CS 33 24.2 Lecture 1 Notes

Department of Computer Science
College of Engineering
University of the Philippines Diliman

1. The Minimum Spanning Tree problem

Problem 1: Given a connected weighted undirected graph, find a connected spanning subgraph with the fewest number of edges, and among them, the one with the minimum total cost.

Problem 2: Given a connected weighted undirected graph, find a connected spanning subgraph with the minimum total cost.

Claim 3: Any optimal answer to Problem 1 is a spanning tree.

Proof: We proceed by contradiction.

Suppose there's an optimal answer, T, that isn't a spanning tree. Recall that a tree is a connected acyclic graph.

Note that T is connected and spanning because it is an answer to Problem 1, so T must *not* be acyclic, i.e., it must have a cycle.

However, any edge in this cycle isn't a bridge in T, so removing it doesn't disconnect the graph. But doing so gives us a new connected spanning subgraph with *strictly* fewer edges than T, which is a contradiction. Therefore, every optimal answer must be a spanning tree. □

Claim 4: Any optimal answer to Problem 2 is a spanning tree.

Proof: We proceed by contradiction.

Suppose there's an optimal answer, T, that isn't a spanning tree. Recall that a tree is a connected acyclic graph.

Note that T is connected and spanning because it is an answer to Problem 1, so T must *not* be acyclic, i.e., it must have a cycle.

However, any edge in this cycle isn't a bridge in T, so removing it doesn't disconnect the graph. But doing so gives us a new connected spanning subgraph with a *strictly* smaller total cost than T, which is a contradiction. Therefore, every optimal answer must be a spanning tree. □

Note that the proofs are very similar, but the second one uses the fact that the edge weights are *positive* in an important way. (What goes wrong if the weights are allowed to be nonpositive?)

Definition 5: Given a weighted undirected graph, a **minimum spanning tree** (MST) is a spanning tree with the minimum total cost.

2. Kruskal's algorithm

Kruskal's algorithm (named after American computer scientist Joseph Kruskal) works as follows:

```
def mst_kruskal(graph G = (V, E)):
    initialize T to be an empty graph on V
    sort E by increasing cost
    for each edge in E:
        if adding the edge to T doesn't form a cycle:
            Add the edge to T
    return T
```

2.1. Correctness

Kruskal's algorithm is a *greedy* algorithm, so we need to prove that it is correct.

Lemma 6: Let $a_1, a_2, \ldots, a_e$ be the e edges of the graph in increasing order of cost. Let S_k and E_k be the sets of edges included and excluded, respectively, by Kruskal's algorithm after k steps. Then for each k such that $0 \leq k \leq e$, there is an MST containing all edges in S_k and no edges in E_k.

Proof: We proceed by induction.

- **Base case**. the statement is (vacuously) true for $k = 0$: S_0 and E_0 are both empty, so any MST contains all edges of S_0 and no edges of E_0. (An MST exists because the graph is connected.)
- **Inductive case**. Suppose the statement is true for k. We will show that it is true for $k + 1$.

  By the inductive hypothesis, there is an MST, say T, containing all edges in S_k and no edges in E_k.

  Let x and y be the endpoints of a_{k+1}.

  We now consider two cases:

1. **x and y are connected using edges in S_k.** Kruskal's algorithm excludes a_{k+1}, so we have $S_{k+1} = S_k$ and $E_{k+1} = E_k \cup \{a_{k+1}\}$.

   Then T contains all edges in S_{k+1}. It also doesn't contain a_{k+1}, because x and y are connected in $S_k \subseteq T$, which means adding a_{k+1} would form a cycle, contradicting the fact that T is a tree. Therefore, T doesn't contain any edge in $E_k \cup \{a_{k+1}\} = E_{k+1}$. Since T is an MST, this proves the inductive hypothesis for this case.

2. **x and y are connected using edges in S_k.** Kruskal's algorithm includes a_{k+1}, so we have $S_{k+1} = S_k \cup \{a_{k+1}\}$ and $E_{k+1} = E_k$.

   Consider the path from x to y in T (which exists because T is connected). Because there is no path from x to y in S_k, at least one of the edges in this path must be outside of S_k, say a, so that $a \notin S_k$. Note that $a \notin E_k$ because T doesn't contain any edge in E_k. Thus, edge a hasn't been considered yet by the algorithm, and must be one of $a_{k+1}, a_{k+2}, \ldots, a_e$. Since the edges are sorted in increasing order of cost, we have $\text{cost}(a_{k+1}) \leq \text{cost}(a)$.

