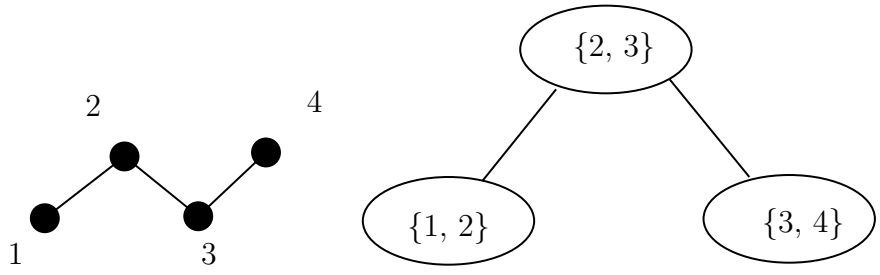
Steiner Tree - DP Over Treewidth

Aditya Sheth - IMT201800x
Rathin Bhargava - IMT2017522

May 14, 2021

Contents



(a) $\mathcal{P}_4$ (b) Tree Decomposition

1 Introduction

1.1 Intro

The Steiner Tree Problem in an NP-Complete Problem. It's a generalization of two polynomially solvable problem - shortest path and the minimum spanning tree. The problem is defined as follows. Given a weighted undirected graph G, and $T \subseteq V$, we would like to find a sub-graph of G which is a tree and spans all vertices in T.

It reduces to the shortest path problem on graphs if $|T| = 2$ and to the minimum spanning tree if $|T| = |V|$.

This problem is a PACE 2018 Challenge problem.

1.2 Our work

We implemented algorithms of both tracks involving exact algorithms - Track 1 and Track 2. Track 1 had test cases with a low number of terminals. Track 2 on the other hand had test cases with arbitrarily high number of terminals (3k) but graphs with low treewidth.

The Track 1 algorithm was an $\mathcal{O} * (3^k)$ algorithm, where k is the number of terminals. Our Track 2 algorithm is a DP over Treewidth algorithm which takes $\mathcal{O}^*(k^{O(k)})$ time.

2 Nice Tree Decomposition

In this section, we'll go over how we converted a tree decomposition into a nice tree decomposition.

The idea that we keep the nodes which are in the tree decomposition as is, and insert new nodes in between the older nodes in order to "smoothen" the change from one node to another.

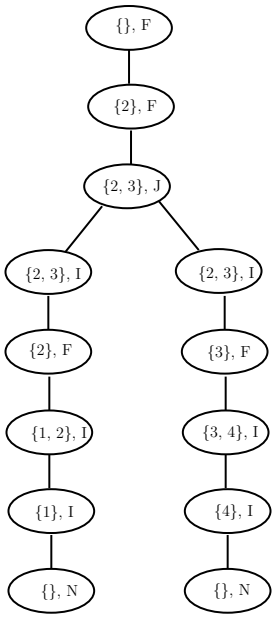


Figure 2: Nice Tree Decomposition of tree in Fig. 1b

2.1 Join Nodes

If a node has multiple children (c), we have to create join nodes. An easy way to do so is to duplicate the current node, add those two duplicates as children to the current node and then assign the first child to the left duplicated node. Now, we have $c-1$ children left, and we recurse on the right duplicated child. The current node is a join node.

This procedure increases the height of the tree by c if the original node had c children. This is because we keep adding vertices only to the right. However, it doesn't matter as the algorithm does not depend on the depth. However if we did, we could minimise the height by making the binary tree of joined nodes balanced.

We can refer to the example in Fig. 1b where the node $\{2, 3\}$ has been duplicated as shown in Fig. 2. To make the tree nice w.r.t the duplicated node and its child, we'll go to the next subsection .

2.2 Introduce and Forget Nodes

At this point, any node which has child with a different bag of vertices has only one child. By making the transition for that nice, we end up making the entire tree nice.

Given nodes X and Y, where Y is the only child of X, we want to insert nodes between X and Y such that there is a nice path from Y to X as we go up the tree.

To compute this, while going from $Y \to X$, we should add forget nodes for all vertices in $Y \backslash X$ and then add introduce nodes for all vertices in $X \backslash Y$. Note that this order is im-

portant. If we reverse the order, we can potentially increase the treewidth.

This has been done in Fig. 2 where we go from $\{3, 4\} \rightarrow \{2, 3\}$ in this manner.

