

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

Abstract

The abstract gives a short summary of the project. Begin by stating the motivation of the research at hand, describe the problem and shortly describe what methods you used to solve this problem. Finally, name the most important findings and provide a brief conclusion of your work.

1 Introduction

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 polynomial-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 maximization (or minimization) 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:

$$\text{maximise } \sum_{uv \in E} z_{uv} \quad (1)$$

$$\text{subject to: } z_{uv} \leq x_u + x_v, \quad (2)$$

$$z_{uv} \leq 2 - (x_u + x_v) \quad (3)$$

This describes 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 which 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 $E_{v_i} = \{(v_i, u) | u \in \{v_1, \dots, v_{i-1}\}\}$ 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:

$$|C| \geq \frac{1}{2} \sum_v |E_v|$$

$$|C| \geq \frac{1}{2} |E|$$

Input: Graph $G = (V, E)$
Output: Cut C
begin
 foreach $v \in V$ **do**
 if $|neigh(v) \cap S| \leq |neigh(v) \setminus S|$ **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

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.

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

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 shows the pseudo-code for this heuristic.

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 considered v in S in order to figure out if it is contained in S , we can generate a lookup table over all vertices 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.

The theoretical running time for the heuristic on

the other hand cannot be reduced. But it is possible to parallelise this algorithm very easily. We can simply divide the vertices into p equally sized, disjoint partitions, where p is the number of available processors. Each processor then gets assigned one of the partitions and adds each v in the partition to S randomly.

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 datastructure consists of an array over all vertices. For each vertex the set of outgoing edges is saved as an array of the indices of its 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 `solver` 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 its 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.

- **Advanced Algorithm:**

Give and explain the advanced algorithms that you used, and compare them to the basic algorithms.

- **Implementation:**

Explain how you implemented these algorithms and state what external libraries you used.

- **Algorithm Engineering Concepts:**

State the algorithm engineering concepts that you used and explain why they were helpful (if applicable).

4 Experimental Evaluation

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

4.1 Data and Hardware

Here, the input data and the applied parameters are described. Further, information about the underlying hardware such as main memory and CPU is provided.

4.2 Results

In this part, the results are presented. This includes comparison of running time and memory usage. To visualize the results we recommend one of the following graphing tools:

- **matplotlib:**

matplotlib is a library for python which allows the direct visualization of the produced results.

- **R:**

R is a programming language for statistical computations, which provides different graphing possibilities. One of the better known is ggplot2.

- **gnuplot:**

gnuplot is a command-line plotting program. It can also be used to graph data directly from different programming languages, such as Java and C++.

5 Discussion and Conclusion

In this section, the results are discussed and interpreted. Finally, the work is summarized shortly.

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: ?.