1 Greedy Algorithms

Suppose we want to solve a problem, and we're able to come up with some recursive formulation of the problem that would give us a nice dynamic programming algorithm. But then, upon further inspection, we notice that any optimal solution only depends on looking up the optimal solution to one other subproblem. A greedy algorithm is an algorithm which exploits such a structure, ignoring other possible choices. Greedy algorithms can be seen as a refinement of dynamic programming; in order to prove that a greedy algorithm is correct, we must prove that to compute an entry in our table, it is sufficient to consider at most one other table entry; that is, at each point in the algorithm, we can make a "greedy", locally-optimal choice, and guarantee that a globally-optimal solution still exists. Instead of considering multiple choices to solve a subproblem, greedy algorithms only consider a single subproblem, so they run extremely quickly – generally, linear or close-to-linear in the problem size.

Unfortunately, greedy algorithms do not always give the optimal solution, but they frequently give good (approximate) solutions. To give a correct greedy algorithm one must first identify optimal substructure (as in dynamic programming), and then argue that at each step, you only need to consider one subproblem. That is, even though there may be many possible subproblems to recurse on, given our selection of subproblem, there is always optimal solution that contains the optimal solution to the selected subproblem.

1.1 Activity Selection Problem

One problem, which has a very nice (correct) greedy algorithm, is the Activity Selection Problem. In this problem, we have a number of activities. Your goal is to choose a subset of the activies to participate in. Each activity has a start time and end time, and you can't participate in multiple activities at once. Thus, given n activities $a_1, a_2, ..., a_n$ where a_i has start time s_i, finish time f_i, we want to find a maximum set of non-conflicting activities.

The activity selection problem has many applications, most notably in scheduling jobs to run on a single machine.

1.1.1 Optimal Substructure

Let's start by considering a subset of the activities. In particular, we'll be interested in considering the set of activities $S_{i,j}$ the start after activity a_i finishes and end before activity a_j starts. That is, $S_{i,j} = \{a_k | f_i \leq s_k, f_k \leq s_j\}$. We can participate in these activities between a_i and a_j. Let $A_{i,j}$ be a maximum subset of non-conflicting activities from the subset $S_{i,j}$. Suppose some $a_k \in A_{i,j}$, then we can break down the optimal subsolution $A_{i,j}$ as follows

$$|A_{i,j}| = 1 + |A_{i,k}| + |A_{k,j}|$$

where $A_{i,k}$ is the best set for $S_{i,k}$ (before a_k), and $A_{k,j}$ is the best set for after a_k. Another way of interpreting this expression is to say "once we place a_k in our optimal set, we can only consider optimal solutions to subproblems that do not conflict with a_k."

Thus, we can immediately come up with a recurrence that allows us to come up with a dynamic programming algorithm to solve the problem.

$$|A_{i,j}| = \max_{a_k \in S_{i,j}} 1 + |A_{i,k}| + |A_{k,j}|$$

This problem requires us to fill in a table of size n^2, so the dynamic programming algorithm will run in $\Omega(n^2)$ time. The actual runtime is $O(n^3)$ since filling in a single entry might take $O(n)$ time.

But we can do better! We will show that we only need to consider the a_k with the smallest finishing time, which immediately allows us to search for the optimal activity selection in linear time.

Claim 1. *For each $S_{i,j}$, there is an optimal solution $A_{i,j}$ containing $a_k \in S_{i,j}$ of minimum finishing time f_k.*

Note that if the claim is true, when f_k is minimum, then $A_{i,k}$ is empty, as no activities can finish before a_k; thus, our optimal solution only depends on one other subproblem $A_{k,j}$ (giving us a linear time algorithm).

Here, we prove the claim.

Proof. Let a_k be the activity of minimum finishing time in $S_{i,j}$. Let $A_{i,j}$ be some maximum set of non-conflicting activities. Consider $A'_{i,j} = A_{i,j} \setminus \{a_l\} \cup \{a_k\}$ where a_l is the activity of minimum finishing time in $A_{i,j}$. It's clear that $|A'_{i,j}| = |A_{i,j}|$. We need to show that $A'_{i,j}$ does not have conflicting activities. We know $a_l \in A_{i,j} \subset S_{i,j}$. This implies $f_l \geq f_k$, since a_k has the min finishing time in $S_{i,j}$.

