Algorithm Engineering
WS 22/23

Jakob Sanowski
Fiona Riesterer

Comparison of exact and 2-approximation Algorithm for Vertexcover-Problem

Abstract

We implemented and analyzed the running times of a 2-approximation and a exact algorithm for finding a minimal vertex cover of a graph. First we explain the basic algorithms and their implementations, then we look at improvements we made to the exact algorithm and finally look at further possible improvements. We tested the algorithms on 200 different graphs of different size. The results show finding a minimal vertex cover is very expensive.

1 Introduction

A vertex cover has a number of applications especially in finding ideal places for example for ATMs. In this report, we analyse the difference between a 2-approximation and an exact algorithm for calculating a minimal vertex cover. These algorithms are max-matching and a simple Boundary Search Tree. The remainder of this work is structured as follows. Section 2 will explain the two basic algorithms, while Section 3 provides information about the used algorithm engineering techniques and gives some implementation details. Furthermore we look at some more possible improvements we didn't use. In Section 4, we describe the experimental setup and provide empirically results. Finally, in Section 5, we discuss the outcomes and draw further conclusions.

2 Preliminaries

We consider the following problem: Given a -not necessarily simple- graph $G = (V, E)$. We'd like to get a minimal set of vertices C so that $\forall e \in E : e \cap C \neq \emptyset$. In other words, every edge of G should be represented by at least one vertex in C. To solve this problem, we consider two algorithms. One is a 2-approximation algorithm 2.1 that actually finds a maximal matching and the other one is an exact algorithm using Simple Bounded Search Trees 2.2.

2.1 2-Approximation Algorithm

The 2-approxiamtion algorithm is actually computing a matching of a graph $G = (V, E)$, but this yields a valid vertex cover of at most double the size of the optimal. The algorithm works as follows: it takes any $e \in E$ and adds the vertices of e to the cover C. Then it deletes all incident edges to e. Do this as long as there are edges left. The pseudocode is shown in algorithm 1. The algorithm has a running time of $O(|V| + |E|)$ if the graph is given as adjacency list.

Algorithm 1: 2-approximation Algorithm

```
Input: Graph $G = (V, E)$
Output: Vertexcover $C$
begin
    while $E \neq \emptyset$ do
        $e \leftarrow x \in E$
        $C \leftarrow C \cup \{e\}$
        $G = (V \setminus \{e\}, E \setminus (E \cap \{e\})$
```

2.2 Simple Bounded Search Trees

The Simple BST algorithm for Vertexcover is an exact algorithm to solve this problem. Given a graph $G = (V, E)$ and a number k, it tries to find a vertex cover C with $|C| \leq k$ vertices. It therefore is taking any edge $\{v, w\} = e \in E$ and first tries to compute a Cover with $v \in C$. To get the next vertex, it removes all the incident edges of v and finds the next vertex in the modified graph. If it couldn't compute a vertex cover, it tracks back and tries with the right hand side of e. This computes a tree structure with at most k levels, which leads to a running time of $O(|E| \cdot 2^k$. The pseudocode is in algorithm 2.

Algorithm 2: BST-VC

```
Input: Graph $G = (V, E)$, $k$, Cover $C$
Output: Vertexcover $C$ with $|C| \leq k$
begin
    if $|C| = kE \neq \emptyset$ then
        return $\emptyset$
    if $E = \emptyset$ then
        return $C$
    $e \leftarrow x \in E$
    $G\prime \leftarrow G$ with node $v$ and it's incident
      edges removed
    $C\prime \leftarrow C \cup \{v\}$
    if BST-VC($G\prime, C\prime, k$)$\neq \emptyset$ then
        return $C\prime$
    $G\prime\prime \leftarrow G$ with node $w$ and it's incident
      edges removed
    $C\prime\prime \leftarrow C \cup \{w\}$
    if BST-VC($G\prime\prime, C\prime\prime, k$)$\neq \emptyset$ then
        return $C\prime\prime$
```

3 Algorithm & Implementation

Since the Simple BST algorithm results in a long runningtime for big graphs and high value for k, we have to find a suitable implementation. What we improved can be found in section 3.1 and how we actually implemented it in section 3.2.

