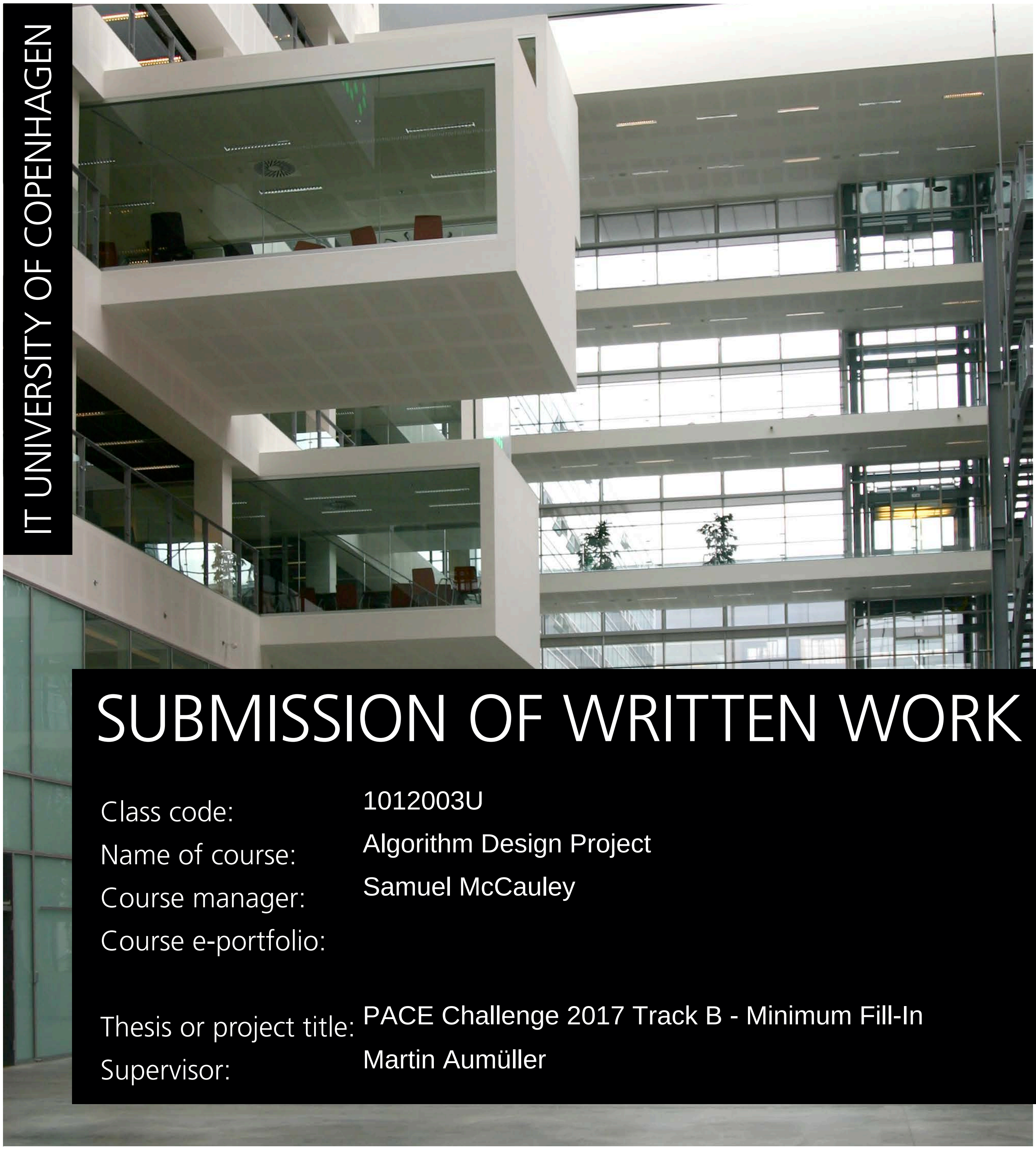


IT UNIVERSITY OF COPENHAGEN

SUBMISSION OF WRITTEN WORK

Class code:	1012003U
Name of course:	Algorithm Design Project
Course manager:	Samuel McCauley
Course e-portfolio:	
Thesis or project title:	PACE Challenge 2017 Track B - Minimum Fill-In
Supervisor:	Martin Aumüller