All $a_t \in A_{i,j} \setminus \{a_l\}$ don't conflict with a_l, which means that $s_t \geq f_l$, which means that $s_t \geq f_k$, so this means that no activity in $A_{i,j} \setminus \{a_l\}$ can conflict with a_k. Thus, $A'_{i,j}$ is an optimal solution. □

Due to the above claim, the expression for $A_{i,j}$ from before simplifies to the following expression in terms of $a_k \subseteq S_{i,j}$, the activity with minimum finishing time f_k.

$$|A_{i,j}| = 1 + |A_{k,j}|$$
$$A_{i,j} = A_{k,j} \cup \{a_k\}$$

Algorithm Greedy-AS assumes that the activities are presorted in nondecreasing order of their finishing time, so that if $i < j$, $f_i \leq f_j$.

Algorithm 1: Greedy-AS(a)

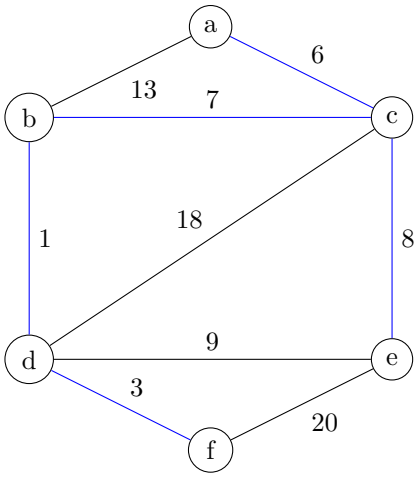
```
A ← {a_1} // activity of min f_i
k ← 1
for m = 2 → n do
    if s_m ≥ f_k then
        //a_m starts after last acitivity in A
        A ← A ∪ {a_m}
        k ← m
return A
```

By the above claim, this algorithm will produce a legal, optimal solution via a greedy selection of activities. The algorithm does a single pass over the activities, and thus only requires $O(n)$ time – a dramatic improvement from the trivial dynamic programming solution. If the algorithm also needed to sort the activities by f_i, then its runtime would be $O(n \log n)$ which is still better than the original dynamic programming solution.

2 Minimum Spanning Trees

The minimum spanning tree problem is structured as follows:
Input: $G = (V, E)$ undirected, w: $E \to \mathbb{Z}$
Output: A tree connecting all of V with minimum total weight

Example Below is an example of the minimum spanning tree of a graph. In the example, the edges forming the minimum spanning tree are colored blue while edges that are not part of the minimum spanning tree are colored black.



3 Finding Minimum Spanning Trees

Algorithms for finding Minimum Spanning Trees tend to be greedy algorithms. These algorithms maintain and build upon some subset of the edges of the graph $A \subseteq E$ with the invariant that A is always a subset of a Minimum Spanning Tree. In each step, edges will be greedily added to A such that A will continue to be a subset of a Minimum Spanning Tree. When the algorithm terminates, A will contain a set of edges that form a Minimum Spanning Tree of the graph.

The **cut property** of the MST problem is defined as follows:

Theorem 3.1. *Let $A \subseteq E$ be a subset of some MST, let $S \subseteq V$ be a subset such that there is no edge in A connecting S to $V \backslash S$, and let (u, v) be the edge in G with minimum weight such that $u \in S$, $v \notin S$, then $A \cup \{(u, v)\}$ is a subset of some MST.*

Proof of cut property. Let T be some MST containing A. Since A has no edges from S to $V \backslash S$ but T is spanning, T has some edge (x, y) with $x \in S$ and $y \notin S$. If (x, y) is removed from T, two subtrees are obtained which we can label T_x and T_y. Without loss of generality we assume that $u \in T_x$ and $v \in T_y$. Adding (u, v) results in the tree $T' = T \backslash \{(x, y)\} \cup \{(u, v)\}$ which is still a spanning tree. weight(T') = weight(T) - weight((x, y)) + weight((u, v)). Since weight((u, v)) $\leq$weight((x, y)) we have that weight(T') $\leq$ weight(T) which consequently means that T' is an MST. □

Next, we will present three different algorithms for finding an MST. Each algorithm will be greedy and correctness will follow by maintaining the cut property. As we've seen before, there will be tradeoffs between running each algorithm, so it's good to understand how each of them works.

