Algorithm Engineering
WS 21/22

Nick Beckenlechner

Comparison of different algorithms on the Feedback Vertex Set for vertex-weighted graphs

Abstract

The Feedback Vertex Set is a NP-Complete graph problem that plays an important role in the computer world. In this work we compare different algorithms that solve the Feedback Vertex Set for vertex-weighted graphs. We use a heuristic approach, then the constant factor approximation called Bafna-Berman-Fujito and at last we use an exact algorithm by using a bounded search tree with a parallelized implementation. We found that even tho the heuristic approach is the fastest, the approximation works very well in run time as well as in precision and the exact algorithm is pretty slow caused by the asymptotic exponential run time.

1 Introduction

There exist several algorithms for solving the Feedback Vertex Set on a graph. Solving the Feedback Vertex Set plays a fundamental importance in combinatorical optimization. It also plays a role in practice for solving deadlocks, as well as for designing VLSI chip's and for many other occasions.
In this report we will compare three algorithms that solve the Feedback Vertex Set on vertex-weighted graphs. One is a simple heuristic approach, the other is the 2-Approximation Bafna-Berman-Fujito [Vin99] and the last one is a bounded search tree that uses the approximation as bound and parallelization for speedup.
We start our work by explaining the problem and the algorithm approaches in section 2, then we discuss improvements and implementational details, as well as algorithmic concepts that lower the run times of the algorithms in section 3. Then we have a look at the experimentational evaluation of the algorithms in section 4 and finally we discuss about these results and make a conclusion in section 5.

2 Preliminaries

First we introduce some definitions.
Cycle: is a set of vertices that include a path of edges where every vertex is visited once and the beginning of the path is the same as the end of the path.
Fundamental Cycle: is a cycle which holds that no subset of that cycle is also a cycle.
Semidisjoint Cycle: is a cycle that holds that all of the cycle vertices have a degree of 2 excluding at most 1 vertex.
Semidisjoint Vertex: is a vertex of a semidisjoint cycle that has a degree larger than 2.
Clean Graph: a graph is clean when every vertex has at least a degree of 2.
$\mathbf{n} = |V|$ $\mathbf{m} = |E|$ **c**: number of cycles

The Feedback Vertex Set (short: FVS) for vertex-weighted graphs is definded by the following problem: Given a undirected graph G = (V,E) and a weight function $\omega : V \mapsto \mathbb{R}_+$ that maps vertices to a weight. What is the set $S \in \mathcal{P}(V)$ that holds "$G[V \backslash S]$ is a forest (acyclic)", that has least weight, that means the optimization goal is to minimize $\omega(S) = \sum_{v \in S} \omega(v)$.
To solve this problem we have the following algorithms:

2.1 Heuristic Algorithm

The heuristic algorithm follows the approach of a greedy algorithm. First it finds cycles by doing a DFS (Depth-First Search) where whenever it finds a vertex that has already been marked it backtracks to

get the complete cycle (see algorithm 2). Since the FVS must contain a vertex of every cycle we take the lightest vertex (smallest weight) of each cycle and remove it from the graph until there are no cycles left. Since we remove every round at least 1 vertex and we do a DFS every round we have a asymptotic run time of $\mathcal{O}(n \cdot (n+m))$.
Algorithm 1 covers the pseudo-code of the algorithm.

Algorithm 1 *heuristic*(*Graph G*)

```
solution ← ∅
while ($cycles \leftarrow findCycles(G)) \neq \emptyset$ do
    for $cycle \in cycles$ do
        $v \leftarrow$ lightest vertex in $cycle$
        $G \leftarrow G - \{v\}$
        $solution \leftarrow solution \cup \{v\}$
    end for
end while
return solution
```

Algorithm 2 *findCycles*(*Graph G*)

```
$Cycles \leftarrow \emptyset$
while G has unmarked vertex do
    $u \leftarrow$ any unmarked vertex
    mark u and put on a new stack
    while stack not empty do
        $x \leftarrow stack.top()$
        if x has unmarked edge then
            $y \leftarrow$ incident vertex of edge
            if y marked then
                $cycle \leftarrow$ vertices over y on stack
                Cycles.add($cycle \cup \{y\}$)
            else
                mark y and put on stack
            end if
        else
            stack.pop()
        end if
    end while
end while
return Cycles
```

2.2 Bafna-Berman-Fujito

The Bafna-Berman-Fujito [Vin99] is a algorithm for the FVS that gives us a solution that is guaranteed to be at worst case 2 times worse than the optimum. This is done by relaxing the vertex weights:

First we start by doing a cleanup where we recursively remove all vertices that have a degree of 1 or less since they are not part of the solution.
After the cleanup we are left with cycles only (graph is clean).