2.3 Introduce Edge Nodes

After applying the above steps, we get a nice tree decomposition which adds and forgets vertices one by one. If we would like to add edges one by one as well, Given an edge containing two vertices u and v, let $f(u)$ and $f(v)$ denote the nodes where u and v are forgotten. Let T_u and T_v denote the subtrees of the tree decomposition rooted at $f(u)$ and $f(v)$ respectively. Now, it's easy to see that either $T_u \subset T_v$ or $T_v \subset T_u$ where $\subset$ denotes the "subtree" property. If it isn't, then there is no node in common between T_u and T_v and that contradicts the property that there exists a bag containing u and v if there is an edge (u, v) in the graph.

In such a scenario, without loss of generality, assume $T_v \subset T_u$. Then, we add an Introduce Edge node just before v is forgotten, i.e. as a child of $f(v)$. This ensures that the graph isn't joined and the edge is only introduced once.

3 Algorithm Overview

In this section, we describe the DP over Treewidth algorithm in detail along with any implementation details or edge cases we had to explicitly take care of.

As defined by the nice tree decomposition, we have 4 types of nodes as described above in Section 2. For each node, we've described the recurrence below.

We add a terminal u^* to all bags, which leads to an increase of the bag size by 1. This is done so that any solution for node t we construct always has the invariant that it is covering the terminals for the sub graph corresponding to the subtree rooted at X_t. Otherwise, till the time we introduce our first terminal, we would've seen a portion of the graph already. Inserting a terminal node would mean that we check all possible forests to see if they can be extended to this terminal node. This is a very high number of possibilities (even though it's asymptotically the same). By adding that one node in the beginning, we prune off all these possibilities.

We have implemented this algorithm in a top down fashion so that we only go to the required states and don't enumerate each and every possible state. Top down approaches prune the search tree and tend to be more efficient. At each step we memoise the solution.

3.1 Leaf Node

Since every bag has size one, there are two options for X. Either $X = \{u^*\}$ or $X = \emptyset$. Hence, we have two possible reachable DP states. $dp[t, \{u^*\}, \{\{u^*\}\}] = 0$ and $dp[t, \emptyset, \emptyset] = INF$.

We can't build a solution from the second case since we don't uphold the invariant that the forest spans all the terminals introduced in the subtree of the node.

3.2 Introduce Vertex Node

While introducing a vertex v without it's edges, we have two possibilities. If $v \in X$, then the partition should have a new block containing a single vertex v. This adds a new component to the forest. In this case, we recurse to the state where v is not contained in X and the block $\{v\}$ is not contained in $\mathcal{P}$ and recurse on that state. Otherwise, we ignore the vertex and recurse. Note that if v is a terminal then v has to be in X, otherwise it's impossible and we return INF. Also, if $v = u^*$, we just simply recurse to the child with the same X and $\mathcal{P}$. This is because u^* had already been introduced initially in every leaf node. We shouldn't introduce it again.

$$dp[t, X, \mathcal{P}] = \begin{cases} dp[t', X \backslash \{v\}, \mathcal{P} \backslash \{\{v\}\}] & v \in X \\ dp[t', X, \mathcal{P}] & \text{else} \end{cases}$$

3.3 Forget Vertex Node

When we forget vertex v, we forget all it's edges as well. We have two possibilities here. If v is not in the child solution, then we don't need to do anything. We just recurse to the child with the same X and $\mathcal{P}$. We do the same if $v = u^*$ as u^* can only be forgotten at the root.

If v is in some child solution, then we have a variety of child states to get our answer from. Formally we check all states which include v in their solution i.e. $X \cup \{v\}$. However, there can be a lot of partitions. We can generate all the child partitions by simply adding v to a block in $\mathcal{P}$. Essentially, if we had b blocks in $\mathcal{P}$ we are generating b new partitions $\mathcal{P}'$ and recursing on them. Note that we cannot create a new block for v as by doing so, we wouldn't ensure that the tree is a single connected component.

$$dp[t, X, \mathcal{P}] = \min\{\min_{\mathcal{P}'} dp[t', X, \mathcal{P}], dp[t', X \cup \{v\}, \mathcal{P}']\}$$

3.4 Introduce Edge Node

We add an edge (u, v) in this node. If $u \notin X$ or $v \notin X$, then there's no point in adding the edge (u, v) to our solution. Also, if this is not the case, but u and v are in different blocks in $\mathcal{P}$, then we don't add this edge either and just recurse to the child with X and $\mathcal{P}$.

