in G[*$\{w\}\cup$*B], implies* $w \in F$*, where F is FVS-bipartition of G.*

*Proof.* (Remember that G[B] is a forest, and that this property must be maintained.) By construction of the FVS-bipartition problem, F⊆A, and no other vertex of A is involved in the mentioned cycle. w being in F is only way to "cover" this cycle. □

**Rule 3.** *If there is a cycle in G[*$\{w\}\cup$*B], then reduce (G,A,B,k) to (G,A*$\setminus\{w\}$*,B,k-1).*

Algorithms 3 and 4, solves the optimization problem minimum FVS-bipartition (Problem 3.1) and the deciding problem FVS-bipartition (G,A,B,k) (Definition 3), respectively.

Though algorithm 3 and 4 solve different problems, they are exactly alike in how they use the rules, we have mentioned (rules 1, 2 and 3).

Note that on Line 6 of Algorithm 3; seeing as G[B] is a forest (by construction), all cycles are {w,w}-path's, {w,w}-path's in G[$\{w\} \cup B$].

**Algorithm 3** pedagogicalFVS-bipartition(G,A,B)

, where G is a bipartition on A and B

```
1  if  G is a forest  then
2     return ∅
3  end if
4  if  ∃_{w∈A} deg_{G[{w}∪B]}(w) ≥ 2 then
5     pick w ∈ A. s.t. deg_{G[{w}∪B]}(w) ≥ 2
6     if  a cycle in G[{w} ∪ B]  then
7        return {w}∪ pedagogicalFVS-bipartition((A∪B)\{w},A\{w},B)
8     else
9        return min( {w}∪ pedagogicalFVS-bipartition(G[(A∪B)\{w}],A\{w},B),
         pedagogicalFVS-bipartition(G,A\{w},B∪{w}))
10    end if
11 else
12    pick w ∈ A. s.t. deg_{G[A]}(w) ≤ 1
13    if  deg_G(w) ≤ 1  then
14       return pedagogicalFVS-bipartition(G[(A∪B)\{w}],A\{w},B)
15    else
16       return pedagogicalFVS-bipartition(G,A\{w},B∪{w})
17    end if
18 end if
```

**Algorithm 4** FVS-bipartition(G,A,B,k), where G is a bipartition on A and B, and integer k≥0. Let x∪NO=NO

```
1  if G is a forest then
2    return ∅
3  end if
4  if k ≤ 0 then
5    return NO
6  end if
7  if ∃_{w∈A} deg_G[{w}∪B](w) ≥ 2 then
8    pick w ∈ A. s.t. deg_G[{w}∪B](w) ≥ 2
9    if a cycle in G[{w} ∪ B] then
10     return {w}∪ FVS-bipartition(G[(A∪B)\{w}],A\{w},B,k-1)
11   else
12     t={w}∪ FVS-bipartition(G[(A∪B)\{w}],A\{w},B,k-1)
13     if t=NO then
14       return FVS-bipartition(G,A\{w},B∪{w},k)
15     else
16       return t
17     end if
18   end if
19 else
20   pick w ∈ A. s.t. deg_G[A](w) ≤ 1
21   if deg_G(w) ≤ 1 then
22     return FVS-bipartition(G[(A∪B)\{w}],A\{w},B,k)
23   else
24     return FVS-bipartition(G,A\{w},B∪{w}),k)
25   end if
26 end if
```

We will now show that Algorithm 3 produce the correct answer. As recursion is used, one only has to show that all inputs are handled and handled correctly, while assuming that procedure is inductively correct. It should also halt. That the number of possible paths, decrease, for every recurrence.

**Lemma 1.** *Algorithm 3 produce the correct answer to the FVS-bipartition Problem.*

*Proof.* Line 1-4 are trivial base cases. Line 4-11 branches on adding w to FVS-bipartition or not, where Line 7 applies Rule 3. Line 12 is always possible due to Observation 1. Line 14 is safe du to Rule 1, and finally Line 16 due to Rule 2. □

Algorithm 4 differs from Algorithm 3 in that it yield "NO" if there is no FVS-bipartition F, s.t. $|F| \leq k$.

Note that Algorithm 4 is formulated with an assumption in this text; that integer k≥0. If this assumption is met, then the algorithm will meet this requirement during the course of the algorithm. On Line 7 of the algorithm, k will be at least 1. The recurring on lines 10 and 12 meets the requirement, as 1-1≥0, and it is similar for the other recurrences.

You may have noticed that Definition 3 is a decition problem that can be used to answer Problem 3.1. This will be addressed later.

## 3.2 minimum FVS

Here, we show a simple way; how we can use what we have introduced so far, to solve our main problem, the minimum FVS. This algorithm, already is FPT, but that will be elaborated on, later. We use Algorithm 4, that solves the decision problem defined in Definition 3.

We will not reduce Problem 3, the minimum FVS problem, to decision problem defined in definition 3.

We solve Problem 3, using decision problem defined in Definition 3.

A minimum FVS F, is sought.

Let F′ any set of vertices. Let F be a/any minimum FVS, that we have yet to find.

We have (F\F′)⊆(V\F′).

We could try all possible I⊆F′ or equivalently all (F′\I)⊆F′ s.t. I⊆F′, used from now on. "Some I must satisfy (F′\I)⊆F."

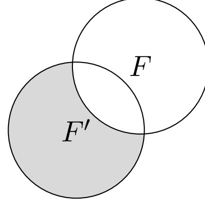



Figure 3.1: An illustration of how the **”compression“** is achieved.

**Definition 4.** *Hereditary - a property is hereditary for graph G, if the property holds for all G[X], where* $X \subseteq V$*.*

**Observation 3.** *Forest is hereditary.*

Written fully out; given a FVS $F$ of G=(V,E), $\underset{X \subseteq V}{\forall} G[X \setminus F]$ is a forest. A less formal formulation; ”you can not make something that is not a forest, into a forest, by adding vertices/edges“.

Per construction G[V\F] is a forest, and as I⊆(V\F), and with forest being an hereditary property, G[I] is a forest. Recapping; graph G[I] is a forest, and can by construction freely choose what the content of vertex set F′ is. We choice F′ to be a FVS of G, with the result that G[V\F′] will be a forest. With forest G[V\F′] and forest G[I], we have what is required of the FVS-bipartition's.

We can search G[V\(F′\I)] for a FVS Y, where Y⊆ V\F′. A FVS where we immediately have a decrease the upper bound on number of elements of |(F′\I)| elements.

$$\text{(G[V],k)} \Leftrightarrow \underset{\text{I} \subseteq \text{F}'}{\exists} \text{ where G[I] is forest, and (G[V\textbackslash(F}'\text{\textbackslash I)],V\textbackslash F}',\text{I,k-}|\text{(F}'\text{\textbackslash I)}|)$$

See Figure 3.1, where B is drawn in grey.

**An example on how finding minimum FVS could be proceed on a problem instance.** This example will find the minimum FVS of graph G=(V,E), the graph in Figure 3.2. Let $V_i = v_1, v_2, \ldots, v_i$, where $v_i$ is the element of the vertices in V, in the order they are considered. Will assume alphabetical order.

In this example, Algorithm 4 will not proceed beyond Line 5, until i=7, $V_7 = V$. Will not describe any steps of Algorithm 4 until than.

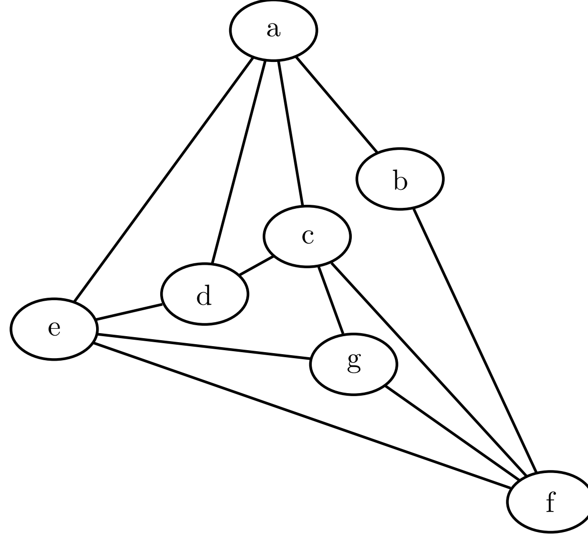



Figure 3.2: A graph to serve as an example of how the solving of minimum FVS proceed.

Start with $V_1$. We have that $V_1 = \{a\}$, and that F′={a} is a FVS of G[$V_1$]. There are two subsets of F′, ∅ and {a}, both G[∅] and G[{a}] are forests, hence {∅,{a}} is an enumeration of the induced forests. We iterating the enumerated forests, and use Algorithm 4, FVS-bipartition(G[{a}],{a},∅,-1)=NO, but FVS-bipartition(G[{a}],∅,{a},0)=∅. We have found that F=∅ is a minimum FVS of G[{a}].

