Algorithm Engineering
WS 21/22

Jakob Sanowski

# Comparison of an approximation, heuristic and integer-linear program for the max-cut problem on unweighted, undirected graphs

## Abstract

I implemented and compared a heuristic, a approximation algorithm and an Integer-Linear-Program formulation for the maximum-cut problem on graphs. First I explain the algorithms and show their theoretical guarantees. The implementations are tested and evaluated on 60 graphs of different size. The results show the basic heuristic being the fastest while the approximation algorithm achieves the higher quality results. Solving maximum-cut with an Integer-Linear-Program solver is only really feasible on sparse graphs.

## 1 Introduction

The max-cut problem is a widely known problem on graphs. It is known to be NP-Hard but there exist a wide variety of approximation algorithms and heuristics for it. In this report we will look at a simple heuristic and approximation algorithm as well take a look at an Integer-Linear-Program formulation for it [Xiao]. We will focus on the performance and quality of the result on differently sized graphs. The heuristic and the approximation algorithm have in common, that they both achieve a 0.5-approximation. This report is structured as follow. In Section 2 we will look at the underlying ideas of the algorithm. Section **??** will provide details about the implementation and ideas on how to improve the algorithms. Following that, in Section **??** the setup of the experiments is outlined and the results are presented. Finally, Section **??**, we will discuss the findings of this report and draw further conclusions.

## 2 Preliminaries

The maximum-cut of a graph is the cut which size is at least the size of any other possible cut. A cut is a partition of all vertices into two complementary sets $S$ and $T$, which size is defined by the number of edges between $S$ and $T$.

For finding an exact solution we can formulate an integer-linear program $ILP$. Because the max-cut problem is NP-Hard there do not exist any polynominal-time algorithms for it. There are .. numerous approximation algorithms and heuristics for getting an approximate solution. In the following I will briefly explain the ILP-formulation, 0.5-approximation algorithm and heuristic I used.

### 2.1 Integer-Linear Program

An Integer-Linear program is a maximazation (or minimazation) of a objective function subject to a number of constraints with some or all of the variables being integers. The objective function and the constraints have to be linear. The maximum-cut problem can be formulated as follows:

Given a graph $G = (V, E)$, let $\{x_v\}_{v \in V}, \{z_e\}_{e \in E} \in \{0, 1\}$ where $x_v$ encodes the partition $v$ belongs to and $x_e$ encodes whether $e$ is part of the cut or not. Then the ILP can be formulated as follows:

$$\begin{aligned} \text{maximise} \sum_{uv \in E} z_{uv} \quad (1) \\ \text{subject to: } z_{uv} \leq x_u + x_v, \quad (2) \\ z_{uv} \leq 2 - (x_u + x_v) \quad (3) \end{aligned}$$

This discribes our problem because $z_{uv}$ can only be 1 if $uv$ is part of the cut. When $u$ and $v$ are in the same partition, $x_u + x_v = 0$ (or 2 respectively).

Because of (2) and (3) $z_{uv}$ then only can be 0. Only when $u$ and $v$ are in different partitions $x_u + x_v = 1$ and $z_uv$ can become 1 and thus be part of the cut. This formulation has $|V|+|E|$ variables (for each vertex and edge) and $2 * |E|$ constraints.

## 2.2 0.5-Approximation Algorithm

The 0.5-approximation algorithm is a greedy which iteratively inserts each vertex in the partition which improves the cut the most. For each vertex $v$ we count the the number of neighbors in $S$ and $T$ and then insert $v$ in the partition wich contains less neighbors of $v$ in order to increase the cut size as much as possible. Algorithm 1 shows the pseudo-code of this algorithm.

This algorithm finds a 0.5-approximation because if we let $C_{v_i}$ be the cut with all vertex added before $v_i$ and $E_{v_i} = \{(v_i, u) | u \in \{v_1, \ldots, v_{i-1}\}\}$ the edges between $v_i$ and all vertices added before $v_i$ then at least $\frac{1}{2}|E_{v_i}|$ get added to $C_{v_i}$.