However, if there is a block i $\mathcal{P}$ such that u and v are a part of it, then we need to find suitable partitions to recurse on for the child node. Note that X stays the same for the child. Let the block be b. Now, we need to generate partitions such that after adding the edge (u, v), we get $\mathcal{P}$. All blocks except b are unaffected. We can split b by making two blocks, one with u and the other with v. Now each of the other $|b| - 2$ other vertices have two choices. Hence, we can generate $2^{|b|-2}$ new partitions $\mathcal{P}'$.

$$dp[t, X, \mathcal{P}] = \min\{\min_{\mathcal{P}'} dp[t', X, \mathcal{P}'] + w(u, v), dp[t', X, \mathcal{P}]\}$$

3.5 Join Node

In join nodes, $X_t = X_{t_1} = X_{t_2}$. The only change is in the partition $\mathcal{P}$. Now, we want to combine all partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ such that while merging the forests from their respective solutions, we get a forest in the current solution. The only way this can happen is if no cycles are formed in this process. While we can add cycles and show that the minimum will always be a forest, by ensuring the pairs we compute the DP for merge to form a forest, we are pruning out all the cyclic merges which further optimises the solution.

The merge of two partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ are said to be acyclic if $\forall u, v \in X, \nexists b_1 \in \mathcal{P}_1, b_2 \in \mathcal{P}_2$ such that $u \in b_1$ and $v \in b_1$ and $u \in b_2$ and $v \in b_2$. This is because if $\exists b \in \mathcal{P}$ where $u \in b$ and $v \in b$, then it implies that they are connected indirectly via a path in G_t. This is because the only time we merge two blocks into one is in introduce edge node.

In this case, that would imply that there is a path from u to v in G_{t_1} and in G_{t_2} which forms a cycle when the two forests are merged into G_t.

The calculation of such pairs for each partition has been described below in Section 4. This generates all partitions for vertices in the range $[0 \dots n]$. For example, the pairs of the partition is $\{\{0\}, \{1, 2\}\} \rightarrow (\{\{0\}, \{1, 2\}\}, \{\{0\}, \{1\}, \{2\}\})$. Now if the set $X = \{1, 3, 4\}$, then the pairs for the partition $\{\{1\}, \{3, 4\}\} \rightarrow (\{\{1\}, \{3, 4\}\}, \{\{1\}, \{3\}, \{4\}\})$ can just be mapped from the earlier case since the structure of the partition is the same.

$$dp[t, X, \mathcal{P}] = \min_{\mathcal{P}_1, \mathcal{P}_2} \{dp[t', X, \mathcal{P}_1] + dp[t', X, \mathcal{P}_2]\}$$

3.6 Weird Nodes

Since the tree decompositions given in the input weren't the most optimal, there were times when the bags were repeated. As in, the contents of two bags in the tree decomposition were the same and they were connected by edges. Ideally, we would like to merge them, but that would've greatly altered the nice tree decomposition algorithm explained in Section 2. This is because such a node couldn't be classified into the 4 regular nodes since there's no chance in the bag. Hence, we decided to keep them as is, make join nodes if necessary and just simply call the recurrence for the child.

$$dp[t, X, \mathcal{P}] = dp[t', X, \mathcal{P}]$$

3.7 Running Time Analysis

If the treewidth is k, then the bag size is $k + 2$ (We added one terminal node to each bag). The total work done in each (non-join) node is $O(k^k * 2^k * nlogn)$. In each join node however,

k^{2k} because we are recursing over all pairs whose acyclic merge forms the node. The total time complexity is $O(k^{O(k)})$. However, in practice it's a little better since $k = 6$ worked under 5 minutes. If we do calculate 8^16 (taking $k + 2$), that's over 1e14 instructions which would take over a day to compute.

The k^k for the join node is actually $B(n)$ which is the n^{th} Bell number. Hence, the bound for each join node cannot be more than $B(k+2)^2$. Asymptotically however, the time complexity is the same.

4 Generating Partition Pairs for Join Node

In the DP recurrence for join node, given a partition $\mathcal{P}$, we need to find all pairs of partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ such that the acyclic merge of $\mathcal{P}_1$ and $\mathcal{P}_2$ form $\mathcal{P}$.

4.1 Generating Partitions

We can generate partitions recursively. Let $\mathcal{P}_n$ denote the set of all partitions of n numbers. Each partition is made up of *blocks*. A set of mutually disjoint blocks which collectively exhausts n numbers forms a partition.

To generate $\mathcal{P}_n$, we can do so recursively by first generating $\mathcal{P}_{n-1}$. Now, for each partition $p \in \mathcal{P}_{n-1}$, for each block $b \in p$, we append $\{n\}$ to that block and call the resulting partition a new partition. Lastly, we make a block from $\{n\}$ add add that to the partition p. Hence, if p has c blocks, we would've created $c + 1$ new partitions from it.