   Now, consider

   $$T' := (T \setminus \{a\}) \cup \{a_{k+1}\}.$$

   Note that $T \setminus \{a\}$ consists of two trees, one containing x and another containing y (because the unique path from x to y in T contains a). Thus, adding edge a_{k+1} doesn't form a cycle and connects the two trees back together, so T' is a spanning tree. Because $\text{cost}(a_{k+1}) \leq \text{cost}(a)$, we have

   $$\text{cost}(T') = \text{cost}(T) - \text{cost}(a) + \text{cost}(a_{k+1}) \leq \text{cost}(T),$$

   and since T has the minimum cost among all spanning trees, we have $\text{cost}(T') = \text{cost}(T)$, and T' is an MST as well. Also, note that T' contains all edges in $S_{k+1} = S_k \cup \{a_{k+1}\}$ (because $a \notin S_k$) and doesn't contain any edge in $E_{k+1} = E_k$ (because $a_{k+1} \notin E_k$), thus the inductive case is proven. □

The above is an example of an **exchange argument**.

Note also that in the second case, a and a_{k+1} could be equal, which would imply that $T = T'$. However, the proof doesn't break; it simply means that a_{k+1} was already in T to begin with!

Theorem 7: Kruskal's algorithm correctly returns an MST.

Proof: Once Kruskal's algorithm finishes, the edges in S_e will have been included and the edges in E_e will have been excluded. Lemma 6 guarantees that there is an MST containing all edges in S_e and no edges in E_e, say T. But note that $S_e \cup E_e$ is the set of *all* edges $\{a_1, \ldots, a_e\}$, so T must be exactly S_e, i.e., Kruskal's algorithm successfully outputted an MST. □

Notice that we've used the assumption that the graph is *connected*; without it, there wouldn't be an MST, so in particular, the base case of Lemma 6 is false! Contrast this with the proof idea in CS 32 LE 3, which also uses the assumption that the graph is connected, but is used in a different place.

2.2. Implementation

There are two straightforward implementations.

- To check if a cycle is formed when adding some edge, say (x, y, c), to T, one might perform a traversal/search (BFS or DFS) along T. This runs in $\mathcal{O}(n)$ time, because T will be acyclic at every step. Since this traversal needs to be performed for each of the e edges, this runs in $\mathcal{O}(ne)$ time.
- One could use a **disjoint-set union data structure** (DSU) to maintain the connected components, because the only operations we need are:
  - check if two elements are in the same component.
  - merge two components into one.

  A straightforward way to do this would just be to use a list of lists, or a list of sets. With such a structure, each of the two operations above can be performed in $\mathcal{O}(n)$ time. Since there are $\mathcal{O}(1)$ such operations for each of the e edges, there are $\mathcal{O}(e)$ operations, so this runs in $\mathcal{O}(ne)$ time as well.

Note that the sorting step in both cases runs in $\mathcal{O}(e \log e)$ if we use a fast sorting algorithm, but this is dwarfed by $\mathcal{O}(ne)$.

We can optimize the second implementation by using a faster DSU structure. For example, using parent pointer trees with path compression and union by rank/weight, each operation now becomes $\mathcal{O}(\alpha(n))$ where α is the slow-growing inverse Ackermann function.

Now, because of this, the main loop now runs in $\mathcal{O}(e \cdot \alpha(n))$, so the $\mathcal{O}(e \log e)$ step now dominates. This gives us an $\mathcal{O}(e \log e)$ implementation of Kruskal's algorithm, which is fast!

The $\mathcal{O}(e \log e)$ running time can be improved to $\mathcal{O}(e \log n)$ if one uses a min heap with a *decrease key* operation, which decreases the value of any element in the heap (regardless of its location in the heap).[1] This can be further improved to $\mathcal{O}(e + n \log n)$ with the use of more advanced heaps where the *decrease key* operation runs in $\mathcal{O}(1)$, e.g., a Fibonacci heap.

3. Prim's algorithm

Prim's algorithm (named after American computer scientist Robert Prim, but developed earlier by Czech mathematician Vojtěch Jarník, so it's sometimes called the Jarník-Prim algorithm) works as follows:

```
def mst_prim(graph G = (V, E)):
    let x be an arbitrary node in G
    initialize T to be the graph containing only x
    while T doesn't contain all nodes V:
        let a be the minimum-cost edge connecting a node in T and a node not in T
        add a to T
    return T
```

[1] Note that $\mathcal{O}(e \log e)$ and $\mathcal{O}(e \log n)$ are basically the same for simple connected graphs. (Why?)

3.1. Correctness

Prim's algorithm is also a greedy algorithm, so we also need to prove that it's correct. The correctness can be proven by induction similarly to Kruskal's algorithm, by finding an analogue of Lemma 6. It is left as an exercise to the reader.

3.2. Implementation

Note that the main loop runs $n - 1$ times, because each iteration increases the number of nodes in T by 1, and it starts at 1 (because of x).

