Danmarks Tekniske Universitet

Computationally Hard Problems

Assignment Project

Jonathan Højlev
s194684

Hans Henrik Hermansen
s194042

4. november 2024

Division of labor

The following is our division of labor. Throughout the project, we have developed and discussed our ideas in collaboration, and no part of this project is solely a single person's responsibility. That being said, the following is the closest to a division of labor.

Report:
Hans: c,d,e,f
Jonathan: a,b,g,h

For the algorithm in part f), we both contributed equally to designing it and wrote most of the code through pair programming. That being said, Hans wrote most of the code.

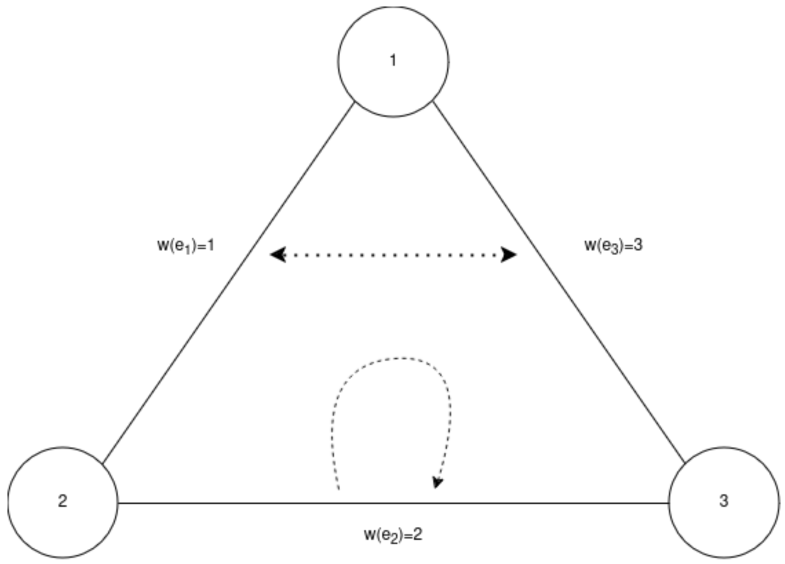
a) Understanding the problem

The problem stated in the project is very similar to the problem of finding a minimum spanning tree (MST) in a graph, with an important difference. Here, each edge is paired with another edge in the graph, possibly itself. If the edges are $E = \{e_1, e_2, \ldots, e_m\}$, then edge e_i is paired with edge e_{m+1-i} and these edges are called each others' 'mirror'.

In colloquial terms, the problem is to determine if there exists a spanning tree T in the graph G with a mirror M such that the weight of T and M are both below the threshold B. The weight of a spanning tree is defined as the sum of the weights of its edges, and its mirror M is defined as the set of edges that are the mirror edges of the edges in T. The weight of a set of edges is denoted $|\cdot|$.

Importantly, while there is a restriction that the edges in T, namely that they form a spanning tree of G, there is no restriction on the edges in M. These can belong to T or some other part of G and generally be thought of as a random subset of $(n-1)$ edges.

This problem is computationally hard to solve, as this report will show. As an example, the solution to the first example problem is shown below.



Figur 1: First example, where the dotted lines illustrate an edge's mirror edge.

For $B = 4$, the answer is "YES". This is because the edges $T = \{e_1, e_3\}$ with mirror $M = \{e_3, e_1\}$ have the weights $|T| = |M| = 4 \leq B$.

b) The word problem

In order to describe the formal language that is used in the `.uwg` file format and to solve the word problem, the alphabet and the language must first be defined.

The alphabet is $\Sigma_{MFMST} = \{0, 1, \ldots, 9, SPACE, LF\}$, where "SPACE" is the blank symbol and "LF" is the line-feed symbol.

The language $L_{MFMST,B}$ consists of all inputs whose graph has a mirror-friendly minimum spanning tree T with a mirror M with $max\{|T|, |M|\} \leq B$. In other words, the language is the set of all inputs for which the output is "YES".

