Danmarks Tekniske Universitet

02249 Computationally Hard Problems

# Course Project

Mirror Friendly Minimum Spanning Tree

Mikkel Riber Bojsen
s093255

Martin Kasban Tange
s093280

Casper Tollund
s093037

November 4th 2013

# Contents

## 1 Problem Description

The problem resembles the Minimum Spanning Tree problem, with the addition of having to calculate the complete weight of the so-called mirror value. The mirror of an edge is the edge found when the middle of the edge list is used as a mirror. The mirror of a spanning tree is found by adding up all the mirrors of the edges in the tree. The MirrorFriendlyMinimumSpanningTree (MSFMST) problem deals with finding a tree where the highest value of the tree and the mirror of the tree is at most some value $B$.

The given problem is a graph with 3 nodes and three edges, so all nodes are connected in a cycle, as seen in Fig. 1. The problem will return true, since the spanning tree with edges $e_1$ and $e_3$ have a complete weight of 4, and the mirror edges, which are also $e_3$ and $e_1$ will have a complete weight of 4 also. If any other spanning tree is chosen, either the spanning trees complete weight or the complete weight of the mirror edges will be 5 or more.

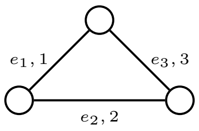

Figure 1: Example graph

## 2 MFMST $\in \mathcal{NP}$

To show that MFMST is in $\mathcal{NP}$ we design the following polynomial time algorithm which takes a string of random values $R$ as input as well as a graph $G$.

### 2.1 Algorithm

- Let the string $R$ consist of index numbers for the edges in $G$: $R = r_1, r_2, \ldots, r_l$.
- If the number of edges in $R$ does not equal $n-1$, where $n$ is the number of vertices in the input graph $G$, then return NO.
- Check whether the edges in $R$ form a tree in $G$. If not, then return NO.
- Let the string $Q = q_1, q_2, \ldots, q_l$ consist of the mirror edges of the edges in $R$, such that if $r_1 = e_k$, then $q_1 = e_{m+1-k}$: .
- Calculate the complete weight, $W_1$, of the spanning tree formed by the edges in $R$.
- Calculate the complete weight, $W_2$, of all the edges in $Q$.
- If $\max\{W_1, W_2\} \leq B$, then return YES, else return NO.

## 2.2 Conditions

Assume the true answer is YES

- There exists a spanning tree in $G$, with a total weight being less than $B$, and the total weight of the mirror edges is also less than B.
- Construct a string of edges $R* = r_1, r_2, \ldots, r_l$ containing all the edges in the spanning tree.
- When the algorithm receives $R*$, it will construct the spanning tree, calculate the weight of it, and calculate the weight of the mirror edges and answer YES.
- Therefore there is a string of length $n-1$ that will return YES. The probability of creating it is positive.

Assume the true answer is NO

- No set of edges can create a spanning tree where both the total weight of the tree and the total weight of the mirror edges will be less than $B$.
- If the length of $R$ is not $n-1$ then we answer NO.
- If the length of $R$ is $n-1$, then the algorithm will check to see whether the edges in $R$ form a tree. If not then it will return NO.
- If the edges form a spanning tree, the algorithm will calculate the complete weight of the spanning tree and the complete weight of the mirror edges.
- Since both weights cannot be less than B, as the true answer is NO, the algorithm will return NO.

## 2.3 Running time

- We can check if there are $n-1$ edges in $R$ in time $O(n)$.
- We can check if the edges form a tree using an algorithm like depth-first-search, in time $O(n+m)$, where $n$ is the number of nodes and $m$ is the number of edges.
- $Q$ can be created by looping over $R$ in time $O(n)$, assuming a suitable data structure to hold edges, such as an array.
- The total weight of $R$ is calculated in $O(n)$.
- The total weight of $Q$ is calculated in $O(n)$.
- The complete running time is therefore $O(n+m)$.

As we have a positive chance to answer YES when the true answer is YES, we always answer NO when the true answer is NO, and the running time of the algorithm is polynomial, we can conclude that MFMST is in $\mathcal{NP}$.

