The Prize-Collecting Steiner Forest Problem

Kiss Bendegúz
Supervisor: Dr. Király Tamás

1 Introduction

The Prize-Collecting Steiner Forest Problem is about finding a forest F in a weighted undirected graph with a given penalty function over all unordered node pairs, which minimizes the total of its edge costs and penalties for pairs not connected in F. We will give a proper definition later.

This semester I continued my project with the Prize-Collecting Steiner Forest Problem often referred as PCSF. My main goal was to get familiar with a 2-approximation algorithm described in [1] which calls a 3-approximation algorithm as a subroutine also described in the book. During this semester I have finished implementing the 2-approximation algorithm along with a heuristic algorithm, so I could start testing them. The results will be described in the last section.

2 Description of PCSF

Definition: 1 *Given an undirected $G = (V, E)$ with non-negative edge costs $c : E \mapsto \mathbb{R}_0^+$ and a non-negative penalty function $\pi : V \times V \mapsto \mathbb{R}_0^+$ which is 0 over node pairs (i, j), where $i \geq j$. The goal is to find a forest, which minimizes the following expression:*

$$\sum_{e \in F} c_e + \sum_{(i,j) \in Q} \pi_{ij}$$

where Q contains all disconnected node pairs in F.

The PCSF is a general version of the PCST problem (Prize-Collecting Steiner Tree), where we search for a tree and pay penalties for the nodes not included in the solution. A simple generalization is the following: We try every node as a root, which means we include them in the solution anyway. Now, the penalties paid for unreachable nodes are the same as the penalty paid for the disconnected node pairs containing the root.

3 Algorithms for PCSF

3.1 3-approximation algorithm

The algorithm which I have worked the most is a 3-approximation algorithm described in [1] from which we can easily obtain a 2-approximation algorithm by iterating it. The algorithm is based on a coloring procedure. During the algorithm we maintain active sets, initially every node. The main idea is that these sets color their outgoing edges for some time until one edge becomes fully colored, which means that the coloring duration of the edge reaches its edge cost. This is called the static coloring procedure, and its duration in an iteration is denoted by deltaE.

To deal with the penalties we use an other coloring procedure called dynamic coloring. Now we color the node pairs until one of them reaches its penalty. This one is a bit difficult to visualize, so I explain it briefly. The main idea is that we build a digraph called SetPairGraph where we have nodes for all sets that we colored with before and for all node pairs. We add a source node and a sink node as well. There will be edges from source to the set nodes with capacity y_S where y_S is the coloring duration of S on its outgoing edges. There will be edges from every S set node to the (i, j) pair nodes based on whether S cuts the (i, j) pair (contains only i or j) with infinite capacity. Finally, we have an edge from every (i, j) pair node to the sink with π_{ij} capacity. We want to represent a static coloring procedure but this time the bounds are the penalties. Coloring with a set now is like increasing the capacity of the active sets and running a max-flow algorithm. If the set edges (from source to set nodes) are full then we call this procedure valid static coloring. We need to find a coloring like this by uniformly increasing the capacity on the active set edges. This duration is denoted by deltaP.

Given deltaE and deltaP we take a minimum. Let it be delta. We color the edges for delta time. It is important to note if an edge connects two active sets it will be colored twice as fast than the others. If an edge is fully-colored we add it to the forest. In the background the pairs are also being colored during the process. To visualize it we just need to run a max-flow algorithm in our SetPairGraph with the new capacities on the edges from source to set nodes and the flow value on the pair edges (from pair to sink) will be the coloring duration of the pairs. If an edge like this is full, then we call it tight. If we have a set, which cuts only tight pairs we call it tight. This means we are willing to pay the penalties cut by the set. After a coloring iteration we remove all tight sets from the active sets.

The algorithm terminates when there are no active sets left. In the end we have a nice procedure for reducing the tight pairs, for which we would pay penalties. After the non-tight pairs will be connected in our forest and we can delete those edges which do not contribute to connect a non-tight pair.

About the running time we can say that is polynomial, since all subroutines we use during the algorithm are polynomial, and there can be a linear number of active set, but it mostly depends on the number of edges. If the graph is sparse the components are build much slower due to low-connectivity but if it is dense it happens much faster. I will write about this during the testing chapter.

3.2 2-approximation algorithm

This algorithm uses the previous approach by running it two times. First, runs it normally and in the second run it sets the penalties 0 for pairs for which we paid penalties in the

first run. After it evaluates it with the given penalties and compares the two solutions. The book proves that it gives us a 2-approximation solution.

3.3 A heuristic approach

I developed a heuristic algorithm, which sometimes works quite efficiently compared to the 2-approximation algorithm. The main idea is that we want to put the nodes in different categories depending on we are willing to include them in our solution or they can be left out. Two obvious observation can be made. If we have a node and we sum all penalties including this node and it is bigger than the cost of the maximum cost outgoing edge, then we definitely include this node in our forest, since we do not want to penalties for it. If the sum of the penalties is less than the cost of the minimal cost outgoing edge, then we can leave it out because including any edge from this node will be a worse solution. We separate the nodes by these parameters into necessary nodes and useless nodes. Obviously, there are nodes that cannot be put any of these categories. We will call them the question nodes.