Then we look whether we have a semidisjoint cycle. If we do we relax all weights of the cycle by decreasing it with the minimum weight of the cycle vertices.

If we do not have a semidisjoint cycle we calculate $\gamma = min\{\omega(v)/(deg(v) - 1) \wedge v \in V_i\}$ where V_i is the set of vertices left contained in G at the ith round. We relax all vertices with γ by reducing their weights $\omega(v) := \omega(v) - \gamma(deg(v) - 1)\ \forall v \in V_i$. Notice that deg(v) - 1 is the number of fundamental cycles v is part of.

After we relaxed (semidisjoint cycle if available, else whole graph) we collect all vertices with relaxed weight of 0 onto a stack and into our solution set. Then we clean up and begin with relaxation again until we have an empty graph left.

As soon as we have an empty graph we go through the stack (reverse order as we have found them) and check if our solution without the current element is as well a valid FVS, if so we remove the current element from the solution.
After removal we have a valid FVS that holds the approximation factor, see reference [Vin99] for proof.

Since every round we remove at least 1 vertex we have at most $|V|$ rounds. Every round we go through all vertices for relaxing but also through all edges for finding a semidisjoint cycle. That leaves us with a run time of $\mathcal{O}(n \cdot (n+m))$.
Algorithms pseudo-code can be seen in 3.

Algorithm 3 *Bafna-Berman-Fujito*(*Graph G*)

```
$solution \leftarrow \emptyset$
$stack \leftarrow$ create empty stack
clean(G)
while $V \neq \emptyset$ do
    if G contains semidisjoint cycle C then
        $\gamma \leftarrow min\{\omega(v) | v \in C\}$
        for $v \in C$ do
            $\omega(v) \leftarrow \omega(v) - \gamma$
        end for
    else
        $\gamma \leftarrow min\{\omega(v)/(deg(v) - 1) | v \in V\}$
        for $v \in C$ do
            $\omega(v) \leftarrow \omega(v) - \gamma(deg(v) - 1)$
        end for
    end if
    for $v \in V \wedge \omega(v) = 0$ do
        $G \leftarrow G - \{v\}$
        stack push $v$
        $solution \leftarrow solution \cup \{v\}$
    end for
    clean(G)
end while
while $stack \neq \emptyset$ do
    $v \leftarrow stack.pop()$
    if $solution \backslash \{v\}$ is a valid FVS then
        $solution \leftarrow solution \backslash \{v\}$
    end if
end while
return solution
```

Algorithm 4 *clean*(*Graph G*)

```
while G has vertex $v$ with $deg(v) \leq 1$ do
    $v \leftarrow$ vertex with $deg(v) \leq 1$
    $G \leftarrow G - \{v\}$
end while
```

2.3 Exact Algorithm

The exact algorithm is an algorithm that creates the optimal solution. This is done by using a so called Bounded Search Tree (short: BST), that branches every possibility by deciding whether something is in the solution or not.

Concretely the algorithm takes a random non-marked Vertex v of the Graph G and marks it. Then it makes one recursive call where v is in the solution and one recursive call where v is not in the solution. After comparison of both solutions the better one is taken and returned.

A BST makes use of fixed-parameter tractability, that means it uses a specific bound such that if a solution would exceed a solution size, instead it stops the search in that specific path. Through that we do not search for solutions that are worse than our bound.

Since we split every function call into 2 recursive calls we have a total of 2^n function calls. In every function call we have to look if we already have a valid solution what has a asymptotic run time of $\mathcal{O}(n + m)$. That means we have a total asymptotic run time of $\mathcal{O}(2^n \cdot (n + m))$.

Algorithms pseudo-code can be seen in 5. The "solution" parameter is the solution that was found until the call was made (in the first call equal to $\emptyset$).

Algorithm 5 *exactBST*(*Graph G*, *bound*, *solution*)

```
if $bound < 0$ then
    return "No Solution"
end if
if G is acyclic then
    return solution
end if
if G has no unmarked vertex then
    return "No Solution"
end if
$v \leftarrow$ random unmarked vertex of G
mark $v$
$G' \leftarrow G - \{v\}$
$bound' \leftarrow bound - \omega(v)$
$solution' \leftarrow solution \cup \{v\}$
$solution1 \leftarrow exactBST(G', bound', solution')$
$solution2 \leftarrow exactBST(G, bound, solution)$
if $\omega(solution1) < \omega(solution2)$ then
    return $solution1$
else
    return $solution2$
end if
```

3 Algorithm & Implementation

In the following chapter we introduce some algorithmic concepts, then talk about advancements of the basic algorithms, as well as some implementional details that have impact on run time of the algorithms.