In order to solve the word problem, the problem will be broken into steps. Firstly, the input word w will be read and parsed as a graph, as described in the problem statement. Then some algorithm can be used to give a heuristic of which edges must be or must not be in the optimal solution. Then, check all spanning trees until one with $max\{|T|, |M|\} \leq B$ is satisfied. Then output "YES". If no solution exists, output "NO".

c) From decision variant to optimization variant

We now assume that we have an algorithm for the decision problem where a call takes one computational step. From this we will construct an polynomial time algorithm for the optimization problem. The pseudo code is shown in algorithm 1.

Correctness

The first part of the algorithm simply binary searches for the correct B. After this step B has the smallest possible value for our given G and we now need to find the solution. To find the solution we use the same procedure as presented in the course for finding the solution for the MaximumClique problem.
We go through each edge and try to remove it. If A_d still answers YES it means that there still exists a solution and we can remove this edge. If A_d answers NO we keep the current edge. We will now argue that this always finds a correct solution. When $|E| = n-1$ we return our solution E.

One solution

We will first look at the case where there only exists one solution in G. A solution to this problem is spanning tree which is a set of edges ST and in this case there only exists one such set. We now look at an edge e. If $e \in ST$ and we remove it there no longer exists a solution in G and A_d answers NO and we keep e. If $e \notin ST$ we still have $ST \subseteq E$ and the solution still exists in G. A_d will therefore answer YES and we can remove e and still have a solution. As G is finite we at some point will have removed all edges not in ST. This final G is our solution as it is a spanning tree with edge set ST.

Multiple solutions

We now assume there are multiple solutions $ST_1 \ldots ST_k$ present in G. We now look at an edge e. If e is an all ST_i and we remove it then G no longer has any solutions and A_d answers NO so we keep e. Thus we never remove all solutions from G. If e is present in at least ST_j but not

Algorithm 1 Find the minimum mirror friendly spanning tree

```
$B_{upper} \leftarrow \sum_{i=1}^{m} w(e_i)$
$B_{lower} \leftarrow 0$
if $A_d(G, B_{upper}) = \text{NO}$ then
    return NO
end if
$B \leftarrow \left\lfloor \frac{B_{upper}+B_{lower}}{2} \right\rfloor$
while true do
    if $A_d(G, B) = \text{YES}$ then
        $B_{upper} \leftarrow B$
    else
        $B_{lower} \leftarrow B + 1$
    end if
    $B \leftarrow \left\lfloor \frac{B_{upper}+B_{lower}}{2} \right\rfloor$
    if $B_{lower} = B_{upper}$ then
        break
    end if
end while
for $i$ in $1 \ldots m$ do
    if $A_d((V, E \backslash \{e_i\}), B) = YES$ then
        $E \leftarrow E \backslash \{e_i\}$
    end if
    if $|E| = n - 1$ then
        return $E$
    end if
end for
```

present in at least ST_i then removing it does not destroy all solutions. Therefore G will still have at least the solution ST_i. Then A_d would answer YES and we would have destroyed at least ST_j and thus decreased the number of solutions. If e is present in no solution then we of course can just remove like the case with one solution. Since each all the solutions are different then each ST_i has at least one edge not present in all the other solutions. Therefore while there are more than one solution present in G there will always be an edge which does not destroy all solutions which we can remove. Doing this iteratively we at some point only have one solutions present in G and we know from before that in this case we will find the correct solution.

Polynomial running time

We will now argue that the algorithm presented above runs in polynomial time.