3.1 Algorithm Engineering

Since it is a recursive algorithm, we do not want to pass a copy of the graph or the cover each time we call the method. So instead, we will use static variables for that and keep an extra stack of the removed edges of the graph to add them again to the graph if we are backtracking. This helps us preventing a stackoverflow.
Now, we have to come up with an efficient method for finding and removing edges. This depends on the graph data structure we use. We decided to use adjacency list. If we would use an edge list, finding an edge would be very efficient, but deleting all incident edges would take long time. For adjacency matrix, finding an edge takes long time, but deleting is very efficient. In an adjacency list, if vertices with no edges are removed from the graph, we can find an edge in $O(1)$ since the first vertex has to have a neighbor. Deleting all the incident edges in a non-directed graph, we would only have to delete the vertex and go through all the neighbors to delete said vertex. This is taking less time than in an edge list.
We also tried to use an iterative approach, but had the same problems, namely efficiently finding and removing an edge, so we decided to stick to the recursive one.

3.2 Implementation Details

We implemented the algorithms using Java 19. As described in 3.1, we used adjacency lists to store our graph. Each vertex has list of neighbors only containing the id's of that adjacent vertex. So in case of self loops, we prevent any problem with concurrent modifications.
Instead of copying and passing the graph and Cover along the recursive calls, we used static variables and reset the Cover to the one before going down left, to then go right down the tree.
To find an optimal Vertexcover we first computed the Vertex cover found by the 2-approximation algorithm and started simple BST with a k of half the size of that cover. If it couldn't find a cover, we try next with $k + 1$ until we found a solution which is then optimal.

3.3 Further Possible Improvements

There are numerous improvements which can be made to improve the running time of SimpleBST. One such improvement would be to remove all vertices from the graph which have a self loop and calculate a minimal vertex cover for the resulting graph. If we then just add the removed vertices to the found vertex cover we get a minimal vertex cover for the original graph. This works because vertices with self loops have to be in the vertex cover otherwise the loop would not be coverd.

Following the approach of reducing the problem size we could look at kernelization. Kernelization tries to reduce the instance to an equivalent instance with size bounded by $f(k)$. One such method is discribed on Slide 7 in SLIDES #6. When calculating the vertex cover with the kernel graph it is also possible to use a k with the number of vertices which have to be in the vertex cover subtracted.

4 Experimental Evaluation

In this section, the experimental setup is described and the results are presented.

4.1 Data and Hardware

The Max-Matching algorithm was tested on an AMD Ryzen 7 with a base clock of 1.8 GHz and a boost clock of up to 4.3 GHz. For SimpleBST we used an Intel i7 6700K with a clock speed of 4.2 GHz. Both algorithms were tested on the 200 PACE19 graphs which can be found here: `https://pacechallenge.org/2019/vc/` For SimpleBST we set a time limit of 3 minutes because otherwise it would take too long to test all 200 graphs. The 3 minute limit was chosen because it limits the maximum time of testing all 200 graphs to 10 hours.

4.2 Results

Figure 1 shows the results for the max-matching algorithm. As expected running time increases on graphs with more vertices. The two anomalies for graphs with 1000 to 1999 and 30000 to 39999 vertices correlate with the average number of edges for these graphs. (Figure 2)

Unfortunately SimpleBST didn't compute a solution for any graph. It either timed out or stoped with a stack overflow. A stack overflow occurred for all graphs where the maximum k computed by the max-matching algorithm was bigger than roughly 4200.

5 Discussion and Conclusion

In this work we implemented two algorithms for finding a vertex cover of a graph and reviewed their performance. We found the 2-approximation algorithm finds a solution in a few milliseconds and thus is very suitable when no exact solution is sufficient. Our implementation of a exact algorithm in contrast was not able to find a solution for any of the test graphs in a acceptable amount of time. Suffice it to say the SimpleBST algorithm can be improved in multiple ways, some of which we discussed in Section 3.3

6 References

The references list the external resources used in the work at hand. LaTeX offers special ways to list those resources. In this template the references are stored in the 'refs.bib' file and can be referenced with the '\cite{REF}' command, where REF is a label defined in the .bib file. This example shows how such a reference looks like: **?**.

