Algorithm Engineering
WS 22/23

Sanket Datta
Toru Tsukamoto

# Fixed-Parameter Tractable Algorithms

## Abstract

In order to solve the minimum vertex covering problem, the brute force algorithm, the 2-Approximation algorithm, and the BST algorithm were used. We also made the BST algorithm faster by taking advantage of the fact that either a vertex or its neighbors must be part of the solution.
These algorithms were implemented and it was confirmed that for relatively small graphs, the force-fed algorithm and BST compute exact solutions, while the 2-Approxmation algorithm finds approximate solutions that are at most twice as large as the exact solution.
These algorithms were used to try to find the PACE 2019 benchmark's minimum vertex coverage. 2-Approximation was able to get solutions for most problems. But neither the brute-force algorithm nor BST were able to solve these problems in a reasonable amount of time.

## 1 Introduction

By providing a computer with an appropriate algorithm, many significant issues can be resolved. Although an issue could be theoretically solved, it is of little benefit in practice if the computing complexity is unrealistic. If the computational complexity is determined not by a polynomial but by an exponential function or a function with weaker qualities, then the issue becomes difficult to solve if the number of components is just slightly bigger. It is true that a large variety of problems cannot be resolved in polynomial time.
Approximation is one strategy for solving these issues. The upper or lower limit of the answer is obtained after approximating the original issue to one that is simpler to solve. This enables the issue to be solved with practical, if not precise, precision.
Another solution to this problem is the fixed-parameter tractable algorithm. Using algorithms that can solve problems in polynomial time for the input size is an alternative strategy that often takes use of the problem's specificity. For parameters that reflect the problem's complexity, such methods are still slower than polynomial time. The issue can be solved for considerably higher input sizes, however, if the parameters are within a narrow range and the computational complexity grows nearly exclusively in polynomial time.
The minimal vertex cover problem for graphs, which is covered in more detail in the next section, is one issue that may be solved using this strategy.

## 2 Preliminaries

Finding the lowest set of vertices in a graph that contains any endpoint of all edges is known as the minimum vertex coverage problem.
First, there is the Brute Force Algorithm, a simple solution. In order of increasing number of elements, all subsets of the set of vertices are checked to see if they are vertex covers to determine the lowest vertex cover. Theoretically, this approach also produces an accurate solution. However, when searching for subsets with up to k elements, the computational cost is $O(n^k * m)$ if the input graph has n vertices and m edges. When the input data is huge, these algorithms are impractical because the computational complexity explodes.
It is well known that, in principle, no algorithm can solve the vertex cover issue in polynomial time. As a result, using approximations and parametric algorithms, practical solutions to this problem are experimented with in the parts that follow.

# 3 Algorithm & Implementation

The 2-approximation algorithm, Bounded Search Tree and its enhanced variant were implemented in this assignment.

## 3.1 2-Approximation Algorithm

The 2-approximation algorithm is first described as follows. The algorithm, depicted in pseudo code 1, is a recognized solution to the minimum vertex coverage problem. This algorithm selects an arbitrary edge and sequentially adds the vertices at its two ends to the list of vertex coverings. Once all edges are covered by vertices, it comes to an end.
Even if the edge is chosen well, this algorithm may not find the lowest number number of vertices to cover. For example, if the graph contains isolated pairs of two vertices connected by one edge, twice as many vertices as the minimum coverage are chosen. On the other hand, the minumum vertices cover (two vertices) are picked as its vertex cover in a graph with triangular edges. As the name implies, even in the worst-case this approach can find a vertex cover that has twice as many vertices as the minimal coverage.
For n vertices and m edges, the computational complexity of this 2 approximation algorithm is known to be O(n+m)

**Algorithm 1** 2-Approximation Algorithm

2-APX $(G, C, k)$
G:Graph,
E:Graph Edge Set,
C:Found Cover Vertex Set,
Skip covered edges instead of remove them

1: **while** $E' \neq \emptyset$ **do**
2: $e = \{v, w\} \leftarrow$ pick arbitrary edge from $E$
3: **if** $v \notin E$ **and** $w \notin E$ **then**
4: $C' \leftarrow C \cup \{v, w\}$
5: **end if**
6: **end while**
7: **return** $C$

