Technical University of Denmark

02249 - Computationally Hard Problems

Project Exercise

Mirror-Friendly Minimum Spanning Trees

Author:	*Student number:*
Lukas Villumsen	s144451
Sebastian Forman Nyholm	s144434
Søren Oest Jacobsen	s172968

Technical University of Denmark

November 5, 2017

Contents

Graph element abbreviations

ST	Spanning Tree
MST	Minimum Spanning Tree
MFST	Mirror-Friendly Spanning Tree
MFMST	Mirror-Friendly Minimum Spanning Tree

Mirror-Friendly Minimum Spanning Tree Problem

The mirror-friendly minimum spanning tree (MFMST) problem is about determining whether or not there is a *spanning tree* of a graph, such that neither it or its mirror tree has a weight greater than some natural number. Mathematically, these weight condiitons are expressed as follows:

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

The difference between the standard MST problem and the MFMST problem, is that we take into account also the *mirror* of the spanning tree. The MFMST has more restrictions in that, the MFMST of a graph is subjugated to uphold the weight condition of B for both itself and its mirror. The MFMST is not necessarily the spanning tree with the minimum weight, as its mirror may falter the conditions.

Following is an example showing how to solve the MFMST problem for the graph $G_1 = (V, E, w, B)$ illustrated in figure 1 where: $G_1 = (V, E, w, B)$ with $V = \{1, 2, 3\}$, $E = \{e_1 = \{1, 2\}, e_2 = \{2, 3\}, e_3 = \{1, 3\}$, $w(e_i) = i$ for $i \in \{1, 2, 3\}$ and $B = 4$.

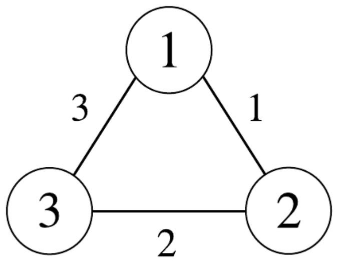


Figure 1: Resulting graph G_1 for the stated problem input

To find an MFMST, three possible spanning trees are shown in figure 2. If we take a look at the left-most spanning tree, the spanning tree itself has a weight of $w = 3$. However, taking the mirror spanning tree results in a weight of $w_m = 5$. Therefore the first spanning tree will not work, since max $(w, w_m) > B$. Taking a look at the middle spanning tree, we will find that it has a weight of $w = 4$. Since the mirror spanning tree in this example gives the same weight, $w_m = 4$, the second spanning tree will solve the MFMST problem to the above input.

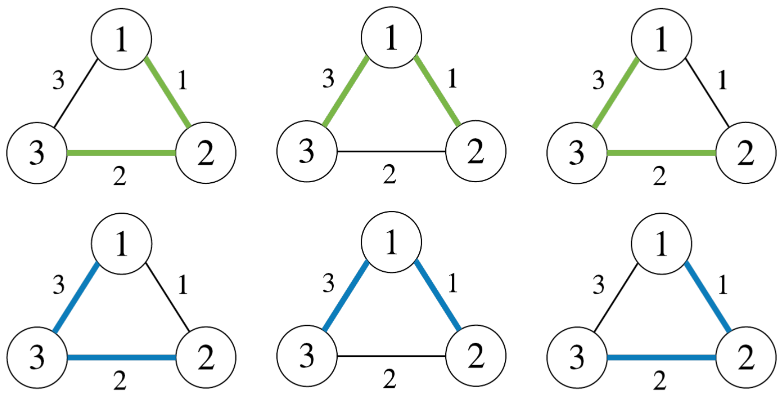


Figure 2: 🟩 Spanning trees of G_1
🟦 Mirrored spanning trees of G_1

Membership of $\mathcal{NP}$ for MFMST

In this chapter we will show that the MFMST problem is in the $\mathcal{NP}$ class of problems.

Input:
The algorithm A receives an input $G(V, E, w, B)$, i.e. a graph containing n vertices, m edges, a weight for each edge and an integer which the total weight of the spanning tree has to be less than or equal to. Further, A takes random string R, consisting of m bits $r_1 \dots r_m$.

Interpretation:
For each bit in R, if the i'th bit is 1, use the edge e_i as a part of the spanning tree, otherwise leave it out.

Verification:
If the weight of the spanning tree and the weight of the mirrored spanning tree are both less than or equal to B, then answer YES, otherwise NO.

