Algorithm Engineering
WS 22/23

Fiona Riesterer

Comparing Algorithms for the Maximum Cut Problem

Abstract

We implement and compare four different algorithms to solve the maximum cut problem. We used an exact algorithm using an ILP formulation, a two-approximation algorithm, a heuristic and a parallel implementation of the heuristic. We first describe the basic algorithms, then describe how we implemented and improved the algorithms. In the experiments we compared the running times and the solution quality of these algorithms. The results show that the approximation algorithm has a higher solution quality than the heuristic, while the heuristic has a better running time.

1 Introduction

The maximum cut problem is a well known NP-complete graph problem. There exist many algorithms to solve this problem. There are ILP formulations to find an exact solution and greedy approximation algorithms. In this report, we focus on the practical running times and the solution quality of the different algorithms. Therefore we compared an exact algorithm using an ILP formulation [1], a two-approximation algorithm and a heuristic [2].

The report is structured as follows. First in section 2, we describe and explain the basic algorithms. In section 3 we describe our implementation of the algorithms and the improvements we made. Next, the experimental setup and the results are described in section 4. Finally in section 5, we discuss the results and summarize our work shortly.

2 Preliminaries

Given an undirected graph $G = (V, E)$, a cut is a distribution of vertices into two sets $(S, V \setminus S)$. A cut-edge $e = \{u, v\}$ is an edge between these two sets such that both vertices are in different sets, e.g. $u \in S$ and $v \in V \setminus S$. The maximum cut problem is the problem to find a cut such that the number of cut-edges is maximized.

2.1 Exact Algorithm

To find an exact solution, we use an ILP formulation. Every edge $\{u, v\} \in E$ is assigned a variable $e_{uv} \in \{0, 1\}$ and every vertex $u \in V$ is assigned a variable $x_u \in \{0, 1\}$. Goal is to maximize the number of edges with value 1. The whole ILP formulation looks as follows.

$$\begin{array}{lll}
\textbf{Maximize} & \sum_{\{u,v\}\in E} e_{u,v} & \\
\textbf{Constraints} & e_{uv} \in \{0,1\} & \forall \{u,v\} \in E \\
 & x_u \in \{0,1\} & \forall u \in V \\
 & e_{\{uv\}} \leq x_u + x_v & \forall \{u,v\} \in E \\
 & e_{\{uv\}} \leq 2 - (x_u + x_v) & \forall \{u,v\} \in E
\end{array}$$

The last two lines are making sure that an edge can only be counted as a cut-edge if the two vertices are on different sides of the cut, i.e. the two vertices are not assigned both 1 or both 0.
The problem will then be solved with an ILP-solver.

2.2 Approximation Algorithm

The algorithm starts with a random cut. For every vertex $v \in V$, it then switches the group, if it would improve the number of cut-edges. That is done until there are no more improvements. The pseudocode for this algorithm is shown in algorithm 1.

Algorithm 1: Approximation Algorithm

```
Input: Graph $G = (V, E)$
Output: Number of cut-edges
begin
    for $v \in V$ do
        $v \leftarrow$ random group
    while still improving do
        for $v \in V$ do
            if swapGroupIsBetter($v$) then
                swapGroup($v$)
    return numberOfCutEdges($G$)
```

This algorithm has an approximation ratio of two [2].

2.3 Heuristic

The Heuristic just randomly groups all the vertices. This is done a certain number of times. The best result is then picked as a cut. The pseudocode for this algorithm is shown in algorithm 2.

Algorithm 2: Heuristic

```
Input: Graph $G = (V, E)$
Output: Number of cut-edges
begin
    $max \leftarrow -1$
    $tries \leftarrow 100$
    while $tries \geq 1$ do
        for $v \in V$ do
            $v \leftarrow$ random group
        $numberCutEdges \leftarrow 0$
        for $e = \{u, v\} \in E$ do
            if $u.group \neq v.group$ then
                $numberCutEdges++$
        if $numberCutEdges > max$ then
            $max \leftarrow numberCutEdges$
        $tries--$
    return $max$
```

This algorithm has an expected solution value of two, i.e. producing a cut with at least half as many cut-edges as the optimal solution [2].

3 Algorithm & Implementation

This section provides information about the actually used algorithms and their respective implementations.

3.1 Implementation Details

We implemented the algorithms using Java 19. As graph data structure we used a vertex list, an edge list and an adjacency list. The adjacency list is only created if necessary. We used Arraylists to store the elements. Vertices have a unique identifier and a group identifier.