For example, consider $\{\{0, 1\}, \{2\}\} \in \mathcal{P}_2$. Now, from this partition, we can create 3 more partitions after adding the number 3. They are - $\{\{0, 1, 3\}, \{2\}\}$, $\{\{0, 1\}, \{2, 3\}\}$ and $\{\{0, 1\}, \{2\}, \{3\}\}$.

This process takes $O(B(n))$ where $B(n)$ is the n^{th} bell number. Practically, we were able to generate partitions up to 12 in less than 5 minutes.

Now, we go over all pairs of the partitions formed above. For each pair, we would like to see if an acyclic merge is possible, and if it is then what partition is the result of the merge. By keeping a record of the pair of partitions and the merged partition, we can build a map when given a partition, we return the list of pairs of partitions which when merged form the given partition.

This is really useful because instead of enumerating over all possible ways of getting pairs p_1 and p_2 for a partition p in a join node, we can just look up this table and recurse to the correct states without wasting computation time to check if the partitions formed are legitimate or not.

4.2 Acyclic Merge

Given two partitions p_1 and p_2, we need to check if their merge is acyclic and then actually merge the two partitions. Recall that a block in a partition represents a connected component. Hence, if two vertices belong to the same connected component in p_1 and p_2, after the merge we will form a cycle.

To prevent this, we could simply enumerate over all possible pairs of vertices and check if they belong to the same block in both the partitions or not. This takes $O(k^2)$ where $k-1$ is the treewidth of the graph. We could optimise this further to $O(k * \alpha(k))$ by using the DSU data structure.

The idea is that we could think of p_1 as a graph where each block is a connected component and add that to the data structure. This can be done in Now, for each block in p_2 all the vertices have to belong to different components. If not, by merging the two partitions, we create a cycle. This check can be done in $O(b * \alpha(b))$ time overall, where b is the size of the block. Summing over all blocks, we can do this in $O(k * \alpha(k))$. Hence, the overall time taken is $O(k * \alpha(k))$.

Finally, if the above conditions are satisfied and the two partitions can be merged, we can treat p_2 as a graph where each block is a connected component and accordingly add the edges to the DSU. Since p_1 is already added, all vertices connected in the DSU form a new block. Let's see an example.

We want to merge partitions $\{\{0,1\},\{2\}\}$ and $\{\{0\},\{1,2\}\}$. Now, note that their merge is acyclic. When we add p_1 to the DSU initially, 0 and 1 form one component while 2 forms its own second component. Now, when we add p_2, since 1 and 2 are connected in p_2, once we add that edge, 2 gets connected to 1 resulting in one single block $\{\{0,1,2\}\}$ which is the result of merging our partition.

Since we are checking this merge condition for every pair of vertices, this entire procedure of generating pairs takes $O(B(k)^2 * k * \alpha(k))$. Practically, we could run it for $k \leq 8$ under 5 minutes.

5 Implementation

6 DP with low terminals (Track 1)

7 Observations

We created 11 sample test cases of small graphs of around 2-10 nodes ranging from a weighted C_5 to a P_4. We individually verified each case and concluded that they were correct.

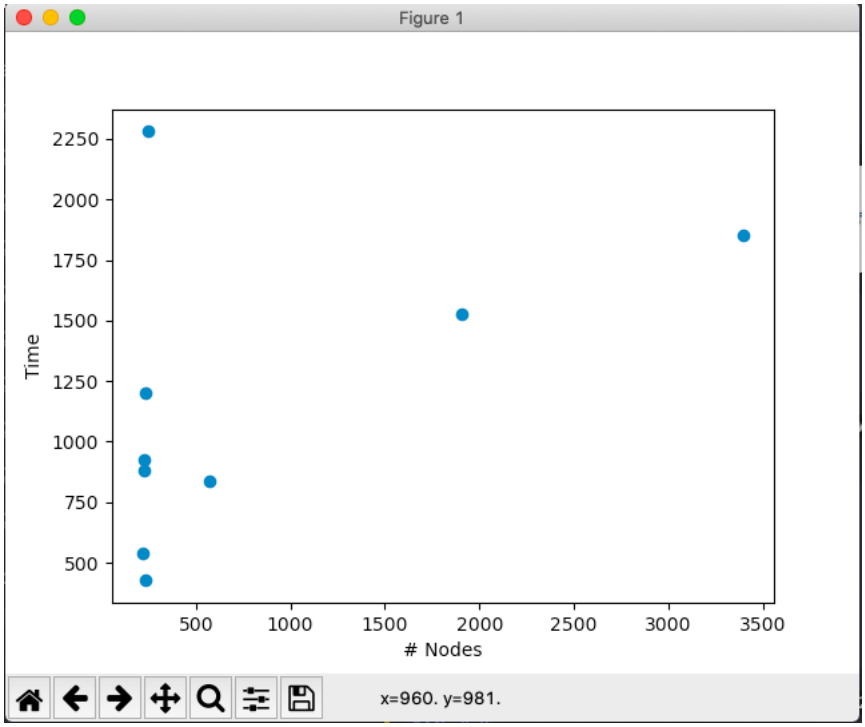


Figure 3: Plot of $|V(G)|$ vs time

7.1 PACE Dataset Observations

We then ran our program on the testcases in the PACE dataset. We were able to run 14 tests successfully within the time limit of 30 minutes.