Without self-loops and multi-/parallel-edges, there can not be any cycles in a graph, when vertex set consists of fewer then three vertices, so let's skip to $V_3$. We have that $V_3 = \{a, b, c\}$, and that F′={c} is a FVS of G[$V_3$]. As previous, we calculate FVS-bipartition(G[$V_3$],{a,b,c},∅,-1)=NO, and FVS-bipartition(G[$V_3$],{a,b},{c},0)=∅, We find out that that F=∅ is a minimum FVS of G[$V_3$].

We have that $V_4 = \{a, b, c, d\}$, and that F′={d} is a FVS of G[$V_4$]. Calculate FVS-bipartition(G[$V_4$],{a,b,c,d},∅,-1), and FVS-bipartition(G[$V_4$],{a,b,c},{d},0), with the answer NO in both instances. Since the answer was NO in all instances. Vertex set F={d} must be a minimum FVS of G[$V_4$].

Now $V_5 = \{a, b, c, d, e\}$, and F′={d,e} is a FVS of G[$V_5$]. There are four subsets of F′, and {∅,{d},{e},{d,e}} is an enumeration of the induced forests.

As previous, we calculate FVS-bipartition(G[$V_5$],{a,b,c,d,e},$\emptyset$,-1)=NO , FVS-bipartition(G[$V_5$],{a,b,c,e},{d},0)=NO, FVS-bipartition(G[$V_5$],{a,b,c,d},{e},0)=$\emptyset$, and FVS-bipartition(G[$V_5$],{a,b,c},{d,e},1)={a}s. Both {a} and {d} are minimum FVS of G[$V_4$]. Arbitrarily chooses one as we continue calculating the minimum FVS of G. May as well choose the first found minimum FVS of G[$V_5$], which was the vertex set {d}.

Next step $V_6 = \{a, b, c, d, e, f\}$, and F′={d,f} is a FVS of G[$V_6$]. Algorithm 4 returns NO for every induced forest. FVS {d,f} is a minimum FVS of G[$V_6$].

Eventually we get to i=7, $V_7 = V$. F′={d,f,g} is a FVS of G[$V_7$]. There are eight subsets of F′, and all of them are in the induced forest enumeration. For all but {d,f,g}, FVS-bipartition returns NO. FVS-bipartition(G[$V_5$],{a,b,c},{d,f,g},2)={c,e}. In Algorithm 4; Line 8 picks vertex c followed by Line 10, and likewise for vertex e. FVS {c,e} is a minimum FVS of G[$V_6$].

## 3.3 A FPT algorithm

So far. Algorithm of Figure 4 has been shown to solve the FVS-bipartition (G,A,B,k), and how one could use it. We now show that the algorithm of preceding section, actually is a FPT algorithm. For information on iterative compression (I.C.), you may look at [52].

Let's assume |F′| $\leq k+1$, where k is the cardinality of any/all minimum FVS. (Does not have to be one, picking a larger integer work also.) We iterate all the induced forests G[I] of G[F′]. (May skip I=$\emptyset$, as we already have that F′ is a FVS.)

$2^{k+1}$ forests, and for each forest I, (G,A,B,k+$|I|$-$|F'|$)=(G,A,B,k-$|(F' \backslash I)|$) is calculated.

**Lemma 2.** *Algorithm of Figure 4 is FPT.*

*Proof.* Let l be the number of connected components in B, have $l \leq |B| = |I| \leq k+1$, and have k as usual. Looking at Algorithm 4, one sees that when branching on one path; l decreases and the other path; k decreases. Neither $l$ nor $k$ decreases below zero. $k \leq 0$ is explicitly tested for ($k \leq 0$ and G not forest is a rejecting state). $l = 1$ and $deg_{\mathrm{G}[\{w\} \cup B]}(w) > 1$, together implies that G[$\{w\} \cup B$] is not a forest, hence no branching.

$l$ never increases, Line 22 ensures that Line 24 only occurs when $deg_{\mathrm{G}[\{w\} \cup B]}(w) = 1$. Thus the total number of operations that must be done, is not greater than $2^{2k}$, as k+l$\leq$ 2k. □

For any graph, the number of connected components $\leq$ the number of vertices.

# Chapter 4

# Experiments

In this project we decided to make implementations that do whole problem fast opposed to timing each procedure and focusing on the procedure that use most time. The times that are presented from running on a machine, would likely change if ran on a different machine. Running all on same machine, gives an idea about the performance of the experiments.

This including the cost of reading graphs measuring the time taken to solve problem instances.

Favouring doing small and simple steps. It is more difficult to get an idea of what changes caused an effect, if many changes has been applied. If none of these experiments perform better than others, that would be interesting too.

Although it is natural to do kernelization, as a preprocessing before solving the problem, kernelization was not in any experiment. It could be a nice experiment that might say something about how well the algorithm that follows perform, however it is just a reduction or reductions just as the ones used in algorithm seen in Chapter 3. None of the experiments try to place the data more tightly in memory either, which could increase the likelihood, that such a kernelization is useful, however the motivation behind kernels is usually to bound the worst running time, which it wont for the algorithm seen in Chapter 3. We went the route of trying to make the implementation of the algorithm faster, prioritizing experiments that was believed to give improvements. Did not prioritize experiments, based on the possibility that they would give information about how well the implementation performed.

## 4.1 Random graph set

To make a meaningful comparison of experiments, it became important to make a set of problem instances to efficiently compare implementations, a random graphs set. The random graph set was not constructed before a first implementation was made, as there was no previous known implementation on this problem. The implementation that generated the random graph set, also took considerable time computing on the machine. For convenience this chapter is laid out as if the random graph set was always available.

**Random graph** creation was implemented using BOOST Graph Library, which is a c++ library. Storing it in graphviz file format. Note that the algorithm was made with little performance in mind. We choice to limit the graph benchmarked to connected, because unconnected graphs are one of several graph types that can be solved quicker.

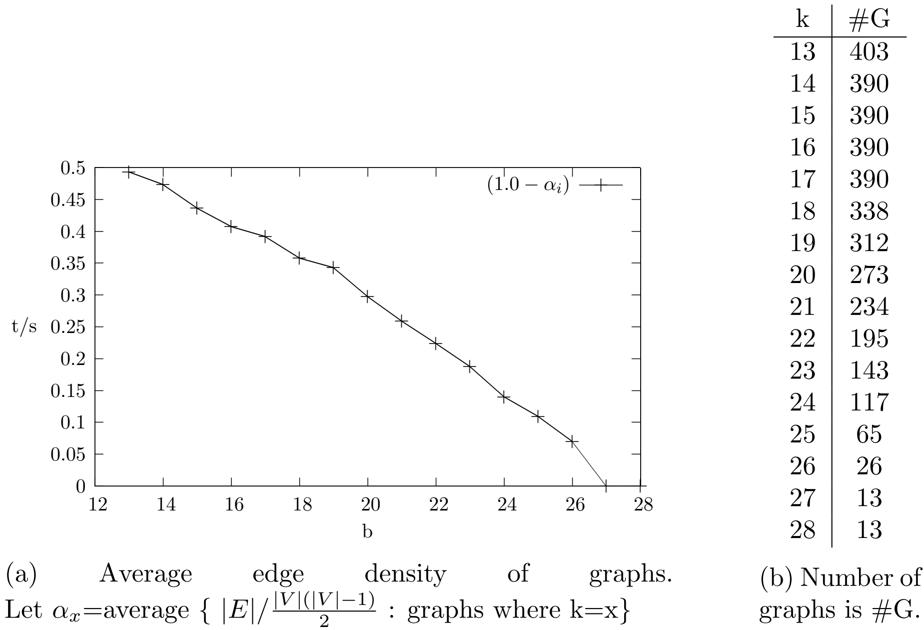
**Algorithm 5** random-connected-graph-creator(n,m) , returns a graph, with n vertices and m edges (E not a multi-set, and no {u,u} edge)

```
1 while TRUE do
2   make a graph with n vertices and no edges.
3   for for every pair {u,v} do
4     do E ← E ⋃{u, v}, with probability m/(n(n−1)/2)
5   end for
6   if m=|E| and graph is connected then
7     return the graph.
8   end if
9 end while
```

* $\frac{(n-1)((n-1)-1)}{2} < |E| \Rightarrow$ connected.
* $|E| < n - 1 \Rightarrow$ not connected.
* $\frac{n-1 \leq |E| \leq (n-1)((n-1)-1)}{2}$ varying chance of being connected between the extremes.

