Algorithm Engineering
WS 20/21

Luca Bosch
01/851145

Exam: Comparing Approaches for Finding a Maximum Cut in Graphs

Abstract

I implemented and compared various algorithms that tackle the problem of finding a maximum cut in a finite, simple, undirected graph, that is, a partition of the vertices into two sets, such that the weight of the edges between them is as great as possible. You can find a heuristic, an approximation, and an exact solution in this paper.
I describe the problem's characteristics, the ideas behind the different approaches, and the consequences of the implementation decisions in this report.
Furthermore, I performed various test cases on a range of different graphs whose results I provide as well to show how the algorithms are affected by parameters and graph characteristics.

1 Introduction

First of all, it is necessary to clarify the terminology. The term *maximum cut problem* is often used synonymous with *weighted maximum cut problem*. The difference is, that for the maximum cut problem, the input graph has no edge weights, whereas for the *weighted maximum cut* problem, the input graph is weighted and the goal is not to optimize the number of edges that are cut, but instead to maximize the weight of the cut edges.
Both problems are NP-complete [4, 8], but since the maximum cut problem can be emulated using an algorithm for a weighted maximum cut problem by setting all weights to 1, I will focus on the more general version and refer to it as **Max-Cut** from now on.

As for real-world scenarios, this problem is particularly interesting for finding ground states of spin glasses with exterior magnetic field, which is relevant for the field of physics, as well as planning of layouts of integrated circuits by minimizing the number of holes on a printed circuit board, or contacts on a chip [1].

Max-Cut is usually demonstrated visually by drawing an actual cutting line between the vertices and looking at which edges are crossed by it. The only requirement is that the drawn line does not cross any edge multiple times. An example can be seen below.

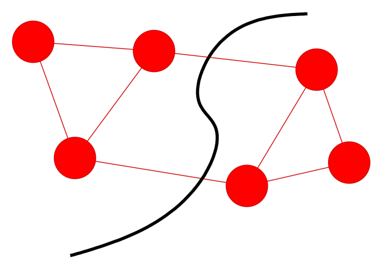


Figure 1: **Example of a 2-edge cut** [*public domain*]

As for formulas and algorithms, it is easier to work with two colors, denoting all the edges that connect vertices of a different color as being cut. This definition is equivalent but easier to express mathematically or implement into code. The decision problem formulation is determining whether the following is true or false for some graph $G(V, E)$ and given x:
$\exists A, B \subset V$ with $A \cap B = \varnothing \wedge A \cup B = V, \forall \{u, v\} \in Z \subset E$ with $(u \in A \wedge v \in B) \vee (v \in A \wedge u \in B)$: $\sum w(\{u, v\} \in Z) \geq x$
Where w denotes the weight function.

Here is an overview of what the rest of this report consists of:
Section 2 introduces the two algorithms, their characteristics and correctness and explains how they work, while in **Section 3** I write about the used algorithm engineering techniques and give imple-

mentation details. **Section 4** includes information about the experimental setup and provides empirical results. **Section 5** draws further conclusions from the gained knowledge. Finally, **Section 6** explains how the reader can execute the test cases that are provided.

2 Preliminaries

As said, there are three algorithms I will have a look at. The first one is a heuristic that is simple and quick, especially for large graphs. However, it might calculate an arbitrarily bad result for some inputs, though it is always a valid one. The second algorithm is a 0.5-approximation, meaning the result is always at least half as good as the optimal one. In most cases, it takes longer than the heuristic, but also provides a better result. The last algorithm is an Integer Linear Programming approach which can find the optimal solution but as the problem is NP-hard, it takes a lot of time, especially for medium-sized or large graphs.

The goal of all these algorithms is to find a partition into two subsets of V, such that there is no partition where the sum of all weights of edges between these subsets would be higher.

2.1 Heuristic

The idea that I had was that we always take the heaviest edge and add its two vertices to different subsets. Then we remove all edges adjacent to nodes which are already in one of the subsets, because else we would in many cases subtract cut cost, by putting two neighboring vertices into the same subset. This leads to the problem that in a complete graph, we would remove all edges after the first step, which on an arbitrarily large (unweighted) graph leads to an arbitrarily bad solution.

Correctness: I only have to prove that $A \cap B = \varnothing$ and $A \cup B = E$, since there is no quality guarantee. This is indeed true since in line 12 we add the difference of V and A to B so it matters not how many elements are in A, though in practice it can be up to half of all.

Algorithm 1 Heuristic $G(V, E)$

1: sort E descendingly by weights
2: $A := \varnothing$
3: **while** $E \neq \varnothing$ **do**
4: $\quad \{u_0, v_0\} := E$.get(0)
5: $\quad A$.add(u_0)
6: $\quad$ **for** $\{u, v\} \in E$ **do**
7: $\quad\quad$ **if** $u = u_0 \lor v = u_0 \lor u = v_0 \lor v = v_0$ **then**
8: $\quad\quad\quad E$.remove($\{u, v\}$)
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: **end while**
12: $B := V - A$
13: **return** A, B

It is also important that the algorithm terminates, which can be ensured by removing every edge sharing a vertex with the one added to A from E and that must be at least be one, which is the one added to A itself. By reducing the remaining edges in E by at least 1 per loop, E will at some point be empty.