- **Algorithmic Concepts:**

  First we introduce **kernelization**.
  The goal of kernelization is to reduce one problem instance into another instance in polynomial time. This can be very helpful, for problems that are NP-Complete since we save a lot of time by reducing the problem in polynomial time so that our exponential growing algorithm just has to deal with a reduced smaller problem.

  Another concept is **parallelization**.
  With parallelization we make use of multiple cores by using multiple threads with each having its own code. These threads then can be executed by multiple cores at the same time, saving some valuable time.

- **Advanced Algorithm:**

  We made many adjustments forward the advanced version of the exact algorithm. First of all we used the before introduced concept of kernelization. The basic idea is if we find any vertices that we can exclude or include in our solution beforehand, we can remove these vertices from the graph and include them later on in our solution. By this we can reduce our graph and save valuable time. The kernelization reduction rules are applied at the beginning of every function call of the exact algorithm.

  The first kernelization reduction rule is pretty obvious and already used in the approximation algorithm Bafna-Berman-Fujito. For every vertex that is not part of cycle we do not include them in our solution. Since the FVS is a set of vertices, whose removal would lead the graph to be acyclic, these vertices are already not part of a cycle, what means that their removal would not lead into less cycles. That means they are not necessary and with that not part of a minimal FVS.

  The second kernelization reduction rule optimizes a semidisjoint cycle. As before declared a semidisjoint cycle is a cycle that has at most one vertex with a degree higher than 2. That means if we have a semidisjoint cycle either the graph consists only of this cycle or the semidisjoint cycle intersects only with one vertex with the rest of the graph (intersection vertex is denoted as semidisjoint vertex). In both scenarios this cycle has at most one collision point with another cycle. That leads to that only one vertex of this cycle is part of the optimal solution,
  since every FVS that includes the semidisjoint vertex covers the same cycles as including only the semidisjoint vertex and every FVS that excludes the semidisjoint cycle would have the same cover as if we just take one vertex.
  Because now the only vertices to be considered in the semidisjoint cycle would be the semidisjoint vertex and the minimum weight vertex

of that cycle our reduction rule would be to go to every semidisjoint cycle and remove all vertices that are not the lowest weight or a semidisjoint vertex while containing that the remaining vertices of that cycle still form a cycle.

Next we changed the way of picking a vertex since the basic algorithm has many paths in the search tree that lead to no valid solution. Instead of choosing a random vertex of the graph we chose a cycle. We know that the FVS must contain at least one of the vertices of the cycle. So now we do a recursive call with every vertex v of the cycle where v is included in the solution. Through that we are guaranteed that every recursion call removes at least one cycle less. We now have a recursion depth that is bound with the number of cycles.
Also we do not pick any random cycle each call, instead we pick the smallest cycle. Through that our decision tree stays small at the beginning and would expand at the end. That leads to the big cycles that would leave many decisions just being considered if it is in our bound. Through that our bounded decision tree stays smaller as if we would take the big cycles first.

Another improvement we did is the usage of multiple cores by making use of parallelization. Since our search tree splits every call into multiple recursive calls which are independent of each other we can simultaneously do the recursive calls in different threads, which then can be handled by multiple cores. With that the solutions of the recursive calls can be calculated at the same time and afterwards be compared as soon as all threads are done.

The last improvement is done by updating the bound. As soon as we find a new solution we use the new solution as a new bound since searching for any solutions that are worse than the new bound are also worse than our solution that we already found. Through that we only get solutions that are consecutive better until the optimum is found.

- **Implementational Details:**

  We implemented all three algorithms in Java (JavaSE-15 (jdk-17.0.1)).

  No extern libraries were used.

  As graph structure we used a adjacency list. Concretely we have an array of vertex objects. Each vertex holds its weight, boolean values whether the vertex is marked and the list of edges that are incident to that edge. An edge just consists of both vertex id's and a boolean value whether it is marked.
  Through that we can access a vertex in $\Theta(1)$ and iterate over all adjacent vertices of a vertex v in time $\Theta(deg(v))$ where $deg(v)$ is the degree of v. These run times are minimal and both very important for our algorithms, in example for DFS for finding cycles which is used in different forms in every algorithm.