Assume the answer is YES for some appropriate input. Then there is a subset of edges, which forms a spanning tree T with a total weight less than or equal to B. Moreover, the mirrored spanning tree's, $\overline{T}$, total weight is also less than or equal to B.

Let $L \subseteq \{1, \dots, m\}$ be the set of the indices of T. Then construct a bit string $R^* = r_1 \dots r_m$ such that $r_i = 1$ **iff** $i \in L$.

Thus the probability of creating a random string, which given as an input to A, resulting in the answer YES is positive - seeing as the string is of length at most m, and its value is drawn uniformly at random from a finite set.

Assuming the answer is NO, either there exists no spanning tree or no spanning-/mirror tree pair exists, such that both have a weight less than or equal to B. A assigns the edges according to the input and its interpretation, then proceeds to evaluate the total weight of the spanning tree and its mirror. Since such spanning tree does not exist, no random string can satisfy the problem bounds, thus A answers NO.

Running Time

The running time of the algorithm is evaluated by terms of the input, consisting of n edges and m vertices:

1. Each edge is either assigned to the spanning tree or not, $O(m)$.
2. Check if edges form a spanning tree, by BFS (to counteract cycles) and vertex checking (to ensure coverage), $O(2n + m)$.
3. Check whether the spanning tree's total weight is at most B, $O(m)$.
4. Check whether the mirrored spanning tree's total weight is at max B, $O(m)$.
5. Return the answer, $O(1)$

$\Rightarrow O(n + m)$ which is bound by the linear polynomial $p(n, m) = cn + km$, for some constants c, k.

$\mathcal{NP}$-Completeness proof of MFMST

In this chapter we will prove that the MFMST problem is in the class of $\mathcal{NP}$-completeness, by reduction from 1-In-3-Satisfiability to MFMST.

$$\text{1-IN-3-SAT} \leq_p \text{MFMST}$$

The proof is done by component design:

- ⋆ Given an instance (X, C) of 1-IN-3-SAT
- ⋆ Construct an instance $(G = (V, E, w), B)$ of MFMST
- ⋆ G is built of many small components
- ⋆ Components ensure consistent assignment and satisfiability of the clauses

Construction of the graph G

Let $X = \{x_1, \ldots, x_n\}$ be boolean variables and $C = \{c_1, \ldots, c_m\}$ be 1-IN-3-SAT clauses over X. Meaning that for each clause $c_i = (c_{i_1} \vee c_{i_2} \vee c_{i_3})$, any c_{i_j} is a literal based on the variables of X.

Setting $B = 3m$ for the MFMST instance.

The set of vertices V is defined for G as

$$V = \{\Theta\} \cup \bigcup_{i=1}^{m} \{c_{i_1}, c_{i_2}, c_{i_3}\}$$

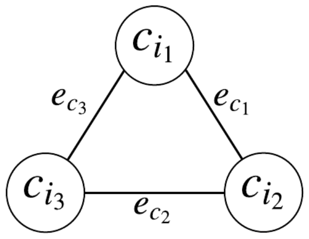
Each edge is defined with a *(S)ource* and a *(T)arget* vertex - although undirected for the MFMST instance.

Component construction

First the components for clause satisfiability are defined:

$$\forall c_i \in C: \text{ Let } E_C \text{ be the set of edges}$$

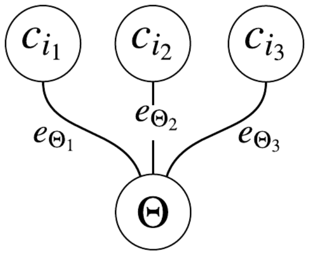
$$E_C = \bigcup_{i=1}^{m} \{ \overbrace{\{c_{i_1}, c_{i_2}\}}^{e_{c_1}}, \overbrace{\{c_{i_2}, c_{i_3}\}}^{e_{c_2}}, \overbrace{\{c_{i_3}, c_{i_1}\}}^{e_{c_3}} \}$$



The idea here is that two edges per triangle are needed to form a spanning tree. The two edges chosen leaves one edge not chosen, corresponding to the *one* **true** literal c_{i_j}, satisfying c_i.