Complexity: As we have just seen, the while-loop can run at most $|E|$ times whereas the inner for-loop, in the n-th run, can at most run $|E| - n$ times. Therefore the complexity is technically $\mathcal{O}(|E|^2)$.

Nevertheless, there is one interesting observation to be made: In every while-loop run, there is one vertex added to A. As we cannot have more than half of the total number of vertices in A (since we dismiss all neighboring vertices), the outer while-loop can in fact only run $\frac{|V|}{2}$ times, giving us a $\mathcal{O}(|V||E|)$ complexity. In practice, vertices without edges do not matter, as we can see in the loop conditions, so this is always better. There exist sorting algorithms in $\mathcal{O}(|E| \log |E|)$ [9], so the sorting part would only dominate if $|V| < \log |E|$ This can never be the case, as there cannot exist a simple graph with $2^{|V|}$ or more edges [2].

2.2 Approximation

For the approximation algorithm, I decided to implement Sahni et al.'s idea of a 0.5-approximation [12]. This is a greedy best-in algorithm. Such an algorithm usually starts with an empty graph, while members of the original graph are considered to be candidates for inclusion in the constructed feasible solution. The algorithm successively adds a candidate, which provides the best contribution to the objective. In this case, our objective is to maximize edge cut costs. Therefore we start with $A = E$ and for each vertex, we evaluate how much we would profit by transferring it from A into B.

Algorithm 2 Approximation $G(V, E)$

```
 1: Let w be the weight function
 2: B := ∅
 3: A := V
 4: repeat
 5:     max := 0
 6:     Vertex heaviest := null
 7:     for u ∈ V do
 8:         if u ∉ B then
 9:             weight := 0
10:             for {u, v} ∈ E do
11:                 if v ∈ A then
12:                     weight += w({u, v})
13:                 else
14:                     weight −= w({u, v})
15:                 end if
16:             end for
17:             if weight > max then
18:                 max = weight
19:                 heaviest = u
20:             end if
21:         end if
22:     end for
23:     B.add(heaviest)
24:     A.remove(heaviest)
25: until no improvement is possible
26: return A, B
```

Correctness: Again, it is obvious that $A \cap B = \varnothing$ and $A \cup B = E$ are fulfilled with the same argument as before. Though it does not follow easily how this produces a 0.5-approximation, which can be found as a Lemma 2.3 in Sahni et al.'s paper [12], and there is a more recent description from Kahruman et al. [5].

The algorithm terminates because there cannot be improvement possible forever as there has to exist an optimal solution for every finite graph. It is also impossible, that the algorithm makes the solution worse and better again forever since *max* is set as 0, so only improvements are allowed.

Complexity: If we have a closer look, we can see that the outermost loop can run at most $|V|$ times since every time an improvement is possible, a vertex gets added to B. if every vertex would be added to B, the if-condition in line 7 would never be true and we would not add anything to B in that run, therefore not making an improvement. In practice, we would of course stop way before that. The inner for-loop runs exactly $|V|$ times, and the innermost one $|E|$ times, which is how we get a running time complexity of $\mathcal{O}(|V|^2|E|)$.
Note: You can also find a discussion of a parallelized version of this algorithm in ***Section 3.2****.*

2.3 Exact Solution

I decided to state Max-Cut as an Integer Linear Programming (**ILP**) problem, which can be solved optimally. For that matter, it is important to find an expression of the problem with the variables, which in this case are in which subset a vertex is, as conditions. We can use the following:
Let $n = |V|$, $u < v$ and $u, v, \{u, v\} \in \{0, 1\}$.

$$\max \sum_{u,v=1}^{n} w(\{u, v\}) \cdot \{u, v\}$$

$$\{u, v\} - u - v \leq 0$$

$$\{u, v\} + u + v \leq 2$$

Note: $\{v, w\} = 1$ if the edge is cut and 0 else.
Note: $u = 1$ if $u \in A$, $v = 1$ if $v \in A$ w.l.o.g.
To give some quick example for this rather confusing formula let us assume, u and v are in the same subset, say they are both in A. Then $u = v = 1$.

Therefore, for the formula $\{u,v\} + u + v \leq 2$ to be true, $\{u,v\}$ needs to be 0. That means, that the edge is not cut, just as we expected, when we put both vertices u and v into the same subset. Let us now instead assume u and v are both in B. Then $u = v = 0$. $\{u,v\} - u - v \leq 0$ can then only be true if $\{u,v\}$ is 0, again the edge is not cut. Only if $u = 1, v = 0$ or vice versa $\{u,v\}$ can be 1, meaning the edge can be cut. Of course, the naming of A and B does not matter.
Since we want to maximize the sum of weights of cut edges, we can assume the solving algorithm will ensure that the right edges are cut. After that, we can get the result of which edges are 0 and which ones are 1 and assign them to their respective subsets.