3.1 Borůvka's Algorithm

Borůvka's Algorithm only works for graphs with distinct edge weights. At a high level, the set A maintained by the algorithm can be thought of as a set of disjoint trees. In each step, each tree in A picks the edge with minimum weight leading out of it and merges with the tree at the other endpoint of this selected minimum edge. This process is repeated until only one tree remains, which is then the returned MST.

Before we delve further into the exact functionality of Borůvka's Algorithm, we define two terms: supernodes and superedges. A supernode is defined to be a collection node that contains multiple nodes within it. A superedge (t, t') exists between supernodes t and t' if $\exists x \in t, y \in t'$ such that $(x, y) \in E$. If multiple such edges exist, (x, y) is selected to be the edge with minimum weight that connects a node in t and a node in t'.

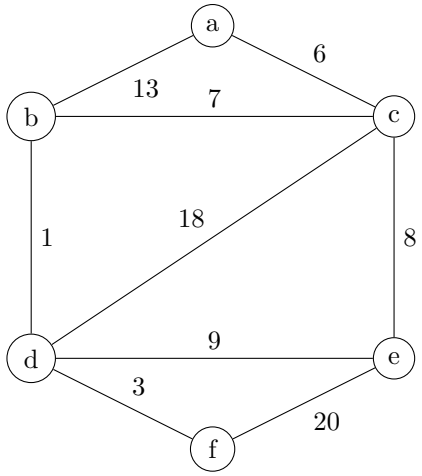
In Borůvka's Algorithm, every node is first initialized to be a supernode and every edge is initialized as a superedge. In each step of the algorithm, each supernode t locates its minimum weighted superedge (t, t').

t is then merged together with t' to create a new supernode, and the real-edge corresponding to the selected superedge (t, t') is added to the MST.

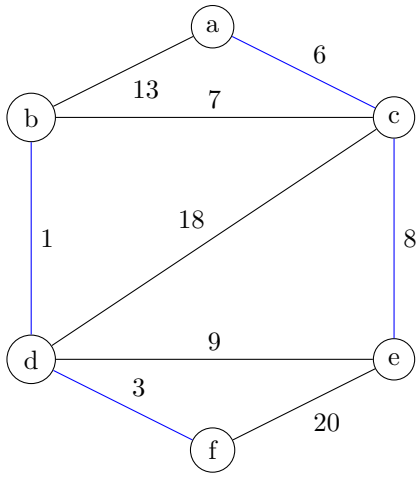
Algorithm 2: Borůvka($G = (V, E)$)

```
T ← empty set
foreach v ∈ V do
    Create supernode t containing v;
    T ← T ∪ {t}
foreach (u, v) ∈ E do
    Create superedge (t, t') where u ∈ t and v ∈ t';
while More than one supernode in T do
    X ← empty set
    foreach t ∈ T do
        find min weight superedge (t, t');
        X ← X ∪ {(t, t')};
    foreach (t, t') ∈ X do
        //to optimize runtime, iterate in DFS order on X Merge(t, t');
        A ← A ∪ real-edge((t, t'))
return A
```

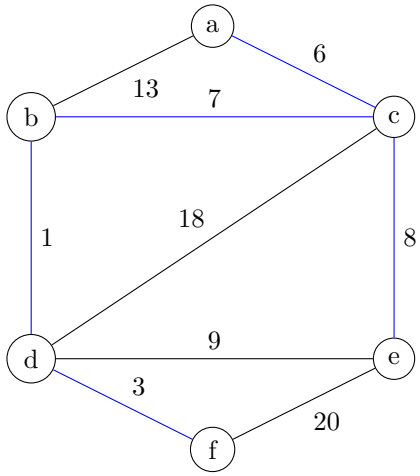
Example In this example we will go through the steps of applying Borůvka's Algorithm in order to find the MST of the following graph:



In the first iteration, we select the minimum superedge of each supernode. Since after initialization every node is a supernode and every edge is a superedge, this is equivalent to selecting the minimum weighted edge of each node.