Next, we define the truth setting components:

$$\text{Let } E_\Theta = \bigcup_{i=1}^{m} \{ \overbrace{\{c_{i_1}, \Theta\}}^{e_{\Theta_1}}, \overbrace{\{c_{i_2}, \Theta\}}^{e_{\Theta_2}}, \overbrace{\{c_{i_3}, \Theta\}}^{e_{\Theta_3}} \}$$



Here, the idea is to continue the spanning tree, requiring exactly one edge to do so. This is due to the triangular (connected) clause shapes from clause satisfiability. Hence only one edge chosen will result in a continued spanning tree, determining the truth setting: c_{i_j} chosen indicates $c_{i_j} = \textbf{true}$.

The connecting component here is clearly the Θ vertex, connecting all occurrences of the true literal in every clause to each other. Acting as the junction from which the 'venation' of the spanning tree occurs.

These components make up the edges of the graph:

$$E = E_C \cup E_\Theta$$

Weights of the edges are set to be 1, and are given by the set w:

$$w = \bigcup_{i=1}^{m} \{1, 1, 1, 1, 1, 1\}$$

Since each clause c_i prompts the construction of a total of 6 edges.

Altogether these components are used to build G. A small example can be seen on figure 3.

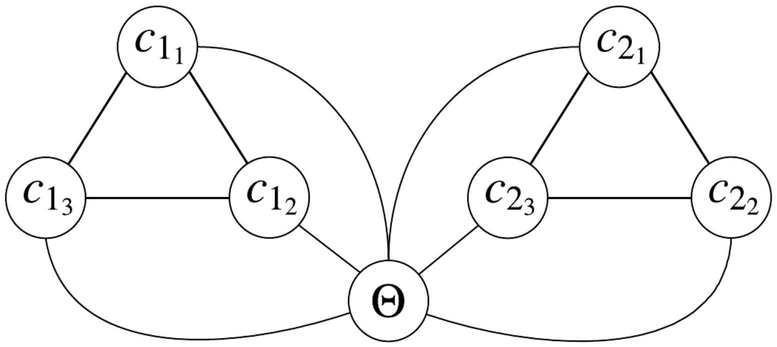


Figure 3: Sketched construction of G over $C = \{c_1, c_2\}$, for which $B = 3 \cdot 2 = 6$

Altogether the construction of G is possible in polynomial time in m.

Correctness of the reduction

To show correctness, we show that $G = (V, E, w)$ has a Mirror Friendly Minimum Spanning Tree of weight at most B, **iff** the 1-IN-3-SAT clauses of C have a satisfying assignment.

This will be done in two steps:

1)	1-IN-3-SAT	$\Longrightarrow$	MFMST
2)	MFMST	$\Longrightarrow$	1-IN-3-SAT

Step 1

Let us assume that $\alpha : X \to \{0, 1\}$ is an assignment satisfying all clauses c_i.

We construct the spanning tree as a set of edges $E' \subseteq E$.
As with E, the construction of E' will also be done in similar parts:

$$E' = E'_C \cup E'_\Theta$$

Starting with edges from the triangles, from which we need 2 edges to reach every vertex:

$$e_{c_j} \in E'_C \iff \alpha(e_{c_j}(S)) = 1 \;\vee\; \alpha(e_{c_j}(T)) = 1$$

That is, either the source $e_{c_j}(S)$ or the target vertex $e_{c_j}(T)$ of an edge is a representation of a clause literal with a truth value of **true**. Since all clauses are satisfied, only one literal is true in each clause. Thus every vertex $c_{i_1}, c_{i_2}, c_{i_3}$ is covered by the minimum number of edges.
By this point we have selected exactly $2m$ edges, the minimum required to cover all vertices $v \in V \setminus \{\Theta\}$.

Next we need one more edge from each triangle to the Θ vertex. By selecting the edge which source vertex is the clause literal representation with the **true** assignment, only one is selected:

$$e_{\Theta_j} \in E'_\Theta \iff \alpha(e_{\Theta_j}(S)) = 1$$

Hence connecting the triangles by means of a minimum number of edges and covering the last vertex of G. Since one edge was selected from each triangle, we selected exactly m edges.

The set $E' = E'_C \cup E'_\Theta$ thus covers all vertices and is a spanning tree $T \subseteq E$ for G.
The total weight of T is:

$$\sum_{e_i \in T} w(e_i) = \sum_{e_i \in E'} 1 = 3m$$

And the mirrored spanning tree $\overline{T}$, also of size $3m$ edges, likewise has a total weight of $3m$. Thus the inequality native to the MFMST problem becomes:

$$\max\{3m, 3m\} = 3m$$

And $B = 3m$. ■

Step 2

To show this implication, we now assume that $T = E' \subseteq E$ is a spanning tree for G such that

$$\max\left\{w\left(T\right), w\left(\overline{T}\right)\right\} \leq B = 3m$$

If the answer to the transformation is YES, it is to be shown that it is so as well for the original problem, 1-IN-3-SAT.