## 3 $\mathcal{NP}$-completeness

To prove $\mathcal{NP}$-completeness, we look at the problem PartitionByPairs (PBP).

### 3.1 Transformation

Given an instance $X$ of PBP, we do the following transformation $T(X)$. We start off by calculating the value $B = \frac{1}{2}\sum_{i=1}^{2n} s_i$. For each pair $(s_{2i-1}, s_{2i})$ in $S$, where $i \in \{1, \ldots, n\}$, we construct a graph as show in Fig. 2, where the labels are the weights of the corresponding edge.

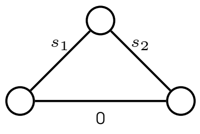

Figure 2: Transformation of a single pair

We order the set of edges, so the mirror of the edge with weight $s_{2i-1}$ is $s_{2i}$ and the mirror of an edge with weight 0 is another edge with weight 0. If there is an odd number of pairs, the middle edge will be a zero-edge that mirrors into itself). For multiple pairs, we chain multiple graphs together as shown in Fig. 3.

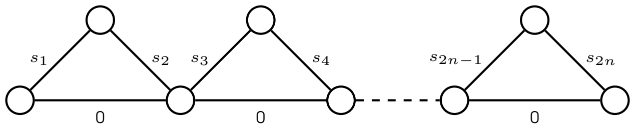

Figure 3: Transformation of several pairs

For example, the set $S = \{1, 2, 3, 4, 5, 6\}$ is transformed into a list of weights $W = [1, 3, 5, 0, 0, 0, 6, 4, 2]$, where the weight of edge $i$ is the $i$'th element in $W$. Using this graph and the calculated value $B$ we can query MFMST to answer PBP.

### 3.2 Proof

We do our transformation in polynomial time. The calculation of $B$ is done in $O(n)$. For each pair, we construct a constant number of nodes and edges, so the graph can be created in $O(n)$, this means our transformation can be done in $O(n)$.

If the answer to the original problem instance $X$ is YES, it means that a partition where we pick one from each pair, equals $B$. It is possible to pick a spanning tree where we pick one from each pair as shown in Fig. 4. As the answer to the original problem was YES, and we can pick a spanning tree that uses an edge from each pair, the answer must also be YES to $T(X)$.

We now assume the answer to the original problem is NO. In this case, we know that it is not possible to pick one edge from each pair in the transformation and get a value that is exactly $B$.

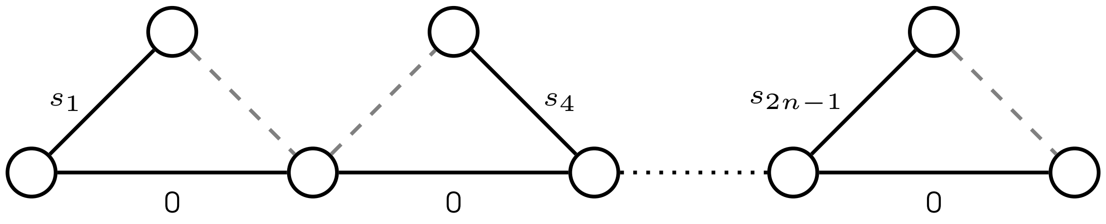

Figure 4: A spanning tree

Any spanning tree must pick at least one from each pair, but can pick two (ignoring the 0-edge) and all edge weights are non-negative.

If we pick more than one edge from a pair, we increase both the weight of the spanning tree and the mirror, as the two edges we pick are mirrors of each other, and therefore will be in both sums.

In summary, this means that if the original answer is NO, we cannot pick a spanning tree using one from each pair, where the maximum value of the tree and the mirror is exactly $B$. As we need to pick at least one from each pair to have a spanning tree, and picking more than one, always increases both values, we cannot get a spanning tree that will make MFMST answer NO.

As our transformation runs in polynomial time, the original problem is $\mathcal{NP}-$*complete*, our transformation preserves the original answer and MFMST is in $\mathcal{NP}$, we can conclude that MFMST is $\mathcal{NP}-$*complete*.

