Minimal Spanning Trees and Greedy Algorithms

Suhas Arehalli

COMP221 - Spring 2024

1 Weighted Graphs and Minimal Spanning Trees

- Here we have a new kind of graph problem!
- Suppose we have a *weighted* undirected graph, which we will formally represent as an unweighted graph $G = (V, E)$ of vertices and edges with a *weight* or *cost function* $c : E \to \mathbb{R}$ that maps each edge $e \in E$ to a real-valued weight $c(e) \in \mathbb{R}$.
  - Weighted graphs can be formalized in different ways too! For instance, we can have edges be 3-tuples: $G = (V, E)$, with V the set of vertices and E the set of edges with $(v_1, v_2, c) \in E$ representing an edge from $v_1 \in V$ to $v_2 \in V$ with weight c.
  - We could also choose to call an unweighted graph $G = (V, E)$ and a weighted graph $G = (V, E, c)$ with $c : E \to \mathbb{R}$ still a cost function.
  - All of these formalizations are equivalent, so I will choose the one above because it's the one that's most intuitive to me. You can swap out the details below for the one that makes the most sense to you, but you will probably encounter a variety of formalizations in different courses.
- What we're interested in during this lecture are algorithms to construct a **Minimal Spanning Tree**, or an MST. An MST is...
  1. A *Sub-graph* of G, $M = (V', E')$, with $V' \subseteq V$ and $E' \subseteq E$.
  2. *Spanning*: $V' = V$ (i.e., the subgraph contains all of the vertices in G)
  3. A *Tree*: M is connected (there is a path between any two vertices in M) and acyclic (M contains no cycles).
  4. *Minimal*: We want to minimize the total cost of M, which we define as the sum of the weights of the edges we include: $\sum_{e \in E'} c(e)$. That is, for any **Spanning Tree** that fits the above criteria, we wnat the one that has the minimal cost.

2 Prim's Algorithm

2.1 The Core Idea & Greedy Algorithms

- Note that since we must have a spanning sub-graph at the end of the process, the set of vertices in M must be exactly V!

- Then we can reframe the problem of constructing an MST to the problem of selecting edges from E!
- In Prim's algorithm we will do this sequentially, choosing edges one by one.
  - At the end of the process, we will need to have a minimal spanning tree, and so in Prim's our idea is to choose edges that maintain the tree structure (i.e., acyclic and connected) of M as we build it.
  - Note that adding an edge with both vertices within M already would create a cycle, and that adding an edge with both vertices outside of M would make our graph disconnected!
  - Thus we add only edges where one vertex is already in V' and one vertex is in $V \setminus V'$.
  - We'll stop when we're spanning, and we can guarantee that we'll have a spanning tree!
  - We also want the spanning tree find to have minimal cost. The idea of Prim's is to do this is a **greedy** fashion: We will choose the *lowest cost* edge that keeps M a tree!
- We apply these principles and get Prim's algorithm (Alg. 1

Algorithm 1 Pseudocode for (an unoptimized) Prim's Algorithm

```
function PRIMS(G = (V, E), c, s)
    V' ← {s}
    E' ← ∅
    while |V'| < |V| do
        m ← ∞
        e ← NULL
        for (v', v) ∈ E s.t. v' ∈ V' and v ∈ V \ V' do
            if c((v', v)) < m then
                e ← (v', v)
                m ← c((v', v))
            end if
        end for
        (v', v) ← e
        V' ← V' ∪ {v}
        E ← E ∪ {e}
    end while
    return M = (V', E')
end function
```

- Prim's falls into the class of **Greedy Algorithms**.
  - Algorithmic problems often take the form of sequential decision problems
  - A greedy solution will always make the locally optimal choice
  - For many problems, a greedy solution is not optimal! Think about the robber problem from Activity 12!
  - However, when greedy solutions *are* optimal (i.e., where the locally optimal choices lead to globally optimal solutions), greedy solutions are typically very efficient!

