1 Introduction

In the last lecture we looked at two greedy algorithms, Kruskal’s and Prim’s, for finding a minimum spanning tree (MST) in a connected, undirected graph. Then we said that it can be proved, by transformation of the optimal MST, that the spanning tree computed by the greedy algorithm is really minimal. So in this lecture, first we will present how can one do this transformation, and what can be learned from it.

Then we will continue with Huffman codes. We will give short introduction what are Huffman codes, we will look at the greedy approach of generating optimal coding tree and finally we will give a proof that this greedy algorithm actually gives an optimal tree.

Last we will talk about the travelling salesperson problem, with which we will show one example when greedy is not giving the wanted solution.

2 Transforming MST

Assume that $T^*$ is an optimal MST, and assume that $T^{(G)}$ is the spanning tree computed by the greedy algorithm. In order to prove that $T^{(G)}$ is MST, we need to transform $T^*$ into $T^{(G)}$, step by step, without increasing its total length.

$$T^{(O)} = T^* \rightarrow T^{(1)} \rightarrow T^{(2)} \rightarrow \ldots \rightarrow T^{(k)} = T^{(G)}$$

By this we are proving two things at the same time:

(i) $T^{(G)}$ is an optimal tree, since it has the same length (cost) as $T^*$;

(ii) there are a number of different optimal MST, since every single tree on the path of transformation is an optimal MST. These trees does not have to be unique.

By looking the proof into more details, we would realize that at some point, in the greedy algorithm, there is a tie in the construction of the tree. The greedy algorithm gets two equivalent different choices for building the MST, and whichever choice it will make, it will construct an optimal tree.

What does one of the steps of transformation look like?

The optimality of $T^{(G)}$ can be proven by induction. We will induct on the behaviour of the greedy algorithm, as it stops building a tree when it will get an optimal tree. So, for every edge picked by the greedy algorithm, need to be
checked if that edge is also in the optimal tree. As long as greedy picks edges that are also in T*, we know that it works correctly - it picks optimal edges. However, the first time greedy picks an edge that is not into the optimal tree, there is a potential problem.

This problem can be managed by the following induction step, consisted of three operations:

1) Let the $i^{(th)}$ edge, $\{u,v\}$, chosen by the greedy algorithm be the first edge that is not also in T*.

2) Modify T* by putting $\{u,v\}$ in it. This will form a cycle.

3) Remove an edge from the formed cycle, different from $\{u,v\}$, by what the structure will be restored to be tree again. The resulting tree will have same cost as T* and will agree with T($G$) at least until the next step.

**What does this cycle look like?**

The Kruskal’s and Prim’s algorithms start by putting every single vertex in its own equivalence class. After this, they chose greedily which edge will be added next and merge the two equivalence classes of the vertexes that this edge is connecting. Then, they repeat the last step until the minimum spanning tree is not built.

So, at any time there is a collection of equivalence classes, and each of these classes is a tree in a forest. Most of the trees are single vertices, and the others are subtrees that the greedy algorithm has build.

On the figure below, is represented the optimal tree T*, before the $i^{(th)}$ edge is added. The circles represent the equivalence classes build by adding the $i - 1$ edges from the greedy algorithm, which are trees with one or more vertices, and the blue lines represent the edges which are connecting the given equivalence classes and are still not chosen by the greedy algorithm. Until this point T* and T($G$) agree with each other.

![Figure 1: T* before adding the edge $\{u,v\}$.](image)

When the $i^{(th)}$ edge will be added, a cycle will be formed in the optimal tree. In order to break this cycle, one must go around it, and find an edge which will be removed.
From Figure 2, may be concluded that creating a cycle is same as connecting two equivalence classes. So, the cycle formed by adding the red edge \( \{u,v\} \) will need to exit from the class of the vertex \( u \), maybe go through another equivalence classes, then enter the class of the vertex \( v \), and go back to \( u \). What is happening inside the classes of \( u \) and \( v \) is not a problem, because these green edges are also chosen by the greedy algorithm. So, we should remove one of the two edges which represent a connection between two equivalence classes - either the one that is going out of the equivalence class of \( u \), or the one that is going in the equivalence class of \( v \). These two edges can be identified if one look for the first and the last edge of the cycle. An important thing to be noted is that these two edges are known not to be chosen among the first \( i-1 \) edges by the greedy algorithm. So, these two edges are candidates for removing from the cycle, and we must to know that they are not shorter then \( \{u,v\} \).

What can be said about the value of the length of the edge \( \{u,v\} \) versus the value of the length of the two candidate edges?

Since the two greedy algorithms choose the edges differently, we must analyse them separately.