Let $W = \sum_{i=1}^{m} w(e_i)$. First we binary search for the correct B in the interval $[0, W]$. This takes $O(\log(W))$ time. However we know that $W \leq 2^{\|\mathbf{X}\|}$. So we get that this takes $O(\log(W)) = O(\log(2^{\|\mathbf{X}\|})) = O(\|\mathbf{X}\|)$, so this step takes linear time. For each edge we some constant time work and make a call to A_d, which we assume takes constant time. As a result this step takes $O(m)$ time. Since each edge is specified in the input we know $m \leq \|\mathbf{X}\|$. So we finally get a running time of $O(\|\mathbf{X}\| + m) = O(\|\mathbf{X}\|)$ which means the algorithm A_o runs in polynomial time.

d) MFMST in in $\mathcal{NP}$

To prove that MFMST is in $\mathcal{NP}$ we will design an algorithm which takes the problem instance $\mathbf{X}$ and a random sequence R and solves the decision problem. This random sequence R consists of integers in the range $[1, m]$. Our algorithm is as follows

1. Consider $R = r_1, r_2 \ldots r_k$. If $k < n-1$ return NO. Otherwise check if the set $ST = \{e_{r_1}, e_{r_2} \ldots e_{r_{n-1}}\}$ is a spanning tree. If it is not return NO.
2. Now check if $\sum_{i=1}^{n-1} w(e_{r_i}) \leq B \wedge \sum_{i=1}^{n-1} w(e_{m+1-r_i}) \leq B$. If this is true return YES otherwise return NO.

This clearly runs in polynomial time. We check if something is a spanning tree which can be done in linear time with for example BFS and then we compute two sums. If the answer to the MFMST problem is NO then there exists no set ST that is both a spanning tree and satisfies $\sum_{i=1}^{n-1} w_{r_i} \leq B \wedge \sum_{i=1}^{n-1} w_{m+1-r_i} \leq B$. Therefore our algorithm will also answer NO either in step 1 or step 2. If the answer to the MFMST problem is YES then there must exists at least on set ST which is a solution to the problem. Since we uniformly at random draw a set of numbers in range $[1, m]$ of at most size $n-1$, we draw from a finite set of possibilities. Therefore the probability of drawing ST from a finite set of possibilities is positive.
So we have a deterministic algorithm which takes $\mathbf{X}$ and a random sequence R as inputs which when the true answer is NO answers NO for all R. When the true answer is YES there exists a polynomial size R where the algorithm will answer YES. Therefore MFMST is in $\mathcal{NP}$.

e) MFMST is $\mathcal{NP}$-complete

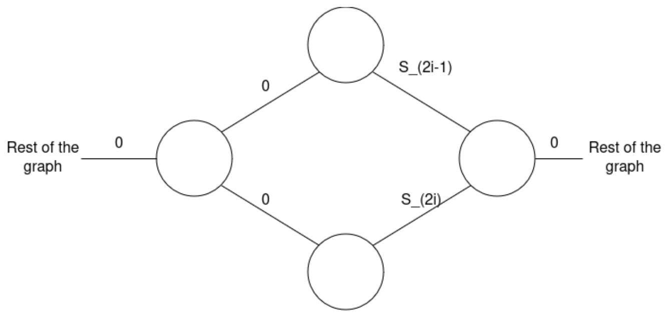
To prove that MFMST is $\mathcal{NP}-$complete we will use the problem PartitionByPairs which we will shorten to PBP and then show

$$PBP \leq_p MFMST.$$

First we will show how we transform an input to PBP to an input to MFMST in polynomial time and then argue that it is a proper transformation.

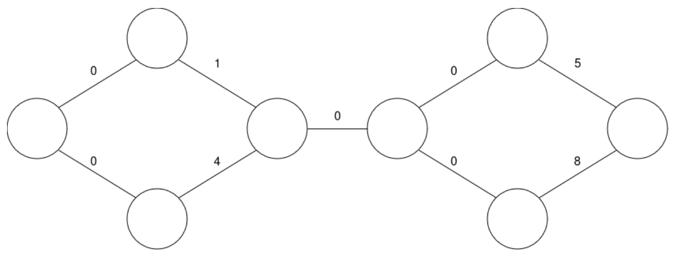
Transformation: PBP $\rightarrow$ MFMST

For an input sequence of numbers to PBP $(s_1, s_2, \ldots, s_{2n})$ we will look at the pair (s_{2i-1}, s_{2i}). This we will model in our graph as seen below in figure 2.