Algorithm 5 is used to make a random graph with exactly n vertices and m edges. Example of an issue; connectivity is not uniform over the m's. This is alleviated by the m=|E|.

There are many graph attributes that are not uniform. And also many attribute dependent on the application that uses the minimum FVS problem.




(a) Average edge density of graphs. Let $\alpha_x$=average { $|E|/\frac{|V|(|V|-1)}{2}$ : graphs where k=x}

| k | #G |
|---|---|
| 13 | 403 |
| 14 | 390 |
| 15 | 390 |
| 16 | 390 |
| 17 | 390 |
| 18 | 338 |
| 19 | 312 |
| 20 | 273 |
| 21 | 234 |
| 22 | 195 |
| 23 | 143 |
| 24 | 117 |
| 25 | 65 |
| 26 | 26 |
| 27 | 13 |
| 28 | 13 |

(b) Number of graphs is #G.

Figure 4.1: Data on the generated random graph set. Where k is $|F|$.

For simplicity, more graph types will not be looked at, in this text? There are likely examples/classes that would be more valuable, however in some cases, there might be problems with regards to whether it is practical and/or feasibility to implement such a study. An example of graphs it could be interesting conduct a study, is expander graphs.

**Graph set.** Let mFVS be any minimum FVS. If for a fixed f=|mFVS| there exist a graph with n=|V|, there exist such a graph with |E|=min$\{\frac{n(n-1)}{2}, \frac{f(f-1)}{2} + n - f\}$. Assuming all graphs are connected, a lower bound on |E| is $n - 1 + f \leq |E|$. In following Algorithm 5, we also assume that if there exist graph for |E|=a and for |E|=b, then there exist graphs for any j, where $a < j < b$.

The random graph set was generated using Algorithm 6. Before I ran graph-set-maker(X,30,13,13), I ran a similar program for a night to make a initial X that was not empty. It ran for a some weeks and program was restarted a several times.

Some information regarding the constructed random graph set can be seen in Figure 4.1.

**Algorithm 6** graph-set-maker(X,n,c,r), where $c \leq |mFVS|$, r occurrences of each $(|V|, |E|, |mFVS|,_{)}$ present

```
1  while a new graph was added to graph set X do
2    for i,j,k do
3      t ← (#graphs in X with i vertices, j edges, and |mFVS|=k)
4      while r < t do
5        discard oldest (i, j, k, _)
6        t ← t − 1
7      end while
8    end for
9    for i = n, (n-1), (n-2), ... do
10     s ← i(i−1)/2
11     for j = s, (s-n), (s-2n), ... do
12       for k = (n-2), (n-3), (n-4), ... do
13         t ← (#graphs in X with i vertices, j edges, and |mFVS|=k)
14         if (0 < t) ∨ ( min_l{(i, l, k, _) occuring} < j < k(k−1)/2 + k(n −
           k) + (n − k) − 1 ) then
15           while t < r do
16             G ← random-connected-graph-creator(i,j)
17             f ← |mFVS(G)|
18             if c ≤ f then
19               X = X ⋃(i, j, f, G).
20               if f = k then
21                 t ← t + 1
22               end if
23             end if
24           end while
25         end if
26       end for
27     end for
28   end for
29 end while
30 discard all (i, j, k, _) with fewer than r occurrences.
```

## 4.2 A first implementation

We choose to represent graphs using an adjacency matrix data structure. Adjacency matrix was chosen as it is simple , and that the graphs will be small.

The time consumed by the part of the implementation not known to be polynomial is liable to out-way the time used by any polynomial time process added.

In this experiment, enumerating induced forests, was done by iterating all subsets of the vertices, and the forest test (Algorithm 2) was applied after inducing. Iterating subsets was done doing counting, and the vertices of the graph where ordered to make the counting as straightforward as possible. The induced forest enumeration was made, so that $\{v_i\}$, came first in the enumeration, so that in cases such as $deg_{G[V_i]}(v_i) \leq 1$, an improvement would be immediately found, only involving one FVS-bipartition(G[$V_i$],$V_i \backslash v_i$,$\{v_i\}$,0). Methods to ease the process of swapping position of two vertices in the representation of a graph, and sorting vertices, where implemented, to make the ordering. Although it is nice to view it as sorting, it is not common that one can sort in linear time, even stable sort in linear time. This implementation ultimately, in regards to swapping, involves more then changing two values, when applied to graph representation.

Algorithm of Figure 4 involved some vertex set's, one was referred to as B. The union find data structure used in the forest test was also used in the representing of B. Using union-find without rank or any form of path compression on B.

There are several option for representing the union-find, data structure. C++ has many types that can be used. C++ also comes with a several containers part of it's library. Plain vector as commonly used for the union-find structure, may be the simplest. Went with a container based on balanced binary tree. Maintaining the rooted tree does require more memory, but can be implemented with memory linear in k, opposed to using vector where memory usage is linear in n. In all experiments, balanced tree containers is used, and other options are talked about.

Implementation of balanced tree containers are fairly complicated compare to, let's say, implementing union-find. Balanced tree containers are commonly seen in standard libraries of programming languages. (e.g. c++'s std). Some libraries go out of their way at making misuse impossible, and to get the functionality sought, functionality that can not be gained by practical means writing the programming language, the option one is left with,

| intent | permutation |
| --- | --- |
| where they are | $\sigma$ |
| where they should be | $\sigma^{-1}$ |

Figure 4.2

is to look for other libraries, or do the implementation ones self. Sometimes one is lucky, and can take what one is looking for, from old work or exercises. The balanced tree container used, was taken from old exercise. In [2] there is a nice figure on deletion in red-black trees.

It is tempting to try and somehow try and continue building on the already available union-find on G[B], when doing forest test on G[A∪B]. Have not tried to do so. No attempt in this text has been done towards adding support for undo to union-find.

Determining whether there is a cycle in G$[\{w\} \cup B]$, which is an essential part of Algorithm 4, was crudely implemented, using lookup in the union-find data structure stored in balanced tree container. It involves determining if any vertex in B, adjacent to w, is in the same, connected component (or same set). As there should be limited number of vertices in B, this was crudely implemented using insert sort.

**vertex order** If you have some ordering you want (i.e. using common sort), you have $\sigma^{-1}$, you compute $\sigma$, and order so that $\sigma = identity = \sigma^{-1}$.

You have $\sigma$ and wish to restore to labelling where $\sigma = identity$. You first compute $\sigma^{-1}$. And order so that $\sigma = identity = \sigma^{-1}$., just like previously.

Can have labels on vertices, so that however many times the order of vertices has been changed. The order can be reverted to original ordering in linear time. It may also be faster to only sort labels, and not a whole matrix.

I choice to order vertices such that the ones in current FVS was first. Iterating all subsets is then simple.

First implementation did not do this "sorting" stable. This was later made stable by using two different kinds of labels as I wanted stable to be the default. Stable/unstable has no impact on whether implementation correctly yields the minimum FVS, it could possibly have an impact on performance.

Thought instantiate a new graph for every reduction made, should have little effect on memory usage, it was implemented with everything allocated once. Memory allocation also cost time.

Using $\sum_{i=0}^{n} i \cdot \binom{n}{i} = \frac{n}{2}$. We get that the average number of vertices in B is $\leq \frac{(k+1)}{2}$,where $F'$=k+1, and for some cases it is equal.

It was observed from running the application, that implementation claims barely any memory, less then 1Mb. This could be the fact that the problem instances are small. The way it was implemented implementation, memory used, should be linear in the input size.

The running of this implementation on the random graph set was aborted after 12 hours. The graphs where k∈{24,25,26,27}, remained to be computed.

## 4.3 The number of induced forests

They are both the same recursion, however there may be some difference with regard to parallelization. Two ways for doing the enumerating of induced forest will be considered in Section 4.4.

This enumeration of induced forest proceeds as follows. Initially all vertices of F′ are removed. The recursion works as follows. For $v_i \in V$, remove i and vertices $v_1, v_2, \ldots, v_i$ from V, put $i \in S$, if G[S] is a induced forest, recur. If G[S] is not a forest we backtrack.

The result from Section 4.3, there are 4 comparisons missing at the end. the implementation from the first experiment was not concluded. There is no reason for the missing data to proceed any different. Figure 4.3 also shows the time the implementation of this section used.

Looking at the figure, the implementation of this section actually lookes as if it takes less time as k get's large. This may be more effected by how the graph set was constructed. See Algorithm 6 and Figure 4.1a.

Experimenting with random graphs gave the impression that graphs with some, but few edges, where the ones taking most time to solve.