The Kruskal’s algorithm can choose any edge between two equivalence classes. Since it is not bounded to build a single tree at first, it has no annotation on which edge it can take next. So, it would have look at all of the left edges and will choose the smallest one among all of them. In this case it has chosen the red edge \( \{u,v\} \), so that means that this edge is the smallest edge among all the edges it has looked at. From here it can be concluded that the red edge is not longer then the two candidate edges, so we can remove one of the candidate edges and not change the cost of the tree.

The Prim’s algorithm can not look at all the edges, because it is restricted to the edges that have the same equivalence class - the one that is initialled at the very beginning, and which is containing the starting vertex. However, since it have chosen the red edge that means that the starting equivalence class was either the one with \( u \), or the one with \( v \) in it. So, Prim’s algorithm will compare
\{u,v\} with just one of the candidate edges, depending on which vertex \(u\) or \(v\) belongs to the initial spanning tree.

Also, since \(T^*\) is the optimal spanning tree, it is known that the candidate edges are not longer than \{u,v\}.

So, for two different reasons, these candidate edges must be with the same width as \{u,v\}.

After one induction step is made in this way, we would get a new optimal tree that has a greedy edge in it. The greedy algorithm now agrees with \(i\), instead of \(i−1\) edges with the optimal tree, and we know that it is still optimal. However, it may continue to compare the edges that it is adding and to repeat this step for any chosen edge which is not belonging to the optimal tree.

The other thing that is proved is that if this happened, then there was a tie in the greedy algorithm - there were two or three edges with same cost, and the greedy algorithm had to choose just one of them.

**Corollary:** For a given graph, if all edges have different length, then the minimum spanning tree is unique.

### 3 Huffman Codes

Huffman codes are used as a way to reduce the length of the bits transmitted or to reduce the probability of having the message understood by a third party.

Here, the Huffman code main purpose is to reduce the length of the message to be transmitted by taking advantage of the uneven probability distribution of the symbols in the message. If we take the English for instance, the most common letter by far is the letter \("e\)", so we would like to give to it a very short code. An incredibly rare letter is \(q\). Every word with \(q\) in English is basically pretty much quite formal and a foreign word, so we could give it a long code. We could do this for \("e\)" and \("q\) instead of giving them a code of the same length, which is practically what ASCII code is doing. So by changing the length of code and defining common symbols with short code and rare symbols long code, we will reduce the overall length of the message. We could do this not just at the level of symbol, but also at the level of words. We could encode much larger and much more complicated code book that encodes words, so in English there will be a very short code for the letter \("a"\) as a word, the article. We will also have a very short code for the word \("the". If you transmit a message in Hawaiian, which has lots of vowels and is missing a lot of consonants, there is no point in using an ASCII code when there are bunch of things that are not even used. All of this also depends on the document that is transmitted. If for instance you want to transmit an SMS, the vocabulary used is quite different, the frequencies of letters and words compared to when you want to transmit a book from Shakespeare or Victor Hugo or some old time classic which is going to use
flowery language, long words and have a different frequency distribution. This is why we want to tailor the code to the particular document that is transmitted. This is all the idea behind Huffman code.

The other things we need to decide on when transmitting a code is what conditions we are going to put on the code and the decoding on the other end of the transmission line. This is very important because if you mix the bits on the way of the transmission or something goes wrong, at the end we would like to know that not everything is completely lost.

Coding something in the absolute shortest possible code that could be transmitted is a terrible idea, because it would be completely fragile. If one little thing goes wrong, then the entire message can get thrown away since the message could not be understood if we don’t know what is missing.

Huffman code tends to shorten the message, but not as much as one could with arbitrarily coding, cause they have requirement to be prefix free, which allows the code to be synchronized, so if something goes wrong, later on you can understand what has happened.

The prefix free code means that no final code that you transmit can be the prefix of another code. This is a nice property in terms of decoding and also in terms of genuine code.

There are many more sophisticated codes than this one. This is in many ways from the dark age of coding theory. It is very simple, but it’s also worth looking at it.

Now let’s proceed with looking at how actually we can come up with the code.

Let’s assume that the coding alphabet is binary. This simplifies matters, the algorithm would really change if we have a larger alphabet. The code can be placed on a binary tree. To think about this, we start with the empty code word $\lambda$, meaning if there is nothing to be transmitted. This is not the end of the story. Obviously we need at least one code word. Afterwards we choose between codes that start with a 1 or codes that start with a 0. In this way we will have a collection of codes of 0 prefixes and 1 prefixes. The tree separates them, so the tree will give us the prefix free property, meaning that some code in the left subtree cannot be a prefix of something in the right subtree and vice versa. And recursively this would be true, so we get a binary tree to a series of code. From the English language, the letter e would be next to the root, coded with one bit, and for the the letter q it be better to have a longer code, meaning it’s code word will be somewhere down in the tree and it would cost us more bits to represent it. Now what we need is the probability of occurrence in the text you’ve sent of each of the characters to be transmitted and then we’re going to evaluate the equality of the length of the code in such way.