Following the merge operations the remaining supernodes are $\{a, c, e\}$ and $\{b, d, f\}$. The superedge between these supernodes is defined by the real-edge (b, c). Thus we add edge (b, c) to the MST and following another merge operation we are left with only one supernode at which point the algorithm terminates.



Merging two supernodes t and t' can be done by creating a new supernode t'' and giving it all superedges of t and t' that are not superedges between t and t'. To save on runtime, the merge algorithm below picks the smaller degree node t' and adds all its edges to t without creating multiples. (In the lecture we didn't do this.)

Algorithm 3: Merge(t, t')

foreach *superedge* $(t', x), x \neq t$ **do**
  if (t, x) *not superedge* **then**
    add superedge (t, x);
  if (t, x) *already superedge* **then**
    weight$(t, x) \leftarrow \min\{\text{weight}(t', x), \text{weight}(t, x)\}$
  Delete t' from the graph and T

While we'll see that the other algorithms run faster than Borůvka's, note that because each supernode is built up separately, this algorithm is highly parallelizable – a property which is very desirable. This property makes Borůvka's algorithm particularly useful for analyzing very large networks, possibly in a distributed setting.

Correctness. Most of the correctness follows from the cut property: for any supernode/tree in A, the cut property dictates that picking the minimum weight edge out of the supernode is safe to be added to A. However, due to the parallelism of the Borůvka steps, we need to make sure that the supernodes remain trees, i.e. that A does not contain any cycles. We prove this below, and this completes the correctness proof of the algorithm.

Claim 2. *If the weights of the edges in a graph are all distinct, the set of edges A selected by Borůvka's Algorithm will not contain any cycles.*

Proof. Suppose Borůvka's Algorithm when applied to a graph whose edge weights are all distinct ends up selecting edges that form a cycle of k supernodes on supernodes $v_1, v_2, ..., v_k$ where supernodes with consecutive subscripts are adjacent to one another in the cycle. This means that at some step in Borůvka's Algorithm we will end up selecting superedges (v_1, v_2), (v_2, v_3), ..., (v_k, v_1). Since the weight of each superedge is distinct, there must be a maximum weighted superedge in this cycle. Call this maximum superedge (v_i, v_{i+1}). Since this is the largest weighted superedge in the cycle, we know that superedge (v_{i+1}, v_{i+2}) must have a smaller weight than superedge (v_i, v_{i+1}). Thus (v_{i+1}, v_{i+2}) will be the superedge selected by supernode v_{i+1} during this step of Borůvka's Algorithm. However, a similar argument can be applied to superedge (v_{i-1}, v_i) which must also have a smaller weight than (v_i, v_{i+1}) and consequently will be the superedge selected by Borůvka's Algorithm for supernode v_i at this step. But this leads to a contradiction since neither of the endpoints of (v_i, v_{i+1}) will end up selecting this superedge during Borůvka's Algorithm. Thus there cannot be any cycles in the set of edges selected by Borůvka's Algorithm if the weights of all the edges in the graph are distinct. □

Runtime. The runtime of the merge algorithm is: $O(\min\{\deg(t')\})$ where t' is the node merged into another node t.

In lecture we mentioned that we can run Borůvka's Algorithm so that the superedges out of each supernode are accessed at most a constant number of time in each iteration, so that each iteration runs in $O(m)$ time. Here we give the details on how to do this.

Consider the set X of edges to be merged in a particular iteration. We will do the merges of edges in X in a suitable order. First go through X and create out of it a graph in adjacency list representation. This can be done in $O(n + |X|)$ time where n is the current number of supernodes. Now we can run DFS on the subgraph induced from X. Whenever a superedge (t, t') of X is traversed by the DFS search, we merge t' into t. Since X is a forest (as we proved above), the DFS search will not encounter any backedges. Thus, any supernode that is an endpoint of a superedge in X will be used as t' at most once. Thus its superedges will be added to a node t at most once. To check whether a pair (t, x) is actually a superedge of t, we can store the superedges of a supernode in two formats: the adjacency list format (a list of the superedges L_t) and the adjacency vector format (an array N_t of length n, such that if (t, x) is a superedge, $N[t] = 1$).

