Computational Hard Problems

Emil Bonne Kristiansen (s165200), Johan Hagelskjær Sjursen (s154674)

October 2020

	a)	b)	c)	d)	e)	f)
Johan			100%	50%	100%	50%
Emil	100%	100%		50%		50%

Table 1: Distribution of Labour

$$\max\left\{\sum_{e_i\in T} w(e_i), \sum_{e_i\in T} w(e_{m+1-i})\right\} \geq B \tag{1}$$

a) (Emil)

In colloquial terms the (decision version of the) MST problem asks if an undirected weighted graph contains a spanning tree that doesn't exceed a given number when summing the weight of the edges. The MFMST problem adds to that definition by also requiring the weight of the edges picked in the graph by indices to have a sum less than the same given number when the list of edges is mirrored (reversed order).

In other words there needs to be at least two ways to sum the weight of $n-1$ of the edges in the graph and have it be less than the given number. One has to be a spanning tree, the other has to be those exact same edges, however where every edge is switched by mirroring the list of edges. This second set of edges does not necessarily make a spanning tree in the graph. We will refer to the subgraph induced by this second set of edges as the spanning tree's mirroring and denote it by a mark (i.e. T').

While the MST problem can be answered by using a visual representation of a graph instead of a mathematical structure, this is not the case for the MFMST problem. This is because in the MST problem, the order of the edges is irrelevant, but in MFMST that is not the case.

Solving an instance of the problem:

$$G = \left\{\begin{array}{l} V = \{1,2,3\}, \\ E = \{e_1 = \{1,2\}, e_2 = \{2,3\}, e_3 = \{1,3\}\}, \\ w = i, \text{for} i \in \{1,2,3\} \end{array}\right\}$$
$$B = 4$$

We must first consider possible candidates C for T. Since T has to be a spanning tree we can simply take the power set of E and eliminate all but three possibilities.

$$C = \{\{e_1, e_2\}, \{e_2, e_3\}, \{e_3, e_1\}\}$$

This step of selecting candidates for T was trivially done in this example, but could be harder in bigger graphs since checking if a set of edges form a spanning tree is not trivial. Finding all spanning trees can at worst take polynomial time[1].

Now we will find each candidate's mirroring. That is, we will mirror the list of edges in E, but use the same indices as before:

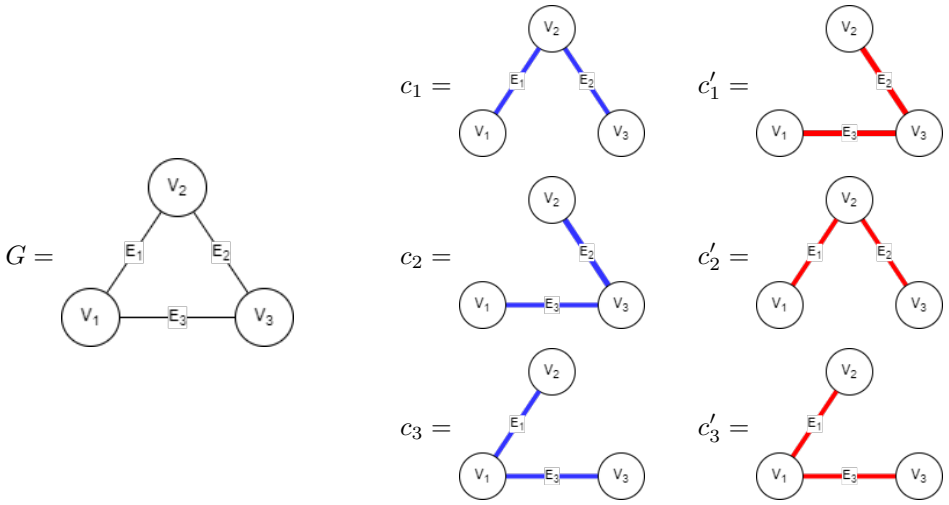
$$c_1 = \{e_1, e_2\} \qquad c_2 = \{e_2, e_3\} \qquad c_3 = \{e_3, e_1\}$$

$$\Downarrow$$

[1] kirckthoffshaska

$$c'_1 = \{e_3, e_2\} \qquad c'_2 = \{e_2, e_1\} \qquad c'_3 = \{e_1, e_3\}$$

Now let us visualise the graph along with the three spanning trees, and their mirrorings.