Algorithm 3 ILP $G(V,E)$

```
A, B = ∅
List cond := {}
for u ∈ V do
    for v > u ∈ V do
        if {u, v} ∈ E then
            cond.add({u, v} − u − v ≤ 0)
            cond.add({u, v} + u + v ≤ 2)
        end if
    end for
end for
solveBinary(cond, maxSum({u, v} ∗ w({u, v})))
for u ∈ V do
    if u = 1 then
        A.add(u)
    else
        B.add(u)
    end if
end for
return A, B
```

Correctness: It is difficult to argue about the correctness of the ILP-solver, since there are many different ones, so I will treat it as a black box. The used formula is rather intuitive once understood and has been proven to be correct [5].
With unlimited time, we can always get a perfect result. Concerning the result quality in limited time, my findings are in **Section 4**.

Complexity: Creating the conditions takes $\mathcal{O}(|V|^2)$ steps, and $|E|$ many conditions are created. Even though there are some very sophisticated approaches as to how to make solving these equations easier, there can never exist an ILP-solver solving this in under $\mathcal{O}(2^{|E|})$ if $P \neq NP$, since Max-Cut is NP-Complete.

3 Algorithm & Implementation

To compare the actual performances of these presented algorithms, it is necessary to convert the ideas into code that works efficiently. In this chapter, I will describe the details of how that has been done and what changes I deemed necessary to improve practical running times. Specifically, you can find pseudocode of a parallelized approach for Algorithm 2 at the end of this section.

3.1 Algorithm Engineering

Formally, a Max-Cut is a partition of all vertices of a graph into two subsets which is why for all pseudocode implementation the return value was two sets of vertices. However, for the results, I am not interested that much in what vertices are in which subset, but much rather in the total weight of the edges. This is why instead of adding vertices to lists and removing them as well as calculating the difference or if an element is contained in a list, which is a rather expensive operation, I avoided all of this by enabling nodes to be marked. The subsets are then all nodes that are marked, and those that are not. I implemented functions for displaying the total weight of cut edges, the subsets, and the cut edges individually.

For the exact ILP solution, we would technically create $|V|^2$ conditions, but a lot of them would not include an edge in the equation, meaning that it would only be $u + v \leq 2$ and $-u - v \leq 0$ which is not an actual restriction for $u, v \in \{0, 1\}$. Though the library immediately sorts these entries out, it still takes some time to create them, which is why I of course did not do so.

3.2 Implementation Details

I implemented all algorithms in Java (Version 15.0.1), using the included standard libraries and data structures as well as the library *LpSolve*. I decided to create separate Graph, Node, and Edge classes since those Objects are used very often and make abstraction easier. Every mentioned algorithm is a method taking the graph on which Max-Cut is to be done as an argument and as mentioned earlier does not return a partition of the graph but instead directly marks nodes in the graph. These marks can then be evaluated by separate functions which determine the weight, number of cut edges, etc. Before running a different algorithm, all nodes are unmarked first.

As promised in the beginning, every problem on unweighted graphs can also be solved with weighted ones, which is why I included the possibility to use both weighted as well as unweighted graphs as input files.

Something that might be confusing is what format the ILP-solver uses for its conditions, which is an array of coefficients, where the positions encode the variable names, which is why the code is way more extensive than the pseudo-code provided in **Section 2**.

Now, I will be discussing the parallelization part that I omitted earlier. Specifically, I parallelized the most work-intensive part of the approximation algorithm which is evaluating the possible improvement in the sum of cut edge weights, if one vertex was transferred to subset B. For that purpose, I split the list of vertices into as many parts as there are cores and calculate a local maximum. The maximum of these is determined sequentially.

Of course this is just as correct as the non-parallelized approach making use of $\max(a, b, c) = \max(\max(a, b), c)$.

Algorithm 4 Parallel Approximation $G(V, E)$

Let w be the weight function
$B := \varnothing$
$A := V$
repeat
    i := 0
    List results = {}
    for core **in** cores **do**
        X = Sublist from $i/|cores| * |V|$ to $(i+1)/|cores| * |V|$ of V
        i++
        results.add(core.**Task**(X, A, B))
    end for
    heaviest := max(results.max)
    B.add(heaviest)
    A.remove(heaviest)
until no improvement is possible
$A := V$
return A, B

Algorithm 5 Task (X, A, B)

for $u \in X$ **do**
    if $u \notin B$ **then**
        weight := 0
        for $\{u, v\} \in E$ **do**
            if $v \in A$ **then**
                weight += $w(\{u, v\})$
            else
                weight −= $w(\{u, v\})$
            end if
        end for
        if weight > max **then**
            max = weight
            heaviest = u
        end if
    end if
end for
return [heaviest, max]

4 Experimental Evaluation

4.1 Data and Hardware

I ran all the tests on the following hardware:
Intel Core i7-6700K CPU, 4x4.4 GHz
(Hyper-Threading enabled)
16 GB RAM, 3200 MHz

4.2 Test Graph Sources

Most graphs I used are from *SteinLib*[1] Additionally, I used datasets from the US road network [10], Bitcoin OTC trust [6, 7], and Last.fm's social network [11].