  To optimize memory management we implemented enabling and disabling of vertices. With that we can disable a vertex in time $\mathcal{O}(1)$ and enable it at some point later in time $\mathcal{O}(1)$ instead of doing a copy of the graph and then delete the vertex which would run in $\mathcal{O}(n+m)$. The only time we copy the graph is when we make use of parallelization since we do not want multiple recursive calls to work on the same graph, instead we do copies for every thread so it has its own instance.

4 Experimental Evaluation

In this section, we will show our experiments and their results.

4.1 Data and Hardware

All experiments were conducted on a AMD Ryzen 7 5800x 8-Core processor with a base speed of 3.8 GHz and up to a boosted speed of 4.5 GHz. For RAM we used 16 GB with 3200 MHz base speed. The algorithms were tested on the public instances of the

PACE 2016 - Track B: Feedback Vertex Set challenge, see [PAC] for reference. These are 100 undirected unweighted graphs. Since our algorithms are all designed for solving the feedback vertex set for vertex-weighted graphs we initialized all vertex weights with random decimal numbers between 1 and 5 (equal distribution) in a graph.

4.2 Results

Since our exact algorithm has a high potential of having horrendous execution times we have set a time limit of 20 minutes for that algorithm to compute the solution. With that we only were able to compute exact solutions for 20 graphs. We did not compare run time of the exact algorithm with any other algorithm because every time the exact algorithm was able to compute the solution both other algorithms did not even take 2 milliseconds for computation.

Still we can use these 20 solutions to compare the solution qualities of the algorithms. That comparison can be seen in the the figure 1. As you can see the approximated solution is very close to the exact solution. Concretely in average it was 3.934% worse than our exact solution. Even the worst approximation was only 15.96% off from the optimal solution. The approximation can be also be the optimal solution, which even was the case for 3 graphs (graph 7, 62 and 99).

The heuristic algorithm on the other hand was definitely more off. In average the solution of the heuristic algorithm was 45.55% worse than the exact algorithm. Its worst solution exceeded the approximation boundary of 2 with a solution that was 124.63% worse than the optimal solution. The best solution boundary is 13.74% what still was quiet close to the worst of the approximation algorithm.

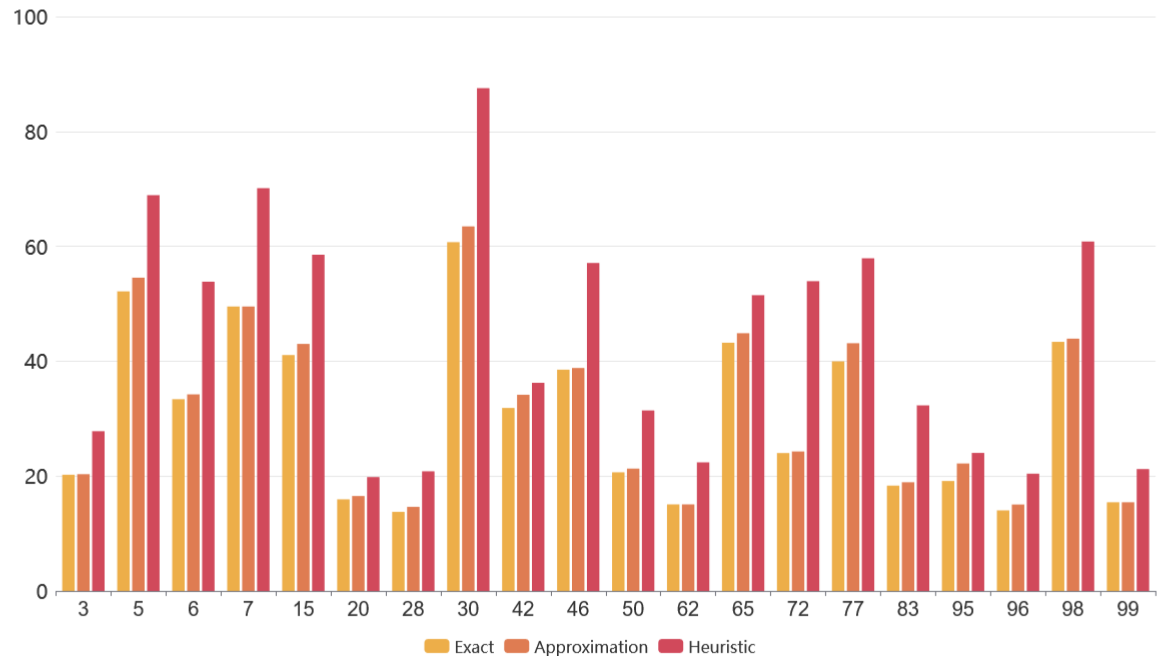

Figure 1: A comparison of the quality of the algorithms. The x-axis indicates the graph number (ID) and the y-axis indicates the solution (sum of all weights of vertices in the FVS).