Now, there are a lot of ways to deal with question nodes. I tried the following: if at least half of the penalties including a question node are 0 then we put in the useless nodes category, otherwise the necessary nodes category. This is a simple process, but if we do not generate penalties for every node pair it is quite efficient.

Finally, we search for a minimum spanning tree/forest (usually the graph is connected) including the necessary nodes. That tree/forest will be our solution.

I have not tried any other possibilities, but for instance a possible way can be if we look at some ratio of the penalties and edge costs for a node. We can think of a rule for separating the question nodes like this. A wrong approach is when we want to separate them by looking at which edge cost is closer to the sum of penalties. This usually relies on the node number.

4 Implementing and testing

4.1 Implementation

This semester I finally implemented the approximation algorithms in Python using the networkx package for graphs. There were some changes, that I made in my code compared to the book. For the coloring durations the book uses an y vector, where y_S means how much time we colored with the S set. If we want to test the algorithm it is not really efficient if we have an array with exponential size, so I had a list for the coloring durations and when one of the sets colors, I put the set in y with its cutting edges and the coloring duration. From the book we can see this y will be a polynomial sized list. Another advantage is that we can determine which set colored a specific edge and for how long.

The algorithm has several subroutines, some of them was easy to implement but for example finding deltaP was quite difficult due to constructing a specific graph and running a max-flow algorithm on it.

I used built-in algorithms in some of the subroutines. For example, to find a max-flow I used a networkx function. At the end of the algorithm I used built-in functions to find shortest paths and check if two nodes are connected. This is needed when we want to decide which edges can be deleted to obtain the final forest.

4.2 Testing

The main issue was to find the right parameters for testing. I will go through each one of them one by one.

First, I always used 100 nodes for testing and I chose the parameters accordingly the number of nodes. I used the Erdős-Rényi model to generate graphs. I chose $p = 1/10$ for the edge probability, so we do not get very dense graphs. In these cases, the algorithm usually does not pay penalties.

Edge costs were integers generated uniformly from [1,10]. The reason behind this the huge running time. For a much wider interval, the components are built much slower, so running the algorithm can take an hour, for instance, if we generate from [1,100].

The penalties were generated by the following procedure: we randomly decide for each node pair, if we want penalties on them. We get two parameters from this procedure: penalty generation probability and the penalty cost. I gave penalty 20% of the nodes and the penalty was 1. The penalties cannot be much bigger because the edge costs are between 1 and 10 and we do not pay penalties in most of these cases. The probability parameter is important because we get nodes that can be easily left out from the solution, but we get also nodes that should be included. It helps the heuristic algorithm as well.

5 Results

I tested the algorithm with the parameters mentioned above. The results will be shown on the last page. We can see that the second iteration helps a lot in most cases. Sometimes the edge cost is the same, but the 2-approximation algorithm does not pay penalties. The heuristic approach sometimes better than the approximation algorithm but if it worse, then it much worse. The approximation algrotithms run slow, approximately 8-9 minutes. The heuristic approach is obviously really fast. The m parameter is the number of edges.

6 Future plans

Firstly, I will search for possibilities to improve the running time. Secondly, I will try to think of any other heuristic approaches (maybe using the idea I mentioned before), so I can compare them with my implemented algorithm. Finally, I will search for possible applications of PCSF.

7 Appendix

#	m	Heuristic			3-Approx			2-Approx		
		Cost	Penalty	Time (s)	Cost	Penalty	Time (s)	Cost	Penalty	Time (s)
1	481	175	0	0.14	183	49	444.16	168	7	552.44
2	500	154	0	0.07	163	47	393.37	163	0	397.29
3	494	168	0	0.09	169	43	461.58	124	21	426.65
4	481	166	0	0.09	181	54	472.60	118	23	421.72
5	464	180	0	0.09	198	59	591.07	190	6	477.45
6	456	164	0	0.07	175	57	399.08	118	19	352.72
7	498	137	0	0.08	157	48	320.73	114	16	274.05
8	475	166	0	0.08	173	52	390.46	173	0	371.17
9	496	174	0	0.07	184	56	396.20	103	31	314.22
10	497	154	0	0.06	168	47	348.00	164	1	336.47
11	505	177	0	0.06	197	64	432.69	189	3	480.04
12	514	178	0	0.09	189	57	716.42	183	6	613.20
13	489	151	0	0.12	168	41	547.06	168	0	554.86
14	510	144	0	0.10	157	48	506.97	154	2	543.36
15	517	172	0	0.11	178	64	551.24	170	11	619.86

Table 1: Summary of PCSF results on 15 randomly generated graphs with 100 nodes and edge probability 0.1. Penalty probability: 1/5, Penalty range: {0, 1}

References

[1] Ahmadi, Ali, et al. "2-approximation for prize-collecting Steiner forest." Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). Society for Industrial and Applied Mathematics, 2024.