We were only able to run tests with $k \leq 6$. All other testcases had a higher treewidth. This is because the bag size becomes 8, and the join node takes up a lot of computation time because we go through all $B(k)$ partitions pairwise to compute their acyclic merges. When the bag size is 9, $B(k) = 21147$, where $\binom{B(9)}{2}$ is huge to store in memory (takes around 700MB). Hence, reading and writing from this pre computed structure takes a lot of time on our PCs.

We believe with a dedicated engine (like an online judge), our performance would be better.

It's worth noting that the participants in the actual PACE 2018 Challenge, either implemented the $O(2^k)$ algorithm or implemented the current algorithm, but switched to a more efficient one when the treewidth crossed a certain threshold.

While analysing the amount of time each testcase (instances 6 - 14 from PACE 2018 dataset), we realised that while the time taken to compute the optimal weight increased as the graph size increased, there was no direct relation showing so. As it can be seen in Fig. 3, there are small graphs which have similar sizes (around 250 vertices) but tend to take

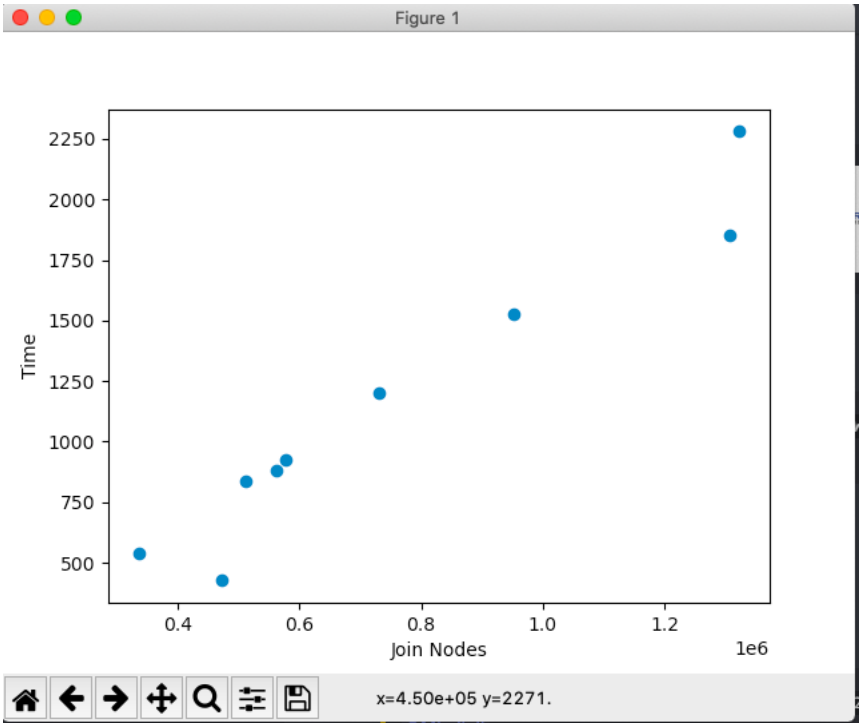


Figure 4: Weighted sum of Join Nodes vs time

different times.

Since this was unusual, we ended up plotting different graphs for edges vs time and vertices * edges vs time to see if there was any difference and there wasn't. These plots looked like Fig. 3.

Our next hypothesis was that since Join Nodes tend to take more time to compute, there might be more join nodes in the graphs which take more time. This is because the treewidth is the same amongst the test cases we are considering. However, even this plot looked exactly like Fig. 3.

The reason was that while the treewidth is the same for all graphs, the join nodes in some graphs could have a smaller bag size than some other graphs. Hence, graphs with more join nodes with bigger bag sizes (but limited by treewidth) should take more time. To compute this, we summed up $|B|^{|B|}$ over all bags B of the Join Nodes in the nice tree decomposition. We got the graph in Fig. 4. As we can see, there is a clear increase in time as this weighted sum increases. This also shows that the running time is asymptotically $O(k^O(k))$.

This algorithm could not be tested on the track A dataset since the tree decompositions weren't given.

8 Conclusions

References