Figur 2: How (s_{2i-1}, s_{2i}) is modeled in our graph input

The module or subgraph shown in figure 2 is repeated for each pair (s_{2i-1}, s_{2i}) and these subgraphs are connected together. We will show how this is done for the example input to PBP $(1, 4, 5, 8)$. This is seen in figure 3.



Figur 3: The MFMST transformation of the PBP input $(1, 4, 5, 8)$

In the following we will identify the edge corresponding to the number s_i simply as the edge es_i and we will refer to the set containing these edges ES.
So to transform a PBP problem to MFMST we create a weighted graph as presented above and then we set $S = \sum_{i=1}^{2n} s_i$ and finally $B = \frac{S}{2}$. We then pair the edges such that es_{2i-1} is paired with es_{2i} and the edges with weight 0 are paired arbitrarily. This we do by constructing the edge sequence $E = \Big(es_1, es_3, \ldots, es_{2n-1}, \text{all the other 0 weight edges}, es_{2n}, es_{2n-2}, \ldots, es_2\Big)$. Since the size this graph is linear in the size of the input to PBP, it is clear that this transformation can be performed in polynomial time.

We never include the both the edge es_{2i-1} and es_{2i} in our spanning tree

We will first argue that any solution to our MFMST transformation will not select both the edge es_{2i-1} and es_{2i}. First we can see from figure 2 that any solution must include at least s_{2i-1} or es_{2i} in the spanning tree. So the spanning tree contains at least n edges corresponding to our input numbers.
Now assume that for some $j \in [1..n]$ that both es_{2j-1} and es_{2j} are in our solution ST. We know that $\sum_{e \in ST} w(e) = \sum_{e \in ST \cap ES} w(e) \leq \frac{S}{2}$, but since every es_i not in ST is in the mirror of ST we also know that $\sum_{e \in ES \setminus ST} w(e) \leq \frac{S}{2}$. Clearly the sets $ST \cap ES$ and $ES \setminus ST$ form a partition of ES. So the sum of all the weights of their edges is exactly S. Therefore the only way both $\sum_{e \in ST \cap ES} w(e) \leq \frac{S}{2}$ and $\sum_{e \in ES \setminus ST} w(e) \leq \frac{S}{2}$ can be true is if $\sum_{e \in ST \cap ES} w(e) = \frac{S}{2}$ and $\sum_{e \in ES \setminus ST} w(e) = \frac{S}{2}$ hold. This however leads to a problem. Since both es_{2j-1} and es_{2j} are in ST they are also both in the mirror of ST. Therefore the sum of weights in the mirror becomes $(\sum_{e \in ES \setminus ST} w(e)) + s_{2j-1} + s_{2j} = \frac{S}{2} + s_{2j-1} + s_{2j}$. As every s_i is a natural number we get that $\frac{S}{2} + s_{2j-1} + s_{2j} > \frac{S}{2}$. This means that our solution ST is not a valid solution and our assumption that both s_{2j-1} and s_{2j} were in ST was false. So we conclude that for all $i \in [1..]$ any solution ST contains either es_{2i-1} or es_{2i}.

PBP reduces to MFMST

Now that we have our transformation from PBP to MFMST and we have proved some additional things we can show that our transformation indeed is valid and therefore PBP $\leq_p$ MFMST in the following.

The answer to PBP is YES

In this case there exists a set A that partitions the numbers $(s_1, s_2, \ldots, s_{2n})$ such that $\sum_{i\in A} s_i = \sum_{i\in\{1..2n\}\setminus A} s_i$. From this we easily get

$$\begin{aligned}\sum_{i\in A} s_i + \sum_{i\in\{1..2n\}\setminus A} s_i = \sum_{i=1}^{2n} s_i \Leftrightarrow \\ 2\cdot\left(\sum_{i\in A} s_i\right) = \sum_{i=1}^{2n} s_i \Leftrightarrow \\ \sum_{i\in A} s_i = \sum_{i\in\{1..2n\}\setminus A} s_i = \frac{\sum_{i=1}^{2n} s_i}{2}\end{aligned}$$