– This means that proofs of correctness for greedy algorithms can be particularly tricky.
– We'll see a few variants of greedy algorithms and their proofs of correctness in the coming days.

2.2 Proof of Correctness

Let's start by considering the structure of the algorithm: We build a subtree of the MST edge-by-edge, and stop when the size of the subtree is the size of the MST. If we can prove that after each iteration of the while loop, $M = (V', E')$ is a subgraph of the MST, then once the while loop ends we will have a subgraph of the MST that spans G. Well, that must be the minimal spanning tree! So this gives us our loop invariant to prove!

Let's proceed by induction:

Loop Invariant: After each interation of the while loop, $M = (V', E')$ is a sub-tree of an MST of G. That is, for an MST $M^* = (V^*, E^*)$, that $V' \subseteq V^*$, $E' \subseteq E^*$, and M is a Tree (acyclic, connected).

To see that we always build a tree, we can simply observe that the only edges we consider adding have one edge in M and one edge ouside of M. This cannot create a cycle, since before adding the edge one vertex is not connected to any vertex in M, and M is connected afterward since after being added one of the two vertices was already within M, and thus connected to every over vertex in M. Thus the new vertex in the new edge must be connected since there is a path through the the vertex that was in M previously to every other vertex in M. It might be helpful to write out this part of the inductive proof more carefully as an exercise, but it won't be the focus of these notes!

Now let's focus on the tricky part: We must show that M begins and remains a sub-graph of M^*!

Base Case: We begin with $V' = \{s\} \subseteq V^* = V$ since the MST must be spanning, and $E' = \emptyset \subseteq E^*$ since the empty set is a subset of every set. We're done!

Inductive Step: Now, we get to the fun bit. Let's prove this by contradiction!

By our inductive hypothesis, we know that after the previous iteration of the loop, M is a subgraph of the MST (i.e., $M \subseteq M^*$). In the current iteration, all we do is at a new edge to E' and add the vertices at either end of that edge to V'. Since $V^* = V$ (it must span!), all we need to show is that the edge (call it e) that we add to E' is actually in E^* (i.e., that $e \in E^*$).

So, let's assume for contradiction that $e \notin E^*$!

Now, since $M*$ is an MST, it must be connected, and thus adding e to the MST will create a cycle! Let $e = (v_1, v_2)$. Since M^* spans G, $v_1, v_2 \in V^*$, and since M^* is a tree, it's connected, and thus there exists a path P through M^* that connects v_2 to v_1. Since $e \notin E^*$, this path crucially does not contain e! So if we take P and add e, $P \cup \{e\}$, we have a path that begins at v_2 and ends at v_2 – a cycle!

Because there's a cycle, and we know that the cycle contains e (an edge with one vertex in M and one vertex outside of M), we know that there exists an edge e' in that cycle that also has one vertex in M and one vertex outside of M. Why? because since e is in the cycle, there exists a path not containing e from v_1 to v_2. Since $v_1 \in M$ and $v_2 \notin M$, that path must, at some point, leave M, there must be some edge that has it's first vertex in M and it's second outside![1]

[1]This is another fairly simple statement to prove from out definition of a path. Since the second vertex of an edge must match the first vertex of the next edge (consecutive edges in a path must be incident to each other!), we can argue by induction over the length of the path that a path starting in M and having no such edge must end in M!

Now, we ask what happens when we remove e'? That is, we construct $E'^* = E^* \cup \{e\} \setminus \{e'\}$, swapping e' for e? First, I'm going to claim $M'^* = (V, E'^*)$ is a spanning tree! It obviously spans (I've defined it to!), but why is it a tree? We broke the cycle by removing e' (you should prove as an exercise that if M'^* has a cycle, M^* must have a cycle, which is a contradiction!), but how do we know it's still connected?