3.1.1 Exact Algorithm

As ILP-solver we used the choco-solver from chocolatey. For every edge and every vertex, we created variables using the edge and vertex list of the graph. We then added the constraints to the solver by adding two constraints for every edge, namely $e_{uv} \leq x_u + x_v$ and $e_{uv} \leq 2 - (x_u + x_v)$.
We also bounded the value of the number of cut-edges using the approximation algorithm. Since it is a two-approximation, at most double as much cut-edges can be in the optimal solution. Also, at least as many cut-edges found by the algorithm have to be in the solution. This adds a constraint to the ILP-formulation from section 2.1, forming the following ILP-formulation, where a is the number of cut-edges found by the approximation algorithm.

Maximize	$\sum_{\{u,v\}\in E} e_{u,v}$	
Constraints	$e_{uv} \in \{0, 1\}$	$\forall \{u, v\} \in E$
	$x_u \in \{0, 1\}$	$\forall u \in V$
	$e_{\{uv\}} \leq x_u + x_v$	$\forall \{u, v\} \in E$
	$e_{\{uv\}} \leq 2 - (x_u + x_v)$	$\forall \{u, v\} \in E$
	$\sum_{\{u,v\}\in E} e_{u,v} \in \{a, y\}$	$y = \min(\lvert E\rvert, 2 \cdot a)$

Creating all variables and constraints results in a running time of $O(n + m + z)$, where z is the running time of the ILP-solver. Since the maximum cut problem is a NP-complete problem, z is exponential in the input size.

3.1.2 Approximation Algorithm

We implemented the approximation algorithm exactly as described in section 2.2. We use a vertex list, an edge list and an adjacency list as data structures.

We start by randomly grouping the vertices to get a cut to start with. To do this, we use the Java class Random and assign a group to each vertex which is represented by either 0 or 1. For this step we use the vertex list, which results in a running time of $\theta(n)$.

We then start to improve the cut.
Every time we find a vertex v that has less cut-edges than non-cut-edges, we swap the group. To calculate this more efficiently than going through the whole edge list, we use the adjacency list for that. So the running time to decide if swapping would be better, would be in $\theta(\deg(v))$, which is expected to be smaller than $|E|$. Swapping groups can then be done in $O(1)$ since only two groups exist.

If we swapped a group, we set a variable to true, to indicate that there were still improvements to make. As soon as nothing got changed in a complete loop of the vertices, nothing can be improved anymore and the algorithm found a solution. Since the cut size can not be greater than $|E|$, improving the cut is done in at most $|E|$ steps [2].

To get the number of cut-edges, we just go through the edge list and increase a counter every time we come across an edge $e = \{u, v\}$, where u and v are in different groups. This results in a running time of $O(m)$, where $m = |E|$.

The overall running time is in
$O(n+m\cdot\left(\sum_{v\in V}\deg(v)\right)+m) \in O(n+m+m^2)$. This running time is probably too high for most instances, since we can expect the algorithm to find more than one vertex to improve in every iteration of the while loop.

3.1.3 Heuristic

We implemented the algorithm exactly as described in section 2.3.
We have set the number of tries to 100. In every try, we randomly group each vertex using the Random Class provided in Java. This step has a running time of $O(n)$, where $n = |V|$. We then calculate the number of cut-edges by increasing a counter every time we find an edge whose vertices are assigned to different groups. This step results in a running time in $O(m)$, where $m = |E|$.

The overall running time is $100\cdot(n+m) \in O(n+m)$.

3.1.4 Parallel Heuristic

In order to make the heuristic algorithm from section 2.3 more efficient, we parallelized grouping the vertices. Instead of grouping the vertices e.g. 100 times and only one at a time, we divide it by the number of processors. So we group the vertices $\#processors$ times in parallel, each doing $\lceil\frac{100}{\#processors}\rceil$ loops, and take the best result over all.

We can not group the vertices directly in the graph as we did for the heuristic algorithm, since we want to avoid copying the graph too many times. Instead we create Arrays in the size of $|V|$ for every instance. Every entry gets a random number, either 0 or 1 using the Java Random Class. The vertex identifier corresponds to the index of the Array.

To count the cut-edges, we simply go through the edge list and find the corresponding groups of the incident vertices by accessing the Array. Every time the vertices are in different groups, we increase a counter.

Every instance keeps track of the best result it found by updating a local variable if the new cut created a higher result. To find the best result over all instances, we use another Array in the size of the number of instances we created. Each instance is assigned a unique identifier, which is used as index to write into the results Array. As soon as all instances finished, we find the maximum in the results Array.