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Anders Wind Steffensen	10/02-1993	awis@itu.dk
2.	Mikael Lindemann	17/02-1992	mlin@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

PACE Challenge 2017 Track B - Minimum Fill-In

Anders Wind Steffensen   Mikael Lindemann
Supervisor: Martin Aumüller

August 27, 2017

Contents

1 Introduction **3**

2 Preliminaries **3**

3 The Algorithm **3**
3.1 Extensions 6

4 Implementation **8**
4.1 Programming Style 8
4.2 Development Challenges 8

5 Results **8**

6 Conclusion **11**

7 References **12**

1 Introduction

The *Parameterized Algorithms and Computational Experiments Challenge* (PACE) is a yearly competition which investigates the applicability of different algorithms especially in the field of parameterized, and fixed-parameter tractable algorithms.[1]

This report describes our solution to the 2017 PACE challenge track B: Minimum Fill-In. Given a graph G The minimum fill-in problem is the task of finding the minimum set of edges needed to make G chordal. A graph is chordal if for every cycle of length at least four there exists a chord, i.e., an edge between non-adjacent vertices in the cycle. The problem was proven to be NP-complete by Yannakakis in [7]. Our solution is based on the subexponential parameterized algorithm by Fomin et al. in [3] which will be referred to as Fomin's algorithm. In addition to Fomin's algorithm, a number of general extensions and optimizations will be described. The source code of the solution can be found at `https://github.com/Awia00/MinFill-SubExponential`.

2 Preliminaries

Let G be an undirected, simple graph with vertex set $V(G) = V$ and edge set $E(G) = E$. We define the following concepts and notation:

Let $N(v)$ be the neighbourhood of a vertex v such that $N(v) = \{u \in V : (u, v) \in E\}$. Let $N(X)$ be the neighbourhood of a set $X \subseteq V$ such that $N(X) = \cup_{v \in X} N(v) \setminus X$. Let $G[X]$ denote the graph G induced by the set $X \subseteq V$. Let G_Ω denote the graph G where the set $\Omega \subseteq V$ has been turned into a clique.

A chordless cycle, is a cycle of length at least four, where there is no edge between non-adjacent vertices in the cycle. A chordal graph is a graph with no chordless cycles.

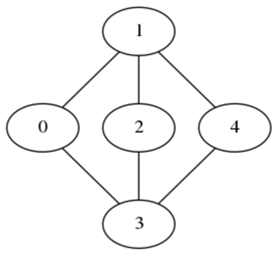
Given a graph G, a *minimal fill-in* F is a set of fill-edges, such that removing any fill-edge from F will make G non-chordal. Given a graph G, a *minimum fill-in* F is a minimal fill-in of smallest possible size for G. An example of minimal and minimum fill-in can be seen in Figure 1.

A separator S is a set of vertices of G for which $G[V \setminus S]$ separates two vertices a and b into two different components. A minimal separator is a separator S for which no proper subset of S separates a and b.

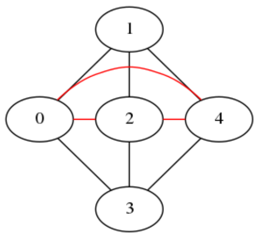
3 The Algorithm

Fomin's algorithm takes as input a Graph G and a non-negative integer k. The algorithm outputs YES if it is possible to make the graph chordal, by adding at most k edges. Otherwise the algorithm outputs NO. In this report an alternative

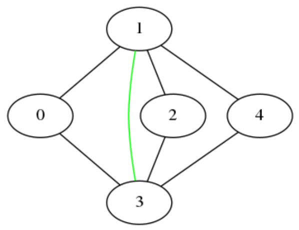
[1]See `https://pacechallenge.wordpress.com/` for more information.



(a) A non-chordal graph G.



(b) A minimal fill-in of G. The red edges are the fill.



(c) A minimum fill-in of G. The green edge is the fill.

Figure 1: A graph, it's minimal fill-in and it's minimum fill-in.

version of Fomin's algorithm is described. This version outputs the edges of the minimum fill-in or NO if infeasible.

The algorithm is divided into four steps: A, B1, B2 and C.