It is imaginable that the number of induced forest's of the FVS $F'_i$ in the algorithm effects the running time greatly, and while more edges often results in large $F'_i$. With fewer edges in a graph, G[F′] often has fewer edges and it is the presence of edges that limits the number of induced forests beyond the trivially $2^n$, where n is the number of vertices.

This motivates looking for implementations that perform better on graphs with the number edges closer to zero than $\frac{n \cdot (n-1)}{2}$. Many interesting graphs have few edges.

Unlike the first implementation, with the use of the enumeration of induced forest, the random graph set was computer and it took 90 minutes
.

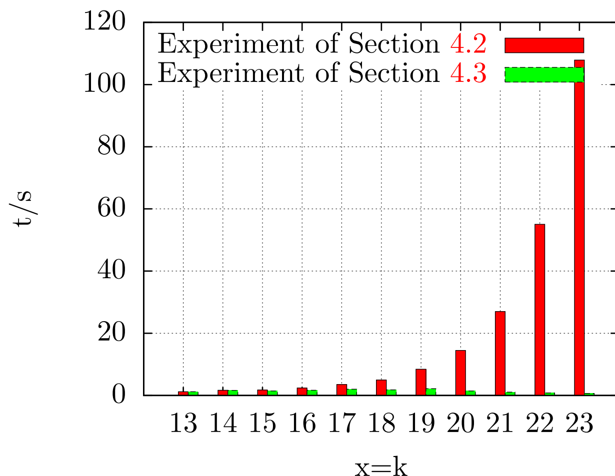



Figure 4.3: Shows a comparison between implementation of experiment of Section 1 and the experiment of this section. The time t, is average time to solve a graph in the set.

## 4.4 parallelization

Parallelization was done on the enumeration of induced forests, a question is how to divide the induced forests. Just as the number of induced forests are not known in advance, the number of tasks are not known in advance, and the benefits of knowing the number of task is limited, as the time it takes to complete a task, can vary. Every task involves computing feedback, and while feedback is FPT, k may in it's self vary among tasks.

The algorithm we are implementing does in it's self, not involve dividing a problem. Every processé could have it's own instantiated graph or it's own vertex set that it induces on a shared graph.

A threshold was chosen, for when there where enough work to warren parallelization. Which was used, through the experiments. The time saved likely vanishes compared to other work, as problem instances become more difficult.

**Two options for parallelization was considered.** Subsection 4.4 Parallelizing with a manager that distributes induced forest, each induced forest being a task. Subsection 4.4 Parallelization dividing the induced forest, where tasks can involve solving several induced forests.

Parallelization was invoked in induced forest enumeration. It would most

likely be better performance wise, to invoking it in the iterative compression (I.C). Invoking it up to n times, likely comes with some cost. Though it vanishes in comparison to how running time is effected by k.

In the implementation there is one copy of graph G, F′ and labelling per thread. However it could be implemented only duplicating G and the labelling. The labelling that maps F′ to G.

**Option a** involves a manager iterating the enumeration witch needs to be communicated with procesés. One could have one procesés designated manager and other procesés designated workers, however it is also an option to have every procesé take the role of manager to get a new task. Later would perhaps be better, as the role of manager, would not involve knowing noticeably more than workers.

This option involves communicating a vertex set representing a induced forest, and there is a cost to communicating induced forests.

**In option b** every procesé having knowledge about how the tasks are divided. The only communication needed with regard to work sharing, is one number, the number of tasks that has been solved or are being solved. One issue however is load balancing. The work involved in solving different tasks, even if tasks only involves one induced forest, may vary, and can as mentioned vary considerably. The main issue is when there are no more tasks for procesés to acquire, and procesés are idly, until all procesés has finished their jobs. One could try and estimate the time tasks take, however to make things simpler, changing the size of tasks where in stead considered. In stead of making more and smaller task, having task become smaller and smaller was sought. We look at the enumeration of induced forest, that has been used since being introduced in experiment of Section 4.3. For instance if the graph we enumerate induced forest from is a forest, and we iterate all $2^n$ induced forests, the recursion includes one vertex every time until all n vertices are in the current induced forest and we are at depth n of the recursion. This depth is never reached again later. This implementation This is not the case for all graphs, but this observation may be a basis for a decent implementation of parallelization, and this was the route we went with; to dividing induced forests into tasks. It may in many instances achieve a decrease in the work task take to complete, much like the scheduling in OpenMP called GUIDED, i.e. trying to achieve an attenuation in the work needed for task to be solved. This setting requires that every procesé has its own graph. The graph that where considered, that looked to be practical

to solve where however small, so this was no issue. This option does involve some tasks, not being solved independently on the procesés. However the number of task that are not independent is bounded by the size of $|F'|$, and the precise depth, this task division is done at.

There is a cost to starting paralelization. The paralelization is in this implementation invoked just prior to the enumeration of induced forests, and a treshold is used so that paralelization only occured when $|F'|$ was of a certain magnitude. The threshold was $4 + ln(p)/ln(2)$. In this option, this subsection, the threshold was used, to determine which depth dividing of tasks occured, and threshold without the added $ln(p)/ln(2)$ for whether or not to do paralization.

There exist algorithm, on which benchmarking a parallelized implementation, the best time out of all runs is a very good measurement. Like for example matrix multiplication on approximated real numbers. This might however not work so well on the minimum FVS algorithm. There may be non-determinism between processies in a machine. As we are seeking an optimum solution to minimum FVS. It is natural to be content with the first solution available. Recall that the minimum FVS problem is also used as an intermediate step in the algorithm of Chapter 3. This can lead to quite unpredictable running time, when solving minimum FVS, with this algorithm, using concurrency, on a problem instance consisting of few graphs. The times you see in this paper are based single runs on the random graphs set, which consist of some graphs. All the 13 graphs where $|F|$=28 are actually the exact same graph.

For results see Chapter 4.1.

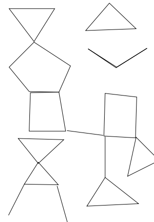
## 4.5 Connectivities

It could be worth while trying to divide the problem instance that one is working with into smaller problem instances. The time it takes to solving the smaller problems and combining the results, may be less than the time it would take, doing the solving with the whole problem on the table. We mentioned that forest is a hereditary property. This experiment exploits that forest is a hereditary property. Connectivities other than 2-edge connected where not realized in the implementation, though there are other options mentioned in this section.

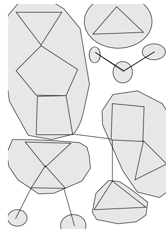
Though as the random graphs where chosen to all be connected, they might not be 2-edge connected, and even if they where, there could still be graphs that are not 2-edge connected, on witch a FVS is sought, as I.C. is

| Component | Implemented data structure |
|---|---|
| 1-edge connected, connected | isolated vertices |
| 2-edge connected, bridge connected | forest |
| 2-vertex connected, 2-connected, biconnected | forest, with additional articulation point 's |

Figure 4.4: A list of how graphs can be divided into components, and a bit about how they might be implemented.

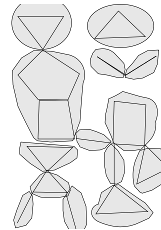


(a) A graph.



(b) The 2-edge connected components of a graph.

.



(c) The 2-vertex connected components of a graph.

.

Figure 4.5: Some examples of components of a graph.

performed. Could even be graphs that are not connected.

Finding 2-edge connected component containing a given vertex w can be solved by depth first search (dfs). Starting at vertex w. All vertices in connected components containing w, is visited by the dfs. The implementation that finds 2-edge connected component, was implemented in such a way that it found component containing vertex w, as the last component found.

In dfs, vertices are commonly either; marked of as seen, or given (usually unique) labels. When labels are in use, not seen is a reserved label that all vertices get's assigned at initialization. 2-edge connected, can simply be found by a dfs algorithm that labels vertices as they are seen and every vertex also has a lowest_label, that is updated as dfs backtracks.

The dfs does not need to conclude before finding all component, this is up to ones wishes/needs, it is a very small difference programming wise. The dfs could simply conclude after searching the (1-edge) connected component containing w.

The dfs was implemented so that label, label represented using a vector , and used in the dfs algorithm , used as is common in dfs algorithms, ends up labelling the 2-edge connected components.

It does dfs twice. First to determine lowest_label, also a vector. Second time to label vertices such that all vertices of a 2-edge connected component has same label.

The algorithm is very simple. The dfs maintaining 3 working variables, as in 3 vertices of consecutive depth. In this regard it resembles how one could make a naive forest test using dfs, making sure that u=w, u first and w last vertex of the 3 v, is not considered a cycle. The reason for doing so when extracting 2-edge connected component, inherently being the same. Though u=w only means lowest_label of vertex between u and w in depth, namely lowest_label[v], will not be updated on dfs backtrack.