4.3 Results

I decided to use a time limit of 180 seconds for the ILP-solver so I could do all the tests at once. The results are consistent with what was expected from the theoretical analysis of the algorithms. The only surprises were slightly increased running times in the first run, which might be attributed to the Java Runtime Environment.

The results of the tests can be found on the following pages consisting of a table and the acquired insights into the algorithms' performances in practice. Moreover, I included three charts on the last page.
An interesting thought for future work would be allowing the approximation algorithm to make small losses in the objective function to avoid getting stuck in a local optimum, perhaps by removing vertices from the subsets at random. Depending on the implementation, this would however make proving its correctness more difficult.

For **Table 1**, consider the following legend and explanation:
H - Heuristic
A - Approximation
AP - Parallelized Approximation
[2] - Odd cycle
[3] - Odd wheel
[4] - Root = 7/8n for n = 3
[5] - Composition of odd wheels as an odd cycle
[6] - Goemans design 4, 3, 2 facett
[7] - Odd antiwheel
[8] - Real life VLSI
[9] - BTC network
[10] - Last.fm social network
[11] - US road map

The ILP-solver is considered timed out after 180 seconds. Since it produces intermediate results, the last one will be used as the final result in the table. Be aware that such an intermediate result is not necessarily optimal. For larger graphs, the ILP was not able to find any feasible solution which is noted in the results. Graphs with even more edges than those for which no feasible solution was found have not been tested, as they would not be solved either.

H *Miss* and ***A*** *Miss* denote the percentage by which the corresponding algorithms result was worse than the optimum, while ***H*** *vs.* ***A*** *Miss* indicates how much better the result of **A** was in comparison to **H**. Lastly, ***AP*** *Speed Change* indicates the difference in time it took to execute **A** and **AP**.

