Algorithm Engineering
WS 22/23

Arsen Ukrainets
Kevin Kusche

Solving the Vertex Cover Problem with a 2-APX and an FPT Algorithm

Abstract

We implement a 2-Approximation Algorithm and an exact Algorithm to compute a minimal vertex cover. First we explain the greedy maximal matching algorithm and the Bounded Search Tree method, then we demonstrate how these two methods are used for an approximative and an exact solution of the vertex cover problem. After that we discuss the implementation of the latter two algorithms and ways to improve them. In the experiments we compare the running times of our algorithms and evaluate the approximation quality on different graphs. The results show us, that the approximate solution is significantly faster but also not much more accurate than the theoretical bound. We also observe, that the practical running time can be improved dramatically with a few optimizations.

1 Introduction

The problem of finding a minimal vertex cover for a given graph is known to be NP-complete. A brute force approach takes time exponential in the number of vertices, which renders it impractical for any big network. However, there are ways to speed the computation of such a vertex cover, despite the hardness of the problem.
This is what we want to take a look at in this report. We first cover a greedy algorithm which computes a maximal matching and the general technique of the Bounded Search Tree. We then show how these methods can be employed to produce a 2-Approximation of the optimal vertex cover and an exact algorithm, which is fixed-parameter tractable. We implement both of these algorithms and compare their running times and evaluate the approximation quality of the approximative algorithm on different graphs. We also compare the running times of an improved version of the exact algorithm to the original one. Through these comparisons we observe that the 2-APX algorithm is dramatically faster than the exact algorithm, but its solutions are only slightly smaller than twice as big. Furthermore we see that the improved algorithm is indeed significantly faster than the unimproved version.
The remainder of this work is structured as follows. Section 2 explains the maximal matching algorithm and the Bounded Search Tree method. Section 3 deals with a 2-Approximation algorithm and the application of the BST method for an exact solution to the vertex cover problem and their implementation, as well as further improvements to the exact algorithm. In Section 4, we describe the experimental setup and provide empirical results. Finally, in Section 5, we discuss and interpret these results.

2 Preliminaries

The problem can be described as follows: Given an undirected multigraph $G(V, E)$, compute $C \subseteq V$ so that $|C|$ is minimal and for every edge $\in E$, there is at least one incident vertex $\in C$. We can approximate the solution using a 2-approximation algorithm based on a maximal matching, but we also look at an exact algorithm, which employs the Bounded Search Tree (BST) method. Because of that, we explain the two underlying techniques, an algorithm for maximal matching and the BST method, in the following subsections.

2.1 Greedy Maximal Matching

First of all, a matching is a subset $M \subseteq E$ so that $\forall e_1, e_2 \in M, e_1 \neq e_2 : e_1 \cap e_2 = \emptyset$. A maximal matching is a matching which can not be extended with another edge $\in E$ without violating the property of a matching.
The algorithm computes such a maximal matching by taking a random edge $\{v, w\}$ adding it to M and removing every edge incident to v or w from the graph. It then selects another random edge from the remaining graph and so on, until there is no more edge left. Since we always delete all the edges incident to the endpoints, we can never have edges with common endpoints in our solution, which renders it a matching. And because we stop only when we can not add any more edges, it is a maximal matching.
The running time of this algorithm is in $\mathcal{O}(n + m)$, as we examine each node and each edge of the graph at most twice.

2.2 Bounded Search Tree method

The Bounded Search Tree works by first identifying parts of the input, for which at least one element needs to be included in the solution. For every possible element, there is a new subtree, which examines all possible solutions if we include this element. For each subtree we again need to find such a subset and build new subtrees. This proceeds until either the input is exhausted, or the depth of the tree reaches k, which is the parameter bounding the size of the search tree.
To then find a solution using this method, we utilize DFS to traverse the BST. We stop once a solution has been found or the whole tree has been traversed. Since the tree has depth $\leq k$, it contains vertices in $\mathcal{O}(b^k)$, where b is the number of subtrees emerging from each vertex. Therefore, as long as the running time for each vertex is polynomial in the input size and b is a constant, the overall running time is exponential only in the parameter k and polynomial in the input size. Such an algorithm is called fixed-parameter tractable (FPT).