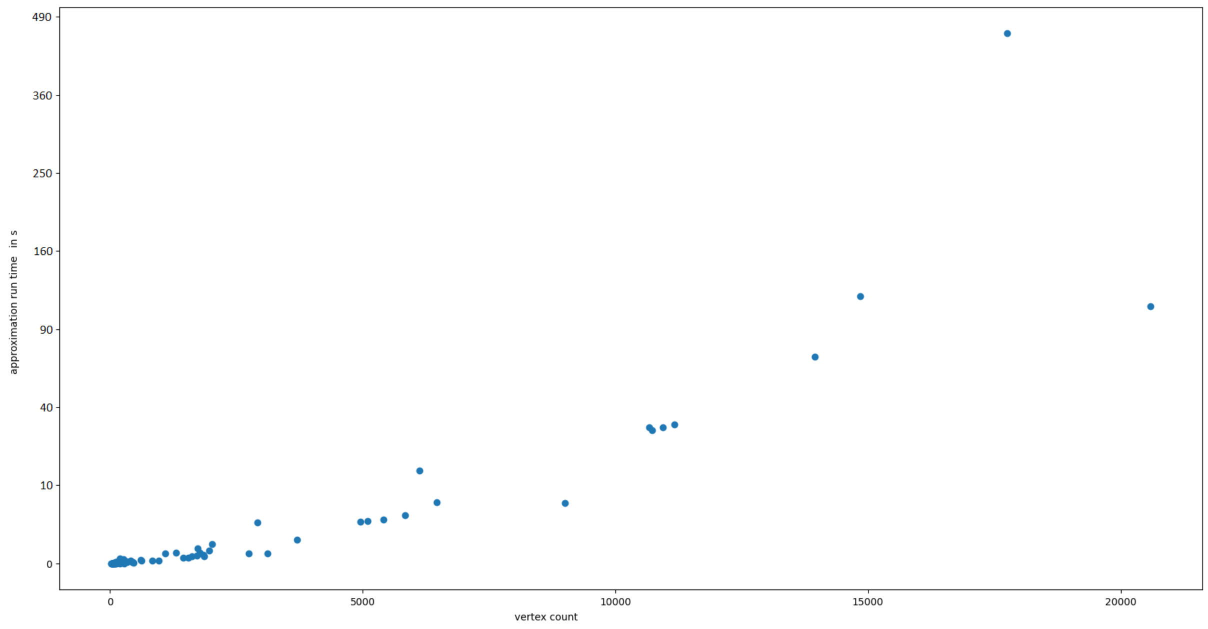

Figure 2: A plot that shows the correlation between the run time of the approximation algorithm and the number of vertices. Note that the axis of the approximation algorithm run time is square-root scaled.

Above in figure 2 you can see the run times of the approximation algorithm in perspective with the number of vertices. Remember that the asymptotic run time of the approximation algorithm was given by $\mathcal{O}(n^2 + nm)$. In the figure you really can see how the run time scales with the square of number of vertices (note that the run time axis has a square-root scale). That means our asymptotic run time seems to hold in practice.

On the right in figure 3 you can see the the run times of the heuristic algorithm with the corresponding number of vertices. Just like the approximation algorithm the heuristic algorithm also has a asymptotic run time of $\mathcal{O}(n^2 + nm)$. In the figure you can see how the run times scale with the square of vertices, since the figure again has a square-root scale for the number of vertices. That means again the asymptotic run time seem to hold in practice.

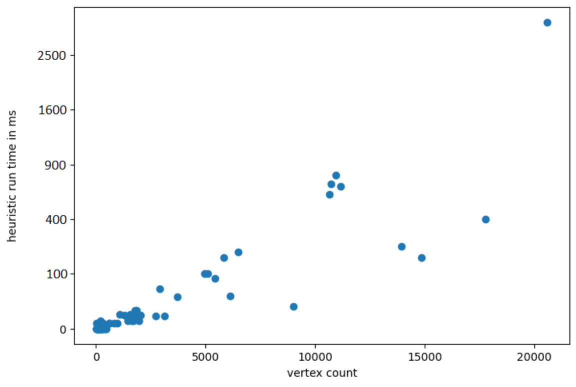

Figure 3: A plot that shows the correlation between the run time of the heuristic algorithm and the number of vertices. Note that the axis of the heuristic algorithm run time is square-root scaled.