[1] `http://steinlib.zib.de/`

#	$\|V\|$	$\|E\|$	**H** Time	**H** weight	**A** Time	**A** Weight	**AP** Time	ILP Time	ILP Weight	**H** Miss	**A** Miss	**H** vs. **A** Miss	**AP** Speed Change	Notes
1	50	63	0,8	219	0,5	323	7,2	13,1	342	36%	6%	32%	1391%	
2	50	63	0,1	255	0,2	329	3,3	6,4	339	25%	3%	22%	1542%	
3	50	63	0,1	310	0,2	310	3,0	19,4	314	1%	1%	0%	1474%	
4	50	100	0,1	369	0,2	493	1,9	556,9	521	29%	5%	25%	799%	
5	50	100	0,1	362	0,3	453	1,9	268,6	470	23%	4%	20%	616%	
6	50	100	0,1	353	0,3	475	3,4	414,5	485	27%	2%	26%	1147%	
7	75	94	0,2	357	0,4	477	1,8	24,8	500	29%	5%	25%	382%	
8	75	94	0,2	389	0,4	456	1,8	50,6	483	19%	6%	15%	382%	
9	75	94	0,2	356	0,4	463	2,2	13,5	476	25%	3%	23%	449%	
10	75	150	0,4	568	1,0	709	1,8	timeout	737			20%	80%	
11	75	150	0,3	488	0,8	636	2,0	timeout	676			23%	155%	
12	75	150	0,2	558	0,3	697	2,3	timeout	754			20%	566%	
13	100	125	0,4	532	0,4	656	2,2	123,3	683	22%	4%	19%	434%	
14	100	125	0,2	452	0,3	630	2,0	20,2	665	32%	5%	28%	509%	
15	100	125	0,2	461	0,3	611	1,8	198,3	657	30%	7%	25%	466%	
16	100	200	0,4	772	0,3	920	1,9	timeout	999			16%	531%	
17	100	200	0,4	692	0,1	879	2,1	timeout	896			21%	2139%	
18	100	200	0,4	773	0,1	960	2,0	timeout	993			19%	2187%	
19	2500	3125	19,1	11695	91,4	15257	91,2	timeout	16102			23%	0%	
20	2500	3125	6,7	11900	92,1	15791	80,9	timeout	16710			25%	-12%	
21	2500	3125	8,3	12175	102,2	15818	88,5	timeout	16590			23%	-13%	
22	2500	3125	6,5	11982	122,7	15797	97,8	timeout	16523			24%	-20%	
23	2500	3125	6,5	12149	91,7	15961	84,8	timeout	16829			24%	-8%	
24	2500	5000	12,0	17849	152,2	22922	88,5	timeout	23456			22%	-42%	
25	2500	5000	12,1	17850	155,4	22914	88,0	timeout	23759			22%	-43%	
26	2500	5000	14,8	17450	152,1	22846	91,6	timeout	23380			24%	-40%	
27	2500	5000	12,1	17759	155,2	23435	90,5	timeout	23781			24%	-42%	
28	2500	5000	11,8	17759	124,3	23066	84,2	timeout	23654			23%	-32%	
29	2500	12500	44,9	38978	370,0	49094	138,6	timeout	infeasible solution			21%	-63%	
30	2500	12500	44,3	39146	358,3	48897	136,6	timeout	infeasible solution			20%	-62%	
31	2500	12500	37,4	39145	306,4	49385	125,4	timeout	infeasible solution			21%	-59%	
32	2500	12500	44,7	39561	357,0	49838	140,3	timeout	infeasible solution			21%	-61%	
33	2500	12500	44,0	39341	360,9	49094	136,1	timeout	infeasible solution			20%	-62%	
34	2500	62500	389,6	178368	2351,2	206026	569,2	not run	not run			13%	-76%	
35	2500	62500	370,6	177592	2229,8	205486	539,8	not run	not run			14%	-76%	
36	2500	62500	375,9	176916	2140,2	207347	504,5	not run	not run			15%	-76%	
37	2500	62500	367,1	177261	2113,5	206511	472,3	not run	not run			14%	-78%	
38	2500	62500	317,2	178527	1980,2	206497	513,1	not run	not run			14%	-74%	
39	53	80	0,0	3424	0,0	4531	0,7	20,3	4920	30%	8%	24%	2810%	
40	55	82	0,0	3936	0,0	4815	1,0	14,4	4932	20%	2%	18%	3573%	
41	57	84	0,0	3843	0,0	4660	1,1	34,3	4915	22%	5%	18%	4173%	
42	157	266	0,1	10910	0,2	13620	2,2	31781,4	14102	23%	3%	20%	1053%	
43	160	269	0,1	11041	0,3	13675	2,2	2591,1	14185	22%	4%	19%	598%	
44	165	274	0,1	10984	0,2	13599	2,4	timeout	14161			19%	1006%	
45	307	526	0,2	26291	0,8	34233	5,1	timeout	35442			23%	562%	
46	311	530	0,2	26226	0,8	34140	3,9	timeout	35422			23%	412%	
47	313	532	0,2	26388	0,8	34143	4,8	timeout	35805			23%	527%	
48	321	540	0,2	26700	0,8	33790	4,3	timeout	34302			21%	440%	
49	816	1460	1,4	68826	6,9	92180	15,3	timeout	94855			25%	121%	
50	818	1462	1,4	68644	6,9	92182	15,2	timeout	94824			26%	119%	
51	822	1466	1,4	68738	7,1	91964	15,7	timeout	94733			25%	122%	
52	828	1472	1,4	69014	7,1	92374	15,2	timeout	94720			25%	116%	
53	840	1484	1,4	69678	7,3	92446	15,3	timeout	94239			25%	110%	
54	1981	3633	9,9	153860	64,4	207952	57,4	timeout	216121			26%	-11%	
55	1989	3641	9,7	154158	63,9	207495	55,4	timeout	214377			26%	-13%	
56	1994	3646	9,8	154328	63,9	207776	58,3	timeout	215197			26%	-9%	
57	2010	3662	10,0	153543	67,3	207741	65,2	timeout	214925			26%	-3%	
58	3675	6709	41,9	280880	270,4	370329	143,7	timeout	379971			24%	-47%	
59	3683	6717	33,4	280978	232,6	370434	138,9	timeout	378516			24%	-40%	
60	3692	6726	42,9	280991	274,9	370061	147,7	timeout	378734			24%	-46%	
61	3716	6750	42,3	281140	276,4	370178	138,6	timeout	379735			24%	-50%	
62	7998	14734	234,6	616357	1440,5	811862	468,8	timeout	infeasible solution			24%	-67%	
63	8007	14743	240,8	615963	1674,5	811597	475,9	timeout	infeasible solution			24%	-72%	
64	8013	14749	227,9	616282	1426,3	811940	474,7	timeout	infeasible solution			24%	-67%	
65	8017	14753	215,9	615929	1535,5	812275	547,9	timeout	infeasible solution			24%	-64%	
66	8062	14798	230,3	615710	1507,0	811674	448,0	timeout	infeasible solution			24%	-70%	
67	19083	35636	1142,7	1443131	7411,7	1937539	1883,8	not run	not run			26%	-75%	
68	19091	35644	1557,6	1443479	11728,7	1937771	2123,7	not run	not run			26%	-82%	
69	19100	35653	1805,4	1443520	12661,5	1938260	1943,7	not run	not run			26%	-85%	
70	19112	35665	1598,1	1443179	12941,0	1938552	2083,2	not run	not run			26%	-84%	
71	19177	35730	1763,8	1443293	12567,0	1937075	2349,9	not run	not run			25%	-81%	
72	38282	71521	11647,7	2780439	72687,5	3804838	7803,1	not run	not run			27%	-89%	
73	38294	71533	7039,1	2780487	50463,9	3804066	8236,1	not run	not run			27%	-84%	
74	38307	71546	6349,3	2780553	43667,4	3804218	8550,2	not run	not run			27%	-80%	
75	38418	71657	11722,0	2781681	67171,1	3802893	9108,2	not run	not run			27%	-86%	
76	6	9	0,1	6	0,0	6	0,7	1,7	6	0%	0%	0%	48050%	[2]
77	7	9	0,0	6	0,0	9	0,5	1,2	9	33%	0%	33%	35550%	[3]
78	13	21	0,0	16	0,0	33	0,6	1,4	33	52%	0%	52%	19023%	[4]
79	3997	10278	55,6	7994	193,3	10278	147,6	3889,2	10278	22%	0%	22%	-24%	[5]
80	783	2262	2,2	2146	7,0	2262	13,5	182,2	2262	5%	0%	5%	93%	[5]
81	1081	3174	4,2	3082	12,8	3174	19,9	337,8	3174	3%	0%	3%	56%	[5]
82	8	20	0,0	32	0,0	32	0,6	1,6	32	0%	0%	0%	28805%	[6]
83	10	15	0,0	10	0,0	12	0,6	1,9	12	17%	0%	17%	23322%	[7]
84	666	221445	1974,2	1001545679	901,9	1062614849	413,8	not run	not run			6%	-54%	
85	640	40896	73,5	3095773	78,9	3375566	36,2	not run	not run			8%	-54%	
86	17127	27352	894,3	399065	5752,3	489847	1279,1	not run	not run			19%	-78%	
87	27019	39407	2773,7	1221574848	20620,9	1483648581	2893,7	not run	not run			18%	-86%	
88	1728	28512	79,4	1562187	229,9	1777010	101,9	not run	not run			12%	-56%	
89	36711	68117	9775,4	487778	40444,3	550923	4399,2	not run	not run			11%	-89%	[8]
90	34479	55494	6116,2	391289	30798,0	443419	3872,4	not run	not run			12%	-87%	[8]
91	5880	35592	161,9	21218	6283,8	39027	39027	not run	not run			46%	-64%	[9]
92	3783	24186	71,5	15968	3187,5	28989	28989	not run	not run			45%	-37%	[9]
93	7624	27806	516,4	13183	9959,9	19087	19087	not run	not run			31%	-60%	[10]
94	129164	165435	144504,2	132500	0	0	0	not run	not run					[11]