Summing over all $v \in V$ we get:

$$|C| \geq \frac{1}{2} \sum_{v} |E_v|$$
$$|C| \geq \frac{1}{2}|E|$$

## 2.3 Heuristic

For the heuristic we can simply do a random flip if we add a vertex to $S$ or not. This also results in a 0.5-approximation. That is, for each $v \in V$:

$$P(v \in S) = P(v \notin S) = \frac{1}{2}$$

Thus the probability for any edge $(v, u)$ to be in the cut is the probability of $u$ and $v$ being assigned to different partitions.

$$\begin{aligned} P(e \in C) &= P(v \in S \wedge u \notin S) + P(v \notin S \wedge u \in S) \\ &= \frac{1}{4} + \frac{1}{4} \\ &= \frac{1}{2} \end{aligned}$$

```
Input: Graph $G = (V, E)$
Output: Cut $C$
begin
    foreach $v \in V$ do
        if $|neigh(v) \in S| \leq |neigh(v) \notin S|$ then
            $S = S \cup \{v\}$
        end
    end
    foreach $e \in E$ do
        if $e$ has only one vertex in $S$ then
            $C = C \cup \{e\}$
        end
    end
    return $C$
end
```

**Algorithm 1:** 0.5-Approximation Algorithm for Maximum-Cut

The overall expected size of the cut is:

$$\begin{aligned} |C| &= \sum_{e \in E} P(e \in C) \\ &= \frac{1}{2} * |E| \end{aligned}$$

Which is a 0.5-approximation on expectation. Algorithm 2 show the pseudo-code for this heuristic. [Xiao]

```
Input: Graph $G = (V, E)$
Output: Cut $C$
begin
    foreach $v \in V$ do
        randomly add $v$ to $S$ with probability $\frac{1}{2}$
    end
    foreach $e \in E$ do
        if $e$ has only one vertex in $S$ then
            $C = C \cup \{e\}$
        end
    end
    return $C$
end
```

**Algorithm 2:** 0.5-Heuristic for Maximum-Cut

# 3 Algorithm & Implementation

In order to compare the performance of these three algorithms I have to come up with implementations for these algorithms. In this chapter I will first talk about possible modifications to the base algorithms which will hopefully increase the performance of them. Secondly, I will provide the implementation details for the three algorithms.

## 3.1 Algorithm Engineering

For the ILP-formulation I could not find any improvements, given it already is very concise.

The approximation algorithm on the other side can be improved by instead of searching for each neighbor of the currently considerd $v$ in $S$ in order to figure out if it is contained in $S$, we can generate a lookup table over all vertex and save whether it got added to $S$ or not. This not only reduces the practical running time but also the theoretical because instead of having to search in $S$ to find out if a vertex is in $S$ or not we can simply do a lookup at the position for the vertex in our lookup-table which is in $\mathcal{O}(1)$ instead of $\mathcal{O}(|S|)$. Furthermore instead of calculating the edges contained in the cut after creating the partitions we can do this during the creation of the partitions. For each vertex $v$ we just add all edges to neighbors which are not in the same partition as $v$ to the cut.

Because the heuristic only results in a 0.5-approximation on expectation, we can execute multiple times at once in and pick the best result. This, of course, does not improve our running time but our quality of our result. Furthermore, if we desire a good quality result more than very fast running times, we can repeat this until our result higher than $\frac{1}{2}|E|$. In order to increase performance we can try to parallelise the generation of the heuristic. This can be achieved by splitting the vertices in disjoint subsets in the number of available cores and assigning each core one subset. Furthermore we can do the same for the edges when searching for the edges contained in the cut.

## 3.2 Implementation Details

All algorithms were implemented in Rust(1.67.1).

As a data structure for the graph I used adjacency lists. This data structure consists of an array over all vertices. For each vertex the set of outgoing edges is saved as an array of the indices of it's neighbors reachable via outgoing edges. I made this decision because this data structure allows iterating over the neighbors for a vertex $v$ in $\Theta(deg(v))$ which is highly beneficial for the approximation algorithm.

The ILP was formulated with the library good_lp. For the solver HiGHS was used. To formulate the problem the library provides a data structure to which the variables and constraints get added and passes this formulation to the solver.