Given the text alphabet $\Sigma = \{a, b, ..., z\}$ and given a frequency for each character, a probability of occurrence for each character $c$, $c \in \Sigma$, is:

$$p : \Sigma \rightarrow [0, 1]$$

We’re going to construct a code tree that minimizes the average length of the code:
This is a sum of the probability of observing the input character $c$ in the text denoted as $p(c)$, times the length of the code of $c$, $\text{length}(\text{code}(c))$ which is the length of the path from the root of the tree to the leaf representing the character. Each code has a certain probability representing the frequency of appearance in the text to be transmitted. An optimal code for a file is always represented by a full binary tree, in which every nonleaf node has two children.

We minimize this sum with an algorithm that starts with building a tree with $|\Sigma|$ number of leaves, one for each letter of the alphabet, and exactly $|\Sigma| - 1$ internal nodes. The starting point of the algorithm is picking the right symbols that need to be encoded. We have all the characters from the text alphabet with their corresponding frequency probabilities:

```
· · · · · · ·
 a b c ... z
```

Using the greedy approach, there are two ways for building this binary tree. The first one is the top down approach where we want to group all the nodes that need to be on the left side of the tree and group all the nodes that need to be on the right side, by having the sum of the frequencies of these two groups to be $\frac{1}{2}$ each. After this some recursive divide and conquer needs to be done. But here this is a difficult approach to be done right.

A much easier greedy approach is the bottom up one, so the algorithm works as follows: First we take the two lowest frequency symbols, and create something that we call a "cherry" (a subtree). For our example, if we assume that the characters "b" and "c" have the lowest frequencies, their cherry will look like this:

```
   p(b) + p(c)
  /        \
 p(b)      p(c)
```

This is the first step in building a tree which is keeping the symbols together by giving them a common parent, so that the parent’s frequency is the sum of the frequencies (probabilities) of it’s children, "b" and "c" in our case. This is just a reverse to the other process. You take two things that you hope are already trees and give them a common parent which merges them and generates a new bigger tree. It represents a bottom up regime of a divide and conquer algorithm. This process results in having a heap since at the bottom are the least frequent symbols and the access frequencies rise as we move towards the root of the tree.
where $p(bc) = p(b) + p(c)$ as said before

So, the cherries are at the bottom of the tree. We continue by doing the greediest thing possible which is: take the two lowest frequency items (symbols or cherries or subtrees), merge them by giving them a common parent which has a probability that is the sum of the two merged items. In this process, in the character set $\Sigma$, we replace these two items and their probabilities with the new pseudo character (the parent of the cherry) and it’s probability. In this way we are moving bottom up and $|\Sigma|$ decreases.

$$prob(bc) = prob(b) + prob(c)$$

We repeat this process until there is one item left in $\Sigma$, meaning that we pair until there is nothing left to pair.

**Pick two items with lowest probabilities and form a subtree**

$$prob(subtree) \leftarrow prob(item1) + prob(item2)$$

Now we claim that this generates an optimal tree, i.e. a tree that has a minimum expected average code length (path length). We need to prove this and we need to think about about the possible equalities especially if we don’t have any fine grain information about the probability, but of course some sort of quick approximation will count.

**Proof by induction**

We will prove this claim by induction on the size of $|\Sigma|$, where $\Sigma$ is the alphabet.

The base case is when $|\Sigma| = 1$ and it is not very interesting.

Note: If you are a mathematician, probably the base case is when the alphabet is empty. If you are a computer scientist, you would find that kind of annoying, so for you the case would be when the alphabet has only one symbol.

So for the case when $|\Sigma| = 1$, this is easy. Every message that has arrived in the original language will be $aaaa...aa$ and the message that needs to be transmitted will be $1111...11$. This means that we cannot deliver a message like this in Huffman coding, but the code is optimal. The probability is 1 and the length of the tree is 1, so we have an optimal tree.

The inductive step: Assume it works for up to $|\Sigma| = n - 1$ symbol, and we add one more symbol (the $n-th$ symbol). And the first step, which is the only step in the algorithm, is to merge two subtrees in a cherry, and replace the two characters by a new pseudo characters. This is the only step we need to worry about, because from here on the induction works for us for free.