Table 1: All Test Results. Do not hesitate to zoom in.

5 Discussion and Conclusion

5.1 Findings

5.1.1 Average Time

Excluding the test #94 on US roads, **H** took an average of 0.76 s, **A** 4.65 s, and **AP** 0.78 s to finish. The average time for ILP, where we could get an optimal result was 1.622 s, whereas if also counting tests, where we got an intermediate result, but no guarantee that this is an optimum, the average would be at 89.28 s instead.

5.1.2 Largest Input Graph Size

The largest graphs that can be efficiently handled on average hardware for each algorithm are approximately $|E| = 250000$ for **H**, $|E| = 130000$ for **AP**, $|E| = 90000$ for **A**, and $|E| = 280$ for **ILP**. Though for **A** and **AP**, it depends more on $|V|$ as well, the above numbers are valid for an average degree of about 10.

5.1.3 H Worst Result

Since ***H** vs. **A** Miss* was at most 46% in #91 for graphs where we could not find the optimal solution, and knowing that **A** provides a 0.5-approximation, we can conclude that at worst, **H** only found a 23%-approximation, which is nice, considering that in theory, it could be arbitrarily bad. The worst solution we can actually prove was a 48%-approximation in #78

5.1.4 A Worst Result

The optimum solution is only known for a few graphs, but there **A** could deliver a 92%-approximation at worst, meaning that it takes a very special graph to worsen the quality to 0.5 of the optimum.

5.1.5 Parallelization Speedup

AP managed to complete runs #72 and #89 in 89% less time than **AP** which means that it can fully use all 8 (virtual) cores on large graphs. Interestingly, an 8x speedup would only result in -87.5% running time, so there might be some further Java, Windows, or hardware optimization techniques at work, or it was just a coincidence.
The average compared running time for graphs with $|E| > 1000$ was -40%, while the global average is a terrible 1967% because of the outliers for tiny graphs.

5.2 Overview

What I conclude from this experiment data for the different algorithms is the following:

5.2.1 H - Heuristic

- Only suitable if a quality guarantee is not required
- An average quality of about 78% of the optimum
- Quality worse on denser graphs and graphs with very similar weights
- Often the fastest in comparison, especially for graphs with $|E| > 100000$

5.2.2 A - Approximation

- Offers a quality guarantee of 0.5
- Very good average quality of about 97% of the optimum
- Quality similar on all graphs
- Almost as fast as **H** for dense graphs with $|E| < 90000$ on modern hardware

5.2.3 AP - Parallelized Approximation

- Same quality as **A**
- Parallelization overhead makes it slower than **A** for $|E| < 3000$ (see #1)
- Performs especially well on real life instances of Max-Cut problems (see #89 and #90)
- Faster than **H** for some large ($|E| \approx 30000$), dense graphs

5.2.4 ILP - Integer Linear Programming

- Only discussed algorithm which guarantees an optimal solution
- Performance for graphs with $|E| < 130$ barely distinguishable on modern hardware if only doing one run
- Offers intermediate results of very good quality for graphs with $|E| < 5000$
- Unable to completely solve most graphs with $|E| > 500$ in a realistic time span (exception see #79)

5.3 Discussion

It is possible to adjust the heuristic in a way that the list of edges will not be sorted. That does not improve theoretical running time in big O notation, but in practice, especially for small, dense graphs, this does have quite some impact (**A** sometimes performed better for tiny graphs as it did not need a sort function), though it worsens the average result quality.