The figure 4 to the right gives us insight of the relation between the run times of the exact algorithm and the number of fundamental cycles.
As you can see the run time axis is logarithmic scaled, since as we discussed in section 3 we now have a search tree height that scales with the number of cycles.
Because of that we now have a asymptotic run time of $\mathcal{O}(k^c \cdot (n+m))$ where k is a constant.
That scaling is exponential because the search tree grows exponential in height.
The plot verifies that this exponential growth is also visible in practice.
The few outstanding points are probably caused by the kernelization that we used.
More to the kernelization later.

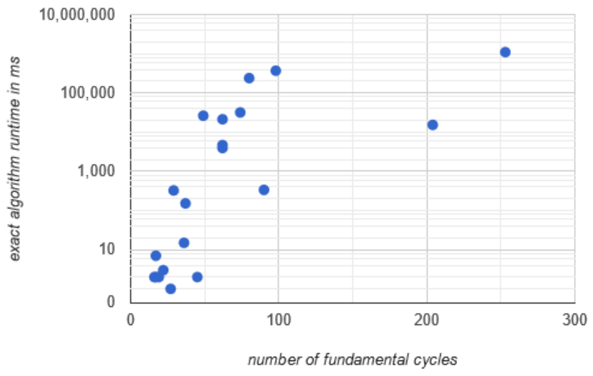

Figure 4: This plot shows the correlation between the run time of the exact algorithm (16 threads used) and the number of fundamental cycles. Note that the axis of the run time is logarithmic scaled.

As you can see in figure 5 to the right there is a big impact by using kernelization. Concretely kernelization has average speedup of 209.27%, but it has many up and downs, depending on the graph. The best speedup was achieved in graph 15 with 565.9% and the worst speedup was achieved in graph 77 with 36.97%.
The cause of these spikes can not be told that easily. Both graphs have a seemingly equal density of 0.0259 (graph 15) and 0.0254 (graph 77), as well as the same vertex count of 118 (graph 15) and 113 (graph 77) and cycle count with 62 (graph 15) and 49 (graph 76). Still they have very different speedup achievements.
There are many random factors such as how fast the BST boundary is updated or how many disjoint cycles are exposed during the procedure, that have a huge impact on run time.
Still we can verify that using kernelization can be very helpful in practice since the run times definetly were improved.

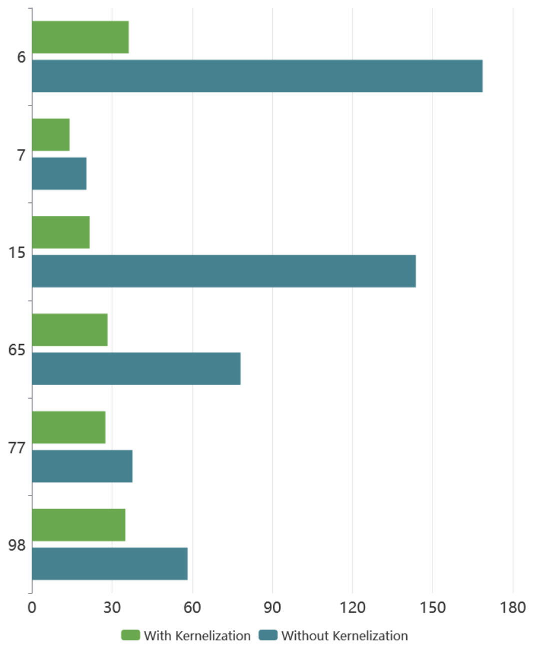

Figure 5: This chart shows the impact using kernelization. The x-axis shows the run time in seconds and the y-axis shows the respective graph number. Both used a non-parallelized version of the exact algorithm for that test, one with kernelization and the other without.

The figure 6 below shows the run times of the exact algorithm on 6 graphs depending on the number of threads used. In general the maximum speedup occurs somewhere between 8 to 16 threads, even tho in this area there is no major difference anymore. This is probably caused by the CPU having 8 physical cores. Even tho hyper-threading should speed up even more there are still limitations like shared memory that play a huge role.
The average speedup between 1 thread and 16 threads is 545.54% with even a spike of 1139.23% in graph 65. The lowest speedup was achieved in graph 77 with a speedup of 133.88%. With these statistics we can say that the parallelization is in practice very successful and helpful.