A solution to MFMST transformation is simply the spanning tree ST where we choose the edges es_i such that $i \in A$. As A contains exactly one i from each pair $(2j-1, 2j)$ for $j \in [1..n]$ we know that our tree includes exactly one of the edges es_{2j-1} and es_{2j} for $j \in [1..2n]$. By adding all the edges in the graph with weight 0 this is ensured to be a spanning tree. We can now calculate the weight of ST and its mirror.

$$\begin{aligned}\sum_{e_i\in ST} w(e_i) = \sum_{e_i\in ST\cap ES} w(e_i) = \sum_{i\in A} s_i = \frac{S}{2} \\ \sum_{e_i\in ST} w(e_{m+1-i}) = \sum_{e_i\in ES\setminus ST} w(e_{m+1-i}) = \sum_{i\in\{1..2n\}\setminus A} s_i = \frac{S}{2}\end{aligned}$$

As we can see from above both the weight of ST and its mirror are less than or equal, in this case equal, to $\frac{S}{2}$ which was our chosen B. Therefore the answer to MFMST is also YES.

The answer to MFMST is YES

If the answer to MFMST is YES we have a spanning tree ST such that the following holds.

$$\sum_{e_i\in ST} w(e_i) \le \frac{S}{2} \wedge \sum_{e_i\in ST} w(e_{m+1-i}) \le \frac{S}{2}$$

However as we have argued earlier ST essentially partitions the edges in ES. The set ST contains exactly n edges from ES with exactly one from each pair (es_{2j-1}, es_{2j}) for $j \in [1..n]$. So we can simplify the above inequalities as follows.

$$\begin{aligned}\sum_{e_i\in ST} w(e_i) = \sum_{e_i\in ST\cap ES} w(e_i) = \sum_{es_i\in ST\cap ES} w(es_i) = \sum_{es_i\in ST\cap ES} s_i \le \frac{S}{2}\wedge \\ \sum_{e_i\in ST} w(e_{m+1-i}) = \sum_{es_i\in ES\setminus ST} w(es_i) = \sum_{es_i\in ES\setminus ST} s_i \le \frac{S}{2}\end{aligned}$$

We also know that since the sets $ST \cap ES$ and $ES\setminus ST$ partition ES the only way that the sum of their weights both are less or equal to half of the total sum of weights in ES is if each sum is actually equal to exactly half of of the total sum. This means that $\sum_{es_i\in ES\cap ST} s_i = \sum_{es_i\in ES\setminus ST} s_i = \frac{S}{2}$. To find a valid partition in PBP we choose $A = \{i \mid es_i \in ST\}$. We now get

$$\left(\sum_{i\in A} s_i = \sum_{es_i\in ST\cap ES} s_i = \frac{S}{2}\right) \wedge \left(\sum_{i\in\{1..2n\}\setminus A} s_i = \sum_{es_i\in ES\setminus ST} s_i = \frac{S}{2}\right)$$
$$\Longrightarrow$$
$$\sum_{i\in A} s_i = \sum_{i\in\{1..2n\}\setminus A} s_i$$

Since we know that ST contains exactly one edge in the pair (es_{2j-1}, es_{2j}) for $j \in [1..n]$ we know that A contains exactly one of the numbers in the pair $(2j-1, 2j)$ for $j \in [1..n]$. This combined with result expressed above we can conclude that the answer to PBP is also yes.

PBP $\leq_p$ MFMST

We have now shown that there exists a polynomial time transformation from PBP to MFMST, such if the answer to PBP is YES so is the answer to the MFMST transformation and if the answer to the MFMST transformation is YES so is the answer to PBP. We can therefore conclude that PBP $\leq_p$ MFMST. Since PBP is $\mathcal{NP}$-complete this means that MFMST is also $\mathcal{NP}$-complete.

f) The algorithm

In this description of our algorithm we will not go into the details of our implementation as this can be seen in our code. We also won't cover basic details like reading input or specific data structures. For a spanning tree ST we will denote the sum $\sum_{e_i\in ST} w(e_i)$ as $W(ST)$ and the sum $\sum_{e_i\in ST} w(e_{m+1-i})$ as $WM(ST)$.