For the approximation algorithms no external libraries were used. In its basic version I just go over all vertices and iterate through all neighbors of each vertex $v$ to count how many are already contained in $S$. If less neighbors are in $S$ than there aren't, $v$ gets added to $S$. $S$ is implemented as a growable array holding the indices of all $v$ contained in $S$. To find all cut edges I iterate over all edges and test if only one of their vertex is in $S$. The engineered version uses a fixed size array holding booleans for $S$. A vertex is contained in $S$ if the entry at it's index is true. It also uses a growable array to collect all edges included in the cut. In each round I now count how many neighbors are in $S$ (and aren't respectively) by checking their entry in the lookup array and add the vertex to $S$ if less neighbors were in $S$ than weren't. Then all edges to neighbors not in $S$ (or in $S$ respectively) get added to the cut. One important thing is to ignore all neighbors with a higher index than the current vertex or edges that aren't in the final cut might be added.

Lastly, for the heuristic the library rand was used for random-number-generation. The for the basic implementation each vertex just randomly gets added to $S$ with probability $\frac{1}{2}$. $S$ is represented by a array of booleans where 'true' indicates, that the vertex with this index is in $S$ In order to parallelise this heuristic all vertices get divided into evenly sized arrays and passed to a separate thread. The threads are the basic implementation provide by the Rust standard library. Each thread then applies the basic heuristic to his slice. For generating random numbers a simple pseudo-random number generator is used in order to increase performance because for this application it is not necessary to use true random numbers. In order to circumvent the process of merging, the threads use one shared array. Because Rusts memory-safety features prevent simultaneous access to a data structure, a Mutex is used for managing the access. The improved heuristic starts $n$ threads, where $n$ is the number of processors available, with the basic heuristic.

After receiving the results, their quality is compared and the best one is returned. The quality, in this case, is just the number of edges contained in the cut. If none of the results are bigger of size than $\frac{1}{2}|E|$ then this routine is repeated until this requirement is satisfied. This implementation has the possibility of never terminating though the probability for this happening is approaching 0.

# 4 Experimental Evaluation

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

## 4.1 Data and Hardware

Experiments were conducted on a 4-core (8 hyperthreads) Intel i7-6700K CPU with a clock speed of 4.00GHz an 16GiB of RAM.

The approximation algorithm and the heuristic were evaluated on 30 graphs of increasing size, selected from the 200 PACE19 graphs. In order to being able assess the quality of the results they were also tested on 30 graphs from the Biq Mac Library. Table 4 shows the number of nodes and edges for each graph.

The ILP was tested on the graphs from 'vc_exact' with a timeout of 10 secons as well on 'g05_60.0', 'g05_60.0', 'g05_60.0' with a timeout of 2 hours. Furthermore, I tested the ILP on 'g05_60.0' without a timeout.

## 4.2 Results

### 4.2.1 Approximation Algorithm

The running times of the two approximation algorithms are shown in Figure **??**.

The results clearly show that the improved algorithm vastly outperforms the basic variant. Compared to all other algorithms, the improved variant performs the best on all instances.

Figure 1 shows size of the computed cuts compared to the optimum found in the Biq Mac Libary [Wiegele [2007]]. We can see that the results of the approximation algorithm are quite close to the optimum. In practice it seems like the results approximation algorithm are only about 10% less than the optimum.

### 4.2.2 Heuristic

As figure 2 shows, the parallel heuristic does not only not improve performance, it actually performs worse on all instances than the basic implementation. The improved heuristic performs very similarly to the parallel heuristic but seems to be slightly faster.

Compared to the approximation algorithm the basic implementation performs slightly better than the improved approximation algorithm. The parallel version does about as well as the improved approximation algorithm for large instances.

In terms of the quality of the result they are about 80% the size of the optimum. Looking at the difference between the improved heuristic and the basic one the improved variant produces better results in nearly all cases. The approximation algorithm beats both heuristics in this aspect. Furthermore they are roughly $\frac{1}{2}|E|$.