In step A a kernel is applied to obtain a graph G', with $\mathcal{O}(k^2)$ vertices, and an integer $k' \leq k$ from the original instance G, k. Any minimum fill-in solution to G', k' is also part of the solution to G, k. The specific kernel algorithm used is from [5] by Natanzon, Shamir and Sharan, which guarantees these reductions in time $\mathcal{O}(k^2nm)$.

G', k' is used as input for Step B1 where G' is of size $\mathcal{O}(k^2)$.

Step B1 is a search tree algorithm which takes as input G, k and finds so called *non-reducible instances* of G. A non-reducible instance is a graph G' with no h-*obscure* paths exist, for $h = \sqrt{k}$. A path is h-obscure if two non-adjacent vertices u and v with a shared neighbourhood X has a chordless path $P = (u, w_1, ..., w_l, v)$ where for all $i \in \{1, ..., l\}, |N(w_i) \cap X| \leq h$. If an h-obscure path is found, the algorithm branches on either adding the edge (u, v) or for every $i \in \{1, ..., l\}, w_i$ has an edge to each vertex of X added. Note that k' is k minus the number of added edges. Fomin proves that there can be at most $|V|^{\mathcal{O}(\sqrt{k})}$ branches on any given graph where each recursive call can be done in $\mathcal{O}(|V|^4)$ which results in a total running time of $|V|^{\mathcal{O}(\sqrt{k})}$.

If a non-reducible instance G', k' is found, this is given as input to Step B2.

Step B2 takes as input G, k and generates *vital potential maximal cliques* of G. A *potential maximal clique* is a set $\Omega \subset V$ which is a clique in *some* minimal fill-in of G. A potential maximal clique is *vital* if it is missing at most k edges from being a clique in G. It is only relevant to consider filling vital potential maximal cliques since any potential maximal clique which is missing more than k edges cannot be part of a minimum fill-in with at most k edges.

A potential maximal clique Ω is a *quasi clique* if some set $Z \subseteq \Omega$ of size at most $5\sqrt{k}$ makes $\Omega \setminus Z$ a clique. To enumerate all quasi cliques of a non-reducible graph, all vertex subsets of size at most $5\sqrt{k}$ are generated. For each vertex subset Z the minimal fill-in H of a graph $G[V \setminus Z]$ is found. For H three different cases are checked:

Case 1: For every minimal separator S of H where $G[S]$ is a clique, if the set $S \cup Z$ is a potential maximal clique in G then $S \cup Z$ is a quasi clique.

Case 2: For every maximal clique X of H where $G[X]$ is a clique, if the set $X \cup Z$ is a potential maximal clique in G then $X \cup Z$ is a quasi clique.

Case 3: For every maximal clique X of H where $G[X]$ is a clique let G' be the graph $G[V \setminus (X \cup Z)]$. For every vertex $y \in Z$, the set Y is the union of the neighbourhoods in G of the components of G' containing y. If $Y \cup \{y\}$ is a potential maximal clique in G then $Y \cup \{y\}$ is a quasi clique.

To enumerate all vital potential maximal cliques, first enumerate all quasi cliques of G and check if they are vital. Then enumerate all vertex subsets of size up to $5\sqrt{k} + 2$ and check if they are vital potential maximal cliques. Last, for each vertex $w \in V$, create a graph H where $N(w)$ is a clique, enumerate the quasi cliques of H and check if they are vital potential maximal cliques in G. The running time of enumerating all vital potential maximal cliques is $|V|^{\mathcal{O}(\sqrt{k})}$.

In step C the set of vital potential maximal cliques Π_k and a set $\Pi_{S,C}$ is used to calculate the fill-in of the remaining instance. A minimal separator $S \subset \Omega$ is the neighbourhood of some component C of $G[V \setminus \Omega]$, and Δ_k is the set of all such minimal separators created from all $\Omega \in \Pi_k$. $\Pi_{S,C}$ is the set of all triples (S, C, Ω), for which $\Omega \in \Pi_k$, $S \in \Delta_k$, C being a full component of $G[V \setminus S]$ associated to S and $S \subset \Omega \subseteq S \cup C$.