Well, M^* is connected, and the only paths in M^* that don't exist in M'^* are those that used the edge $e' = (w_1, w_2)$. Thus the path from a vertex s to t must have the form $P = P_1 \cup e' \cup P_2$, where P_1 is a path from s to w_1 and P_2 is a path from w_2 to t. Now we can construct a path from s to t using e instead of e' in M'^*! Because there was a cycle containing both e and e', we know that there is a path, P_3, not containing e' that connects w_1 to w_2 in $M \cup \{e\}$ (and therefore in M'^*), and thus that for any s and t only connected through e', we can construct the path $P_1 \cup P_3 \cup P_2 \subseteq M'^*$ to show that they are still connected!

So let's review: we assumed for contradiction that $e \notin E^*$. Then we constructed $M'^* = M^* \cup \{e\} \setminus \{e'\}$ that swapped the MST's chosen edge out of M, e', out for e, and showed that it is a spanning tree (See Fig. 1 for a visual of a particular case of this scenario). Now, let's look at the cost of $M'*$! By definition, that cost is

$$\begin{aligned} c(M'^*) &= \sum_{f \in M^* \cup \{e\} \setminus \{e'\}} c(f) \\ &= \sum_{f \in M^*} +c(e) - c(e') \\ &= c(M^*) + c(e) - c(e') \end{aligned}$$

Now let's consider one last fact: Prim's algorithm chose e out of all edges considered in the for-loop. This means that e was the minimum cost edge out of all edges that for loop considered, which is all edges in E such that one edge is in V' and the other is in $V \setminus V'$. Note that we found e' specifically so it fits that bill exactly. This means that Prim's chose e over e', meaning that $c(e) \leq c(e')$ (as per our conditions for selecting one edge over another in the for-loop!). Rearranging that, we get that $c(e) - c(e') \leq 0$, which means that $C(M'^*) \leq C(M^*)$. Two cases!

1. $c(M'^*) < c(M^*)$: Contradiction, since M^* was assumed to be a MST! Thus, by contradiction, $e \in M^*$, and we're done!

2. $c(M'^*) = c(M^*)$: Here, we're in a rare case where there are multiple MSTs! This means that M'^* is a different MST, which M is a sub-tree of! Thus, we still have our loop invariant upheld, and we're still done!

And so we complete the proof of out loop invariant. Then we simply observe that once the while loop complete, $M = (V', E')$ must span, is a tree, and is a sub-tree of the minimum spanning tree, and thus must actually be the minimum spanning tree. To be more formal about it, consider that every spanning tree must have exactly $|V| - 1$ edges, and that being a sub-tree means that $E' \subseteq E^*$. Since the two sets are finite, have the same size, and one is a subset of the other, they must be equal, and if the sets of edges are equal, the spanning trees must be equal (both span, so they both contain all of the vertices of G!).

Why? because to construct a path of length k, we need to add one more edge that has it's first vertex as the last vertex in the path of length $k-1$. If that vertex is in M (as our IH will tell us!), the second vertex must also be in