The pseudocode for an instance of the parallel heuristic is shown in algorithm 3.

Algorithm 3: Parallel Heuristic

```
Input: Integer Identifier
Output: Number of cut-edges
begin
    $id \leftarrow Identifier$
    $group \leftarrow int\,[|V|]$
    $max \leftarrow -1$
    $tries \leftarrow \lceil 100/\#processors \rceil$
    while $tries \geq 1$ do
        for $int \in group$ do
            $int \leftarrow$ random group
        $numberCutEdges \leftarrow 0$
        for $e = \{u, v\} \in E$ do
            if $group[v.id] \neq group[u.id]$ then
                $numberCutEdges++$
        if $numberCutEdges > max$ then
            $max \leftarrow numberCutEdges$
        $tries--$
    // $results$ is the global results array
    $results[id] \leftarrow max$
```

4 Experimental Evaluation

In this section, we describe the experimental setup and present the results of our experiments.

4.1 Data and Hardware

All experiments were run on an AMD Ryzen 7 with a base clock of 1.8 GHz and a boost clock of up to 4.3 GHz, 16GB of RAM and 16 logical cores.
As test data we used all 400 private and official instances from the pace challenge that can be found here: https://pacechallenge.org/2020/td/. For the exact algorithm we limited the number of instances to the 200 exact instances. We also limited the time to solve an instance to ten minutes. To judge the solution quality of the algorithms, we used $(|E| + a)/2$ as exact solution, where a is the number of cut-edges found by the approximation algorithm. If the exact algorithm was able to find a solution, we used this value.

4.2 Results

A comparison of the number of cut-edges found by each algorithm is shown in figure 1 and figure 2.
Figure 1 uses a logarithmic scale on the y-axis and shows the number of cut-edges found by the heuristic and the approximation algorithm. It is clearly noticeable that the approximation algorithm calculates better results for every instance.
In figure 2 the number of cut-edges found by the exact algorithm and the approximation algorithm are compared. For smaller instances the results of the approximation algorithm seem to be closer to the results from the exact algorithm.

In figure 3 the solution quality of the approximation algorithm and the heuristic compared to the instances, that were solved by the exact algorithm, is shown. The approximation algorithm has a solution quality at around 90 percent. The quality decreases for bigger graphs. The heuristic only has about 90 percent quality for graphs with up to 19 vertices and decreasing quality for larger graphs. For graphs with at least 100 vertices, it only finds about 60 percent of the cut-edges of the optimal solution.
In figure 4 the solution quality of the algorithms for all instances, that were not solved by the exact algorithm, is shown. It shows the same trend for the heuristic as figure 3. The solution quality of the approximation algorithm increases. It goes from about 80% for small graphs to about 90% for bigger graphs.

Figure 5 shows the number of instances that were solved in the given time span in percent. Only for graphs with at most 19 vertices all instances were solved. And for graphs with more than 150 vertices, no instances were solved. The graph does not decrease monotonically, it shows a peak for graphs with 80 to 99 vertices.

Figure 6 and figure 7 compare the running times of all algorithms. The exact algorithm is clearly the slowest and the approximation algorithm is faster than the heuristic for all instances. The more vertices a graph has, the slower the heuristic becomes compared to the approximation algorithm.

In figure 8 the running times of the heuristic and the parallel implementation are compared. The y-axis has a logarithmic scale. The heuristic is clearly faster than the parallel version for small graphs up to 499 vertices and larger graphs with at least 399.999 vertices. For graphs in between, the heuristic and the parallel implementation have nearly the same running time, but the heuristic is still a bit faster.

5 Discussion and Conclusion

The running times of the heuristic and the approximation algorithm are unexpected. They do not behave as the theoretical running times suggest. The approximation algorithm has a higher theoretical running time than the heuristic, but is still faster. As mentioned in section 3.1.2, the theoretical running time is probably too high for most of the instances. Usually more than only one vertex switches group per iteration of the while loop and therefore also more than one edge is changed. It is likely that there are less than 100 iterations of the while loop, changing the running time to $n + 100m + m$ which is smaller than the running time of the heuristic with $100(n + m)$. The heuristic has to calculate for example the cut-edges for each of the 100 cuts it creates, while the approximation algorithm only has to count it once in the end.
Also unexpected is the running time of the parallel implementation, which we assumed to be faster than the sequential implementation. Reason for that is for example the additional work to organize the threads. But also the more complex way of counting the cut-edges and grouping the vertices by using Arrays instead of the graph itself, which leads to a higher running time. Even though access times for the graph and the Arrays is in $O(1)$, counting the cut edges takes double as much access time than in the sequential approach. This makes it more understandable why the parallel implementation is slower than the sequential. It is possible that the parallel implementation will be faster, if we would increase the number of tries or decrease the number of processors to minimize the work to organize the threads.