There are many data structures that involve connectivity, though many seem to be motivated by a need for fast query, of information, in regard to two given vertices. (e.g. are two given vertices connected, or cardinality of minimum cut between two vertices). Many mention components, but it may be more productive to focus on getting the part of the implementation not known to be polynomial, faster, before considering using more involved data structures.

With finding 2-edge connected part of implementation, solving the random graph set took 87 minutes, saving 3 minutes.

## 4.6 Adjacency list

Will now focus on adjacency list data structure. Reason for not doing more experimentation with adjacency matrix is that the focus is on improving the performance for sparser graphs. Experiments show that performance for dense graphs is not an issue, although they tend to have larger FVS (see Section 4.3). Graphs seem to be quite a few edges before a decrease in running time is seen. The use of the adjacency list was considered. Adjacency lists are a more efficient way of representing graphs with fewer edges, and could be a better choice for graphs with fewer edges. Though it may mostly be an exercise in programming.

Two version using adjacency list for describing graphs where used. Sub-

section 4.6 Adjacency list, each processor has a vertex set. Subsection 4.6 Adjacency list, each processor has a vertex set graph.

The specific differences are described later. First what is common.

There are several ways of implementing linked lists. In the implemention, the list data structure is circular. To prevent iterating elements of linked list repeatedly, there is a node that indicate end of list, and end nodes are recognized by the data stored, somehow.

While one can view sets as an unordered collection of elements, it is practical to enforce an ordering when realising an implementation.

The adjacency list data structure has a list of adjacency in each vertex, and end of list indicated by node not having valid adjacency information. Adjacency list is representation was even further restriction, by enforced that the list of adjacencies was ordered, ordered as the vertices are ordered. Again G=(V,E). And with the representation that we talk about, working with G, is working with the G induced by the first i elements of V ordered, where i=$|V|$. Inducing is just as simple with $0 \leq i \leq n$, though it is not as general, as inducing on any subset of V, it is just the first i elements of V ordered. This is still useful when inducing graph G[X], and the elements in X, are not consecutive in the representation of V, where G=(V,E).

The point is that having an ordered of list vertices I, makes it easier to determine degree of vertices in G[I], and this is used in the Algorithm 4 with A, B and $A \cup B$, in place of I. The degree could however be a maintained integers in vertices. 3 integers per vertices. Even A, B and $A \cup B$ each having it's own graph data structure, should do just as well, with some labelling to map between the data structures. That B is of length at most 2k, already makes it a little less of a priority to make faster implementation of $deg_G[B](v)$, but the choices seem to be maintaining an integer or maintaining an ordered representation of B. We already have B consecutive in ordering with the container, mentioned in experiment 1. The fact that algorithm is only concerned with small values in regard to degree, should makes the version of Subsection 4.6 implementation perform comparably to an implementation where these values where maintained, as by how vertices in vertex set A are aligned in the implementation of Subsection 4.6. There are also similar regards concerning forest test with the version of Subsection 4.6.

It is straight forward turning adjacency matrix into adjacency list with nodes ordered likewise.

The enumerating of induced forest of G[F′], which was the main focus of experiment of Section 4.3, is quite easy when the vertices are placed consecutive, already seen in experiment of Section 4.2. Continuing in this fashion, a

duplicate graph G[F′] was instantiated. Not altering this instantiated graph, just using it for lookup. With regard to forest testing in induced forest enumeration, you may want to use a vector in stead of balanced tree container, with the G[F′], available for easy lookup.

With this duplication of G[F′], this experiment is the first with duplication of data. If duplication is worth while somewhere, it probably would be on this, as enumerating induced forests, at least not how it is done in all the experiments in this thesis, is a quite expensive job. For instance, if G[F′] is a forest, we are enumerating $2^{|F'|}$ set's.

Instantiating G[F′] also makes the implementation for adjacency list data structure very similar to the adjacency matrix data structure. The instantiation might even makes the forest testing faster.

In implementation, a new instance of the adjacency list data structure is made for every step of the I.C., not just G[F′].

With regards to parallelizing this implementation. The allocating of large continuous blocks, may also reduce cache contention among threads. Cache misses, perhaps even cache cohesion, can be "measures in running time without mentionable cost. The precise cause of cache cohesion may however be more difficult. ICC, supply a memory allocator that looks to be made for the problem of cache cohesion, with regard to concurrency using OpenMP. This was not experimented with. With an easy to use tool for giving an idea of the occurrence of false-sharing, it may have been prioritized. The instantiated G[F′] could be is shared among threads as it is not altered. However, any data that get's altered needs to be owned by procesé or invoke synchronization between procesés.

**Replacing adjacency matrix with adjacency list** The performance seen here is perhaps not of much interest, beyond comparing adjacency matrix and adjacency list.

With finding adjacency list representing the graph, solving the random graph set took 284.272 seconds.

Using 1/19 the time of previous implementation. Though less significant than last improvement, enough so, that it may be difficult to ascribe it to hardware issues alone. Hardware features such as pipeline in cpu. Adjacency list has the advantage, that looking through adjacencies of a vertex does not involve skipping zeros as in the case of adjacency matrix. That a duplicate of graph G[F′] was instantiated, is a possible cause.

It could be due to the ordering of representation of vertices being alteres as decribes in Section 4.2.

There are tools that gathers statistics on breanch prediction, this was not explored.

**A more efficient implementation on adjacency list** In a sequential implementation, or a concurrent implementation, where each procesé has it's own local graph instantiated, such as Subsection 4.4, the more efficient implementation that is being presented here, should outperform the other adjacency list implementation, though this outperforming is not of the same degree seen in Section 4.3.

Observe that; if the set A was the first a elements of V, then determining $deg_{G[A]}(v)$, would not involve accessing nodes that does not contribute to the magnitude of $deg_{G[A]}(v)$,. If there was a way of swapping vertices, than whenever removing a vertex from A, you could swap it to the end of A, prior to removing it, leaving a continuous representation of A.

Experiment of Subsection 4.6 was pretty much doing the first experiment all over again, difference being the data structure used to represent the graph. The experiment now being presented is the only experiment that reorders the vertices in a part of implementation not known to be polynomial.

Will in this implementation implement vertex swapping, implemented much like it was on adjacency matrix, but the swapping will be more extensively used. When a vertex v is moved, swapped with a vertex u, we need to update pointer in nodes that pointing back at v, and determining where a hypothetical node pointing back to v in the list of adjacencies of vertex u. As the list of adjacencies in vertices is to be enforces, more instances of these swappings, than previously seen, may be necessary.

The time it takes to swap two vertices, should be linear in the distance between the vertices in the ordering. As vertex w∈A is unlinked, w is swapped to end of A, and afterwards "adjacency list nodes" pointing back at w are unlinked. Number of swappings, of content, of "adjacency list nodes" during this swapping of vertices is not greater then $|A|$ . And $deg_{G[A](v)}$ is as simple as determining number of elements of the linked list of adjacent vertices of v.

With this choices, A and B, can each be represented by an integer, by swapping vertices around. Doing so for B might be more hassle, and will not be considered.

Some additional operation are used in this implementation, that are referred to as linking and unlinking. These operation are used to remove vertices from graph, in Algorithm 4 and the enueration of induced forests.

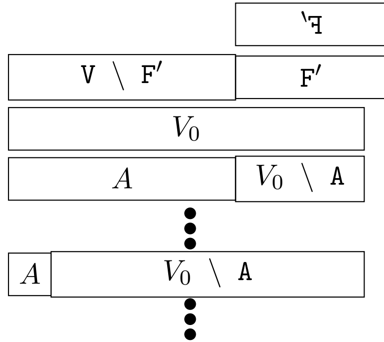



Figure 4.6: An illustration of the more efficient implementation on adjacency list.

The removing of nodes from the list of adjacencies in vertices, and reinserting every node where it was when it was removed from list, is straightfarward when keeping a hand on the removed nodes, keeping only an additional memory address can be enough.

The following is more to give an idea of how data structure was laid out. As to limit the number of allocation of memory, one can count the number of edges/ones and allocate the space needed by "adjacency list's", all at once. Allocating all those nodes in one go, was done in implementation. The data stored in each node u comprise of reference to adjacent vertex v and to the node which represent reference back again from vertex v to vertex u, that u is adjacent to v as well. When an adjacency list is instantiated, the pointer between nodes that together represent an edge, are pointing to neighbouring node in the continuous sequence of allocated node. This is usually not the case for long however. Changing the order of vertices leads to change in the content of nodes in the linked lists. Even if order of vertices return to the initial ordering, this data structure may internally still not return to initial state. As induced forest where enumerated, an extra effort is also made to maintain that B is consecutive. In the implementation, only swapping was done in the forest enumeration, as to maintain that B is consecutive. B start of as consecutive before applying Algorithm 4, as vertices are unlinked from $A \cup B$ and vertices change from being in A to being in B, B is no longer consecutive. Note also that before applying Algorithm 4, any vertices between A and B, also has to be unlinked.