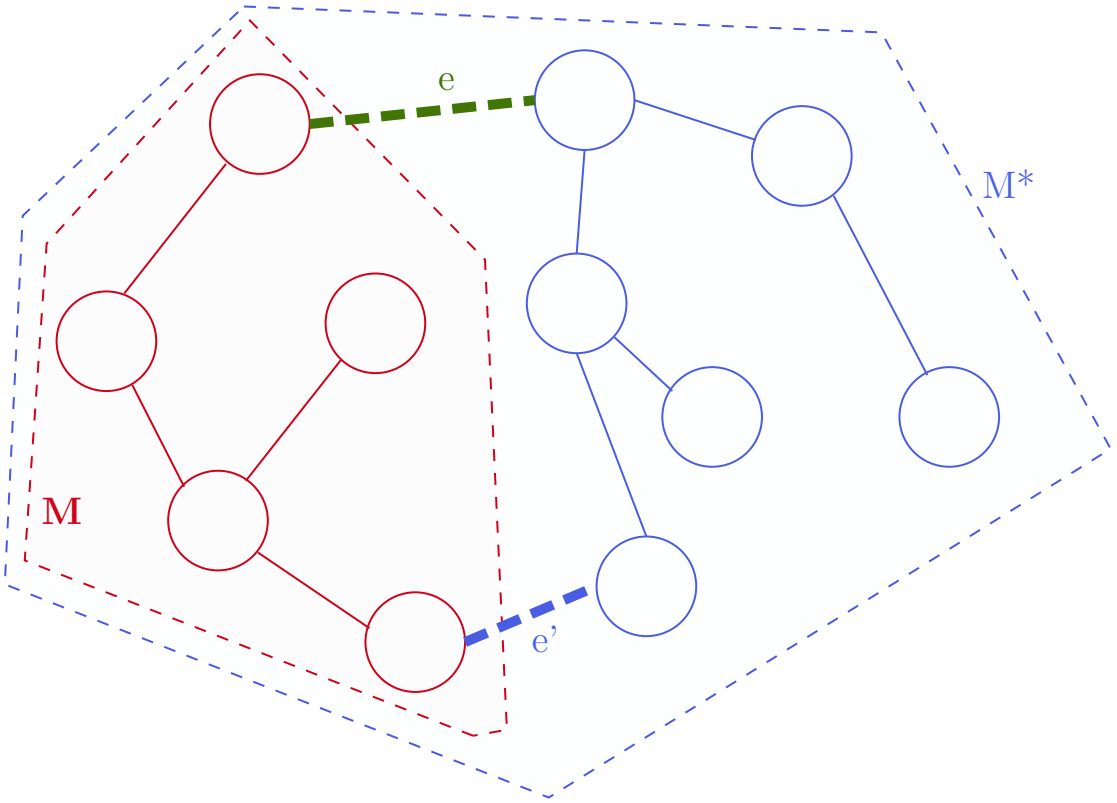

Figure 1: An illustration of the exchange argument at the core of the proof of correctness for Prim's Algorithm. We are constructing the tree in red, M, which so far is a subgraph of the MST M^*, and have chosen to select the edge e to add. we suppose for contradiction that the MST actually selects edge e' that connects M to the rest of M^*.

2.3 Runtime Analysis

- As written, our pseudocode for Prim's Algorithm in Alg. 1 runs in quadratic time.
  - The inner for-loop iterates (at worst) over each edge in the graph, and thus runs in $\Theta(|E|)$ time.
  - The outer while-loop will run until $|V'| = |V|$, which will take $|V| - 1 \in \Theta(|V|)$ steps.
    - Observe that we start with $|V'| = 1$
    - Then observe that we add a new vertex every iteration! When updating at the end of the iteration, we write that $V' \leftarrow V' \cup \{v\}$ when adding the edge (v', v). since $v' \in V'$ already. Then, since we require that $v \notin V'$, we know we're adding exactly one element each time!
    - This means that the algorithm must run $|V| - 1$ times before $|V'| = |V|$!
  - Updating within the while-loop should be dominated by the inner-for loop's time complexity (think about implementations of set operations using HashSets or TreeSets!)
  - Thus the outer-loop will take $\Theta(|V||E|)$ total, and this will dominate the runtime of the algorithm!
  - However, this implementation is fairly inefficient. We are essentially doing a linear search for the lowest cost edge every iteration!
    - We can optimize Prim's further with data structure driven design! Replace the linear search with a priority queue!
    - If we implement the graph as an AdjacencyList, we can look up the edges incident to a vertex in constant time.
    - Add edges to a priority queue when they become incident to at least one vertex in M. This ensures that edge in the priority queue is incident to a vertex in M!
    - At worst, each edge is added to the queue twice (once at each vertex in the edge), and thus we have $\Theta(|E|)$ total enqueues into the priority queue, each of which takes at worst $\Theta(\log|E|)$ time using a heap-based priority queue.
    - At worst, each of these inserts is dequeued and is tested for whether an edge would create a cycle. Each of these also takes $\Theta(\log|E|)$ time.
    - Since those enqueues and dequeues dominate time complexity within the while loop (everything else is constant time, and happens at most $\Theta(|E|)$ times), our total time complexity is dominated by this $\Theta(|E| \log|E|)$ time complexity
    - We could optimize this further (consider a priority that stores vertices ordered by the cost of an edge to connect it to M, or using fancier data structures (Fibonacci heaps!). This can get you to something like $O(|E| + |V| \log|V|)$ time.