In order to cover the connecting vertex Θ, exactly one edge $\{c_{i_j}, \Theta\}$ from each triangle has to be in E'. Requiring m edges to connect all triangles. This in turn also covers one of three vertices of each triangle. In each triangle is left to cover two vertices, using up the last $2m$ edges of E'.

Now define the assignment $\alpha : X \to \{0, 1\}$

$$\alpha(c_{i_j}) = \begin{cases} 1 & \text{if} \quad \{c_{i_j}, \Theta\} \in E' \\ 0 & \text{if} \quad \{c_{i_j}, \Theta\} \notin E' \end{cases}$$

Since E' contain exactly one of c_{i_j} the assignment is well defined. By this assignment, we now show that all clauses are satisfied.

Let $c_i = c_{i_1} \vee c_{i_2} \vee c_{i_3}$ be some clause.

All 3 vertices have to be covered by edges in E' for T to be a spanning tree, meaning exactly 2 edges of E'_C belong to E' - say $\{c_{i_1}, c_{i_2}\}$ and $\{c_{i_3}, c_{i_1}\}$.

The connecting vertex Θ is then covered by the edge $\{c_{i_1}, \Theta\}$ of E'_Θ and $E' = E'_C \cup E'_\Theta$.

Assignment α assigns **true** to c_{i_1} (being the source vertex of the connecting edge).

By construction of G, c_{i_1} is the first and only **true** literal in c_i. By definition of 1-IN-3-SAT, clause c_i is satisfied.

As the clause c_i was chosen arbitrarily, all clauses satisfied, thus answering YES. ■

Having proven the implication of both direction, and shown a polynomial time transformation, 1-IN-3-SAT $\leq_p$ MFMST, the problem of MFMST is shown to be $\mathcal{NP}$-complete.

Algorithm to solve the optimization problem

In this chapter an algorithm is designed for solving the optimization version of the MFMST problem. To see the actual code involved, refer to the attached zip folder or the GitHub repository.

Designing the optimization problem typically requires a decision problem. The solution to the decision problem tells whether or not there exist a spanning-/mirror tree pair T and $\overline{T}$, neither with a weight greater than or equal to some integer B. However, in this design of the optimization problem, those two algorithms are weaved together, since the optimization problem had to use some of the same operations as the decision problem. Nevertheless, the decision problem has been explained briefly below.

Decision Problem

Input:
The algorithm A_d that solves the decision problem, takes as input a weighted undirected graph $G = (V, E, w)$ and an integer B.

Output:
When executed, A_d will return YES if there exists a T and $\overline{T}$ both with a weight not greater than B. Otherwise it will return NO.

The algorithm:
The algorithm first takes all existing spanning trees from G and puts them into a list T^*. Further the algorithm iterates over T^* as T and checks whether or not the total weight of both T and its mirror $\overline{T}$ are less than or equal to B. If A_d finds such a tree, it will return YES. Otherwise NO.

Although inter-weaved, a rough pseudocode example of the decision algorithm can be found on Algorithm 1.

Algorithm 1 Decision Problem

```
1: procedure $A_d(G(V, E, w), B)$
2:     $T^* \leftarrow$ list of all spanning trees in $G$
3:
4:     for $T$ in $T^*$ do
5:         $\overline{T} \leftarrow$ mirror($T$)
6:         if $w(T) \leq B$ and $w(\overline{T}) \leq B$ then return YES
7:
8:     return NO
```

Optimization Problem

Input:
The optimization problem A_o takes an weighted undirected graph as input parameter, $G = (V, E, w)$.

Output:
The spanning tree T, such that $\max\left\{w\left(T\right), w\left(\overline{T}\right)\right\}$ is the lowest possible value for any spanning tree in G.

The algorithm:
In order to determine the lowest possible value for B, for which there still exists a solution, the algorithm

will temporarily assign $B = \infty$ ($2^{31} - 1$ in practice). The optimization algorithm will first find the minimum spanning tree of G, denoted MST. To this end using Kruskal's algorithm. Although MST may be a mirror-friendly spanning tree, MFST, it is not yet know if it is also the minimum, because of its mirror tree having a weight $w\left(\overline{T}\right) \geq w\left(T\right)$. At this point MST is the only MFST known to the algorithm, so it will assign $B = \max\left\{w\left(MST\right), w\left(\overline{MST}\right)\right\}$.