In implementation, every swapping and unlinking is eventually reverted, reverting everything to their initial state, less the actual position in memory.

Figure 4.6 illustrates the presented implementation. The implementation

is perhaps best illustrated by the vertex set's that are ordered. We have the instantiated graph G[F′], the graph used in enumerating induced forests, the graph used in Algorithm 4, and also how this graph is altered in the algorithm. The vertex set B. is not shown in the figure. B$\in V_0 \setminus$ A.

There is also the question of whether the forest test is same as before. In the implementation it is basically testing whether G$[A \cup B]$ is a forest. Applying A followed by B, or B followed by A to forest test gives same return. In implementation, B followed by A was used. If you apply A first, the forest test wont give an answer before you start applying B, and vice versa, unless the number of edges is not less than the number of vertices.

In this implementation, there is less use for an ordered representation of vertex set B, during Algorithm 4, Vertex set B can be represented by a single linked list:

Either way, the way implementation has been made, there is the duplication of B, as previously mentioned. Duplication on linked list, is easier to implement also.

With balanced tree container, it is more affordable to make an exact duplication of the union-find data-structure contained in B, but there is no reason for it to be the same. There should in favour of using plain linked list, be an advantage with regard to determining whether there is a cycle in G$[\{w\} \cup B]$. As mentioned in Section 4.2, testing G$[\{w\} \cup B]$ for a cycle, was implemented quite crudely, and implementation has never been touched.

The realised implementation, does however not realise that balanced tree container is no longer needed, and can be replaced by a simple list, as mentioned. Enhancing implementation on adjacency list solved the random graph set in 930.377 seconds.

## 4.7 Making a better environment

Main motivation of doing this experiment, is to pick a decent setting to be used for results in Chapter 5.

As previous experiments where conducted, there where some implementation choices, that where difficult to decide on, in part, due to that they could interact with other choices.

Not all choices imaginable was tried, as to limit the number of combination, the number of implementation to test. The adjacency matrix representation was all together not considered.

Did for instance not try with and without using rank in forest test, rank was always used. Like forest test improved by maintaining the number

of edges during execution. There where quite a few enhancements whose usefulness where experimented on.

Some things tested was, when enumeration induced forests, whether to call feedback when doing down the recursion, or while going up, and whether or not to do branching in the order seen in the presentation of the algorithm (see Figure 4).

Only implementations that finds 2-edge connected components where tried. As seen in experiment of Subsection 4.5, there where little extra cost of doing so. There where no implementation choices seen as the determining 2-edge connected components was implemented, however the ordering of vertices used by I.C. effect the determined of components, and likely also other things. During this experiments, some simple choices where made up in hopes of a better performing implementation, in addition ones found while implementing other experiments. Choices altering the order of vertices. A new ordering of vertices where made, by greedily picking next vertex of ordering until all vertices where picked. For every possible next vertex w, ever vertex w not already in the ordering, a value was computed, that one could make a greedy decision based on. Which value computed is the interesting question. As said, only some simple ones where considered. Let C′ be the 2-edge connected component in $G_i$ containing w, let F′ be the FVS that would be used in I.C. to compute a minimum FVS of C′. The computed value tried, where some in which could be computed prior to doing I.C. Reversing the order of those where also tried.

The choices for settings allowed for roughly 3525 combinations.

Having a single file for settings, that describes the machine code file that runs on machine.

Conducted on the graphs in the set of random graphs for which k=13. Choosing graphs that algorithm computes faster, such as graphs with low, may be issue as the cost of picking ordering of many of the orderings mentioned are somewhat costly with regard to the number of vertices and edges. k=13 should however be enough as the graphs in the set are also very small. 403 graphs.

There where more graphs in the random graph set for k=13, than for any of the other k's.

The implementation was so that if dependencies in settings are not met, a BOOST_STATIC_ASSERT_MSG should return an explanation as c++ code fail to compile. Some techniques that the language c++ allow, was used. The compiler eagerly instantiates code, however templates defer instantiation, so deference can be achieved, by simply adding a dummy template. Code that is not instantiated, will still produce errors if the operations that

1. $|C'|$, the number of vertices in component.
2. $|C' \backslash F'|$, the number of vertices in component minus FVS.
3. $|C' \cap F'|$, the number of vertices in component that are part of FVS.
4. $deg_{G[C' \cap F']}(w)$, the number of vertices in component that are part of FVS adjacent to w.
5. $|\{$components in G$[V(C \backslash \{w\})]\}|$, the number of components in the whole graph G after deleting vertices C from G.
6. $|\{$components in G$[V \backslash C']\}|$, the number of components in the whole graph G after deleting vertices C from G.
7. $|\{$components in G$[V \backslash \{w\}]\}|$, the number of components in the whole graph G after deleting vertices C from G.

Figure 4.7: Some quantities, where G=(V,E) is a graph, F, C and C′ are vertex set's, and w is a vertex.

are performed are not declared, and an issue with just declaring the operations, is that you do net get much information, with regards to the compile error that occurs, when needed operations are nothing but declared. One can make the assert BOOST_STATIC_ASSERT_MSG only fail on the instantiation of code. Assert are not evaluate before instantiation of code, when the assert is dependent on template value. One can for instance make an assert that fails if some dummy template equals zero and another assert that fails when dummy is unequal to zero. C++ also has a preprocessor, which allows for simple operations such as concatenation. One operation is including files, which makes it easier to reuse code, or to to uses libraries. Many languages have some kind of "include" operations. When including files in C++, one does not have to precisely defined what is included. One does not need to precisely specify filename. There are also little limitations where you can put an include.

A bash script was written, that tries all setting and when c++ code compiles, and preceded to benchmark the assembled machine code. Keep track of the best running time and time-out if exceeding twice of best running time.

It seems that the performance of the implementation presented in Sub-

- Altering graph representation during execution as described in experiment of Subsection 4.6.
- Only altering the graph representation when Observation 2 is applied or doing Reduction 2.
- When doing feedback, iterating the vertices of A, right towards left.
- Sorting the vertices in A with decreasing vertex degree, right towards left.
- Greedily choosing the next vertex of I.C., favouring greater $|C' \backslash F'|$ (see 2 in Figure ).

Figure 4.8: The setting chosen for later uses of implementation.

section 4.6, can be poor if not done right, and that the simpler implementation in Subsection 4.6, that does not altering graph representation under calculation of Algorithm 4, may be a safe implementation choice.. However in spite this, the list of setting seen in Figure 4.8 will be used as the default setting from now on.

Only 173 settings out of the 3525; time-out-ed. All of which had the first option seen in the list of Figure 4.8, an not the two option immediately following the first option. Where as the performance of experiment Subsection 4.6 was mediocre.

It is possible that the reason for so little difference in performance, is due to the timing being done on several random graphs. Not altering the order of vertices of I.C., did seem to do very well. Naturally implementing an random ordering of vertices was not done, as the graphs where going to be random. When solving real life graphs, it is hard to say what the effects of doing I.C on a random order of vertices, will be. There where orderings that did equally well with not altering ordering, at solving the random graph set.

After all this, I played with the order of vertices at the start of the program, before a hypothetical greedy ordering of vertices. And figured that sorting the vertices with decreasing vertex degree, right towards left, immediate halved the running time.

As previously mentioned, all sorts where stable sort. Other options where not assessed.

## 4.8 Other considerations

**There are many things one can do.** For instance; not counting degree of vertex and afterwards compare integer with constant. As mentioned, forest is a hereditary property. Forest test need not be computed with exception of when algorithm use Observation 2, or does Reduction 1. This was however not tested, even though implementing this change likely would improve performance of implementation. Trying to improve the picking of vertices looked like a more rewarding exercise.

**2-vertex connected is another possibility** Mention a bit about finding 2-vertex connected here. It could be part of an implementation. Though the performance gain from finding 2-edge connected components in Section 4.5 was not explicitly found to be significant, there could be comparisons that revile it to be more significant. This would motivate proceeding with the implementing of other connectivities such as 2-vertex. As of now, the main motivation is to show one can implement it.

Though finding 2-edge can be does similarly for adjacency matrix and adjacency list using dfs, an implementation using 2-vertex connected has to maintain information that can make the use of adjacency matrix cumbersome.