When $\Pi_{S,C}$ has been calculated, the formula of Figure 2 is used. The function $\mathit{fill}_G(\Omega)$ computes the set of non-edges in $G[\Omega]$. Notice that C and $S = N_G(C)$ is used to index $\Pi_{S,C}$. The minimum set of edges $M = \mathit{fill}_G(\Omega) \cup \mathit{mfiF}\,(G_\Omega, C, S)$ over all $\Omega \in \Pi_k$ is returned if $|M| \leq k$ and adding M to G makes it chordal. If no such M exists return NO.

$$\mathit{mfi}(G) = \min_{\Omega \in \Pi_k} \left[\mathit{fill}_G(\Omega) \cup \bigcup_{C \text{ is a component of } G \setminus \Omega} \mathit{mfiF}\,(G_\Omega, C, N_G\,(C)) \right]$$

Figure 2: *mfi* computes the minimum fill-in of G. This is a translation of (5.1) from [3] that computes the set of fill edges M rather than $|M|$. Note that *min* calculates the set with fewest elements.

$$\mathit{mfiF}(G,C,S) = \min_{\Omega' \in \Pi_{S,C}} \left[\mathit{fill}_F(\Omega') \cup \bigcup_{C' \text{ is a component of } F \setminus \Omega'} \mathit{mfiF}\left(F_{\Omega'}, C', N_F(C')\right) \right]$$

$$\text{where } F = G[S \cup C]$$

Figure 3: *mfi* computes the minimum fill-in of F. This is a translation of (5.2) from [3] that computes the set of fill edges M rather than $|M|$. Note that *min* calculates the set with fewest elements.

The formula of Figure 3 is calculated via dynamic programming. The memoizer maps graphs to minimum fill-ins. S and C is used to index $\Pi_{S,C}$ to find the sets of vital potential maximal cliques associated to S, c. Notice that if no vital potential maximal cliques are found in $\Pi_{S,C}$ the set of non-edges in F is returned. This answer will always be the worst possible answer for F.

3.1 Extensions

To optimize the Fomin's algorithm, Step A has been divided into two parts A1, and A2 which iteratively reduces the problem more and more, by applying the kernel and performing a number of reductions with polynomial running time.

In step A1, procedure 1 and 2 from the kernel described in [5] is applied to find a lower bound on k as well as the sets $A, B \subseteq V$. A is the set of vertices that contain chordless cycles. B is the set of vertices which where no chordless cycles have been found yet.

Step A2 starts by applying the kernel procedure 3 to A, B, and k. If no instance G', k' can be found, k is increased until the kernel can no longer state that the problem is infeasible with the given k value.

A number of polynomial reductions are applied to the instance. These reductions can both add edges and prune vertices. The reductions exploit that some edges must be part of some minimum fill-in of G' and some vertices can never be part of a chordless cycle. If a reduction adds more than k' edges, then no minimum fill-in of size k' exists.

We propose the following reductions:

A minimal separator which is missing at most one edge from being a clique, must always be a clique in some minimum fill-in H of an instance G. There can be exponentially many minimal separators of a graph, but each minimal separator can be found in polynomial time with the algorithm from [6]. Checking whether a set of vertices is missing an edge from being a clique can also be done in polynomial time, and therefore checking a subset of these minimal separators is possible. This reduction is based on the work in [1].

A *simplicial* vertex is a vertex v for which its neighbourhood is a clique. v can never be part of a chordless cycle. Any cycle of length at least 4 containing

v will also contain two elements $u, w \in N(v)$. Vertices u and v must be adjacent because $N(v)$ is a clique and therefore v can safely be removed. A *universal* vertex v is a vertex whose neighbourhood is $V \setminus \{v\}$. Given that v is connected to every other vertex, it is not possible to add any fill edge $(v, u), u \in V \setminus \{v\}$ to it and therefore v can safely be removed.

If any procedure of step A2 changes the graph either by adding edges or removing vertices, the components of the graph are calculated and the entire algorithm is performed recursively on each component. By intuition, fill-edges will never connect components and therefore each component can be solved separately. If by the end of step A2 there is only one component, the component is checked for minimal separators which are cliques. If any such separator S exists, for each component C of $G[V \setminus S]$ solve $C \cup S$ recursively. Any cycle with two vertices u, v in distinct components after applying the separator will have a chord in the separator, and by intuition adding new edges from any such u, v will not be part of a minimum fill-in.

The results of the recursive calls on components are added together to form the final result.

When no further reductions can be applied, the resulting instance G', k' is forwarded to step B1. If B1 returns a NO answer, k is increased, and step A2 starts over.