From here on out, the algorithm will begin creating *partitions*[Sörensen and Janssens(2005)] from MFST's. Since it only knows one MFST, it will start partitioning MST. Partitions of a tree is defined as seen below:

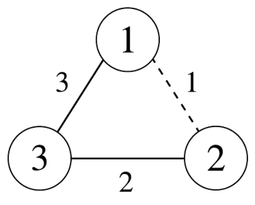
$$\text{Partition}(T) = \left\{\begin{array}{ll} P_1 & = \{\overline{T_{e_1}}\} \\ P_2 & = \{T_{e_1}, \overline{T_{e_2}}\} \\ \vdots & \\ P_s & = \{T_{e_1}, \ldots, T_{e_{n-1}}, \overline{T_{e_n}}\} \end{array}\right\}$$

Where each partition has a reference to edges of the tree to be *included* T_{e_i} and those that should be *excluded* $\overline{T_{e_j}}$. Each partition is then used to generate the MST of G anew. This new minimum spanning tree will indeed be the minimum spanning tree, as we exclude only edges present in the original MST. Another way to think about this is that the partitions each create a new spanning tree that minimizes weight, but the tree cannot contain edges from G that are excluded by the partition, and must use edges from G that are included by the partition.

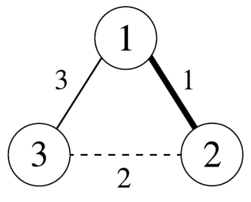
An example of partitioning the MST of G_1 shown in figure 1 is as follows:

$$\text{Partition}(MST(G_1) = \left\{\begin{array}{ll} P_1 & = \{\, \overline{\{1,2\}}\, \} \\ P_2 & = \{\, \{1,2\}, \overline{\{2,3\}}\, \} \end{array}\right\}$$

A visual representation of these partitions can be seen on figure 4. Spanning trees generated from partitions can in turn also be partitioned creating sub-partitions, inheriting already included and excluded edges from the partition which created the tree. To improve performance of the algorithm, we only keep partition trees satisfying the weight conditions for itself and its mirror. The algorithm will partition trees in order of their weight. See more on partition order in section Partitioning. Having found a MFST, a candidate for MFMST, the algorithm will go back to the graph and look for MFST's with a weight strictly less than the candidate's weight. If the trials by partitioning reveal no such tree, the candidate must be the MFMST.



(a) Partition $P_1 = \{\,\overline{\{1,2\}}\,\}$



(b) Partition $P_2 = \{\,\{1,2\}, \overline{\{2,3\}}\,\}$

Figure 4: Partitions for MST$(G_1) = \{\,\{1,2\}, \{2,3\}\,\}$
Dotted lines indicate exclusion and bold lines indicate inclusion

Worst-Case Running Time

Let m be the number of vertices, n the number of edges and N the number of spanning trees of a given graph. The worst case running time of this algorithm is 2^n. This running time is calculated by taking the worst scenario that can happen. The worst case is, if all possible combinations of edges in a graph has to be tested. Since there exist $2^n - 1$ possible ways to construct such a network, 2^n is the upper bound. Hence, $N = 2^n$.

To reduce the needed cycles, one can say that each time the algorithm finds a new lowest B, all combinations of trees so far have been marked. This means, that the algorithm does not have to test those again. Further one can figure out, that if the next minimum spanning tree's own weight is more than B, one do not have to test any further, since all spanning trees from that point onward, will have a weight greater than B. Further one can reduce the number of networks to test on, since a spanning tree must consist of $n - 1$ edges, where n is the number of vertices. However, this does not effect the worst case running time.

Looking at the inner workings of the algorithm, there are a couple of things to note. To generate a minimum spanning tree by Kruskal's algorithm uses $O(n \log n)$ time. Generating minimum spanning trees from a partition obviously also takes $O(n \log n)$ time, as the modifications to allow partition parameters does not mess with the algorithm complexity. The partitions are stored in a `PriorityQueue`, and as the number of partitions is bounded by the number of spanning trees, this gives $O(N)$ time for the number of operations. Each insert/removal operation taking $O(\log N)$ time using `offer()` and `poll()` calls of the queue. Moving on to the main loop of the algorithm, running as long as there are items in the queue. Items in the queue are partition trees based on partitions which of there are at most N, requiring $O(N)$ time to execute. For the optimization loop used to find the minimum of the MFST's, the worst case running time is equal the number MFST candidates. Depending on ordering of the partitions this loop will likely terminate early, but in the worst case each spanning tree is an MFST, requiring $O(N)$ time to complete.
The algorithm thus have time complexity $O(N(N(n \log n + N))) = O(N^2 \cdot n \log n + N^3)$.