Now the question is: is the optimality of the overall system compromised by adding the $n-th$ character?
Figure 3: optimal T* tree with n − 1 nodes and the nth node that needs to be added

For any code tree it will be true that there are minimum two leaves at the bottom level of the tree. We know this is true because we are building a binary tree, so every node has to be a leaf or it has two children, giving us an even number of leaves at the lowest level of the tree. If there is one child, we don’t need to do two steps, we can stop immediately and give it a code word right away.

If the greedy algorithm is used, then these two leaves will have the least probabilities. However, there are optimal trees for which this is not true. So, we may conclude that the greedy algorithm will construct optimal trees, but it can not construct all of them, because of the restriction that it is forced to pick only the least probability leaves from the bottom level.

We will do something very similar to what we did for MST. We will transform the optimal code tree T* into a different code tree whose average path length (average code length) stays the same as the one of T*, and at the bottom level we will have two leaves which will be the leaves with the lowest two probabilities. By doing this we will transform T* into a tree with same bottom level as the tree build from the greedy algorithm.

This transformation is done as described in the following lines:

if T* has the two lowest probability leaves on bottom level
then we’re done
otherwise, exchange these two leaves
    with the leaves that now have the least probability

The resulting tree will have the two lowest probabilities on the bottom level
and by doing this we are not increasing the average path length.

The optimal tree will have the same leaves with the least probabilities at the bottom level, but it may differ in the way that it groups them. For instance, if we take a tree with four leaf nodes with the following probabilities:

0, 0.002
0, 0.0025
0, 0.0026
0, 0.0027

the greedy algorithm for these values will generate only one tree looking like this:

![Figure 4: Tree generated by the greedy algorithm]

For the same path length there are two other optimal trees, so we would have an optimal tree if we also pair the nodes with the probabilities (0, 0.002; 0, 0.0027) and (0, 0.0025; 0, 0.0026), or (0, 0.002; 0, 0.0026) and (0, 0.0025; 0, 0.0027), but the greedy algorithm always does the pairing as shown on the figure.

From this we can conclude that at a fixed level in the optimal tree, one can permute the leaves in any way he wants and the resulting tree will be again an optimal tree. However, the greedy algorithm is constrained to pair according to the probabilities, so it is not capable of constructing every possible tree. Anyway we have shown that the trees it constructs are optimal and they are a subset of all the possible optimal trees.

4 The travelling salesperson

The greedy algorithms are very easy to write and they are simple because they are looking just for a local optimisation. However, for some problems they may fail, and one standard example for that is the travelling salesperson problem.

**Problem:** Given \( n \) cities and a distance function \( d(i, j) = \text{distance between cities } i \text{ and } j \), find a cycle through all the cities that goes through each city exactly once and returns to the starting point, then minimize the total path length.
This is an old fashion problem studied for many years and is NP-hard, the existence of an optimal solution in polynomial time is not proved.

There are few strategies for this problem.

- The first one is a simple greedy strategy: go to the next closest city (nearest neighbour algorithm).
  
  This strategy looks good at first, because at the beginning we have a lot of choice. But as we move through the cities, the choices are fewer and the distances are bigger. At the end, moving from the last visited city to the first one can be really expensive in terms of distance.

- The second one is a cleverer strategy: pick the next cheapest subject without
  - creating a vertex of degree 3;
  - closing the cycle (except for the very last edge).

  ![Figure 5: Choosing the next cheapest subject without creating a vertex of degree 3](image)

  On the Figure 5, the next closest vertex to the blue ones is the red one, but then the middle blue vertex will have a degree 3. So the algorithm will pick the green vertex.

  This is a much better greedy algorithm in contrast to the first one, primitive greedy algorithm.

- The third one is the smartest one: choose any three vertices and add another vertex to the tour by connecting it to any two of the vertices already chosen. Then remove the edge connecting those two vertices, thus replacing one edge with two new ones.
  
  The algorithm repeats this procedure vertex after vertex, until all the vertices are added.

  Another heuristic for this problem is based on minimum spanning tree. If we represent the travelling salesperson problem as a graph problem, then the desired tour is a minimum spanning tree with and extra edge on a complete graph.

  In order to get an optimal tour, we modify the spanning tree by turning it into a tour, as shown on Figure 7. The tour traverses every edge twice and goes
through some vertices many times, so for the optimization purpose, making shortcuts takes place. The shortcuts must be chosen cleverly, guaranteeing decreasing of the total length, but not excluding vertices.

On the Figure 7 the green line represents the tour on the spanning tree and the orange line shows a possible reduction of the tour length by shortcuts. The total length of the tour is twice the cost of the tree, so the shortcuts can reduce the cost of the tour at most by half. A shortcut is effective only if the triangle inequality applies, otherwise the length of the tour will be increased.

Thus, this heuristic can give an approximation ratio of 2.