## 3.2 Bounded Search Tree

The description of the Bounded Search Tree (hereinafter BST) follows. The size of the graph alone does not always indicate how complex it is. Even intuitively, it appears that a star with 1000 leaves is more complex than a conventional graph with 10 vertices. In actuality, a star with 1000 leaves has just one central point as its minimum vertex cover. In other words, it is possible to use the number of vertices constituting the minimum vertex cover as a parameter to determine how complicated the graph is. This number of points is called the parameter k in the rest of this section. As is explained below, BST is an algorithm that, when the number of vertices composing the minimum vertex cover (parameter k) is small, produces an exact solution for increasing input data in polynomial time.
In pseudocode 2, an overview of the BST algorithm is given: BST makes a binary tree with k levels whose paths are candidates for minimum vertex coverage and checks if each candidate actually does vertex coverage. The depth-first search by recursion, on the other hand, means that it is not necessary to build all of the huge binary trees in memory. In a naive BST, the simple binary tree is made by repeatedly picking random edges and including either of their two ends as candidates for vertex coverage. By doing this, other edges, including the chosen vertex, are taken out of the search. This makes the search much simpler to execute than a brute force algorithm.
The BST was also improved as follows. The set of vertices that make up the vertex cover must always have a vertex at each end of every edge. In other words, if a vertex is not in the vertex cover, then all other vertices in the neighborhood that are connected to it by an edge are in the vertex cover. This fact is used to branch a binary tree based on that whether or not that vertex is in the vertex cover, or whether or not all the neighborhood vertices are in the vertex cover. If this branching is done for nodes with a degree of 3 or higher, the search becomes more efficient because it can be decided at once if more than one point are candidates or not. In this case, branching was set up so that the nodes with the highest degree were chosen one after the other instead of

random vertices.The BST takes at most O(2k * m) time to run, on the other hand, improved BST is more efficient so that its time complexity is O(1.47k * poly(m)).
In our implementation, BST stops immediately as it finds a vertex cover. Even if a vertex cover is found, there is no guarantee that it is a minimum vertex cover. Instead, the 2-Approximate Algorithm is run before BST so that a lower limit on the number of vertices with minimum vertex coverage can be found. The lower limit is half the number of vertex cover vertices found by the 2-Approximate Algorithm. If the BST is run while increasing the parameter k one at a time from this lower limit, the minimum vertex cover is the first vertex cover that is found.

**Algorithm 2** Bounded Search Tree

BST $(G, C, k)$
G:Graph,
E:Graph Edge Set,
C:Found Cover Vertex Set,
k:Assumed Vertex Size Parameter

1: **if** $|C| = k$ **and** $E \neq \emptyset$ **then**
2: **return** $\emptyset$
3: **end if**
4: **if** $E \neq \emptyset$ **then**
5: **return** $C$
6: **end if**
7: $e = \{v, w\} \leftarrow$ pick arbitrary edge from $E$
8: $G' \leftarrow G$ with node $v$ and its incident edges removed
9: $C' \leftarrow C \cup \{v\}$ with node $v$ and its incident edges removed
10: **if** BST$(G', C', k) \neq \emptyset$ **then**
11: **return** $C'$
12: **end if**
13: $G'' \leftarrow G$ with node $w$ and its incident edges removed
14: $C'' \leftarrow C \cup \{w\}$ with node $w$ and its incident edges removed
15: **if** BST$(G'', C'', k) \neq \emptyset$ **then**
16: **return** $C''$
17: **end if**

### 3.3 Graph Data Structure

As the graph data structure, an adjacency list was chosen. In other words, lists for every vertices, that are adjacent to each vertex is provided. This data structure was constructed in Python as a dictionary with lists of neighboring vertices and a vertex as the key. This adjacency list and the adjacency matrix, which represents whether there are edges between vertices as 0,1, are well-known graph data structures. Since we are dealing with a relatively large sparse graph, we decided to use the adjacency list, which is spatially efficient for sparse graphs.

## 4 Experimental Evaluation

### 4.1 Hardware

Table 1 shows the specifications of the computers used in the experiment.

Table 1: Computer Specifications

| | |
|---|---|
| Chip | Apple M1 Pro |
| Core | 8 CPU, 14GPU, 16 Neural Engine |
| Memory | LPDDR5 16GB |
| Storage | APPLE SSD |

### 4.2 Preliminary Tests

First, as a preliminary test, minimum vertex coverings for the small graphs shown in Fig.1 were searched for to make sure that the four algorithms shown below worked correctly.(VC: Number of Vertex Cover Set, BF: Brute Force, 2-APX: 2-Approximation Algorithm, BST: Bounded Search Tree, iBST: Improved Bounded Search Tree.)

### 4.3 Data

As a benchmark for the implemented algorithms, the Parameterised Algorithms and Computational Experiments 2019 (PACE 2019) (https://pacechallenge.org/2019 ) benchmark was