3 Algorithm & Implementation

After explaining the techniques used for our algorithms, we now cover these particular algorithms and describe, how they utilize these underlying concepts. After that we look at the implementation details and ways to further improve the practical running time of the algorithm.

3.1 2-APX algorithm

The algorithm starts by computing a maximal matching as described in subsection 2.1. To turn this maximal matching into an approximation of the vertex cover problem, we add both incident vertices of every edge $\in M$ to C. Since M is a maximal matching, every edge of the graph is incident to at least one $v \in C$. Furthermore it is a 2-approximation, because to cover an edge, at least one of its endpoints needs to be included in the cover. Therefore, $|C| \geq |M|$, so by including $2|M|$ vertices, we get a solution which is at most twice as big as the optimal solution.
The running time is unchanged and stays in $\mathcal{O}(n+m)$.

3.2 BST for vertex cover

For this particular problem the Bounded Search Tree method concerns itself with the decision version of the vertex cover problem. I.e. does a vertex cover of size k exist? We build a search tree by selecting a part of the input for which at least one element needs to be included in the solution. As already discussed in 3.1, every edge of the graph fulfills this property, as at least one of its endpoints must be part of a vertex cover. Therefore there are two subtrees, in one the first endpoint is included, in the other the second one. We then remove the incident edges to the respective vertex from the graph, because they no longer need to be covered. However, choosing e.g. the first vertex at this point does not mean the second vertex can not be included further down in the subtree.
By traversing the tree with DFS, we either find a cover if, after removing the incident edges of a node, there are no edges left in the graph, or we do not find a solution and answer the question with no. In the latter case we need to traverse the whole tree, while

in the former we can abort once we find a cover.
Since the tree has depth $\leq k$ and every vertex has two children, it contains $\mathcal{O}(2^k)$ vertices. In each vertex we remove the incident edges of the 2 selected vertices in the graph G, which is in $\mathcal{O}(m)$. This means we end up with a running time in $\mathcal{O}(m \cdot 2^k)$. Therefore the algorithm is fixed-parameter tractable, as its running time is polynomial in the input size and exponential only in the size of the parameter k.
To find the minimal vertex cover we start by running the 2-APX algorithm. The obtained vertex cover is larger or equal to the optimal one. We then use its size as the first k for our exact algorithm. We then decrease k until the algorithm returns no cover. In that case we know, that the previously found vertex cover was minimal.

3.3 Implementation

We implemented the algorithms in Java.
Our Graph class contains an array which contains all the nodes of the graph, where the ID of the node corresponds to its index in the array. Additionally it contains a TreeSet of all the remaining nodes of the graph. The elements inside the TreeSet are ordered non-ascendingly with respect to their degree. The remaining nodes contain only nodes with degree > 0, because they are only added, once they appear in some edge. Every Node has an ID and a Set of neighbors. We used a LinkedHashSet for fast removal and insertion of neighbors, because this happens a lot in the algorithm. Also notice, that by using a set a potential multigraph is turned into a graph without parallel edges.
The 2-APX algorithm is implemented in a slightly different way than described in subsection 3.1, since the edges are not stored explicitly in our graph structure. So instead the algorithm goes through all the nodes of the graph and for every node which has not been removed yet, it removes the node itself and one of its neighbors.
The remove method of the graph removes the respective node from the graph and its incident edges by removing the node from the neighbor sets of its neighbors. Additionally it removes those neighbors, which now have a degree of 0, as they are no longer interesting for a vertex cover.
The exact algorithm uses an upper bound for k to repeatedly call the BST approach as described in subsection 3.2. The application of the BST algorithm has been improved however. Instead of selecting an edge and including either one of the endpoints, it selects a node and removes either the node itself or all of its neighbors. This is still correct, since the only way to cover all the edges incident to a node, is to include that node (endpoint 1 of the edges) or all of its neighbors (endpoint 2 of the edges). This way we remove a bigger part of the graph with each step which improves the running time.
To further increase the removed part we always select the node with the highest degree, which is why the TreeSet of remaining nodes is sorted this way. This has the additional benefit that if the node we extract from the TreeSet has degree at most 2, then every node of the graph does. And if that is the case, the graph contains only cycles and paths, for which the vertex cover problem can be solved in linear time. So we do exactly that, if we detect it.
One last difference compared to the theoretical algorithm is, that instead of always calling the algorithm with a new copy of the graph, we constantly use the same graph. To make this work we remember the nodes we remove from the graph before a recursive call and reinsert them after it. This saves a lot of time compared to copying the whole graph every time. The reinsert method reinserts the node into the graph and into the neighbor sets of the neighbors the node previously had. This works because the removed nodes still retain their neighbors they had when they were removed. Since this might change depending on which nodes were removed when, we need to reinsert the nodes in the opposite order of removal.