Now, when t' is merged into t, for any superedge (t', x) out of t', one first checks if $N_t[x] = 1$. If so, then just the weight of (t, x) is modified. Otherwise, $N_t[x]$ is set to 1 and (t, x) is added to the end of L_t. In addition, $N_x[t]$ is set to 1 and (t, x) is added to the end of L_x. When x is accessed later in the algorithm, both (t, x) and (t', x) will be accessed when searching for the minimum edge weight. Then, (t', x) will be discarded since t' is no longer active, and we can think of this operation on (t', x) as being paid for already by the addition of (t, x) to x. (We omitted all these details in lecture to save time.) Thus each iteration of the algorithm can run in $O(m)$ time (while also paying for discarding future (t', x) superedges out of inactive supernodes t').

Another way to make the merge operation fast is to use a union-find data structure (see Kruskal's algorithm below).

Now we address the number of iterations. In each step of Borůvka's Algorithm, the minimum size supernode is merged with another supernode. The size of the minimum sized supernode will thus at least double after each step. It starts at 1 and ends at n. This means that the number of iterations in Borůvka's Algorithm is $\leq \log(n)$ which gives us the final runtime of Borůvka's Algorithm: $O(m \log n)$.

3.2 Prim's Algorithm

At a high level, the set A maintained by Prim's Algorithm is a single tree. In each step, the edge with minimum weight leading out of A and connecting to a node that has not yet been connected to A is selected and added to A. Once A connects every node in the graph, it is returned as an MST of the graph.

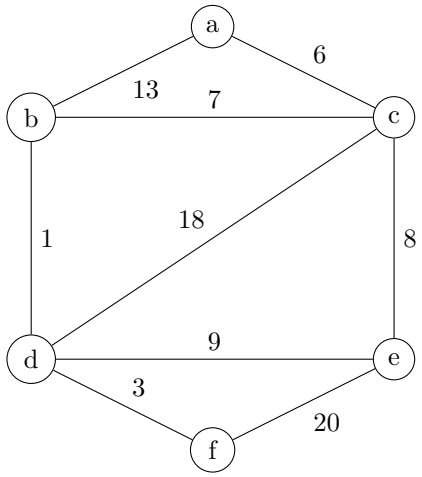
Prim's Algorithm is similar to Dijkstra's Algorithm in that estimates of the distance to each node are maintained and updated as the algorithm progresses. $V(A)$ is defined to be the vertices that edges in A are incident to. $d(v)$ $\forall v \in V$ is defined to be the minimum edge weight from a node of $V(A)$ to v.

Algorithm 4: Prim($G = (V, E)$)

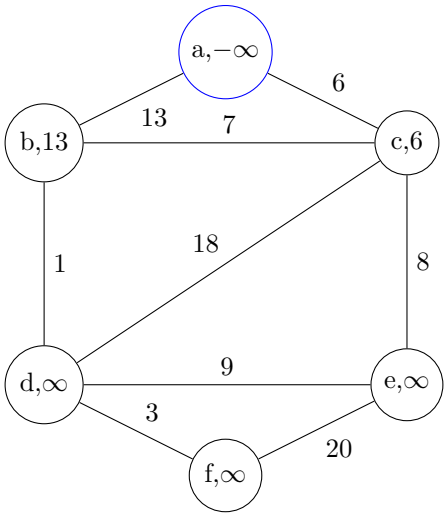
```
A ← empty set  F ← empty Fibonacci heap foreach v ∈ V do
    d(v) ← ∞;
    π(v) ←NIL;
    put v in F with key d(v);
pick s from V;
d(s) ← −∞;
while F is not empty do
    u ← F.extractmin;
    A ← A ∪ {(π(u), u)};
    foreach (u, v) ∈ E do
        if d(v) > weight(u, v) then
            d(v) ←weight(u, v);
            π(v) ← u;
return A
```

Prim's Algorithm is a direct modification of Dijkstra's Algorithm and has a runtime of $O(m + n \log n)$.