The solution quality of the approximation algorithm and the heuristic is as expected. Both have an (expected) approximation ratio of two. Both algorithms have never less than 50% of the cut-edges of the optimal solution and are therefore within the expected ratio. The approximation algorithm is performing better, because it basically starts with a cut from the heuristic algorithm and then improves the cut. Also, the probability to find a good cut in a randomized fashion decreases the more vertices a graph has.
The increase of the solution quality of the approximation algorithm for larger graphs is most likely because of the chosen upper bound. Presumably the exact solution is higher than the chosen upper bound and lower than the chosen upper bound for smaller graphs.

The exact algorithm could only find solutions for smaller graphs. The graphs with $80 - 99$ vertices may have a lower edge-density than the other graphs, which would explain the peak.

In conclusion we can obtain, that the approximation algorithm creates results close to the optimum and has a fast running time in practice. Therefore it is safe to say that the approximation algorithm is a good choice to solve the maximum cut problem.

References

[1] B. Roy. Lec. 3: MAXCUT and Introduction to Inapproximability. `https://www.tifr.res.in/~prahladh/teaching/2009-10/limits/lectures/lec03.pdf`. Accessed: 2023-03-13.

[2] A. Xiao. Lecture 17. `https://courses.cs.duke.edu/cps232/fall15/scribe_notes/lec17.pdf`. Accessed: 2023-03-13.

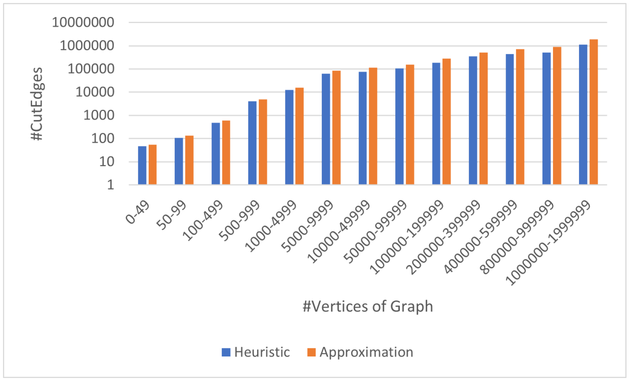

Figure 1: Number of cut-edges found by the Heuristic and Approximation Algorithm

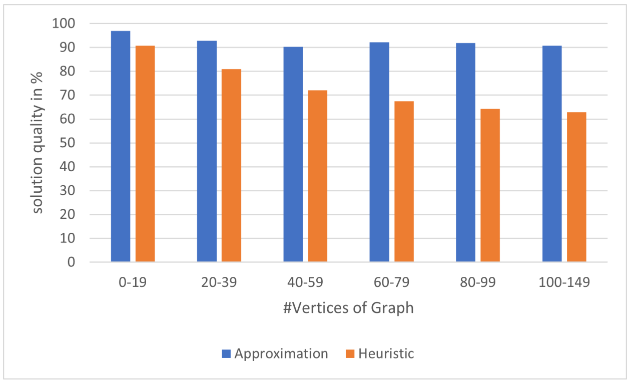

Figure 3: Solution quality of the different algorithms in percent for solved instances of the exact algorithm

