An Analysis of Local Search and Greedy Max-Cut Approximations

Andrew Morato

Abstract

The maximum-cut problem is an optimization problem on a graph comprised of vertices and edges where the goal is to partition the vertex set in such a way as to optimize the number of edges between both partitions of the vertices. This paper compares two maximum-cut approximations under varying conditions, contrasting their speed, accuracy, and robustness in original implementations of both algorithms.

1. Introduction

The maximum-cut problem is unsolvable by any known polynomial time algorithm. Thus, only approximations can be offered when searching for an efficient, "good enough" solution. Two such solutions will be described and thoroughly analyzed in this paper. The first approach, found in *Algorithm Design* by Jon Kleinberg and Eva Tardos, uses a local search technique to search for a good approximation of an optimal cut in the plane of all possible cuts. The second approach is described in *Yet Another Algorithm for Dense Max Cut: Go Greedy* by Clarie Mathieu and Warren Schudy and entails a simple greedy approach where vertices are put into the partition that would maximize the edges in the cut.

1.1. The problem

The maximum-cut problem has several variations which may lead to some confusion. For this discussion, it would be useful to constrain the general maximum-cut problem to a more specific one. We shall then restrict it to the following. Given an undirected graph G = {V, E} with positive integer weights on all edges, we seek to find a partition of the vertex set V → (A, B) such that the weight of all edges with one end in A and the other in B is maximized. We do not consider the weight of edges that connect vertices that belong to the same partition (i.e. edges from vertices in A to vertices in A) and any vertex in V, without restriction, may be assigned to partition A or partition B. The partitions A and B could also be described as a binary vertex coloring of vertex set V.

Alternatively, we can describe the maximum-cut problem as finding a bipartite subgraph of graph G where the edges' combined weight is maximized. Figure 1 shows an example of a maximum cut.

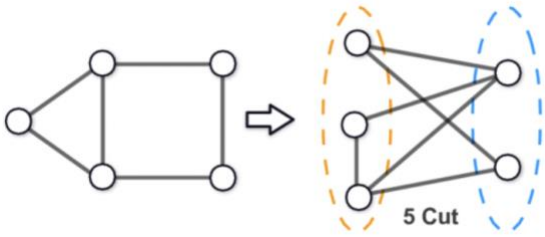

Figure 1: An example of a maximum cut

1.2. Motivation

The maximum-cut problem is similar to the minimum-cut problem in that they both attempt to find a cut in the graph. The minimum-cut problem is, however, solvable in polynomial time by the Ford-Fulkerson Algorithm. On the other hand, the maximum-cut problem is NP-hard; there are no known polynomial time algorithms that solve it. The difference lies in the attempt to maximize the "cost" of the cut rather than minimize it.

Since there are no known polynomial time solutions, we must consider approximation algorithms. These approximations do not guarantee an optimal cut in the graph but give good solutions quickly and efficiently and in some cases have solutions with a reliable minimum quality or "goodness" as we will explore in later sections.

Although we have limited the definition of the maximum-cut problem in this discussion, it may be interesting to note that certain variations of the

problem are solvable in polynomial time. For example, in planar graphs, graphs where edges intersect only at the vertices (resembling a coordinate plane), a cut with maximum weight can be found in polynomial time.

2. Max-cut via local search

The first approximation attempt employs a local search technique that explores the possible solutions (possible cuts) to the maximum-cut problem. The vertex set is initially randomly divided into partition A and partition B and then the algorithm progressively flips a vertex from one partition to the other if it would increase the cost of the cut. This approximation algorithm was proposed in Chapter 12 of *Algorithm Design* by Kleinberg and Tardos. For an in-depth explanation of the strengths and weaknesses of this algorithm, we need to begin with a brief discussion of the local search and its traditional pitfalls.

2.1. A brief discussion of local search

Local search is a general technique that describes any algorithm that explores the space of all possible solutions to a problem sequentially, moving from one solution to a "nearby" one. The idea is to move to better and better solutions to eventually find an optimal solution. In lay terms, we start by considering a solution and then jumping to similar solutions until we conclude that no better solution exists to jump to. There are two main components of any local search algorithm, the neighbor relation and the choice of a neighboring solution.

Once we consider a solution S, a "nearby" solution S' is defined as a neighboring solution to S if S' falls within the neighbor relation to S. Thus, we would say that "nearby" solutions to S are all solutions that have that neighbor relation with S. Local search algorithms have the flexibility to make up any neighbor relation between solutions. Typically, good neighbor relations are not "all-encompassing" but neither too restrictive.