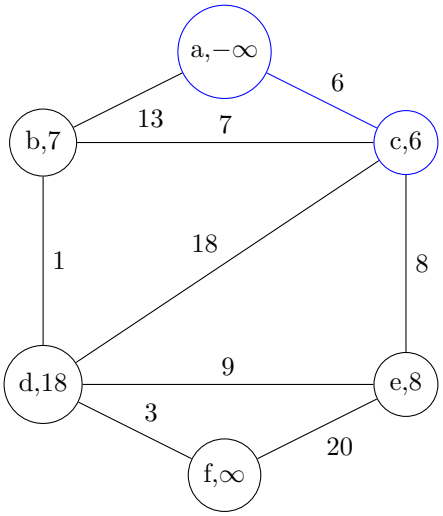
Example In this example we will run through the steps of applying Prim's Algorithm in order to identify an MST for the following graph:



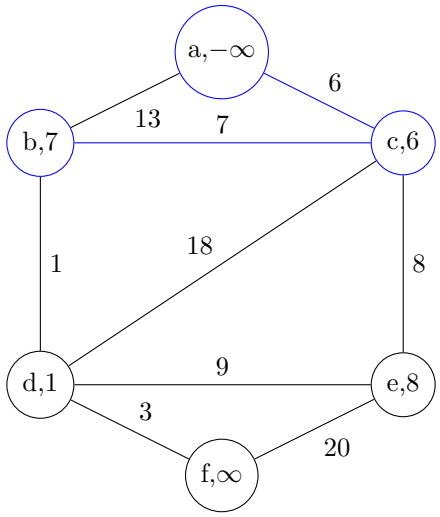
Suppose we select node a to be the source node, s. We then extract node a from the Fibonacci heap and set $d(b) = 13$, $d(c) = 6$ and $\pi(b) = a$, $\pi(c) = a$.



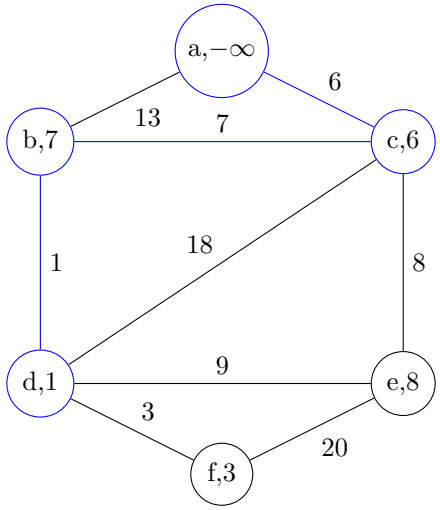
Since $d(c)$ is now the smallest value in the Fibonacci heap, we visit node c. Because $\pi(c) = a$ we add edge (a, c) to the set A. We then update the distance estimates and parent fields of nodes that have edges connecting to c. Thus we set $d(b) = 7$, $d(d) = 18$, $d(e) = 8$, and $\pi(b) = c$, $\pi(d) = c$, and $\pi(e) = c$.



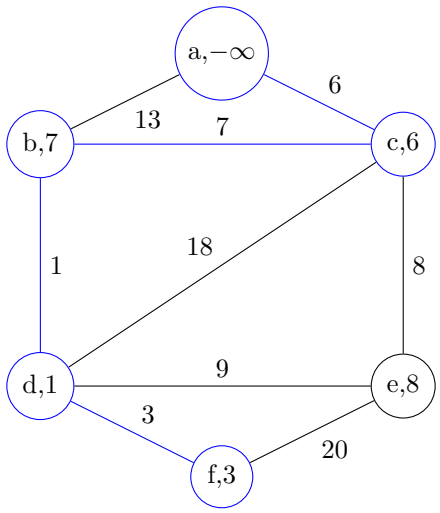
Node b is extracted next and we add edge (b, c) to A. We then set $d(d) = 1$ and $\pi(d) = b$.