1. Attempt to find a minimum spanning tree MST of G. If this is not possible then return NO as G has no spanning tree. Otherwise check if $WM(ST) \leq W(ST)$. If this is true then return $(W(ST), ST)$ as this is optimal.
2. Find a spanning tree $MMST$ where $WM(MMST)$ is minimum. If $W(MMST) \leq WM(MMST)$ return $(WM(MMST), MMST)$ as this is optimal.
3. Set $ST = \emptyset$. Now go through each edge e in G. Remove e from G. If this disconnects G then add e to ST. Then add e back to G.
4. Set $E = E \setminus ST$
5. Now return BruteForce(ST, G)

The subroutine BruteForce is described below

Algorithm 2 BruteForce(ST, G)

```
if $|ST| = n - 1$ then
    return $(\max(W(ST), WM(ST)), ST)$
end if
if $n - 1 - |ST| < |E|$ then
    return $(\infty, \emptyset)$
end if
Choose an edge $e$ in $E$
Set $E = E \backslash \{e\}$
$(B_{\backslash e}, ST_{\backslash e}) \leftarrow$ BruteForce($ST, G$)
if $ST \cup \{e\}$ contains no cycles then
    $ST = ST \cup \{e\}$
    $(B_e, ST_e) \leftarrow$ BruteForce($ST, G$)
    if $B_e \geq B_{\backslash e}$ then
        return $(B_e, ST_e)$
    end if
end if
return $(B_{\backslash e}, ST_{\backslash e})$
```

The first part of our algorithm first deals with the two edge cases where an optimal solution is simple by finding two spanning trees which are minimum with respect to the weight of tree and the weight mirror respectively. This also checks if a spanning tree exists. We then find all bridges in the graph. Every such edge is part of every spanning tree in G and so we immediately add them to ST. Finally the BruteForce subroutine finds the B value for all possible spanning trees and returns the smallest. This is done by choosing an edge $e \in E$ and then recursively calculating all spanning trees not containing e and then all spanning tree containing e.

g) Analysis of runtime

The analysis of the runtime is done in three steps, the check, the filtering step and the brute-force step.

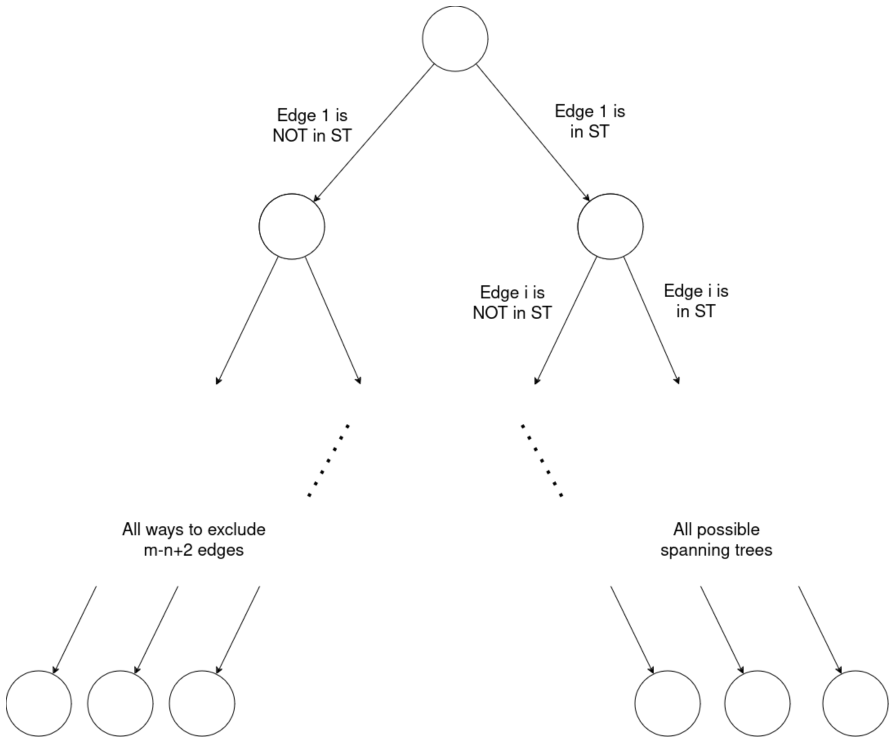
The check is parts 1. and 2. in the algorithm. The edges are sorted in $O(m \log m)$ time and then a minimum spanning tree is found using Kruskal's algorithm. This is repeated where the edges are sorted by their mirror's weight. In total, this takes $O(m \log m)$ time.