An even simpler heuristic would be flipping a coin for each vertex which determines to which subset it belongs, here as well the result is 'just' arbitrarily bad as well.
However, these approaches are not so useful for actual applications.

It has been proven, that it is NP-hard to approximate a Max-Cut to more than $\frac{16}{17}$ of the optimal solution [3], therefore, considering our implementation on average gives a 97%-approximation and in the worst observed case a $\frac{23}{25}$ one, I can be glad about the algorithm choice.

An alternative to the ILP method would be a bounded search tree. One could modify the heuristic to serve as a base for a BST algorithm: instead of always choosing (deterministically or at random) vertex u or v of the heaviest edge, we could branch on that decision, thereby creating every possible subset if we go deep enough. With the employment of appropriate algorithm engineering techniques, this could very well lead to another efficient solving method.

5.4 Algorithm Engineering Success

The idea of marking nodes instead of creating subsets made the code easier to read and stopped the process of finding elements in lists from having an impact on run time analysis. Instead, the subsets are added to lists after stopping the timer.

The idea of reducing ILP constraints was essential to even run LpSolve on larger graphs as it stopped the library from crashing, though it had no real effect on run times since the process of removing unnecessary entries takes less than a millisecond.

6 Test-case Instructions

I included a guide for installing the LpSolve library in readme.txt.
In the folder 'Resources' you can find the table and graphs I used in this document as well as 5 pictures of the graphs I used for the test cases, including optimal solutions. All installation files for LpSolve are included in the folder 'Installation Files'.
I did not include any JRE or JDK installation files since they are large and I assume the reader to have those installed already, but you can find a link to them in the LpSolve installation guide.
First, you have to compile the files. For that matter, navigate to the 'Code' folder and run:

```
javac exam/*.java
```

There might be a warning message for deprecated API and unsafe operations which come from the library, but this should not be an issue. To execute all of the 5 test cases, run:

```
java exam.Test 1
java exam.Test 2
java exam.Test 3
java exam.Test 4
java exam.Test 5
```

If you want to do even more testing, you can also use:

```
java exam.Test <filename>
```

You can find all graph files in the 'Code' subfolder titled 'files'. A good example would be 'b04.stp', which is a rather small graph that can be solved with ILP.

References

[1] Francisco Barahona et al. "An Application of Combinatorial Optimization to Statistical Physics and Circuit Layout Design". In: *Operations Research* 36.3 (1988), pp. 493–513.

[2] Claude Berge. *Théorie des graphes et ses applications*. Dunod, Paris, 1958.

[3] Johan Håstad. "Some Optimal Inapproximability Results". In: *J. ACM* 48.4 (July 2001), pp. 798–859.

[4] Guan-Shieng Huang. *Theory of Computation. Chapter 9*. National Chi Nan University, 2003, p. 8.

[5] Sera Kahruman et al. "On greedy construction heuristics for the MAX-CUT problem". In: *Int. J. of Computational Science and Engineering* 1 (Apr. 2007).

[6] Srijan Kumar et al. "Edge weight prediction in weighted signed networks". In: *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE. 2016, pp. 221–230.

[7] Srijan Kumar et al. "Rev2: Fraudulent user prediction in rating platforms". In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM. 2018, pp. 333–341.

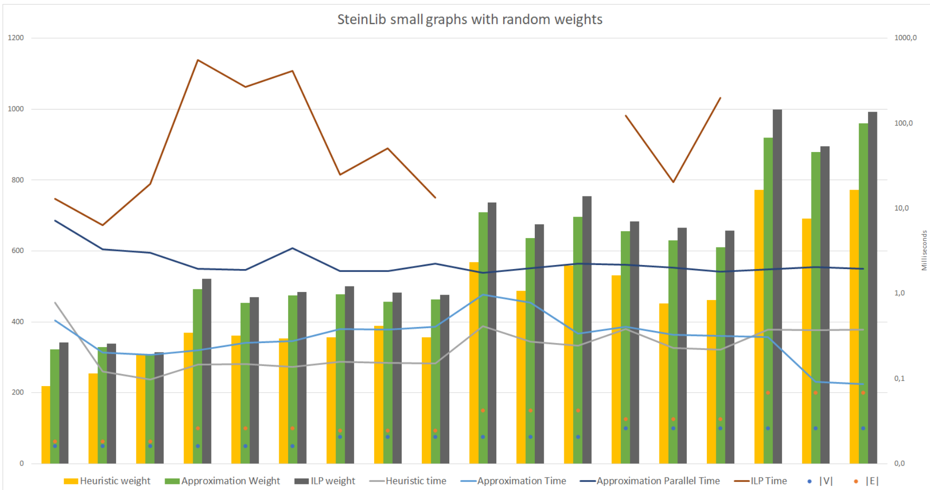
[8] Harry R Lewis. *Computers and intractability. A guide to the theory of NP-completeness*. W.H. Freeman, 1983, Appendix A2.2.