## 4.3 Integer Linear Program

The ILP-solver timed out on most instances. It also timed out on the 3 graphs with the timeout 2 set to 2 hours which are relatively small compared to most other graphs. Interestingly on the instances 'vc-exact_003.gr', 'vc-exact_012.gr', 'vc-exact_027.gr' and 'vc-exact_040.gr' 3 a result was found in less than 10 seconds which is, compared with the running time of

| graph | result size | running time (hh:mm:ss) |
|---|---|---|
| g05_60.0 | 535 | 4:19:40 |

Table 1: Result of the ILP-solver for 'g05_60.0'

'g05_60.0', vastly faster despite the number of variables and constraints being a multiple of 'g05_60.0' 1 If we examine their properties, we recognize, that these are the graphs with the lowest average degree. The result for 'g05_60.0' is one less than the optimum found in the Biq Mac Library.

| graph | result size | running time (hh:mm:ss) |
|---|---|---|
| g05_60.0 | | timeout |
| g05_60.1 | | timeout |
| g05_60.2 | | timeout |

Table 2: Running times of the ILP-solver with a timeout of 2 hours

## 5 Discussion and Conclusion

In this work, I compared a basic heuristic and approximation algorithm and an ILP-formulation for exact computation of the max-cut problem. For the heuristic and approximation we looked into methods of possibly improving their run time. We showed that the basic implementation of the heuristic always is the fastest but gets beaten by the approximation algorithm in terms of quality of the result. It might be surprising, that the parallel heuristic is slower than the basic implementation. But after considering the amount of work which is parallelised is quite small it becomes quite clear, that the time saved by parallelising is negated by the amount of work necessary for merging the results of the individual threads. For this reason it might be better to use parallelism to generate multiple heuristics for achieving better overall results.

Another interesting observation is the better result quality of the approximation algorithm compared to the heuristic. The reason for this observation, are the guarantees we can make about the results. As shown in Section 2.2 the approximation algorithm guarantees that the result size is at least $\frac{1}{2}|E|$ where as the heuristic only produces results with size $\frac{1}{2}|E|$ in expectation. This means the heuristic may produce results which are smaller in size than the expectation.

For computing exact solutions with an ILP-solver it seems like density is the main factor for increase in running time. This becomes especially clear, if we look at the running time of the instance 'vc-

| graph | cut-size | time in ms |
|---|---|---|
| vc-exact_001.gr | | timeout |
| vc-exact_003.gr | 48418 | 5640 |
| vc-exact_008.gr | | timeout |
| vc-exact_009.gr | | timeout |
| vc-exact_012.gr | 42526 | 4319 |
| vc-exact_017.gr | | timeout |
| vc-exact_022.gr | | timeout |
| vc-exact_026.gr | | timeout |
| vc-exact_027.gr | 52435 | 6511 |
| vc-exact_032.gr | | timeout |
| vc-exact_040.gr | 278 | 13486 |
| vc-exact_056.gr | | timeout |
| vc-exact_082.gr | | timeout |
| vc-exact_084.gr | | timeout |
| vc-exact_085.gr | | timeout |
| vc-exact_094.gr | | timeout |
| vc-exact_095.gr | | timeout |
| vc-exact_109.gr | | timeout |
| vc-exact_110.gr | | timeout |
| vc-exact_120.gr | | timeout |
| vc-exact_161.gr | | timeout |
| vc-exact_162.gr | | timeout |
| vc-exact_164.gr | | timeout |
| vc-exact_172.gr | | timeout |
| vc-exact_181.gr | | timeout |
| vc-exact_183.gr | | timeout |
| vc-exact_186.gr | | timeout |
| vc-exact_189.gr | | timeout |
| vc-exact_192.gr | | timeout |
| vc-exact_200.gr | | timeout |