Partitioning

Partitions are stored in a priority queue, comparing weights of the partition trees as they are polled. Initially our idea for the algorithm was to partition by order of *lightest* partition tree first. This worked quite well for smaller graphs, where spanning trees have a higher probability to be the same as its mirror, or close to it. Thus resulting in both trees having a small weight, and more likely to be the MFMST of the graph. However through experiments with larger graphs; MFMST's were shown to have significantly higher weights compared to the MST of the graph. Usually the MFMST's would be heavy, but with a mirror weight very close to it. Whereas spanning trees with weight a lot closer to MST would have very heavy mirrors. In cases where the MFMST of a graph is a heavy spanning tree compared to the MST, many partitions and partition trees must be evaluated before finding the MFMST. Thus having a less optimal performance in these cases. The approach to partition ordering was then changed, polling the heaviest of partition trees first. This change sparked noticeable overall improvements to the running time of the algorithm. As an experiment, we tried picking partition trees from the middle, using the median value of priority queue (sorted). This 'order', however, reverted the performance improvement that was gained by switching from light to heavy ordering. Another experiment that was carried out was randomized partition tree selection. Random selection gained an edge in performance compared to lightest first ordering, but fell short compared to heaviest first ordering.

Using custom test files, offering more variety, some examples of the run time performance can be seen in table 1.

Ordering	Light	Heavy	Median	Random
TestFile01.uwg	83	53	94	74
TestFile02.uwg	8	25	32	8
TestFile03.uwg	53	62	80	66
TestFile04.uwg	304	153	572	178
TestFile05.uwg	1562	373	1893	578
TestFile06.uwg	40	47	39	21
TestFile07.uwg	1536	463	1528	991
TestFile08.uwg	2	1	2	1
TestFile09.uwg	23	70	48	31
TestFile10.uwg	11	14	16	15

Table 1: Example running time(ms) of finding MFMST by partition tree ordering

Larger graphs

With the natural complexity of the MFMST problem, larger graphs quickly become expensive to solve. Larger graphs are regarded as having 50+ vertices with a degree greater than 2 on average. Specifically talking here about the `test04.uwg` file, which is a graph consisting of 100 vertices and 554 edges. To put the amount of work needed to do and evaluate involved using the described algorithm, the number of spanning trees can be expressed by *Kirchhoff's matrix tree theorem*:

$$\#ST = (-1)^{i+j} \cdot \det\left(\begin{bmatrix} \deg(v_1) & \dots & -1|0 \\ \vdots & \ddots & \vdots \\ -1|0 & \dots & \deg(v_m) \end{bmatrix}\right)$$

That is, the degree matrix - adjacency matrix of a graph (Laplacian matrix), then computing the determinant of any co-factor of it.
As there may be exponentially many spanning trees in a graph, the running time of the algorithm does not run in polynomial time. It was actually not possible to deterministically solve the MFMST problem for `test04.uwg`. Many attempts, with all different kinds of partitioning order, were carried out. The initial implementation of the algorithm featured recursive (not optimized for tail-recursion) reduction of B. Letting the algorithm have a go at the test, it ran out of memory before finishing. The reduction method was then implemented iteratively, which is better optimized for Java. Still, the algorithm did not finish before terminating prematurely. The best candidate for MFMST of `test04.uwg` at the time of termination had a weight of $B = 15252$. No correct answer is listed for this graph, so it is hard to tell how close the algorithm got. But for it to run out of memory trying to find an MFST with a smaller weight, the algorithm would have tried a bunch of partition trees without finding a better candidate. The partition ordering used to generate the candidate of the lowest weight was randomized partitioning, as to not be gated by sheer amount of partitions.

References

[Sörensen and Janssens(2005)] Kenneth Sörensen and Gerrit K. Janssens. An algorithm to generate all spanning trees of a graph in order of increasing cost. *Pesquisa Operacional*, 25(2):219–229, 2005.