Algorithms and Datastructures (HS2024)
Week 12

Rui Zhang

December 8, 2024

Contents

1 Revision Theory

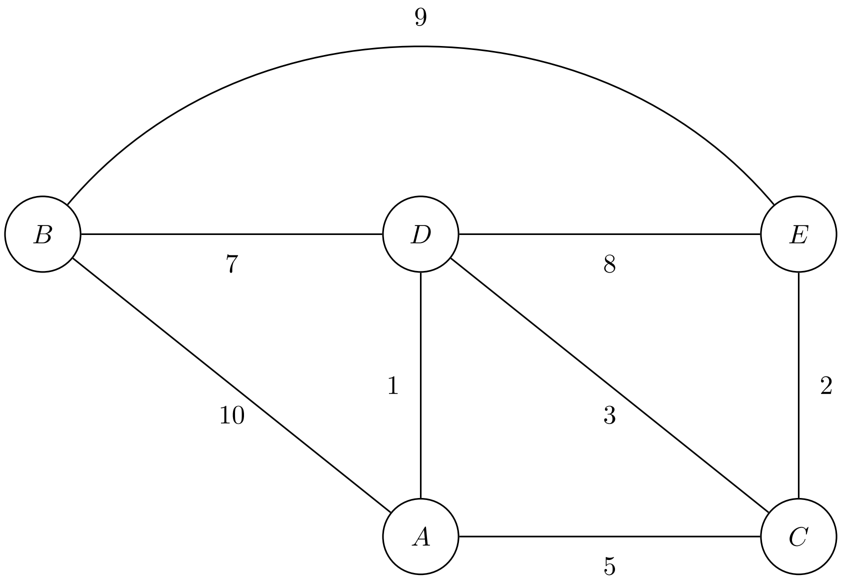
Last week, we looked at two algorithms for finding MSTs in connected, undirected and weighted graphs. This week, we will be looking at one final algorithm:

1.1 Kruskal's Algorithm

Last week, we talked about the concept of safe edges and the cut-principle. To dumb it down, when constructing an MST, we can always add the edge with least weight to the MST, as long as this does not create a cycle (i.e. connect the MST with itself.).

Given a graph $G = (V, E)$ Kruskal makes use of the cut-principle by starting off with just the vertices V of G, without the edges. It sorts the edges E by order of their weight, from smallest to largest. Then, it goes these edges in this sorted order and adds an edge *if this edge connects two different connected components.*

Example:



In this graph, the first edge to be considered is $\{A, D\}$. This edge will be added to the MST. Then comes $\{C, E\}$, which will also be added to the MST. After that, it is $\{C, D\}$'s turn, which will also be added.

The edge that has least weight from here on out is $\{A, C\}$. However, since we have already added the edges $\{A, D\}$, $\{C, E\}$ and $\{C, D\}$, we would be connecting within a connected component if we added $\{A, C\}$. So we skip this edge.

Now comes $\{B, D\}$, which will also be added.

For the remaining edges, we notice that we have already connected all vertices in a tree. So adding any other edge will result in a cycle. Thus, all the remaining edges will be discarded.

Kruskal seems simple enough. We could implement it as a loop which iterates through all edges in sorted order and addes them one by one. To check if an edge connects within a connected component, we would need to perform DFS to even find the connected components of this iteration, which is $O(n+m)$.

Now, since we go through all edges one by one, we would have something like $O(n\cdot(n+m))$, which is much worse than all other MST Algorithms we have seen up until now.

To improve this, we develop a *Union-Find-Datastructure.*

1.2 Union-Find-Datastructure

The goal of this datastructure is to be able to model and form the union of sets in an efficient manner. If we are able to do this, then we can add every vertex in its own set in the beginning and every time we connect two connected components, we would union their sets into one set.

For this, we need the operations *make(V)*, which initializes the datastructure. *same(u, v)*, which returns a boolean value denoting if u and v are in the same connected component / set. *union(u, v)*, which takes the two connected components of u and v and connects them into one connected component.

The pseudocode for union-find is as follows:

```
class UnionFind
  rep = Array[n]
  // for each vertex, denotes the "representative"
  // of the connected component of that vertex

  members = Array of LinkedLists[n]
  // for each representative, saves a LinkedList
  // which contains all vertices in its connected component

  make(V): // V is Vertex Set of Graph
    for each v in V:
      rep[v] = v
      // since in the beginning, we start off with each
      // vertex as its own connected component

  same(u, v):
    return rep[u] == rep[v]
    // if the representatives are the same, then
    // they are obviously in the same connected component

  union(u, v):
    // assume without loss of generality, that the
    // connected component of u is smaller than
    // the connected compoennt of v

    for x in members[rep[u]]:
      rep[x] = rep[v]; // (1)
      members[rep[v]].add(x); // (2)
```

1.3 Runtime of Kruskal and Amortized Analysis

Standard Analysis

Applying Kruskal with this Union-Find-Datastructure avoids the DFS procedure to find the connected components. Every time we add another edge $\{u, v\}$, we would have to call $union(u, v)$.

This part is still going to be the dominating factor in our runtime. Of course, we could analyze the worst case runtime of *union* and multiply it with m, the number of iterations that we will be doing:

The worst case for *union*, is when we have to connect two connected components both of size exactly $n/2$. Since line (1) and (2) in the pseudocode can be done in constant time and we need to iterate through all members of a connected component, we would have a runtime of $O(n/2)$ in the worst case. Multiplying with the number of iterations in Kruskal, we get a total runtime of $O(m \cdot n)$. *Obviously, this worst case does not occur in each iteration of Kruskal!* So this sort of analysis is not realisitic at all.