Table 3: Results of the ILP with a timeout of 10 seconds

| graph | vertices | edges |
|---|---|---|
| vc-exact_001.gr | 6160 | 40207 |
| vc-exact_003.gr | 60541 | 48418 |
| vc-exact_008.gr | 7537 | 72833 |
| vc-exact_009.gr | 38452 | 174645 |
| vc-exact_012.gr | 53444 | 42526 |
| vc-exact_017.gr | 23541 | 34233 |
| vc-exact_022.gr | 12589 | 19775 |
| vc-exact_026.gr | 6140 | 36767 |
| vc-exact_027.gr | 65866 | 52435 |
| vc-exact_032.gr | 1490 | 2680 |
| vc-exact_040.gr | 210 | 625 |
| vc-exact_056.gr | 200 | 1089 |
| vc-exact_082.gr | 200 | 954 |
| vc-exact_084.gr | 13590 | 21240 |
| vc-exact_085.gr | 11470 | 17408 |
| vc-exact_094.gr | 5960 | 10720 |
| vc-exact_095.gr | 15783 | 24663 |
| vc-exact_109.gr | 66992 | 90970 |
| vc-exact_110.gr | 98128 | 161357 |
| vc-exact_120.gr | 70144 | 116378 |
| vc-exact_161.gr | 138141 | 227241 |
| vc-exact_162.gr | 50635 | 83075 |
| vc-exact_164.gr | 29296 | 46040 |
| vc-exact_172.gr | 4025 | 7435 |
| vc-exact_181.gr | 18096 | 28281 |
| vc-exact_183.gr | 72420 | 118362 |
| vc-exact_186.gr | 26300 | 41500 |
| vc-exact_189.gr | 7400 | 13600 |
| vc-exact_192.gr | 2980 | 5360 |
| vc-exact_200.gr | 1150 | 80258 |

| graph | vertices | edges | optimim |
|---|---|---|---|
| g05_60.0 | 60 | 885 | 536 |
| g05_60.1 | 60 | 885 | 532 |
| g05_60.2 | 60 | 885 | 529 |
| g05_60.3 | 60 | 885 | 538 |
| g05_60.4 | 60 | 885 | 527 |
| g05_60.5 | 60 | 885 | 533 |
| g05_60.6 | 60 | 885 | 531 |
| g05_60.7 | 60 | 885 | 535 |
| g05_60.8 | 60 | 885 | 530 |
| g05_60.9 | 60 | 885 | 533 |
| g05_80.0 | 80 | 1580 | 929 |
| g05_80.1 | 80 | 1580 | 941 |
| g05_80.2 | 80 | 1580 | 934 |
| g05_80.3 | 80 | 1580 | 923 |
| g05_80.4 | 80 | 1580 | 932 |
| g05_80.5 | 80 | 1580 | 926 |
| g05_80.6 | 80 | 1580 | 929 |
| g05_80.7 | 80 | 1580 | 929 |
| g05_80.8 | 80 | 1580 | 925 |
| g05_80.9 | 80 | 1580 | 923 |
| g05_100.0 | 100 | 2475 | 1430 |
| g05_100.1 | 100 | 2475 | 1425 |
| g05_100.2 | 100 | 2475 | 1432 |
| g05_100.3 | 100 | 2475 | 1424 |
| g05_100.4 | 100 | 2475 | 1440 |
| g05_100.5 | 100 | 2475 | 1436 |
| g05_100.6 | 100 | 2475 | 1434 |
| g05_100.7 | 100 | 2475 | 1431 |
| g05_100.8 | 100 | 2475 | 1432 |
| g05_100.9 | 100 | 2475 | 1430 |

Table 4: The left table show the graphs from PACE 2019

exact_027.gr' and 'g05_60.0'.