Now we take the sum of all the weights of edges in each candidate for T as well as for each mirroring. We use the cost function w to calculate each edge's weight. In this example we have a very simple cost function that just make each edge weigh as much as its index in E. When applying this cost function these are the resulting sums:

	1	2	3
Spanning tree(c_k) sum	3	5	4
Mirroring(c'_k) sum	5	3	4

We can now see that the candidate c_3 has both sums ≤ 4, and thus we must answer YES to this instance of the problem.

b) (Emil)

In this section we will show that MFMST is in $\mathcal{NP}$.

First we must define a deterministic algorithm A that takes a problem instance X and a guess R, and returns whether or not the guess is right. A problem instance will be just as before: a graph consisting of vertices(V), edges(E) and a cost function(w) as well as a number B. We will encode guesses by a binary string of length m(number of edges). Each binary number will indicate whether or not the edge at that index in E is in our guess of T. So for the problem instance in section a, a correct guess could be encoded as $R = 101$.

This method of encoding a guess R has finite alphabet(2 symbols) and finite range(m). It can encode all subgraphs and not just spanning trees, but this is not an issue since the set of all spanning trees in a graph is a subset all the subgraphs in that graph.

It does mean that for A to verify if R is a correct guess, A must first verify that R does in fact represent a spanning tree. Next A must calculate the sum of the weights of the edges as well as the sum of the weights of the mirrored edges. If they are both below B, then A will approve the guess.

We thus have arrived at a description of A:

1. Check that R encodes a spanning tree, if not return NO
2. Check that R encodes a subgraph with total weights less than B. And that the sum of the edges in the mirroring of that subgraph is also less than B. if not return NO
3. Return YES

Next we will show that if the answer to X is YES, then there exists a string R^* (randomly created with positive probability) such that $A(X, R^*) = \text{YES}$.

The language used for passing guesses to A can be used to represent any subset of edges E in G. T is defined in the MFMST problem definition to be a subset of E. It follows that if an instance of the problem has a solution T, then that solution must be a word in the language used to pass guesses to A. If that word is passed to A then the algorithm would be answering YES. Since the words in the language is of finite range and uses a finite alphabet, the language can also be said to be finite. So there exists a word r^* that represents a correct answer T which will will result in the algorithm returning YES. We know that R^* can be randomly created with possible probability since the language is finite.

Next we will show that if the answer to X is NO, then there exists all strings R will make the algorithm return NO.

Any word R would be interpreted as a guess of a set of edges that form a spanning tree. The first categories of strings we can consider are all the ones that represents sets of edges that is not spanning trees. These would get the answer NO. Next we can look at those words that does represent spanning trees. These can be split into two groups, those that have a sum of edges and a mirrored sum of edges that is less than B, and those that do not. But in the context of problem instances with the correct answer NO, we know that the second group has no members. If it did, then the problem instance would have an answer. Thus we have showed that any word R representing a spanning tree would also get NO.

Finally we must show that A is p-bounded for some polynomial p. We will do this by examining the worst case complexity of A and afterwards seeing if we can find a polynomial that bounds it. To determine if the subgraph given by a R is a spanning tree two properties must be ensured:

1. The subgraph is non-cyclic
2. The subgraph is connected

Both can be detected by a single Depth First Search traversal. By performing DFS and seeing if all vertices gets visited and all edges are being traversed, A can detect if R represents a spanning tree. DFS has a bound complexity of $n + m$. Next we will determine the complexity of calculating the sums, finding the bigger one and comparing it with B. The spanning tree

sum has complexity $n-1$ as it simply iterates and sums up the weights in the subgraph. The second mirrored sum also has complexity $n-1$ as it does the same except the edges are picked by mirroring the list of edges. Finally the act of finding the bigger sum and comparison with B both has linear complexity.