Even if there is not much difference between 2-edge connected component and 2-vertex connected component, there could be significant difference to an implementation using one or the other.

Determining 2-edge and 2-vertex connected components, part of I.C., could be done without dfs. A different way of implementing 2-edge might also have found any hypothetical mistakes during implementing experiment of Section 4.7.

The largest amount of articulation point (ap) in a graph is n-2. Exemplified by a line with vertices on it.

For our purposes, one may disregard the vertices of $G'$, each representing a bridge in G and the two endpoint of the bridge. (The endpoints are in other, 2-vertex connected component of G, and in other vertices of $G'$) All walks in this graph, walking on vertex representing components is followed by walking on dummy vertex and vice versa. Every leafs represent a component.

Rooted tree, is a tree, where a vertex is designated being root.

While the 2-edge connected components as seen in experiment of Section 4.5 was just including an intermediate step in a previous implementation,

an implementation involving finding 2-vertex connected components, would be more involved.

You may have to maintain a rooted tree, which contains information about what 2-vertex connected components to solve. And there could be more than just just one 2-vertex connected component, that has to be evaluate per I.C. step, unlike the one 2-edge connected component per I.C. step seen in Section 4.5. To make an more efficient implementation, one may want to consider, algorithms like LCA and RMQ.

# Chapter 5

# Results

## 5.1 parallelization

The results shown here where based on parallelization where every procesé calculates how the tasks are divided, seen in Subsection 4.4.

Figure 5.1a shows the time running on an Intel E6600, where p is the number of cores used (no hyper-threading).

If you look at previous timed solvings of the random graph set, you see that the first experiment on adjacency list representation of Subsection (4.6), outperformed the more efficient implementation, still after Section 4.7, which lead to the timings presented her. That only k=13 was considered in experiment of Section 4.7, is likely a cause of this observation. That the more efficient adjacency list implementation of Subsection 4.6, is worth while when the problem instance is graphs with fewer vertices. However, as mentioned in Subsection 4.6, the more efficient implementation of adjacency list is not implemented as well as it could. In stead of a plain list, it still used the balanced tree container, same as all other experiments. Balanced tree containers tend to cost a constant factor more than plain containers. And if elements are added in a less than random order, they tend to performs even worse still. It is imaginable that implementation of Subsection 4.6 performs in practice better than implementation of Subsection 4.6, and much better when the number of edges decreases. Which has been the main motivation since Chapter 4.3.

This implementation does involve quite a number induced forests that are solved by every procesé, while it can not be seen in any of the figures here, even for k=13, the speed-up as adding procesés was near perfect in the number of procesés when using OpenMP, however when not using OpenMP,

| p | time/s |
|---|---|
| 1 | 397.537 |
| 2 | 314.333 |

(a) time

| p | speedup |
|---|---|
| 1 | 1.00 |
| 2 | 1.26 |

(b) speedup

| p | efficiency |
|---|---|
| 1 | 1.00 |
| 2 | 0.63 |

(c) efficiency

Figure 5.1: Multi-core processing. Achieved performance using p cores.

there where a significant performance gain, and Figure 5.1 shows for p=1, the performance when not using OpenMP.

With regards to performance, a potential issue is the allocation in induced forest enumeration of experiment of Subsection 4.6, as mentioned, no direct effort was done specificity to prevent cache cohesion. Allocation involves system call, and there is often an additional cost to doing system calls, but this issue was not prioritized either.

## 5.2 real life experiments

The real life experiments are listed in Figure 5.2. Rest of this section is information on the graphs.

Paley graph over GF13, is a graph taken from sagemath.org

A friend network extracted from facebook, named Arvid.

Transeleted from The Stanford GraphBase was the graphs miles1500 and david.

An problem instance of the n-queen puzzly is the queen8_12 graph, which originate from the 2nd DIMACS implementation challenge [38].

On fauskes.net a pretty drawing of the graph example node-transparency. “Lion share” was a race horse, whose pedigree is described as a graph, and an other graph, the graph ngk10_4 where both taken from graphviz.org.

The remaining graph where found searching The University of Florida

| Name | $\|V\|$ | $\|E\|$ | $\|F\|$ | time/seconds |
|---|---|---|---|---|
| IMACS10/chesapeake | 39 | 170 | 12 | 0.020 |
| Newman/karate | 34 | 78 | 7 | 0.004 |
| Newman/dolphins | 62 | 159 | 19 | 33.374 |
| Paley graph over GF(13) | 13 | 39 | 7 | 0.003 |
| ngk10_4 | 50 | 99 | 13 | 2.395 |
| lion_share | 47 | 51 | 3 | 0.003 |
| node-transparency | 50 | 99 | 13 | 2.378 |
| Pajek/GD96_c | 65 | 125 | 19 | 1758.709 |
| Pajek/GD97_a | 84 | 166 | - | - |
| david | 87 | 406 | - | - |
| miles1500 | 128 | 5198 | - | - |
| queen8_12 | 96 | 1368 | - | - |
| arvid | 45 | 166 | 16 | 0.041 |
| HB/dwt_59 | 59 | 104* | 16 | 28.605 |
| HB/dwt_72 | 72 | 75* | 2 | 0.007 |
| HB/bcspwr01 | 39 | 46* | 5 | 0.003 |
| HB/bcspwr02 | 49 | 59* | 5 | 0.004 |

Figure 5.2: Prallelized using 2 procesés. A “-” symbolized timed out. Timeout was at 24hours of attempting to find a solution. The size of FVS was not not already known. Not writing the FVS of the graphs saves paper.

Sparse Matrix Collection [1], which also provides various drawings. All these graphs are connected. The graph Newman/dolphins contains a social network of frequent associations between 62 dolphins in a community living off Doubtful Sound, New Zealand, by Lusseau et al. [46]. Pajek network: Graph Drawing contest, with the challenge of drawing the graphs Pajek/GD96_c and Pajek/GGD97_a, respectively in 1996 and 1997. The graph Newman/karate social network between the 34 members of a karate club at a US university, as described by W. Zachary [56]. The graph DIMACS10/chesapeake is a clustering/chesapeake from the DIMACS10 set.

The following had the edge constituting self-loops removed, opposed to actually removing the vertex. SHB/bcspwr01 and HB/bcspwr02 represents a standard test power system, New England (1981). HB/dwt_59 and HB/dwt_72 are symmetric connection table from DTNSRDC, WASHINGTON.

## Chapter 6

# Concluding remarks

We have made an implementation for solving the FVS problem, shown some experiments that improve on this implementation, and presented information about how the implementations perform on a generated set of random graphs and graphs from real life applications. Some bottlenecks has been discussed and some progress towards getting reliable performance, seems to have been made. The most significant bottle neck presented in Section 4.3, and smaller progress as more experiments where conducted. A direction forward, may be, to examine performance on other types of graphs. There are likely much progress that can be made with regard to sparse graphs, using for instance expander graphs as problem instances. Some options that may lead to improvements have been suggested. With a specific application of the FVS problem in minds, there are likely more improvements. The problem looks to be very parallelizable, and with it's limited requirements on resources like memory, it might be cost-effective to find minimum FVS on undirected graphs. It would be useful to have, better predicting of how much time finding a solution can in practice take, and not just in theory,

# Bibliography

[1] The university of florida sparse matrix collection.

[2] Thomas A. Anastasio. Red-black trees.

[3] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. J. ACM, 45(3):501–555, 1998. Earlier version in FOCS'92.

[4] Vineet Bafna, Piotr Berman, and Toshihiro Fujito. Constant ratio approximations of the weighted feedback vertex set problem for undirected graphs. In John Staples, Peter Eades, Naoki Katoh, and Alistair Moffat, editors, ISAAC, volume 1004 of Lecture Notes in Computer Science, pages 142–151. Springer, 1995.

[5] Vineet Bafna, Piotr Berman, and Toshihiro Fujito. A 2-approximation algorithm for the undirected feedback vertex set problem. SIAM J. Discrete Math., 12(3):289–297, 1999.

[6] Reuven Bar-Yehuda, Dan Geiger, Joseph Naor, and Ron M. Roth. Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and bayesian inference. SIAM J. Comput., 27(4):942–959, 1998.

[7] Ann Becker, Reuven Bar-Yehuda, and Dan Geiger. Randomized algorithms for the loop cutset problem. J. Artif. Intell. Res. (JAIR), 12:219–234, 2000.

[8] Ann Becker and Dan Geiger. Optimization of pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem. Artif. Intell., 83(1):167–188, 1996.

[9] Hans L. Bodlaender. On disjoint cycles. Int. J. Found. Comput. Sci., 5(1):59–68, 1994.