M, and so the path of length k must also end in M! These are the kinds of arguments the first part of the course would have us show rigorously, but are now going to be elided so we can focus on the trickier, but more interesting graph/greedy/contradiction/etc. specific arguments!

3 Kruskal's Algorithm

3.1 The Core Idea

- Like Prim's, we want to sequentially select edges from E such that we will eventually build an MST.
- Prim's is committed to having the intermediate representations of the MST be *trees* - we select edges such that the graph we've build is *connected* and *acyclic*.
- Kruskal's weakens that commitment: It's not a problem to have the graph temporarily be disconnected, since if we eventually converge at an MST, we will connect all of the connected components we temporarily build!
- This means our intermediate representations are simply acyclic undirected graphs. This is equivalent to saying that our graph is constrained such that each connected component is a tree! We call these **Forests**!
- Putting this idea together, we can arrive at an algorithm that looks something like this Alg. 2

Algorithm 2 An unoptimized Kruskal's algorithm.

```
function KRUSKAL(G = (V, E))
    E' ← ∅
    while |E'| < |V| − 1 do
        c_min ← ∞
        e_min ← None
        for e ∈ E do
            if c(e) < c_min and M' = (V, E' ∪ {e}) is acyclic then
                c_min ← c(e)
                e_min ← e
            end if
        end for
        E' ← E' ∪ {e}
    end while
    return M = (V, E')
end function
```

- We'll leave correctness of Kruskal's as an exercise to the reader — it should look a lot like Prim's!
  - Show that the tree you construct is a spanning tree, and construct an exchange argument with a hypothetical different MST you construct for contradiction!

4 Runtime Analysis

Before we analyze the runtime, we should spend some time applying some simple optimizations, as our current implementation is fairly bad! This also let's us review some earlier design principles!

- A simple optimization to the algorithm has us do a little sorting as preprocessing: If we sort the edges by cost first, we can visit them in order of increasing cost and don't have to find any minimums! This gets us to Alg. **??**. Note that we can actually stop the loop early once we've added $|V| - 1$ edges, but asymptotically this algorithm will be just as fast, so it's fine for our purposes!

Algorithm 3 An slightly more optimized Kruskal's algorithm.

```
function KRUSKAL(G = (V, E))
    E' ← ∅
    Let L be a List
    L ← SORT(E)
    for e ∈ L do
        if M' = (V, E' ∪ {e}) is acyclic then
            E' ← E' ∪ {e}
        end if
    end for
    return M = (V, E')
end function
```

- As presented, both versions of Kruskal's can't be analyzed yet: We need to have an implementation that tells us how to determine whether $M' = (V, E' \cup \{e\})$ is acyclic or not!
  - One way to do this is to check whether the two vertices in the new edge e are already connected in $M = (V, E)$!
  - A naive implementation would use a straightforward test of connectivity like a BFS or DFS, which would mean that we get a worst-case runtime of $\Theta(|V| + |E|)$ for each connectivity check in the inner for-loop, which would give us a total time complexity of $\Theta(|E|(|V| + |E|)) = \Theta(|E|^2 + |E||V|)$.
  - Note that the loop dominates the cost of sorting, which would be $|E| \log |E|$!
  - But we can do better if we find a more efficient data structure to keep track of connectivity.

4.1 Union-Find

- We need a data structure that efficiently implements two operations

FIND((v) Map a vertex to some value that represents the connected component that it's a part of. We'll assume that this value is the label of some representative vertex in each connected component.

UNION(v_1, v_2) An operation that updates the data structure to indicate that an edge was added that merges two connected components.

- A HashMap that maps vertices to a representative vertex label would be very good at FIND — $O(1)$ amortized look-ups! But it would be fairly bad at union ($\Theta(n)$ to go through the entire HashTable to update the connected component labels!