Finally we compare the heuristic algorithm and the approximation algorithm, since both algorithms ran on all 100 graphs. In the figure 7 you can see their respective run times. As already discussed both run times scale with the quadratic number of vertices. Still there is a big difference between both algorithms. In average the heuristic algorithm is 73.13 times faster than the approximation algorithm.
This is probably caused by the fact that the heuristic algorithm is guaranteeing that every iteration we remove c cycles where c is the number of fundamental cycles currently in the graph. The approximation algorithm only guarantees to remove 1 vertex per round, what mostly is a lot worse.
But on the other hand the approximation has better results as you can see in figure 8. In average the approximation algorithm gave us a solution that was 2.56 times better than the solution of the heuristic algorithm. The biggest difference was achieved in graph 57, where the solution of the approximation algorithm was 52.82 times better. The approximation algorithm was only worse than the heuristic algorithm for 2 graphs.

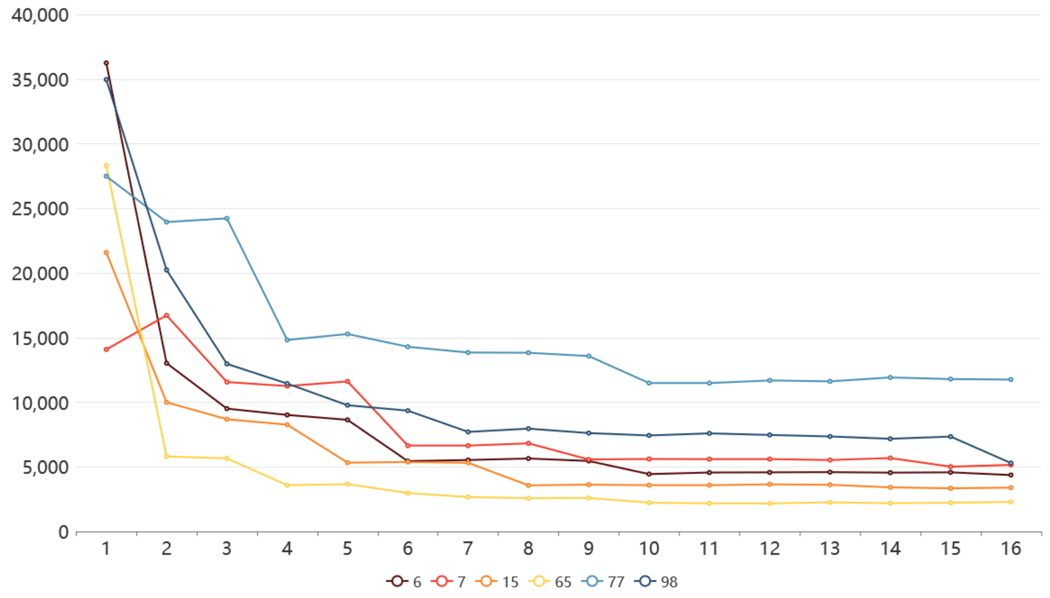

Figure 6: This line chart shows the run times of the exact algorithm for 6 graphs depending on the number of threads used (1-16). The y-axis indicates the run time in milliseconds and the x-axis indicates the number of threads. You can see a legend of which color indicates which graph under the line chart. Kernelization was used for these tests.