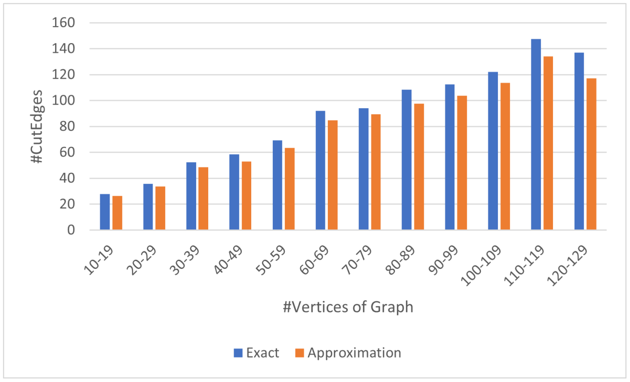

Figure 2: Number of cut-edges found by Exact and Approximation Algorithm

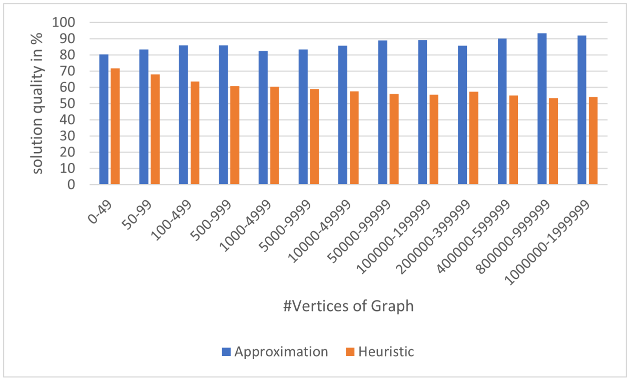

Figure 4: Solution quality of the different algorithms in percent for instances that weren't solved by the exact algorithm

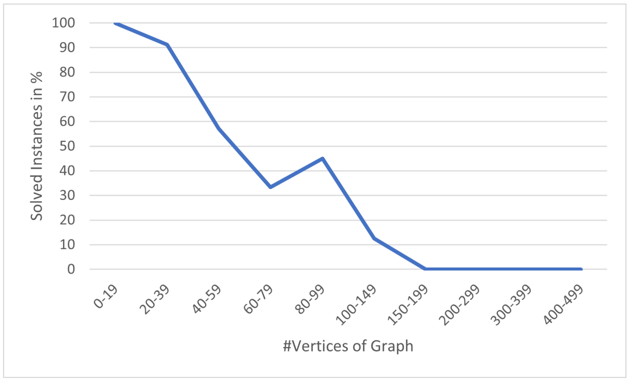

Figure 5: Solved Instances by Exact Algorithm in Percent

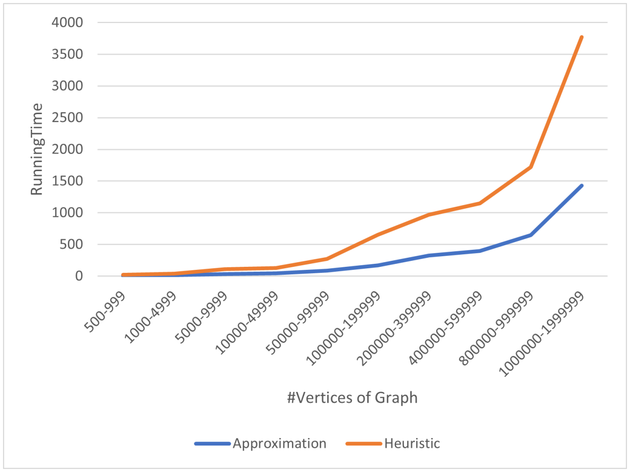

Figure 7: Average Running Times of Heuristic and Approximation Algorithm

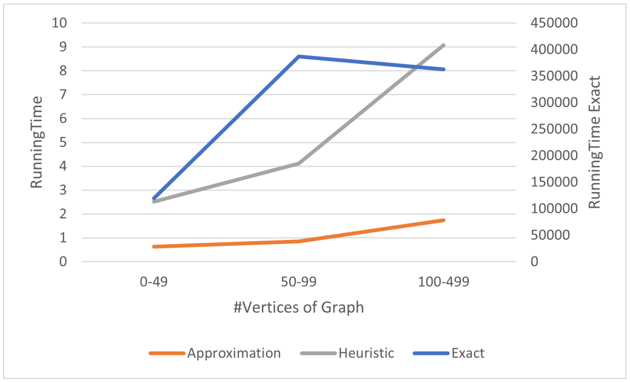

Figure 6: Average Running Time of Exact, Heuristic and Approximation Algorithms

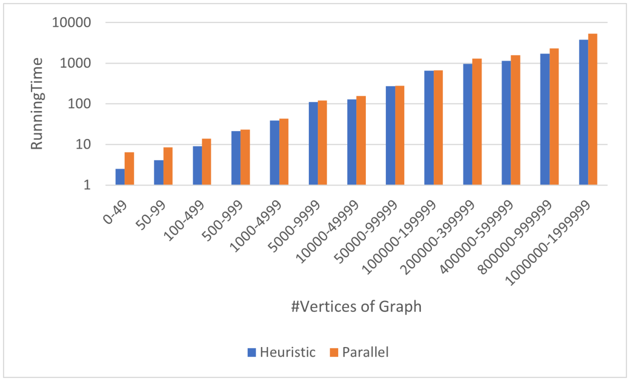

Figure 8: Running Time of Heuristic Algorithm and Parallel Implementation