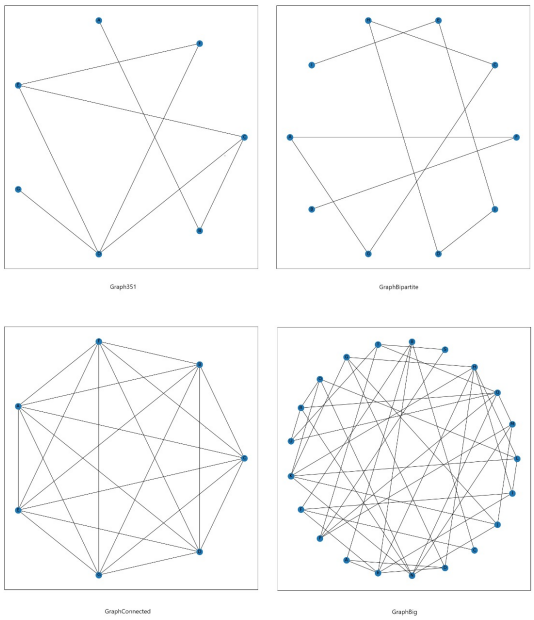

Figure 1: Graphs for preliminary tests

Table 2: Vertex Cover

| Graph | BF | 2-APX | BST | iBST |
|---|---|---|---|---|
| 351 | 3 | 6 | 3 | 3 |
| Bipartite | 5 | 8 | 5 | 5 |
| Connected | 6 | 6 | 6 | 6 |
| Big | 13 | 16 | 13 | 13 |

used. This benchmark included 200 undirected graphs.

### 4.4 Results

In addition to the Brute Force Algorithm and the BST(even with the improved version), the computation could not be done in 60 seconds for all benchmarks.
On the other hand, Figure 2 shows that the 2-Approximation Algorithm could solve 81 benchmarks. Figure 3 shows how the size of the graph (the total number of vertices, edges, and edges) affects how much time it takes to solve the benchmarks. Figure 4 is a scatter plot that shows how the size of

Table 3: Time Consumption (ms)

| Graph | BF | 2-APX | BST | iBST |
|---|---|---|---|---|
| 351 | 0.997 | 0.995 | 1.997 | 0.998 |
| Bipartite | 17.0 | 0.996 | 7.98 | 7.02 |
| Connected | 4.99 | 0.0100 | 4.02 | 7.02 |
| Big | 48900 | 0.997 | 339 | 482 |

the graph affects the size of the minimum vertex coverage.

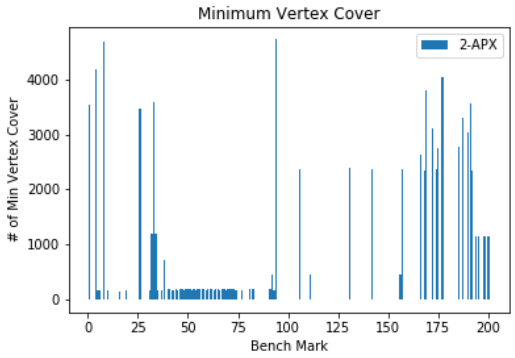


Figure 2: The minimum vertex cove of the benchmark found in 2-APX

## 5 Discussion and Conclusion

In preliminary tests, the Brute-Forced Algorithm and the BST both had the same number of minimal vertex cover. Additionally, as expected by theory, the 2-Approximation Algorithm's vertex count was less than twice that of the exact solution.
In the current studies, the 2-Approximation Algorithm was able to perform 81 benchmarks.As shown in Fig. 3, the execution time increased almost linearly with the total number of vertices and edges.
On the other hand, in addition to the brute-force algorithm, BST was not able to finish the computation within the time limit of 60 s for all benchmarks. In preliminary trials, BST took about 10 seconds if the vertex coverage solution depth was 10, and about 50 seconds if the vertex coverage solution depth was

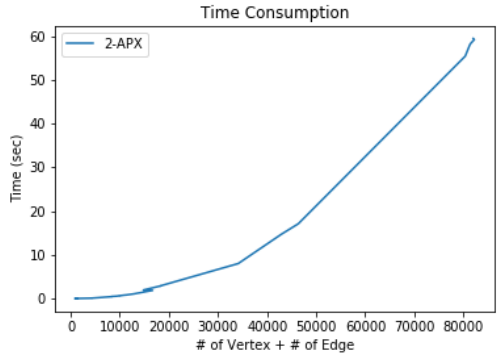


Figure 3: Time Consumption in sorted by size of graph

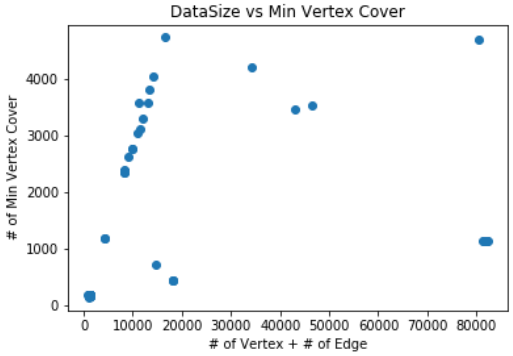


Figure 4: Relationship between the size of the graph and the size of the minimum vertex cover

12. The computational complexity of the improved BST is proportional to 1.47**k, and the results of the 2-Approximation Algorithm show that even for the simplest problems, the lower bound on the number of vertices for the minimum vertex coverage is 70 or more, which means that searching for a solution is impractical. Python recurses.
As shown in Figure 4, the minimum vertex coverage tends to be bigger the bigger the graph is. However, the minimum vertex coverage of some very large graphs is much smaller. In BST, it's easy to find the minimum vertex cover for these kinds of graphs.