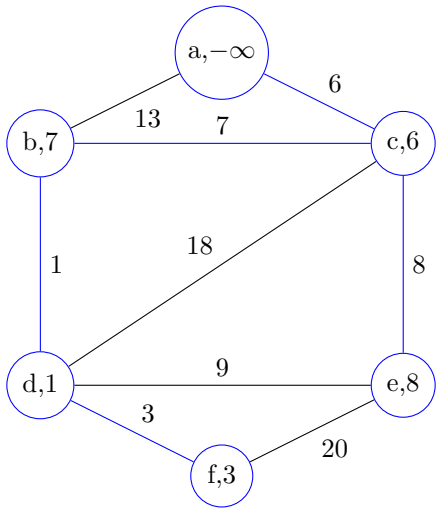
Node d is extracted next and we add edge (b, d) to A. We then set $d(f) = 3$ and $\pi(f) = d$.



Node f is extracted next and we add edge (d, f) to A. No nodes are updated after this step.



Finally node e is extracted and edge (c, e) is added to A. After this step the Fibonacci heap will be empty and Prim's Algorithm will return A.



3.3 Kruskal's Algorithm

At a high level, the set A maintained by Kruskal's Algorithm is a set of disjoint trees similar in a sense to Borůvka's Algorithm. However, Kruskal's Algorithm differs from Borůvka's Algorithm in its update step. During update step i, if the i-th smallest edge connects different trees, merge the two trees connected by this edge. The algorithm progresses until eventually only one tree remains at which point the set A represents an MST of the graph.

Kruskal's Algorithm utilizes the union-find data structure in order to handle the merging of the disjoint trees maintained by the algorithm. The union-find data structure supports disjoint sets with the following operations:

- **makeset(x):** creates a new set containing x provided that $x \notin$ any other sets
- **find(x):** returns the name of the set containing x
- **union(x, y):** merge the set containing x and the set containing y into one set

The algorithm itself can be structured as follows:

Algorithm 5: Kruskals($G = (V, E)$)

```
A ← empty set;
E' ← sort edges by weight in non decreasing order;
foreach v ∈ V do
    makeset(v);
foreach (u, v) ∈ E' do
    if find(u) ≠ find(v) then
        A ← A ∪ {(u, v)};
        union(u, v);
return A;
```

The correctness directly follows from the cut property.

The runtime of Kruskal's Algorithm is dependent on two factors: the time to sort the edges by weight and the runtime of the union-find data structure operations. While $\Omega(m \log n)$ time is required if we use

comparison sorting, in many cases, we may be able to sort the edges in linear time. (Recall, you showed in the homework that CountingSort can be used to sort the edges in $O(m)$ time if the weights are given by integers bounded by a polynomial in m. The corresponding algorithm is called Radix Sort.) In this case, the runtime is bounded by the runtime of the union-find operations and is given by $O(nT(\text{makeset})+mT(\text{find})+nT(\text{union}))$. The best known runtime for the union-find operations is amortized $O(\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function. Interestingly, the value of the inverse Ackermann is for all practical purposes tiny:

$$\alpha(n) \leq 4, \quad \forall n < \texttt{\# atoms in the universe}$$

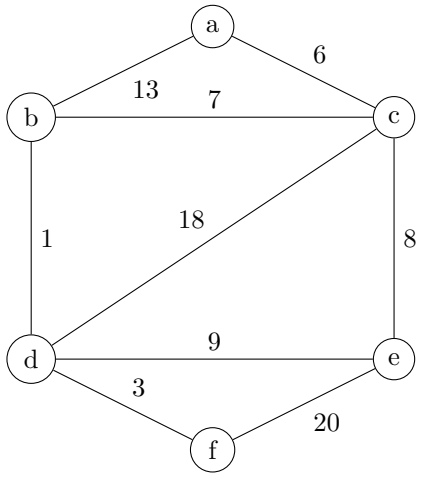
and thus for all practical purposes the union-find operations run in constant time. Thus, in many settings, the runtime of Kruskal's algorithm is nearly linear in the number of edges.

The actual definition of $\alpha(n)$ is $\alpha(n) = \min\{k \mid A(k) \geq n\}$, where $A(k)$ is the Ackermann function evaluated at k. $A(k)$ itself is defined via a more general Ackermann function, in particular, $A(k) = A_k(2)$.