We can thus say that A has a complexity bounded by $O(n+m)$.

c) (Johan)

Now that we know MFMST is in NP, let us show it is NP-complete. To show that MFMST is in NP-complete, we will perform a polynomial time reduction from one of the reference problems given, namely PartitionByPairs. In other words, we will show that PartitionByPairs $\leq_p$ MirrorFriendlyMinimunSpannignTree. This is done in three steps, transformation of input, showing if the answer to the input is yes for PartitionByPairs it is also yes for our transformed input to MFMST, and finally showing only if, meaning the answer to the instance of MFMST is only YES if the answer to the PartitionByPairs instance is YES.

Transformation

First, let us revise what an instance of PBP looks like:

Problem: [PARTITIONBYPAIRS]
Input: A sequence $S = (s_1, s_2, \ldots, s_{2n})$ of natural numbers.
Output: YES if there is a subset $A \subseteq \{1, \ldots, 2n\}$ choosing exactly one from each pair $(2i-1, 2i)$, where $i \in \{1, \ldots, n\}$, such that $\sum_{i \in A} s_i = \sum_{i \in \{1,\ldots,2n\} \setminus A} s_i$, and NO otherwise.

As seen above, an instance of PBP defined only by one input, namely S which is a list of an even amount of natural numbers. Now let us transform this list to edges, vertices, a number B and a weight function.
For vertices, we propose we make $|S|+1$ new vertices. This means if we had 12 natural numbers in S we would have 13 vertices in our graph.
For edges, we propose that the undirected connected graph will consist of $2|S|$ edges. Again, if we had 12 natural numbers, we would construct 24 edges.
The number B would be calculated to the sum of all values in S. If the sum of all values in S is not even, we can just return NO right away, since it is (by definition) not possible to perform PBP in this case anyway.
For the weight function, the weight of an edges would be determined thusly:
There are $2|S|$ edges. The weight of edge number i is $w(e_i)$. If e_i can be mapped directly to a value in S, namely S_i, let $w(e_i) = S_i$. If this is not possible, meaning the index of the edge exceeds legal indexes of S, the the weight of edge i corresponds to the value in S at index $i - \lceil \frac{i+2-|S|}{2} \rceil$ where $|S|$ is the amount of elements in S. This can also be written as:

$$w(e_i) = S_{\lceil \frac{i+2-|S|}{2} \rceil} \text{ if } i > |S|$$

Since all numbers in S are natural numbers, this is a valid mapping for the weight function.

The final hurdle in our transformation is how to create the $|S|$ edges. For this, let us utilise an example:

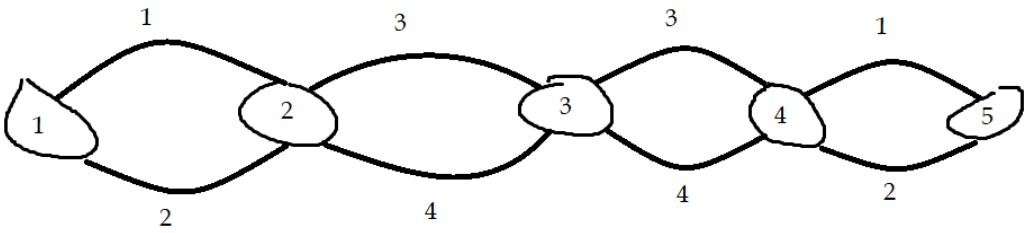
Given $S = (1, 2, 3, 4)$ and $V = \{1, 2, 3, 4, 5\}$ we create E in the following way.
For every vertex in V with index i we create two edges going between vertex $\lceil \frac{i}{2} \rceil$ and vertex $\lceil \frac{i+2}{2} \rceil$ and add it to the list of edges. When we have been through all the vertices in V from the example above, we would have the following edges:

$$E = \{\{1, 2\}, \{1, 2\}, \{2, 3\}, \{2, 3\}, \{3, 4\}, \{3, 4\}, \{4, 5\}, \{4, 5\}\}$$

The weight of these edges would the be, as defined by $w(e_i)$:

[1,2,3,4,3,4,1,2]

Thus the graph would look like this:



All of these transformations can be done in linear time, thus we have a polynomial time reduction from a problem instance of PBP to a problem instance of MFMST.