3.4 Further Improvements

To further improve the practical running time of the exact algorithm, we devised additional changes to the implementation discussed in subsection 3.3.
First, if the node with highest degree has a degree higher than k minus the size of the current cover, then we definitely need to include this node. Other-

wise we would have to include all of its neighbors, but doing that would exceed the size limit of the cover.
In every step we now also look at the node with the lowest degree. If this minimum degree is already bigger than k minus the size of the cover up to this point, we can already abort, since we need to include at least as many nodes as the minimum degree. If this is not the case and the minimum degree is exactly the number of remaining nodes - 1, then the remaining graph is a complete graph and we can include every node except one in the cover and abort.
The last and probably most important observation concerning the minimum degree is, if we have a node with degree 1, we can always include its neighbor, because its neighbor covers the one edge incident to the node and potentially more edges, so it is never worse to include the neighbor. Therefore, if the node with the lowest degree has degree 1, we do just that. And because we always delete nodes with degree 0 from the graph, we can always find a node with degree 1 by getting the node with lowest degree, if there is one.
Using this last observation we also notice, that if we remove all vertices of the graph with degree 1 at the beginning of each call to the decision version of the algorithm, we do the work multiple times. So instead we can preprocess a cover to start with, containing nodes which need to be included no matter the k, and remove them from the graph. Therefore we first look at all the nodes. We ignore nodes which are null or have degree 0. For every node with degree 1, we remove its neighbor and add it to the cover. For every node which contains itself in its set of neighbors, i.e. there is a loop, we include the node itself, as no other node can cover that edge. The preprocess method also takes a maximum k so for each node with degree higher than this maximum k minus the size of the current cover, the node is also included in the cover. Finally we repeatedly look at the node with minimum degree and remove its neighbor, if the degree is 1, until there are no nodes left which have degree 1. This last step is not redundant, since some nodes which previously had a higher degree, might now have degree 1 after removing other nodes.
We then used this preprocessed cover to improve the approximation, by only running the 2-APX algorithm on the remaining graph and adding the result to the preprocessed cover.

4 Experimental Evaluation

In this section we evaluate our two algorithms by running them on the 200 PACE 2019 benchmark graphs. We compare the running times and look at the ratio of the result sizes to evaluate the approximation quality. To confirm that the resulting node sets are indeed valid vertex covers, we created another graph class for test purposes, in which every node contains its incident edges and there is a set of all the remaining edges. We then iterate through the potential vertex cover and remove all the incident edges of every node, one after the other. If at the end the set of remaining edges is empty, the tested node set is a valid vertex cover, otherwise it is not.
Additionally, to make it feasible to run the algorithm on all the instances, we implemented a maximum k of 4000, which is added to the size of the preprocessed cover and a time limit of 4 minutes.

4.1 Data and Hardware

The tests were conducted on an AMD Ryzen 5 4500U with 2.38GHz, 6 cores and a RAM size of 16GB.