Step B2 generates vital potential maximal cliques in $|V|^{\mathcal{O}(\sqrt{k})}$ but in reality the procedure is quite slow, this is due to the hidden factors in the exponent. Since all vertex subsets of size $5\sqrt{k}$ are enumerated to calculate quasi cliques, only values of $k > 25$ will see the subexponential performance increase. Unfortunately at $k = 25$ the graph can have size $\mathcal{O}(k^2) = 600$ which makes the generation of vital potential maximal cliques even more time consuming, when it is based on vertex subset enumeration.

Because of these shortcomings, two algorithms, which have in our experiments shown to be faster, for generating vital potential maximal cliques are implemented. If $|V| < 5\sqrt{k} + 2$ and $|V| < 15$, directly enumerating all vertex subsets instead of performing the entirety of step B2 is enough to find any vital potential maximal clique.[2]

For values of $k < 15$ the search tree algorithm from [4] is used instead of both step B2 and C. The exponential factor of the algorithm is not dependent on the number of vertices in the graph and therefore performs well on a lot of instances with low values of k.[3]

In [2] a polynomial running time algorithm for finding potential maximal cliques in G given the minimal separators of G is provided. The worst case performance is of course still exponential due to the potential number of minimal

[2] $|V| < 15$ being an upper bound since 15! is too big a number of subsets to enumerate on a modern CPU

[3] $k < 15$ since $\mathcal{O}(8^{15})$ is too slow to calculate within the PACE time limits on a modern CPU

separators, but for many graphs using this algorithm gives a huge performance boost.[4]

4 Implementation

4.1 Programming Style

To keep the implementation as close to the articles as possible, the code is implemented in a semi-functional style, where data is immutable, and functions return copies of the changed state. Some elements from imperative programming are allowed, such as loops, print statements, and mutable local variables.

This style definition matches the style used for pseudo-code in many articles, and therefore allows for an easier translation from pseudo-code to implementation.

4.2 Development Challenges

In [3] some of the details needed to implement the algorithm are omitted. Especially with regards to enumerating quasi-cliques the proof is based on information that is first obtained in the end of the proposed algorithm. Furthermore step C leaves some of the implementation details up for the reader to figure out, both in the case of converting the problem from the decision version to the optimization version of the algorithm, but also how edge cases should be handled.

5 Results

It is clear that the main contribution of [3] is the fact that a subexponential algorithm does exist for this NP complete problem. Our implementation however, did not win the PACE challenge, and therefore there must be other algorithms which performs substantially better in general or at least on the instances of the challenge. We assume that for Fomin's algorithm to perform better than other exponential algorithms, the values of k must be bigger than what is feasible today.

Our experiments show that in general the number of non-reducible graphs is relatively small, and finding these is quite fast. The dynamic program in Step C also performs really well which leaves step B2 as the bottleneck of the algorithm. This observation is further supported by the fact that the hidden factors of the $\mathcal{O}$-notation is quite big in finding the vital potential maximal cliques due to the vertex subset enumeration.

[4]As of writing, a bug is present in the implementation of this algorithm which results in not finding every potential maximal clique of G

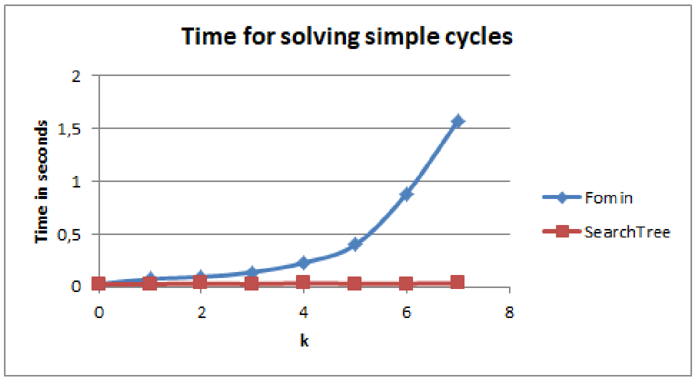


Figure 4: Running times of Fomin's algorithm and the search tree algorithm from [4] when solving simple cycles of size $k + 3$.