After identifying a subset of solutions considered "nearby" we can pick a neighboring solution S' to continue progressing through the algorithm. This criterion is important in order to optimize the solution chosen. If no neighboring solution satisfies the criteria, the local search algorithm ends, and S becomes the approximated solution. A depiction of an iteration of a local search algorithm is shown in Figure 2.

Sometimes, greedily moving to a better and better solution does not lead to the globally optimal solution but ends in a local optimum. Escaping locally optimal solutions is a traditional shortcoming of local search algorithms and a difficult problem to overcome. There are several strategies to mitigate this difficulty, but none are conclusive solutions. In *Algorithm Design*, Kleinberg and Tardos discuss an approach called Simulated Annealing, that eagerly "backtracks" to escape local optimum early on in the algorithm but becomes more and more hesitant to abandon the current, possibly local, optimum as time progresses.

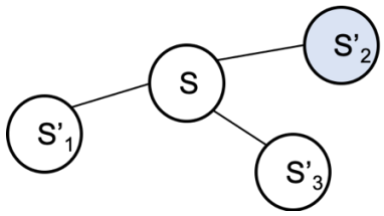

Figure 2: An iteration of a local search algorithm where S'2 is selected as the next solution

2.2. Max-cut via Local search approximation

This issue is not lost on the maximum-cut local search approximation. Again, the goal is to partition the vertex set into set A and set B such that the weight of all edges with one end in A and the other in B is maximized. To that end, we begin by randomly assigning the vertices into A or B. Once the partitions have been made, we set the neighbor relation to solutions that differ from the current one by a single flip of any one vertex from partition A to partition B. We call this difference a single-flip. From the neighboring solutions S' (those that differ from S by a single-flip), our criteria for choosing a neighboring solutions is to pick the flip (if any exists) that most increases the total weight of the edges from A to B.

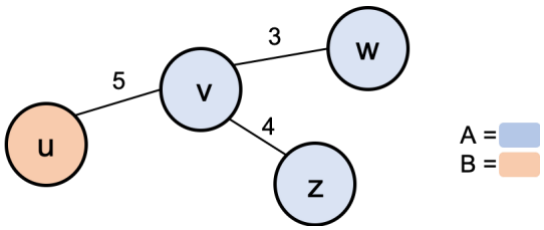

Figure 3: A sample iteration of the local search max-cut approximation algorithm

In other words, in any partition (A, B) of the vertex set, if there is a vertex *v* such that the total weight of the edges from *v* to vertices on its own side of the partition exceeds the total weight of edges from *v* to vertices on the other side of the partition, then *v* should be flipped to the other side of the partition. An example is shown in Figure 3, where the weight of edges from *v* to vertices in A (*w* and *z*) exceeds the weight from *v* to vertices in B (*u*), so *v* should be flipped from A to B. This local search algorithm ends when there is no single-flip that would increase the weight of the edges spanning from A to B.

2.3. Analyzing the local search algorithm

Let us say A and B are the partitions that we have obtained as a result of this local search maximum-cut approximation algorithm. Since this is only an approximation, it is possible (if not probable) that this solution is some local maximum and not the globally optimal solution. If A* and B* are the globally optimal partitions that produce the maximum cut in this graph, Kleinberg and Tardos show that cost of the cut between partitions A and B, cost(A, B), is at least half as good as the globally optimal solution. That is,

$$\text{cost(A, B)} \leq \frac{1}{2}\,\text{cost(A*, B*)}$$

Furthermore, since there is no guarantee that locally optimal solutions can always be found in polynomial time, the run time for this algorithm is only pseudo-polynomial.

2.4. Variations

In search for a guaranteed polynomial time approximation, Kleinberg and Tardos propose a variation on what they call their Single-Flip Algorithm. Rather than continue flipping vertices that increase the cost of the cut, this variation stops when there are no "big enough" improvements. The neighbor relation then becomes a big-improvement-flip, which is defined as a flip that increases the total cut value by at least

$$\frac{2}{|V|}\,cost(A, B)$$

This yields an approximation nearly as good as the Single-Flip Algorithm, with the advantage of guaranteed polynomial run time.

3. Max-cut: go greedy

In *Yet Another Algorithm for Dense Max Cut: Go Greedy*, Mathieu and Schudy present several greedy approaches to approximate the maximum cut in a graph, where we consider the vertices from the vertex set V in random order and assign them to the partition that would most increase the cost of the cut. Although greedy approaches can be seen as overly simplistic, naïvely picking the best choice now without regard to other considerations, they can be effective in their applications.