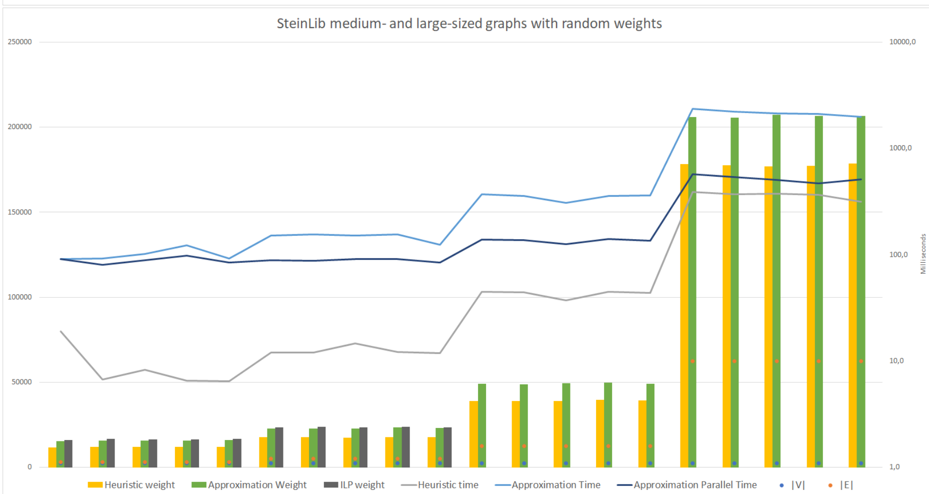
[9] Doina Precup. *MergeSort proof of correctness, and running time*. McGill University, 2014, p. 2.

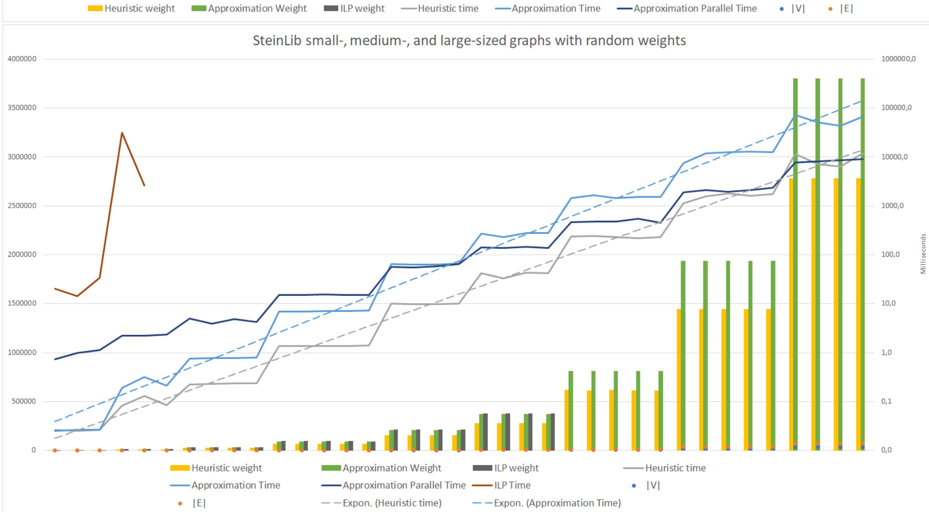
[10] Ryan A. Rossi and Nesreen K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015. URL: http://networkrepository.com.

[11] Benedek Rozemberczki and Rik Sarkar. "Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models". In: *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*. ACM. 2020, pp. 1325–1334.

[12] Sartaj Sahni and Teofilo Gonzalez. "P-Complete Approximation Problems". In: *J. ACM* 23.3 (July 1976), pp. 555–565.


SteinLib small graphs with random weights
1200
1000
800
600
400
200
0
1000,0
100,0
10,0
1,0
0,1
0,0
Milliseconds
Heuristic weight
Approximation Weight
ILP weight
Heuristic time
Approximation Time
Approximation Parallel Time
ILP Time
|V|
|E|


SteinLib medium- and large-sized graphs with random weights
250000
200000
150000
100000
50000
0
10000,0
1000,0
100,0
10,0
1,0
Milliseconds
Heuristic weight
Approximation Weight
ILP weight
Heuristic time
Approximation Time
Approximation Parallel Time
|V|
|E|


SteinLib small-, medium-, and large-sized graphs with random weights
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
1000000,0
100000,0
10000,0
1000,0
100,0
10,0
1,0
0,1
0,0
Milliseconds
Heuristic weight
Approximation Weight
ILP weight
Heuristic time
Approximation Time
Approximation Parallel Time
ILP Time
|V|
|E|
Expon. (Heuristic time)
Expon. (Approximation Time)