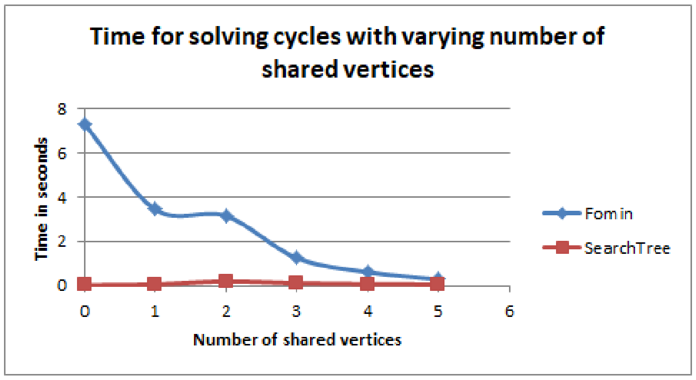


Figure 5: Running times of Fomin's algorithm and the search tree algorithm from [4] when solving cycles with shared vertices.

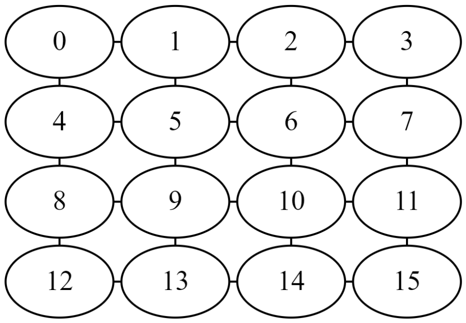


Figure 6: A graph that forms a grid of size $4 * 4$.

On Figures 4 and 5 it seems that Fomin's algorithm is depending a lot on the number of vertices and edges in the graph. In Figure 4 each additional k, adds another vertex and another edge to the cycle. When k is above 4, the running time of Fomin's algorithm starts to increase rapidly, while the running time of the search tree algorithm remains somewhat the same. Early experimentation showed this to be a flaw of Fomin's algorithm, since even small and simple instances can be quite hard to overcome. In Figure 5 the x-axis denotes the number of shared vertices between two, otherwise simple, cycles of length 6. While the search tree algorithm does not decrease in running time when the number of shared vertices increases, Fomin's algorithm does. It seems like the branching-part of Fomin's algorithm selects well-chosen branches on this kind of graph.

We have found a structure, that was hard for both algorithms to solve. The grid in Figure 6 took more than two hours to solve for Fomin's algorithm. After 3 hours, we stopped the search tree algorithm. If we apply the extensions from previous sections to Fomin's algorithm, this instance would be solved within a few seconds.

6 Conclusion

Fomin's subexponential algorithm for the minimum fill-in problem was implemented with a number of optimizations and reductions. Unfortunately our experiments have shown that the theoretically better running time of Fomin's algorithm is not faster for most instances that are already solvable by other algorithms, nor make intractable instances tractable within the time limits of the PACE challenge.

There are several areas of interest which could be examined in trying to improve the running time of Fomin's algorithm. Firstly implementing the algorithm in a low level language such as C, with more focus on optimizing the individual sub procedures would be interesting. Seeing how much can be gained from efficient data structures, less memory usage and less garbage collection would make for a better comparison with some of the other algorithms for minimum fill-in. Furthermore it could be interesting to examine the traits of non-reducible graphs further to see if other algorithms for generation of vital potential maximal cliques could be used efficiently.

We also believe other polynomial time reductions must exist for some classes of (sub)graphs, which would greatly increase the number of solvable instances.

7 References

[1] Hans L. Bodlaender, Pinar Heggernes, and Yngve Villanger. Faster parameterized algorithms for minimum fill-in. *Algorithmica*, 61(4):817–838, 2011.

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

[3] Fedor V Fomin and Yngve Villanger. Subexponential parameterized algorithm for minimum fill-in. *SIAM Journal on Computing*, 42(6):2197–2216, 2013.

[4] Haim Kaplan, Ron Shamir, and Robert E Tarjan. Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs. *SIAM Journal on Computing*, 28(5):1906–1922, 1999.

[5] Assaf Natanzon, Ron Shamir, and Roded Sharan. A polynomial approximation algorithm for the minimum fill-in problem. *SIAM Journal on Computing*, 30(4):1067–1079, 2000.

[6] Hong Shen and Weifa Liang. Efficient enumeration of all minimal separators in a graph. *Theoretical Computer Science*, 180(1-2):169–180, 1997.

[7] Mihalis Yannakakis. Computing the minimum fill-in is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 1981.