4.2 Results

We measured the running times of the 2-APX algorithm and the improved version of the exact algorithm. The results are shown in Figure 1. The red dots mark instances for which the exact algorithm could not find a solution before the time ran out. We can observe, that there are some instances, for which the exact algorithm has a running time comparable to the approximate one, but for the majority, the approximate algorithm is significantly faster. The exact algorithm was only able to solve 15 of the 200 instances in $\leq$ 4 minutes, while the APX algorithm took at worst about 30 ms, but mostly from 0 to 15 ms.
However, the 2-APX algorithm obviously has the disadvantage of only approximating the result. To find out about the quality of this approximation, we also

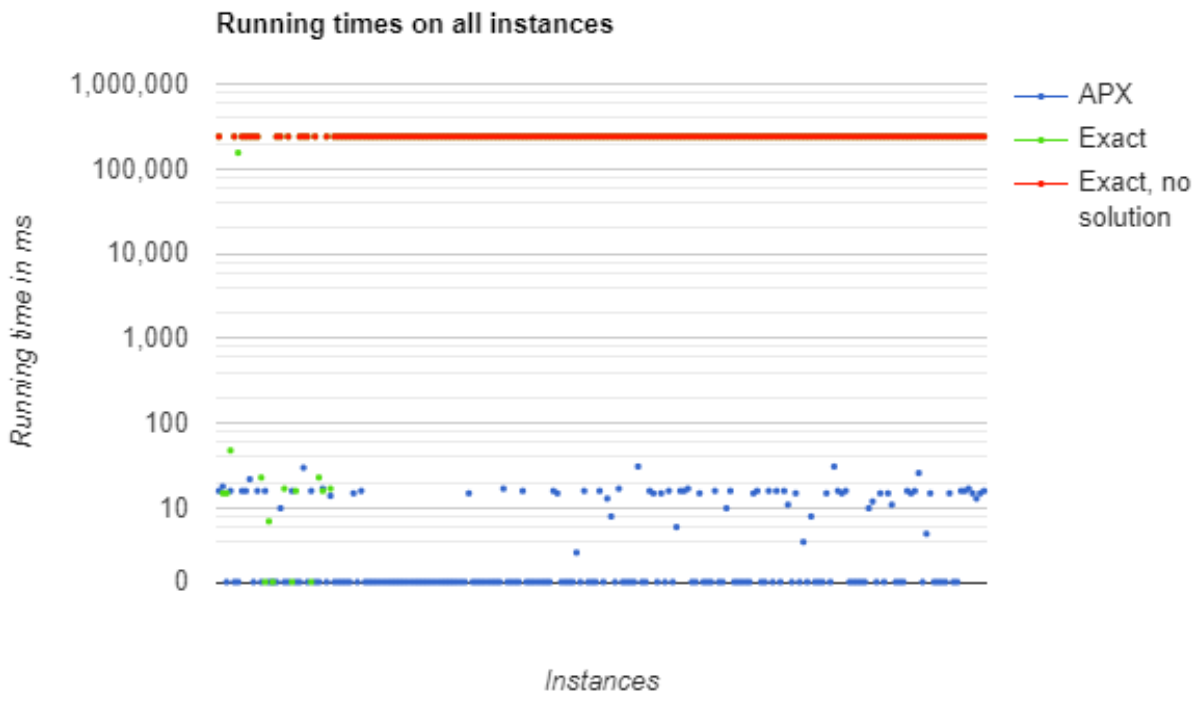

Figure 1: Comparison of the running times of the 2-APX and the exact algorithm

measured the size of the computed cover of the exact and the approximate algorithm. For the 185 instances, where the exact algorithm did not find a solution, we can not judge the quality, so we only look at the other 15 instances. For those the average ratio between the approximate and the exact solution was 1.929 and the maximum ratio was 2.0. This tells us, that the worst case did happen for one of the instances, but even on average the quality was only slightly better than 2. Also, the 2-APX algorithm was never lucky enough to find the exact solution.
Finally, we also recorded the running times of the unimproved exact algorithm, to see the impact our improvements had on the running time. Figure 2 depicts the running times of both algorithms for the instances where at least one algorithm was able to compute a solution. We can see that for one instance, only the improved algorithm did find a solution. But for all the other instances, we also see a very big difference between the two versions. Disregarding the instance for which only the improved algorithm produced a result, the average running time of the unimproved algorithm was 2263.429 ms, while for the improved algorithm it was 14.071 ms.