3.1. Max-cut via greedy approximation

In the first and simplest algorithm described by Mathieu and Schudy, Algorithm 1, we consider all vertices from the graph, one-by-one in random fashion, and place them in either partition A or B, depending on the edge weights to neighbors that are already assigned to A or B. We greedily assign each vertex to the partition that would maximize the cost of the cut. If vertex *v* has no neighbors assigned to A or B (likely the case towards the beginning of the algorithm) or if the combined weights of edges to neighbors in A and neighbors in B are equal, *v* is randomly assigned a partition. Algorithm 1 ends once all vertices are placed in one of the two partitions.

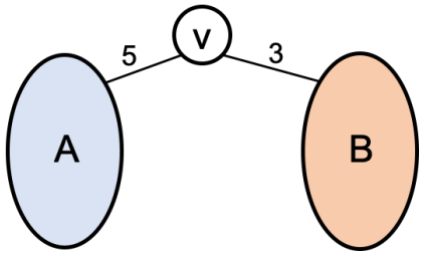

Figure 4: A sample iteration of the greedy max-cut approximation algorithm

For example, in Figure 4 we consider placing vertex *v* in partition A or B. The weight of edges from *v* to *v*'s neighbors in A is 5, whereas the weight of edges from *v* to *v*'s neighbors in B is just 3. Therefore, we'd assign *v* to partition B since that would add 5 to the cut rather than assigning *v* to partition A, which would just add 3.

Algorithm 2 builds off its predecessor. After obtaining the partitions (A, B) by running Algorithm 1, Algorithm 2 revisits all the vertices, again in random fashion, and moves them to the other partition

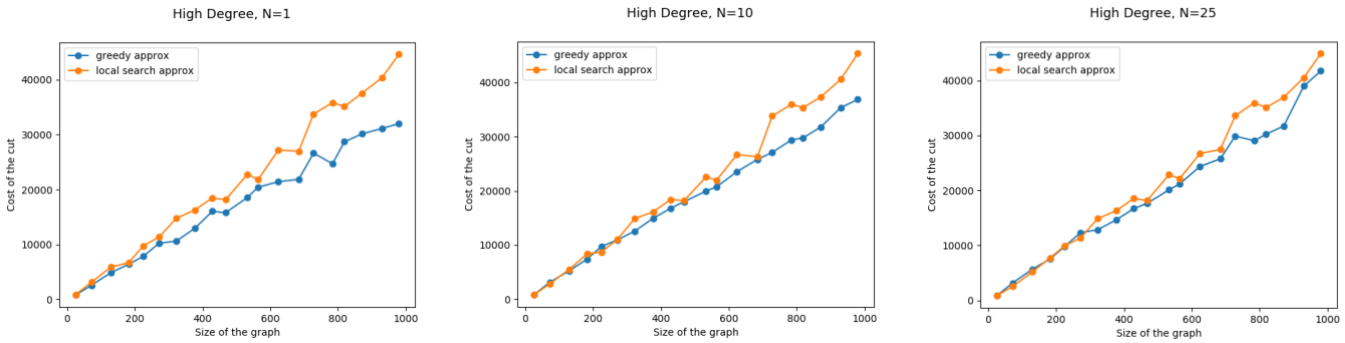

Figure 6: Experimental results on graphs with high average vertex degree

if doing so would increase the total cost of the cut. The main advantage lies in reassigning (if needed) the vertices that Algorithm 1 randomly placed in partition A or B due to having no pre-existing neighbors in A or B. Thus, Algorithm 2 improves over Algorithm 1 by potentially increasing the cost of the total cut by revisiting the vertices Algorithm 1 "ignored".

3.2. Analyzing the greedy algorithms

Analysis of the runtime of Algorithms 1 and 2 is very simple. Both algorithms consider each vertex in the vertex set and furthermore all the edges for each vertex. If the graph were a fully connected graph, where each vertex has an edge with every other vertex, the runtime would be

$$|V| * |E|$$

Since most graphs we deal with likely won't be fully connected graphs, for a rough estimate, we may reduce the runtime to

$$|V| * average\_degree\_per\_node$$

3.3. Variations

One further variation remains, although it may be somewhat impractical. Algorithm 3 involves the repetition of Algorithm 2 an arbitrary amount of times. Algorithm 2 will consistently produce different results since the algorithm is largely based on randomness. Algorithm 3 compares these results and picks the one with the maximal cut value. If Algorithm 2 were repeated enough times, specifically $2^{2^{poly(|V|)}}$ times, we could guarantee near-optimality. Unfortunately, this sort of runtime is far too long for an approximation algorithm.

4. Experiments and results

These two algorithms, the local search technique and the greedy approach, are used to approximate the maximum cut in a graph. Both algorithms were fully implemented in Python3, relying on some external packages for trivial functions (randomness, some mathematical functions, etc.) and OpenCV 4.0.0 and Matplotlib for visualization of the algorithms, to facilitate experimentation and comparison of the results.