## 4 Optimization Algorithm Description

### 4.1 General algorithm

Our algorithm does an exhaustive search of spanning trees, while storing the tree ($T$) scoring lowest in the equation

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

In the algorithm, we treat the graph as being a weighed, directed multigraph, to allow multiple edges between nodes. The spanning trees are found by doing so-called cuts and contracts. This creates a recursive computation as follows:

If no edge remain, stop the calculation. Otherwise, pick an edge and:

1. Create a copy of the graph, where the end-points of this edge is merged into one point and any edges between them are removed. Store which edge was contracted.
   - Use depth first search from an end-point of the contracted edge through the set of contracted edges, to make sure no cycle was created.

- If a cycle is created, stop the calculation of this path, otherwise, calculate all spanning trees on the copy.

2. Create a copy of the graph where the edge is removed.
   - Do a depth first search from one end-point of the removed edge.
   - If the other end-point is reached, calculate all spanning trees on the copy.
   - If the other end-point it not reached, the graph has been disconnected, stop the calculation of this path.

Every leaf of this computation tree contains a unique spanning tree. It can be found by collecting all stored edges from leaf to root.

## 4.2 Computation tree

An example of such a computation tree can be seen in Fig. 5, where all possible cuts and contracts that can be made are shown for the example graph from Fig. 1. Every node corresponds to a copy of the original graph, where edges have been cut or contracted, according to the path to the root. The lines/nodes are grayed out when a cycle has been created (during a `contract`), or the graph has been split into multiple connected components (during a `cut`). Finally one can see that three spanning trees are found, each one marked with a blue leaf node.

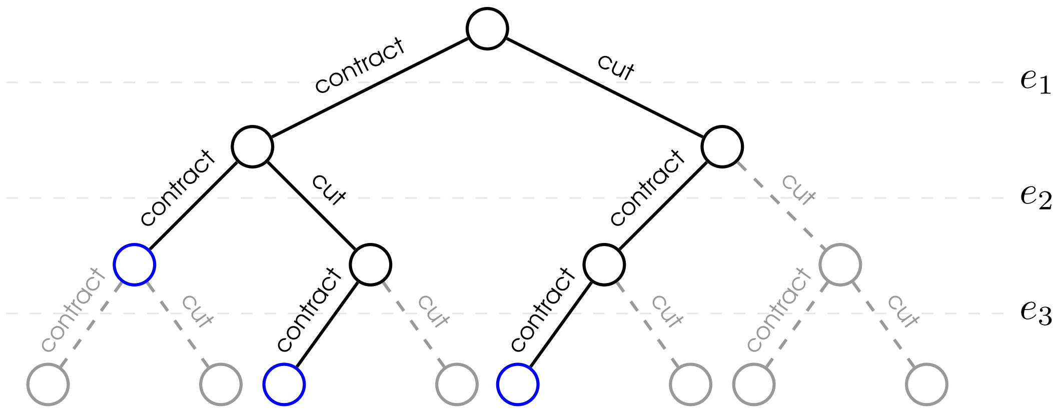

Figure 5: Computation tree for the graph Fig. 1 in order to find all the spanning trees. These have been marked with a blue nodes.

## 4.3 Tricks

Since all edges have positive weights, we can maintain the current "value" of the calculation that has been created, i.e. the edges that have already been picked. If we maintain a best found solution, we can stop finding more trees once we reach this value, thereby hopefully cutting of branches from the computation tree.

To increase the number of branches we can cut off, we can try to guess a good edge to contract at every step, using one of two heuristics:

1. The edge $e_i$ with the lowest score of $\max\{w(e_i), w(e_{m+1-i})\}$ (maximum of edge/mirror)
2. The edge $e_i$ with the lowest score of $w(e_i) + w(e_{m+1-i})$ (sum of edge/mirror)

If we sort all edges based on one of these heuristics, we will automatically try the "best" possibilities first. This can be done using any sorting algorithm, we have chosen QuickSort.

## 5 Correctness

To prove correctness of our algorithm, we need to show that we potentially search all spanning trees, and that the paths in the computation we decide not to follow, cannot contain a tree that is better suited than what we already have.