Amortized Analysis

Instead of considering the number of times (1) and (2) are executed in one execution of *union*, we consider the number of times (1) and (2) are executed during the entire process of Kruskal.

For this we define:

$$N_u := \text{number of times (1) and (2) are executed for } x = u$$

So basically, N_u is the number of times a certain vertex u is added to another connected component during the entirety of Kruskal's m iterations. The total runtime for the union-find-datastructure would then be $O(\sum_{u \in V} N_u)$.

Now, we can use a similar argument as the one we used in Boruvka: Every time we actually execute lines (1) and (2), we would be "merging" two connected components into one. The size of the connected component thus doubles for each execution of union, resulting in the upper bound of $N_u \leq \log_2(n)$ for all $u \in V$, so $O(\sum_{u \in V} N_u) = O(n \cdot \log(n))$. Together with the sorting of edges at the beginning of Kruskal, we have a total runtime of $O(m \cdot log(m) + n \cdot log(n))$.

2 Exercise Sheet

2.1 Exercise Sheet 12 - Priority List

1. - 12.1, 12.3 - Most important for understanding of Kruskal.
2. - 12.5, 12.2 - Not as essential, but still important Kruskal exercises.
3. - 12.4

2.2 Exercise Sheet 10 - Feedback

- When proving correctness, always think about what it is that you actually want to prove. Think of properties that you have to prove that will be sufficient to show that your algorithm is correct.
- $\emptyset$, $\{\}$ is the empty set. Like a box containing nothing in it.
  $\{\emptyset\}$ is the set containing the empty set. Like a box with a box with nothing

in it.
[] is an empty datastructure. Think of a queue with nothing in it.
[$\emptyset$] is a datastructure containing the empty set. Think of a queue with an empty box inside it.
Please, do not mix up these things.

- Do not forget that Array initialization takes as much runtime as the dimensions of the array.

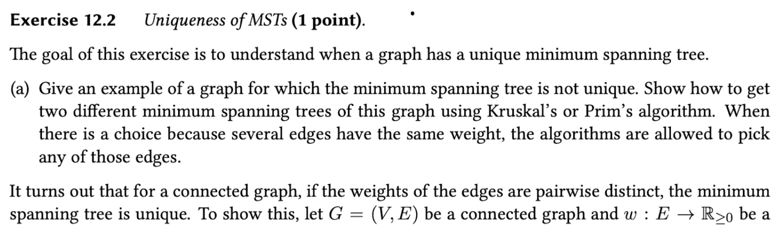
3 Supplementary Exercises

3.1 I love Kruskal

This website makes you apply Kruskals algorithm until you get bored.

3.2 Uniqueness of MSTs

This is an exercise from last years exercise sheet:

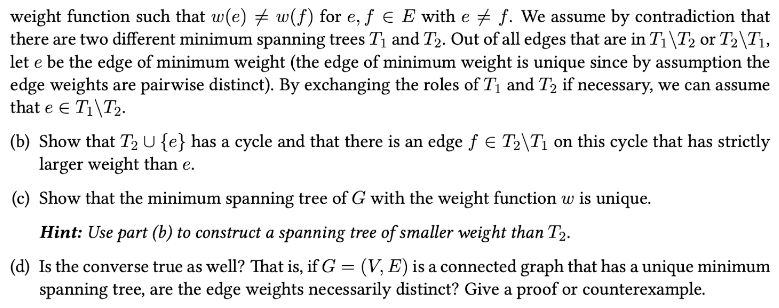


Exercise 12.2 *Uniqueness of MSTs* **(1 point)**.

The goal of this exercise is to understand when a graph has a unique minimum spanning tree.

(a) Give an example of a graph for which the minimum spanning tree is not unique. Show how to get two different minimum spanning trees of this graph using Kruskal's or Prim's algorithm. When there is a choice because several edges have the same weight, the algorithms are allowed to pick any of those edges.

It turns out that for a connected graph, if the weights of the edges are pairwise distinct, the minimum spanning tree is unique. To show this, let $G = (V, E)$ be a connected graph and $w : E \to \mathbb{R}_{\geq 0}$ be a



weight function such that $w(e) \neq w(f)$ for $e, f \in E$ with $e \neq f$. We assume by contradiction that there are two different minimum spanning trees T_1 and T_2. Out of all edges that are in $T_1 \setminus T_2$ or $T_2 \setminus T_1$, let e be the edge of minimum weight (the edge of minimum weight is unique since by assumption the edge weights are pairwise distinct). By exchanging the roles of T_1 and T_2 if necessary, we can assume that $e \in T_1 \setminus T_2$.

(b) Show that $T_2 \cup \{e\}$ has a cycle and that there is an edge $f \in T_2 \setminus T_1$ on this cycle that has strictly larger weight than e.

(c) Show that the minimum spanning tree of G with the weight function w is unique.

***Hint:** Use part (b) to construct a spanning tree of smaller weight than T_2.*

(d) Is the converse true as well? That is, if $G = (V, E)$ is a connected graph that has a unique minimum spanning tree, are the edge weights necessarily distinct? Give a proof or counterexample.