4.1. Experimental setting

Unfortunately, I could not find large datasets of varying size and features for the experimentation and testing of both algorithms. Many datasets were of different formats and contained excess information, so I implemented a graph generator appropriate for this task. As depicted in Figure 7, the generator takes as input the number of vertices and edges (i.e. |V| and |E|) and produces a graph G = {V, E} where the degree of each vertex is the minimum required so that the total amount of edges in the graph would reach the desired inputted quantity |E|.

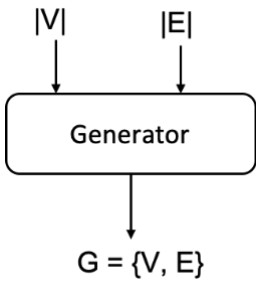

Figure 7: Block diagram of the generator

The generator allowed for the production of datasets with varying features, including datasets where vertices have high average degree, datasets where vertices have low average degree, and datasets of varying vertex set size.

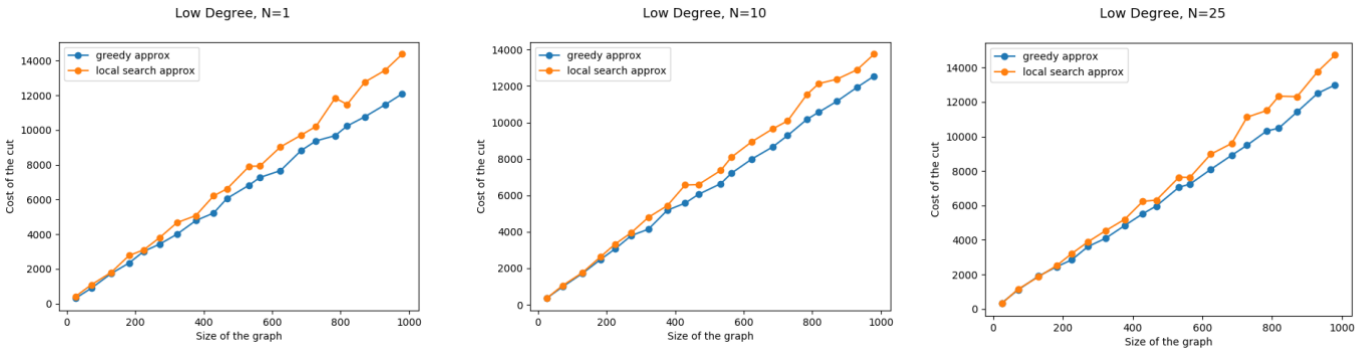

Figure 8: Experimental results on graphs with low average vertex degree

4.2. Experiments

Using this generator, several graphs were created to experiment on. Each instance of an experiment was made on graphs beginning with 25 vertices (|V| = 25), increasing in vertex count by 50, and ending with graphs containing about 1000 vertices (|V| = 1000). Experiments also varied with repeated runs of Algorithm 2 (n=1, n=10, n=25). The following experiments were carried out:

1. Graphs with High Degree and n=1
2. Graphs with High Degree and n=10
3. Graphs with High Degree and n=25
4. Graphs with Low Degree and n=1
5. Graphs with Low Degree and n=10
6. Graphs with Low Degree and n=25

Recall, the greedy approach can be repeated on the same graph to potentially improve the random-seeded result. More and more repetitions, like those done in Algorithm 3, lead to better results but sacrifice runtime.

Visually, graphs with a high average vertex degree resemble connected, "bushier" graphs whereas graphs where vertices have lower degrees tend to have many long walk or paths and appear sparser, albeit larger.

4.3. Results and analysis

Figure 6 shows the experimental results carried out on graphs where vertices have a high average degree (experiments 1-3) and Figure 8 shows the experimental results made on graphs where vertices have a low average degree (experiments 4-6). It is notable that in all the experiments, the local search approximations outperform the greedy approximations, especially the greedy approximations without repetition. As we know from the discussion on Algorithm 3, the more repetitions lead to better results. As more attempts are made, the probability of more optimal (even perhaps globally optimal) results increases. Eventually, as repetitions increase, highly repeated greedy approximations do in fact outperform the local search approximation but at the cost of suffering in runtime efficiency.

Both approximations, but particularly the local search approximation, perform better as the average vertex degree increases. In the case of greedy approximations, that is likely due to an increase in the benefit of assigning vertices to partition A or B (higher cost) and an increase in the probability of neighbors already being assigned into partition A or B due to a greater amount of neighbors and the graph being more connected. In the case of the local search algorithm, the increase in performance is likely due to the greater accuracy of each single-flip since each vertex has more neighbors to consider.