### 5.1 Finding All Spanning Trees

If we disregard the part where we cut of a path early, we first need to prove that we can search all spanning trees.
Assume a spanning tree $T'$ exists which we did not check.

- If we did not find this tree, there is no leaf in our computation tree that corresponds to $T'$.
- If $T'$ contains $e_1$, then we can move down the left edge of the root of the computation tree (contract). If $e_1$ is not in $T'$ we take the right edge from the root (cut).
- This can be done for all edges, except those that would disconnect the graph if removed, or if using the edge creates a cycle in our current tree.
- If $e_i$ is not in $T'$, but it would disconnect the graph, we do not have a leaf corresponding to $T'$.
  - As $e_i$ would disconnect the graph, $T'$ cannot be a spanning tree.
- If $e_j$ is in $T'$, but using it would create a cycle, we do not have a leaf that corresponds to $T'$.
  - As $e_j$ would create a cycle in the tree being built, $T'$ cannot be a tree, and thus not a spanning tree.

The same argumentation works with any ordering of the edges in the algorithm.

### 5.2 Correctness of Early Search Termination

As we only have positive edge weights, we can always assume that an ongoing search for a tree, only increases in total weight. This means that once we reach our current best weight, it is impossible for any tree generated by the current search further down our computation tree to yield a better tree.

### 5.3 Summary

In summary, our algorithm is correct, as we potentially search all spanning trees of the graph and we only terminate a search if it impossible for the result to be better than what we already have.

## 6 Running time

If the input graph has $n$ nodes and $m$ edges, we have the following running times.

- We can calculate all the heuristic values for edges in $O(m)$
- We can sort the edges by their heuristic value in expected $O(m \cdot \log m)$.
- For every inner node in the computation tree we do the following:
  - Perform a depth-first-search to see if an edge added to the contracted set, creates a loop. This can be done in $O(n + m)$.
  - Perform a depth-first-search to see if removing an edge splits the current graph in two components. This can be done in $O(m + n)$.
- The computation tree is binary, and every leaf in the tree corresponds to a spanning tree. Cayleys formula states that a complete graph has $O(n^{n-2})$ spanning trees.
- The path from a leaf to the root of the tree is at most $m$.
- The computational tree has in the worst case, at most $O(m \cdot n^{n-2})$ nodes.

Summing this up, we get a running time of $O((m + n) \cdot m \cdot n^{n-2})$, as we perform two depth-first-searches for every inner node in the computation tree.

## 7 Implementation Specifics

We have implemented the code in `C` in order to get some decent running times on bigger problems.

## 7.1 Sorting

To try to optimize the way we build our spanning trees, we have chosen to use some heuristics to order the edges in some way that is hopefully beneficial. By doing this, we hope to find a potentially small $B$ faster.

Since we did not want to code, test and make sure our own implemented sorting algorithm (i.e. MergeSort) works, we have chosen to use an implementation of QuickSort that we found online[1]. It has been slightly modified to fit our purposes.

## 7.2 Graph duplication

Instead of copying the graph every time we do a cut or contract, we just keep some variables to figure out how the graph will look in the next computation step. This is done by:

**Cuts** Keeping an integer of the current position in our sorted edge list, which increases whenever we go down a step in the computation tree.

**Contractions** We keep a bit-vector of length $m$, which has a 1 at position $i$ if edge $e_i$ is contracted, and a 0 otherwise. This vector is changed in every step of the computation. The benefit is that we can reuse this single vector it as cuts and contracts happen, since we are at exactly one spot in the tree at any given time. We make sure to reset the value of an edge, when we go back up the computation tree.

## 7.3 DFS edge matrix optimization

When we check if we have split the graph into multiple connected components after a cut, one can perform a DFS to see if all nodes are still reachable from a given start node.

However, to make this faster, we start searching from one of the nodes connected to the cut edge and then just try to find the corresponding other node. This way we will likely skip searching the entire graph. We can also optimize our search a bit by trying to see if the current node we are at in our search, is connected to the end node, since we can then confirm that the cut did not split the graph.

[1] http://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Quicksort#C