The filtering step is polynomial in runtime. Here, for each edge in the graph, the edge is removed and a connectivity check is made, after which the edge is put back. The connectivity check is done in $O(\log^2(n))$ time. In total, this step takes $O(m \log^2(n))$ time.

The brute force step is the part taking non-polynomial time. Here, the recursive function `brute_force` is used. To analyze the running time of this algorithm, consider the tree of recursive calls to the function, see figure 4.

In each call of the `brute_force` function, if it is not in a base case (i.e. not a leaf in the tree) it will always make a call to itself without edge e_i and if edge e_i does not create a cycle so far, it will also make a call including edge e_i. It takes $O(\log^2(n))$ time to check for a cycle using a dynamic connectivity data structure. If the number of internal nodes in the recursion tree is I, this contributes $O(I \cdot \log^2(n))$ to the running time.

In the base case, either ST consists of $n - 1$ edges and there is a valid spanning tree, or there have been excluded $m - n + 2$ edges and no spanning tree is possible. In the first case, the value



Figur 4: Recursion tree of the brute force function.

of the spanning tree is computed and the tree is saved, taking $O(n)$ time. In the second case, the empty set is returned, taking $O(1)$ time. Worst case, this takes $O(n)$ time for each leaf node in the tree.

In the recursion tree, a call reaches a leaf node after having either selected $n-1$ edges or excluded $m-n+2$ edges. Imagine starting at the top of the tree and going down a path to a leaf node. When including an edge e_i in ST at depth i, go right and when excluding the edge, go left. The path stops after going right $n-1$ times or left $m-n+2$ times. Combinatorics gives that the number of unique paths is then upper bounded by

$$\binom{n-1+m-n+2}{n-1} = \binom{m+1}{n-1}.$$

This is the upper bound for the number of leaf nodes in the tree. And since each internal node in the tree at most has two children, this is also an upper bound for the number of internal nodes.

Our analysis then gives the total running time of the brute force function as

$$O\left(\binom{m+1}{n-1} \cdot \left(\log^2(n) + n\right)\right) = O\left(\binom{m+1}{n-1} \cdot n\right).$$

And since the other two steps were polynomial in running time, this is also the complexity the algorithm as a whole.

h) Implementation results

The algorithm we have described and analysis up to this point has been implemented in the programming language Rust and run on test cases 1 through 5. One advantage of the Rust implementation was the `DynamicGraph` and `DynamicConnectivity` modules, giving an easy and very fast way to check handle dynamic connectivity in $O(\log^2(n))$ update time.

For the five first test cases, the optimal B values found are shown in table 1. For test case 6, it took more than 10 minutes to run, so it was aborted.

Test case	Optimal B value
test01.uwg	4
test02.uwg	16
test03.uwg	1128
test04.uwg	No spanning tree exists
test05.uwg	1113

Tabel 1: Optimal B values for test cases 1-5.

The corresponding spanning trees for these results can be obtained by running our code.

Finally, we have had many ideas for further implementation and improvements to the algorithm, in order to achieve faster results. And although these were unfortunately not a priority for us to have done in time, we are still satisfied with our final product.