- Idea: Build a forest
  - Each CC/tree in M is a tree in the Union-Find DS.
  - The representative vertex for each CC is the root of it's tree. This means that the time complexity of FIND is proportional to the height of each tree.
  - UNIONs can be done by finding the root of each CC the two vertices belong to and connecting the trees. Merging trees can be done in constant time, and finding the root is equivalent to running FIND.
  - This means that if we figure out a way to make the trees in the Union-Find forest balanced, both operations would be $\Theta(\log n)$!
- Remember that the forest built by Kruskal's is **not** the same as the forest in the Union-Find data structure! Kruskal's must represent the original graph's edge structure, but the forest in the Union-Find DS can have whatever structure makes finding CCs are efficient as possible!
- Now FIND is relatively straightforward to implement: We simply find the root of the tree in the Union-Find Forest (Alg. 4

Algorithm 4 Find in a Union-Find data-structure. F is the forest within the Union-Find.

```
function FIND(F, v)
    w ← v
    while w has a parent in F do
        w ← parent(F, w)
    end while
    return w
end function
```

- For UNION, we need to be careful about how we merge trees! We want to minimize the height of the resulting tree, so our strategy should be to make the root of the shorter tree a child of the root.
  - Let h_1 be the height of taller tree T_1, and h_2 the height of the shorter tree T_2. If $h_2 < h_1$, then making T_2 a child of T_1 will not increase the height, because the depth of every vertex in T_2 will increase by exactly 1, which can't make any depths greater than $h_2 + 1 \leq h_1$.
  - IF $h_1 = h_2$, then the height of the merged tree will be exactly $h_1 + 1$
  - Using this, we can show by induction that any sequences of merges of this sort will result in a balanced tree (i.e., logarithmic in height with respect to the number of vertices in the tree).
- That gives us the follow UNION algorithm

4.2 Improved Kruskal's

- With Union-Find, we can build a more optimized version of Kruskal's, shown in Alg. 6.

Algorithm 5 A Union implementation in a Union-Find data structure. F is the forest in the Union-Find data structure, and directed edges indicate parenthood.

```
function UNION(F = (V, E), v_1, v_2)
    r_1 ← FIND(F, v_1)
    r_2 ← FIND(F, v_2)
    h_1 ← HEIGHT(F, r_1)
    h_2 ← HEIGHT(F, r_2)
    if h_1 > h_2 then
        E ← E ∪ {(r_2, r_1)}
        HEIGHT(r_1) ← max(r_1, r_2 + 1)
    else
        E ← E ∪ {(r_1, r_2)}
        HEIGHT(r_2) ← max(r_2, r_1 + 1)
    end if
end function
```

- The runtime of this algorithm can be determined as follows:
  - Each iteration of the for-loop takes $\Theta(\log|V|)$ time to check for connectivity. If we add the edge, that only takes an additional $\Theta(\log|V|)$ time, which means it will always take logarithmic time.
  - We have to do this once for each edge, so the for-loop takes $\Theta(|E|\log|V|)$ time.
  - Pre-processing is dominated by sorting, which takes $\Theta(|E|\log|E|)$ time.
  - This means our total time complexity is $\Theta(|E|(\log|E| + \log|V|))$ time.
  - Remember that keeping track of both the number of edges and vertices in time complexities helps us differentiate performance on sparse (i.e., $|E|$ is small relative to $|V|$) vs. dense (i.e. $|E|$ is large relative to $|V|$, nearing $|V|^2$).

Algorithm 6 An more optimized Kruskal's algorithm using a Union-Find data structure.

```
function KRUSKAL(G = (V, E))
    E' ← ∅
    Let L be a List
    Let F = (V, ∅) be a forest.
    L ← SORT(E)
    for (v, w) ∈ L do
        if FIND(F, v) == FIND(F, w) then
            E' ← E' ∪ {e}
            UNION(F, v, w)
        end if
    end for
    return M = (V, E')
end function
```