## 6 References

## References

A. Wiegele. Biq mac library - a collection of max-cut and quadratic 0-1 programming instances of medium size, 2007.

A. Xiao. Compsci 532: Design and analysis of algorithms.

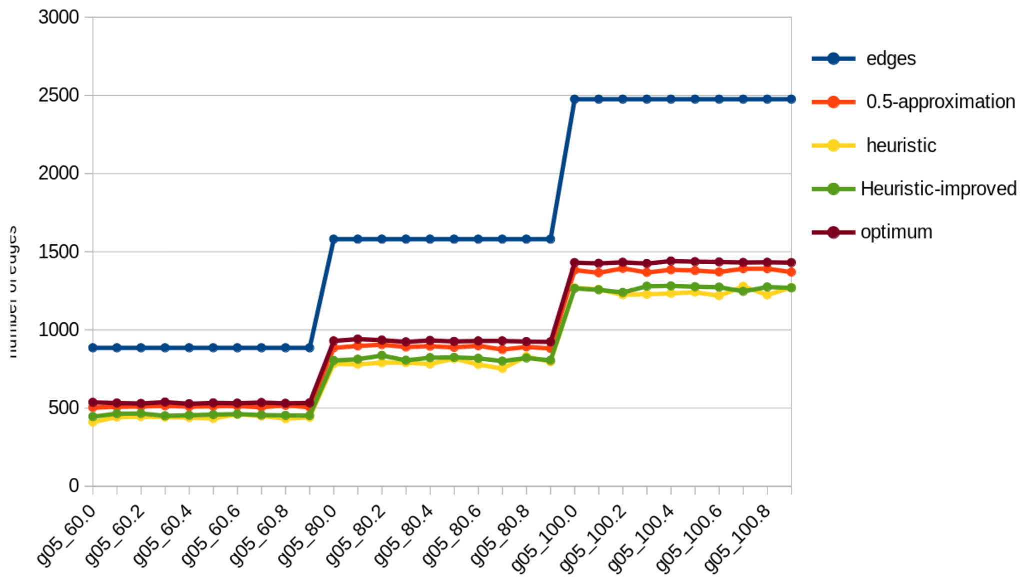

Figure 1: Size of the results compared to the optimum (edges marks the total number of edges in the graph)

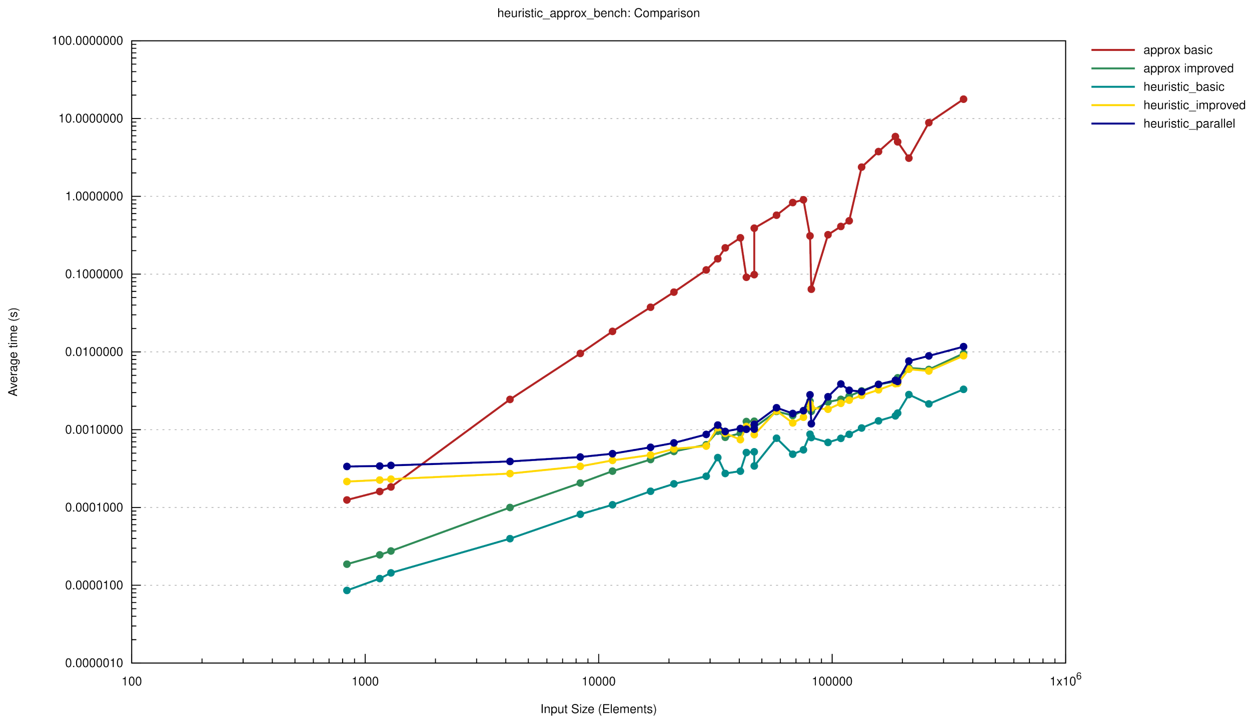

Figure 2: Running times of every polynomial-time algorithm