Show If and Only If

A tree in the constructed graph will have to through exactly one of two edges between two vertices. Traversing a tree through the first half of vertices to the middle vertex corresponds to "choosing exactly one from each pair $(2i - 1, 2i)$" in PBP. Traversing a tree through the second half of the vertices from the middle vertex corresponds to the same thing. If the edges traversed in the first half of the tree does not correspond to a sum equal to half of the total sum of S, but is, lets say, smaller, the corresponding sum of the traversal of the second part of the by the mirror part of MFMST would be too big. This necessitates that if you make choices that are not corresponding to a valid subset for PBP in the first half of your traversal, you would have to make the exact opposite choices in the traversal of the second half in order for you to have the sum of the edges in the tree and the mirror be equal. If they are not equal, on of them will be to big for our B by exactly the amount that the other will be smaller than B, and you will have to try another tree to satisfy MFMST. Since it is impossible to the sum of a tree and a mirror both be less than the total sum of all elements in S in our construction, we have shown if and only if.

d) (Both)

In this section we will design an algorithm that solves the MFMST optimization problem. First we will design it from a very high level and then we will dive into parts of it that are challenging. The overview of our algorithm is as follows;

1. Find all the spanning trees in the graph.
2. Calculate their sum and the sums of their mirroring, take the highest of those values.
3. Find the spanning tree that has the lowest one and return that tree.

The only challenging part of this algorithm is step 1. about finding all the spanning trees in the graph. The two other steps is simply finding the highest and the lowest value of some numbers. So let us look at how to find the spanning trees.

1. Find all subsets containing exactly $n-1$ edges.
2. Perform DFS-traversal on them all and keep only the ones that visits every vertex.

We know that every spanning tree in a graph G must have exactly $n-1$ edges. We can start with every subset of edges with $n-1$ members as a candidate for a spanning tree. This does not mean that every one of them will be spanning trees, in fact most of them will not be. We make a Depth First Search to see if they are in fact a spanning trees. By traversing through the subgraph and checking if they visit all vertices, we can filter away the ones that are not spanning trees. We can be certain that, if the DFS tree of the subgraph visits every vertex, then the sugraph does not have any cycles and must in fact be that exact same tree. This is because the subgraph has exactly $n-1$ edges and the only way to reach every vertex will be by each edge includes one new vertex. Thus there is not enough edges in the subgraph to produce cycles if there exists a tree that visits all nodes.

Smart additions (that we did implement)

Instead of running through all $\binom{m}{n-1}$ subsets of n-1 edges with DFS, we added a "smart" middle step to trim some of the subsets away. We did this by checking whether or not each subset visited all vertices, and if they didn't, we cut them away. This allowed us to drastically reduce the amount of potential trees that we had to perform DFS on in order to check if they were actually trees. This also reduced the running time of our implementation from too long for CodeJudge (above 300 seconds) to okay for CodeJudge ($\sim$170 seconds) for the third mandatory test case.

Smart additions (that we did not implement)

We have found references[2] to better ways of finding all the spanning trees of a graph. These include ones that have vastly better time complexity than our approach. We have not implemented any of these as we did not have time for it, but finding spanning trees is the biggest bottleneck in our algorithm so improving it would decrease time complexity.

[2] Algorithms for generating all possible spanning trees of a simple undirected connected graph: an extensive review

e) (Johan)

The running time is dominated by the time it take to find all trees. As described in D), this is a two step process, first finding all $\binom{m}{n-1}$ subsets of edges with n-1 edges in them, and thereafter determining if they are a tree or not. This is in the worst case done with DFS for all of them. The runtime of DFS is $O(n+m)$ where m is the number og edges and n the number of vertices. Therefore the total running time of out algorithm is:

$$O(\binom{m}{n} \cdot (n+m))$$

f) (Both)

We have uploaded code to CodeJudge that has passed all 4 tests. The code is also in a py file under the learn assignment.