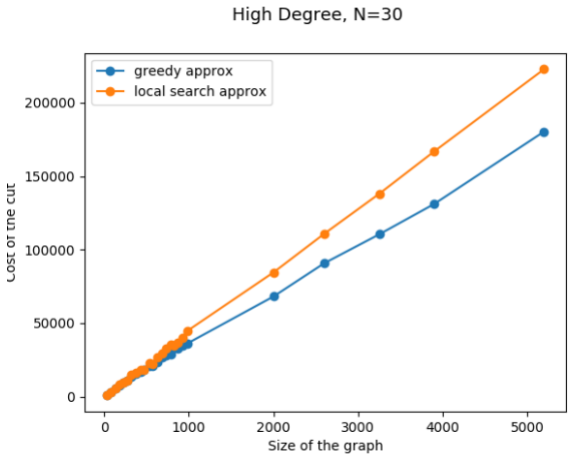

Figure 9: Experiment with large graph sizes

Finally, as the graphs increase in size, there is more deviation between the approximation results. This can be seen more clearly in Figure 9, where the graph sizes have been significantly increases to show this phenomenon. The greedy approach maintains a

roughly linear curve, indicating constant performance regardless of the graph size, whereas the local search approximation increases not only the value of the cut, but its performance as the graph size increases. This behavior arises due to the local search approximation's penchant for finding a local maximum. Local optima are likely to be more drastic, and therefore better, as the graph increases in size, thus the local search approximation will improve as well. In contrast, the greedy approach is expected to produce constant results regardless of the graph size due to its graph-agnostic approach in assigning vertices to partitions.

5. Discussion

Both the local search and greedy maximum-cut approximations differ in runtime and quality of solutions. It seems however, that the local search approximation thrives and yields better solutions on particular graphs while the greedy approximation offers more constant (constantly mediocre) results. The local search approximation may offer more upside albeit at a slightly higher runtime cost. Furthermore, to match the quality of solution offered by the local cost approximation, the greedy approximation must employ a repeated approach with a significantly high number of repetitions. Unfortunately, this adjustment would yield an unfavorably high runtime for the greedy algorithm, thus disqualifying it from consideration. It remains then, that the local search approximation is favorable in most cases.

5.1. Improvements

After researching, implementing, and comparing both approaches, I think they can be somewhat combined to form promising results. Local search algorithms tend to "fall in" to the same local optima if they begin at similar starting points. However, if the starting points are "different enough", it is likely that the same local search algorithm will fall into different local optima. If we examine Figure 10, we can see that the blue and green points will fall into the same local minima whereas the orange point will fall into a different one.

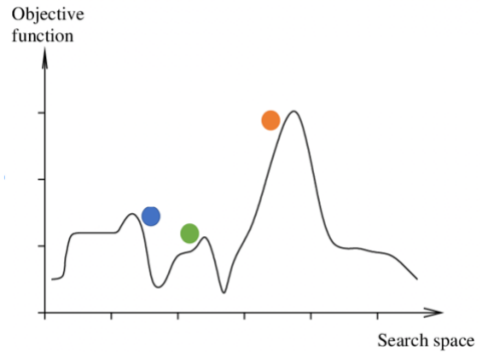

Figure 10: Landscape of a local search problem

If we began with random partitions, filtering out randomly generated partitions of the vertex set that are too "similar" and keeping only the partitions "different" enough to fall into different local optima, and then apply the local search approximation we can perhaps improve the total output by only considering different local maxima. This increases the chance of one of them being the globally optimal solution.

References and disclosure

This paper is a comparison of two algorithms that are not original to me. The only original work is the implementation of both algorithms and any comparisons made between them in section 4 and the discussion in section 5. The local search algorithm may be attributed to Jon Kleinberg and Eva Tardos and the greedy algorithms may be attributed to Claire Mathieu and Warren Schudy. Furthermore, I have heavily relied on several references in this discussion. They are as follows,

[1] *Algorithm Design 1st ed.* – by Jon Kleinberg and Eva Tardos (2006).

[2] *Introduction to Algorithms 3rd ed* – by Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein (2009).

[3] *Yet Another Algorithm for Dense Max Cut: Go Greedy* – paper by Claire Mathieu and Warren Schudy (2008).

[4] *Max-cut, Hardness of Approximations* – lecture by Anupam Gupta at Carnegie Mellon University (2014)

[5] *Discrete Mathematics: An Open Introduction 3rd ed* – by Oscar Levin (2013)