5 Discussion and Conclusion

In this work we looked at how to solve the minimal vertex cover problem. To this end we implemented a 2-APX algorithm to first approximate the result and an exact algorithm which uses this result as an upper bound for the bounded search tree method. We saw that the exact algorithm was only able to compute a solution for 15 of the 200 instances in under 4 minutes, while, the 2-APX algorithm was much faster, solving every instance in at most 31 ms. This is not surprising, as the approximative algorithm has a linear running time, while the exact algorithm has a running time, which is linear in the size of the graph, but exponential in k. The instances, for which the exact algorithm had a running time comparable to the 2-APX algorithm, most likely did not need many steps until the linear time algorithm could be applied. We also observed that the approximation quality of the approximative algorithm was on average only slightly better than the theoretical guarantee.
Lastly we did see that our improvements had a large impact on the running time. The improved algorithm was, on average, about 160 times faster than the unimproved algorithm and was able to solve one additional instance. This shows that, although we did not improve the theoretical running time, we were able to drastically reduce the practical running time.

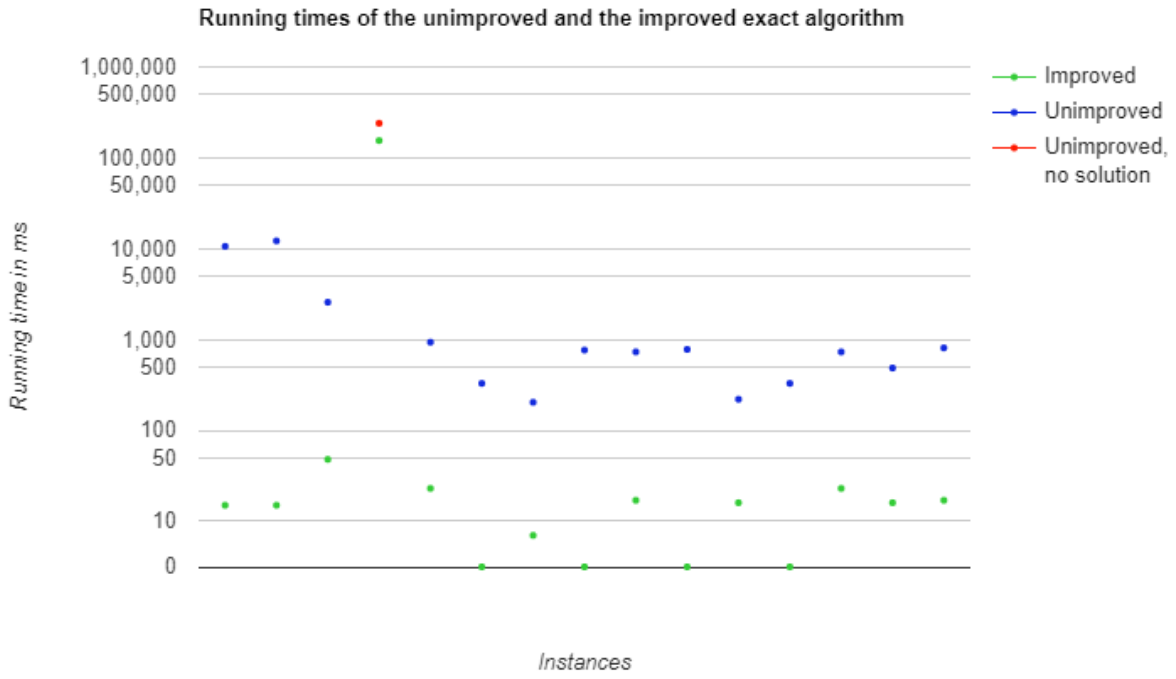

Figure 2: Comparison of the running times of the unimproved and the improved version of the exact algorithm