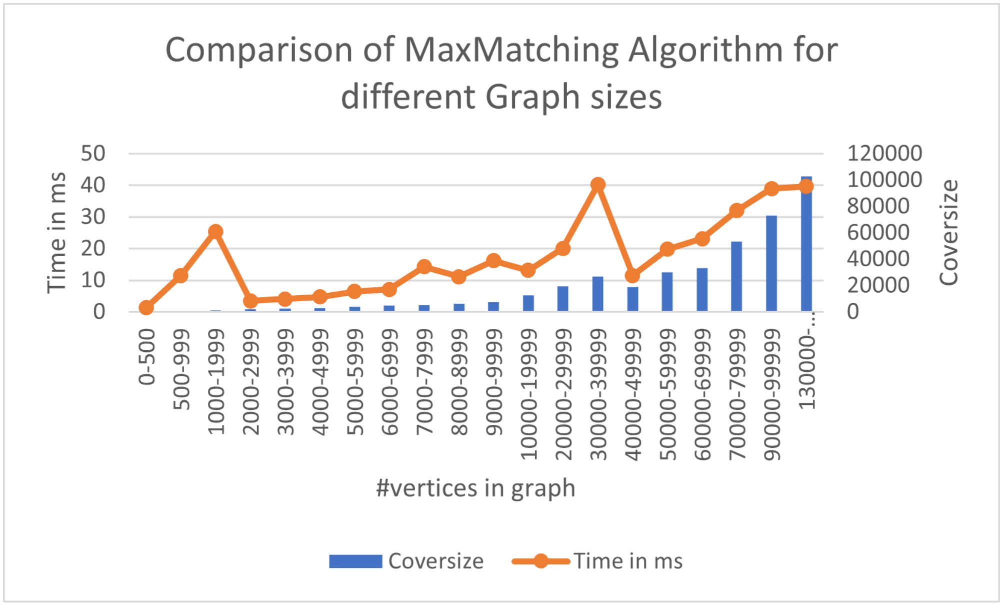

Figure 1: Average coversizes and running time for the 2-approximation Algorithm

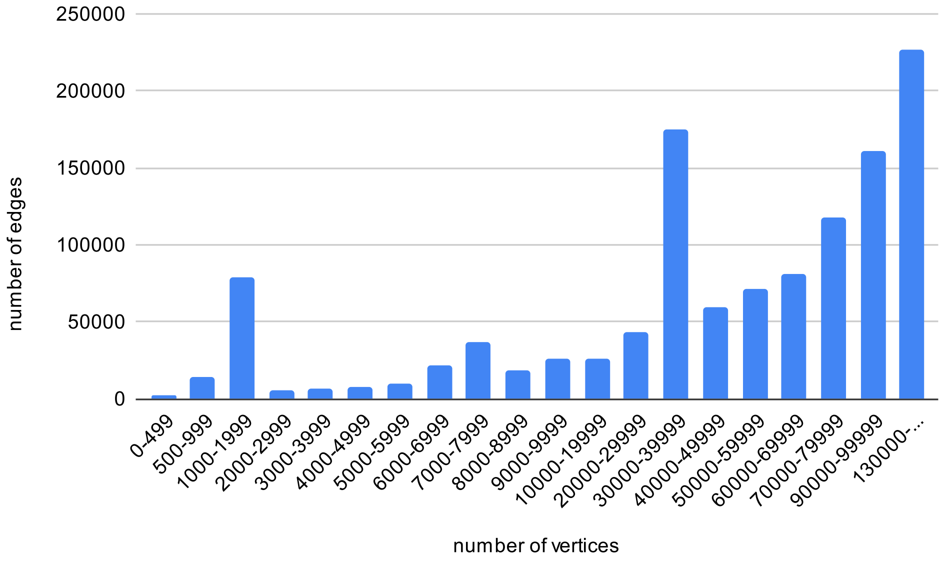

Figure 2: Average number of edges for all tested graphs