A straightforward implementation would be, for every iteration, to go through all edges to get all edges connecting a node in T to a node outside of T, and to let a be the one with the minimum cost. This runs in $\mathcal{O}(n + e)$ time, which is just $\mathcal{O}(e)$ if the graph is connected (why), and because the main loop runs $\mathcal{O}(n)$ times, this means this implementation runs in $\mathcal{O}(ne)$ time.

We can improve this by maintaining the *minimum-cost* edge to each node that connects it to some node in T.

Formally, let C_i be the minimum-cost edge connecting i to some node in T. We initialize the C_i by considering the edges connected to x. (Thus, for some nodes i, C_i may be empty initially, if i is not adjacent to x.)

Now, every time we add a node y to T, we simply go through all neighbors of y and possibly update their C_i. With this implementation, we can look for the minimum-cost edge in $\mathcal{O}(n)$ instead of $\mathcal{O}(e)$ because we can just go through the C_is. Since each edge is only considered twice (for each direction), the total updates runs in $\mathcal{O}(e)$, so the whole algorithm now runs in $\mathcal{O}(n^2 + e)$. For simple graphs, this becomes $\mathcal{O}(n^2)$. (Why?)

We can further improve this. Notice that we only need to get the *minimum* value among the C_is. Thus, we can use a *min heap* to optimize it! By putting the C_i values in a min heap, we can extract the minimum one quickly: $\mathcal{O}(\log n)$ time.

However, an issue is that the C_i values are updating as Prim's algorithm executes. However, each update only decreases C_i and never increases it (why?), so a solution would be to just *push* C_i *again* every time it updates. And then when we pop from the min heap, we only include each node at most once; we ignore the rest of the C_i values since they are "outdated". This implementation now runs in $\mathcal{O}(e \log e)$ time. (Why?)

4. "Generic" MST algorithm

Kruskal's algorithm and Prim's algorithm have a similarity: they start off from an "empty" graph and incrementally include more nodes and edges until an MST is formed. In other words, they both *grow* the MST. They simply differ in the exact way they do this. In CLRS terms, they both fall under the umbrella "generic MST algorithm".

In both cases, the main idea is that every new edge they add must be "safe"; i.e., if T is the set of edges included so far, then T must be a subset of some MST, and if a is the next edge they add, then $T \cup \{a\}$ must be a subset of some MST as well. The way they find a "safe" edge just differ.

5. Borůvka's algorithm

Borůvka's algorithm (named after Czech mathematician Otakar Borůvka) is another algorithm to compute the MST.

It proceeds in several steps, each producing a *forest* with fewer components than the previous one, until it ends up with a forest containing a single tree, which will be the minimum spanning tree. Initially, the forest consists of n trees, each containing a single node.

In each step, the algorithm looks for the minimum-cost edge connecting each tree of the forest to a different tree; all of those edges are simultaneouly added. (Of course, if the same edge is the minimum-cost edge for different trees, then the edge is only added once.) This step merges some of the trees, producing a forest with strictly fewer components. The algorithm ends if there is only one tree remaining.

Note that there's an ambiguity here if there are edges with the same cost, and unlike Kruskal's and Prim's algorithms, this time it *matters* a lot how you break the ties; you can't just arbitrarily break ties! If you break ties in the wrong way, you might not end up with an MST! It is left as an exercise to the reader how to correctly break ties so that the final result of the algorithm is an MST.

The beauty of Borůvka's algorithm is that it is more easily *parallelizable*, because each tree in the forest can be processed (relatively) independently of the other trees. This is unlike Kruskal's and Prim's algorithms which are harder to parallelize, since each step depends on the outcomes of the previous steps in an essential way. (In fact, this algorithm is frequently called Sollin's algorithm in the parallel computing literature.)

6. Read More

- Minimum-cost spanning trees (NOI.PH Training: Bootcamp)
- Standard Algorithms on Graphs (NOI.PH Training: Graphs 2)

7. OJ Practice Problems

- Shichikuji and Power Grid
- Anti Brute Force Lock

Important Note: To get Accepted in these problems, you might need a fast implementation *and* a fast language; Python might not make it. You can use C, but if it proves too inconvenient to implement things in C, you can opt to use C++, which is more-or-less a superset of C, with additional conveniences. See this Python-to-C++ quick guide for more. To compile your program, enter the command

```
g++ -O2 -std=c++17 prog.cpp -o prog
```

This produces a file named `prog`. To run the program, enter:

```
./prog
```

In the compilation command, `-O2` is an optimization flag, `-std=c++17` means you can use the features up to version C++17[2], and the `-o prog` part means that you want the compiled program to be saved to the file `prog`.

If you want more practice problems, go to this ProgVar.Fun problem set.

There are more problems at the end of the reading materials above, e.g., Section 5.2 of NOI.PH Graphs 2.

[2]you could also use `c++20` or `c++23`