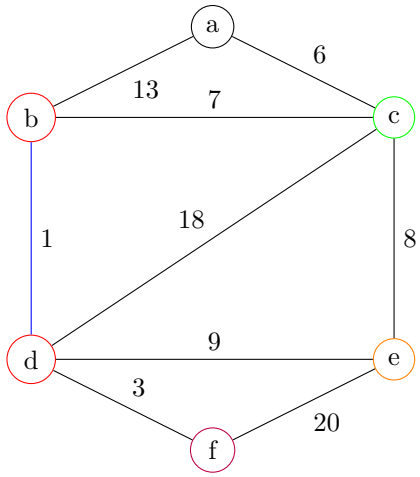
$A_k(x)$ is defined recursively.

- $A_0(x) = 1 + x$, so that $A_0(2) = 3$.
- $A_1(x) = 2x$ (A_0 iterated x times), so that $A_1(2) = 4$
- $A_2(x) = 2^x x$ (A_1 iterated x times), so that $A_2(2) = 8$
- $A_3(x) \geq 2^{2^{2^{\cdots}}}$, a tower of x 2s (A_2 iterated x times), so that $A_3(2) \geq 2^{11}$
- $A_4(x)$ is at least a tower of x towers of x 2s, and $A_4(2)$ is at least a tower of 2048 twos! This number is larger than the total number of atoms in the known universe, and also larger than the number of nanoseconds since the Big Bang. (Thus, $\alpha(n) \leq 4$ for all practical purposes.)

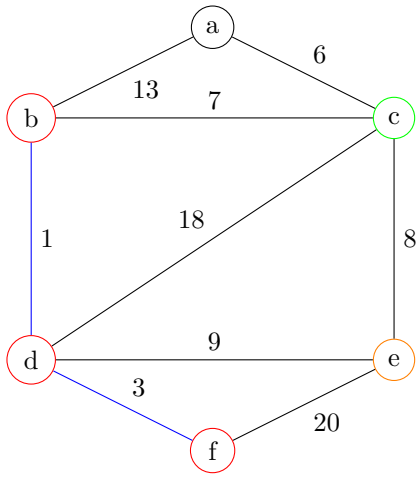
Example In this example we will run through the steps of applying Kruskal's Algorithm in order to identify an MST for the following graph:



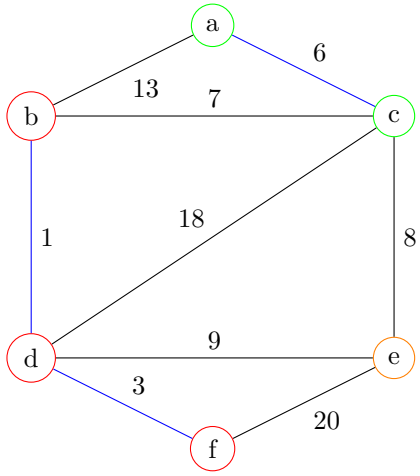
We begin by creating a new set for each node in the graph. We then being iterating over the edges in non decreasing order. The first edge we examine is (b, d). This edge connects nodes b and d which are currently not part of the same set. We thus include this edge in A and union the sets containing b and d.



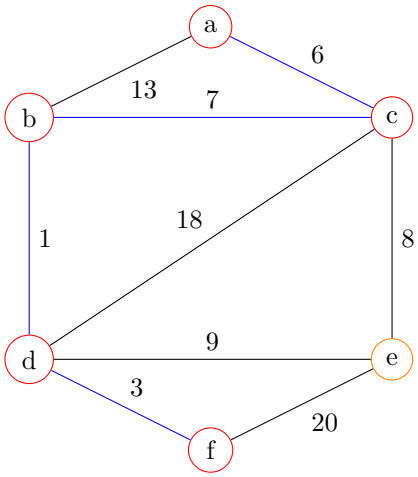
The next edge in our sorted order is edge (d, f). Since d and f are not part of the same set we include this edge in A and union the sets containing d and f.



The next edge in the sorted order is (a, c). We union the sets containing a and c and add edge (a, c) to A.



The next edge in the sorted order is (b, c). We union the sets containing b and c and add edge (b, c) to A.



The next edge in the sorted order is (c, e). We union the sets containing c and e and add edge (c, e) to A. At this point all nodes are contained in the same set and so no further edges are added to A.