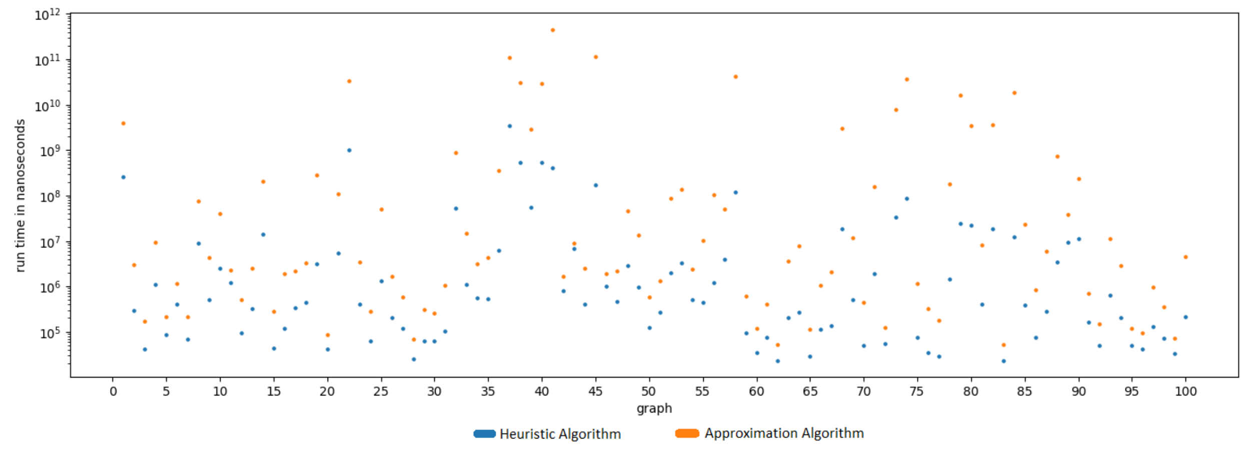

Figure 7: This scatter plot shows the run time for all graphs solved by the heuristic algorithm (blue) and the approximation algorithm (orange). Note that the run time axis has a logarithmic scale.

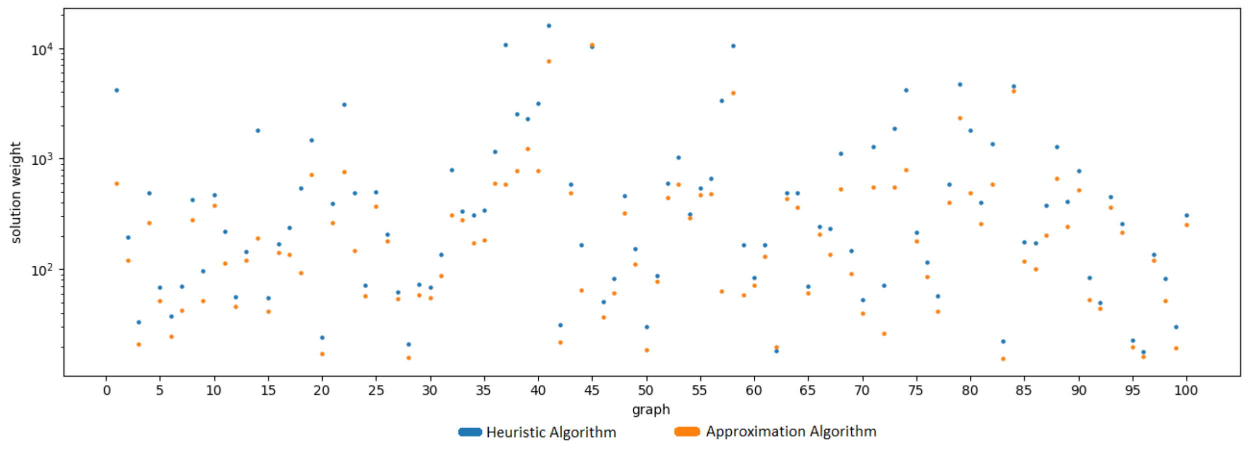

Figure 8: This scatter plot solution weight for all graphs solved by the heuristic algorithm (blue) and the approximation algorithm (orange). Note that the solution weight axis has a logarithmic scale.

5 Discussion and Conclusion

In this work we implemented three different algorithms for the Feedback Vertex Set, discussed and analyzed improvements and finally compared their practical performance. We showed that there are really good improvements for our exact algorithm. Still it has a exponential growth since the problem is NP-Complete. For smaller instances (in example vertex count of 28) the exact algorithm can solve all instances in a feasible amount of time.
All instances that are bigger can be solved by the exact algorithm if kernelization is able to reduce the problem dramatically. But generally we can make use of the 2 other algorithms that do really well on bigger instances.
The heuristic algorithm is as expected the fastest algorithm but has no guarantee how well the solution is compared to the optimum. Still if it is important to solve big instances in a small amount of time it definitely is recommended to use the heuristic algorithm.
The approximation algorithm (by Bafna-Berman-Fujito) meets both algorithms half way. Its not the fastest, neither the most accurate. Still it has a really good approximation factor of 2 which does even better in practice (approximately 4% worse than the optimum in average). Run time wise it still is pretty good since it scales with the squared number of vertices, but is a lot slower than the heuristic algorithm on bigger instances.

References

[Vin99] Toshihiro Fujito Vineet Bafna Piotr Berman. "A 2-Approximation Algorithm for the Undirected Feedback Vertex Set Problem". In: (1999). DOI: `https://doi.org/10.1137/S0895480196305124`.

[PAC] PACE. *PACE 2016 – Track B: Feedback Vertex Set.* URL: `https://pacechallenge.org/2016/feedback-vertex-set/`.