[10] Hans L. Bodlaender. A cubic kernel for feedback vertex set. In Wolfgang Thomas and Pascal Weil, editors, STACS, volume 4393 of Lecture Notes in Computer Science, pages 320–331. Springer, 2007.

[11] Hans L. Bodlaender, Hans L. Bodlaender, and Hans L. Bodlaender. On disjoint cycles. International Journal of Foundations of Computer Science, 5:59–68, 1990.

[12] Hans L. Bodlaender, Bart M. P. Jansen, and Stefan Kratsch. Preprocessing for treewidth: A combinatorial analysis through kernelization. CoRR, abs/1104.4217, 2011.

[13] Lorenzo Brunetta, Francesco Maffioli, and Marco Trubian. Solving the feedback vertex set problem on undirected graphs. Discrete Applied Mathematics, 101(1-3):37–51, 2000.

[14] Kevin Burrage, Vladimir Estivill-Castro, Michael R. Fellows, Michael A. Langston, Shev Mac, and Frances A. Rosamond. The undirected feedback vertex set problem has a poly(k) kernel. In Hans L. Bodlaender and Michael A. Langston, editors, IWPEC, volume 4169 of Lecture Notes in Computer Science, pages 192–202. Springer, 2006.

[15] Yixin Cao, Jianer Chen, and Yang Liu. On feedback vertex set new measure and new structures. In Algorithm Theory-SWAT 2010, pages 93–104. Springer, 2010.

[16] Jianer Chen, Fedor V. Fomin, Yang Liu, Songjian Lu, and Yngve Villanger. Improved algorithms for feedback vertex set problems. J. Comput. Syst. Sci., 74(7):1188–1198, 2008.

[17] Mao cheng Cai, Xiaotie Deng, and Wenan Zang. A min-max theorem on feedback vertex sets. Math. Oper. Res., 27(2):361–371, 2002.

[18] Fabián A. Chudak, Michel X. Goemans, Dorit S. Hochbaum, and David P. Williamson. A primal-dual interpretation of two 2-approximation algorithms for the feedback vertex set problem in undirected graphs. Oper. Res. Lett., 22(4-5):111–118, 1998.

[19] Rina Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. Artif. Intell., 41(3):273–312, January 1990.

[20] Rina Dechter and Judea Pearl. The Cycle-Cutset Method for Improving Search Performance in AI. In Proc. of the 3rd IEEE Conference on AI Applications, pages 224–230, Orlando FL, 1987.

[21] Frank K. H. A. Dehne, Michael R. Fellows, Michael A. Langston, Frances A. Rosamond, and Kim Stevens. An o(2o(k)n3) fpt algorithm for the undirected feedback vertex set problem. Theory Comput. Syst., 41(3):479–492, 2007.

[22] Rodney G. Downey and Michael R. Fellows. Fixed parameter tractability and completeness. In Klaus Ambos-Spies, Steven Homer, and Uwe Schőning, editors, Complexity Theory: Current Research, pages 191–225. Cambridge University Press, 1992.

[23] Rodney G. Downey and Michael R. Fellows. Parameterized Complexity. Springer-Verlag, 1999. 530 pp.

[24] John Doyle, Bruce Francis, and Allen Tannenbaum. Feedback control theory, 1990.

[25] P. Erdős and L. Pósa. On the maximal number of disjoint circuits of a graph, publ. math. debrecen, 9, 3–12., 1962.

[26] Guy Even, Joseph Naor, Baruch Schieber, and Leonid Zosin. Approximating minimum subset feedback sets in undirected graphs with applications. SIAM J. Discrete Math., 13(2):255–267, 2000.

[27] Paola Festa, Panos M. Pardalos, and Mauricio G. C. Resende. Feedback set problems. In Christodoulos A. Floudas and Panos M. Pardalos, editors, Encyclopedia of Optimization, pages 1005–1016. Springer, 2009.

[28] Fedor V. Fomin, Serge Gaspers, and Artem V. Pyatkin. Finding a minimum feedback vertex set in time o (1.7548n). In Hans L. Bodlaender and Michael A. Langston, editors, IWPEC, volume 4169 of Lecture Notes in Computer Science, pages 184–191. Springer, 2006.

[29] Fedor V. Fomin, Serge Gaspers, Artem V. Pyatkin, and Igor Razgon. On the minimum feedback vertex set problem: Exact and enumeration algorithms, 2007.

[30] Fedor V. Fomin, Serge Gaspers, Artem V. Pyatkin, and Igor Razgon. On the minimum feedback vertex set problem: Exact and enumeration algorithms. Algorithmica, 52(2):293–307, 2008.

[31] Meinrad Funke and Gerhard Reinelt. A polyhedral approach to the feedback vertex set problem. In William H. Cunningham, S. Thomas McCormick, and Maurice Queyranne, editors, IPCO, volume 1084 of Lecture Notes in Computer Science, pages 445–459. Springer, 1996.

[32] M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences). W. H. Freeman, first edition edition, 1979.

[33] Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA, 1979.

[34] Jiong Guo, Jens Gramm, Falk Hűffner, Rolf Niedermeier, and Sebastian Wernicke. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. J. Comput. Syst. Sci., 72(8):1386–1396, 2006.

[35] Jiong Guo, Jens Gramm, Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. J. Comput. Syst. Sci., 72(8):1386–1396, December 2006.

[36] Johan Håstad. Some optimal inapproximability results. Electronic Colloquium on Computational Complexity (ECCC), 4(37), 1997.

[37] D.S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems, discrete applied mathematics, 6, 243-254., 1983.

[38] David S Johnson and Michael A Trick. Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993, volume 26. American Mathematical Soc., 1996.

[39] Iyad A. Kanj, Michael J. Pelsmajer, and Marcus Schaefer. Parameterized algorithms for feedback vertex set. In Rodney G. Downey, Michael R. Fellows, and Frank K. H. A. Dehne, editors, IWPEC, volume 3162 of Lecture Notes in Computer Science, pages 235–247. Springer, 2004.

[40] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, Complexity of Computer Computations, page 85?103. Plenum, New York, 1972.

[41] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, Complexity of Computer Computations, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[42] Jon M. Kleinberg and Amit Kumar. Wavelength conversion in optical networks. J. Algorithms, 38(1):25–50, 2001.

[43] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. Algorithmica, 6(1):5–35, 1991.

[44] John M. Lewis and Mihalis Yannakakis. The node-deletion problem for hereditary properties is np-complete. J. Comput. Syst. Sci., 20(2):219–230, 1980.

[45] Carsten Lund and Mihalis Yannakakis. The approximation of maximum subgraph problems. In Andrzej Lingas, Rolf G. Karlsson, and Svante Carlsson, editors, ICALP, volume 700 of Lecture Notes in Computer Science, pages 40–51. Springer, 1993.

[46] David Lusseau, Karsten Schneider, Oliver J Boisseau, Patti Haase, Elisabeth Slooten, and Steve M Dawson. The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. Behavioral Ecology and Sociobiology, 54(4):396–405, 2003.

[47] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. J. Comput. Syst. Sci., 43(3):425–440, 1991.

[48] Panos M. Pardalos, Tianbing Qian, and Mauricio G. C. Resende. A greedy randomized adaptive search procedure for the feedback vertex set problem. J. Comb. Optim., 2(4):399–412, 1998.

[49] Venkatesh Raman, Saket Saurabh, and C. R. Subramanian. Faster fixed parameter tractable algorithms for undirected feedback vertex set. In Prosenjit Bose and Pat Morin, editors, ISAAC, volume 2518 of Lecture Notes in Computer Science, pages 241–248. Springer, 2002.

[50] Venkatesh Raman, Saket Saurabh, and C. R. Subramanian. Faster fixed parameter tractable algorithms for finding feedback vertex sets. ACM Transactions on Algorithms, 2(3):403–415, 2006.

[51] Igor Razgon. Exact computation of maximum induced forest. In Lars Arge and Rusins Freivalds, editors, SWAT, volume 4059 of Lecture Notes in Computer Science, pages 160–171. Springer, 2006.

[52] Bruce A. Reed, Kaleigh Smith, and Adrian Vetta. Finding odd cycle transversals. Oper. Res. Lett., 32(4):299–301, 2004.

[53] Abraham Silberschatz and Peter Galvin. Operating System Concepts, 4th edition. Addison-Wesley, 1994.

[54] Stéphan Thomassé. A 4k2 kernel for feedback vertex set. ACM Transactions on Algorithms, 6(2), 2010.

[55] D. Younger. Minimum Feedback Arc Sets for a Directed Graph. Circuit Theory, IEEE Transactions on, 10(2):238–245, 1963.

[56] Wayne Zachary. An information flow model for conflict and fission in small groups. Journal of Anthropological Research, 33:452–473, 1977.