CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 12

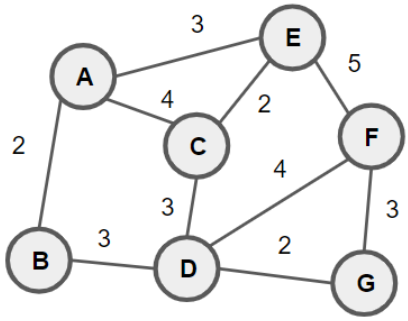
For: April 7–April 11

Goals:

- Priority Queue
- Union-Find review
- MST

Problem 1. MST Review

Problem 1.a.



Can you use the cycle and cut property of MST that we learnt in class to determine which edges must be in the MST?

Perform Prim's, Kruskal's, and Boruvka's (optional) MST algorithm on the graph above.

Solution:

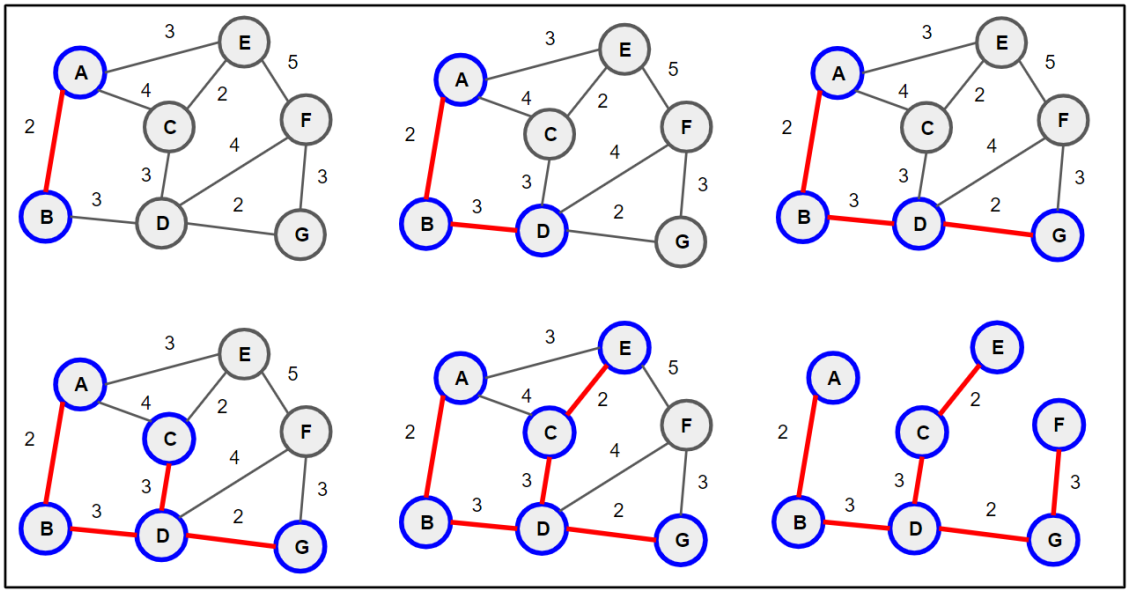


Figure 1: Prim's Algorithm

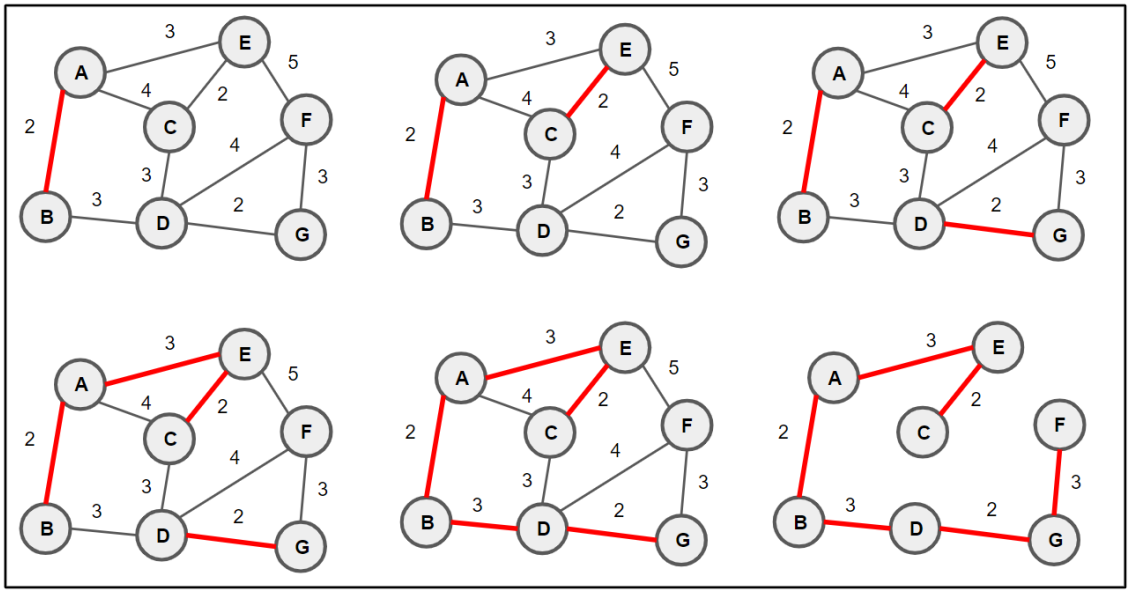


Figure 2: Kruskal's Algorithm

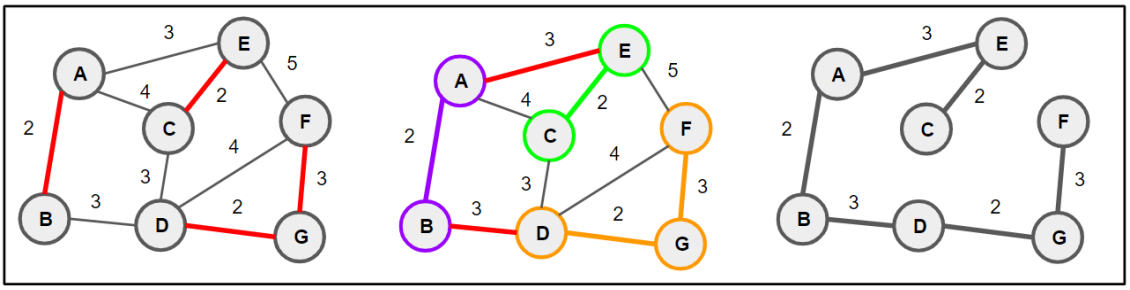


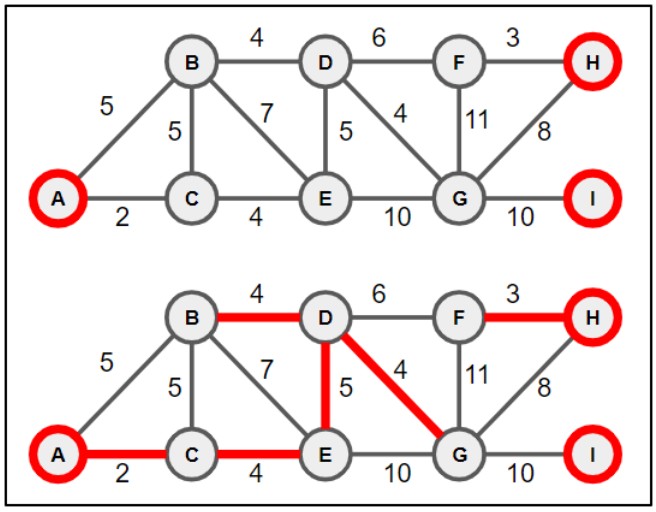
Figure 3: Borůvka's Algorithm

Problem 1.b. Henry The Hacker has some ideas for a faster MST algorithm! Recently, he read about *Fibonacci Heaps*. A Fibonacci heap is a priority queue that implements insert, delete, and decreaseKey in $O(1)$ amortized time, and implements extractMin in $O(\log n)$ amortized time, assuming n elements in the heap. If you run Prim's Algorithm on a graph of V nodes and E edges using a Fibonacci Heap, what is the running time?

Solution: In this case, each node is added into the priority queue at most once, which costs $O(V)$; each edge leads to one decreaseKey operation, which has cost $O(E)$; and each node is extracted once from the priority queue, which results in a cost of $O(V \log V)$. Hence, the total cost is $O(E + V \log V)$.

Problem 2. Power Plant

Given a network of power plants, houses and potential cables to be laid, along with its associated cost, find the cheapest way to connect every house to at least one power plant. The connections can be routed through other houses if required.



In the diagram above, the top graph shows the initial layout of the power plants (modelled as red nodes), houses (modelled as gray nodes) and cables (modelled as bidirectional edges with its associated cost). The highlighted edges shown in the bottom graph shows an optimal way to connect every house to at least one power plant. Come up with an algorithm to find the minimum required cost. You may assume that there exists at least one way to do so.

Solution: A way to solve this problem is to run Prim's algorithm with the priority queue initialized to contain all of the power plant nodes. This initial setup ensures that every node discovered later can be traced back to exactly one of the power plant nodes.

You can also run Kruskal's or Boruvka's algorithm with all the power plant nodes initialised to be in the same component.

Alternatively, you can insert an additional super node into the graph that is connected to each of the power plant nodes via edges of weight 0, then run any MST algorithm.

All of these approaches will run in $O(E \log V)$, where E is the number of potential cables and V is the sum of the number of power plants and houses.

Problem 3. (Horseplay)
(Relevant Kattis Problem: https://open.kattis.com/problems/bank)

There are m bales of hay that need to be bucked and n horses to buck them.

The i^{th} bale of hay has a weight of k_i kilograms.

The i^{th} horse has a strength s_i—the maximum weight, in kilograms, it can buck—and can be hired for d_i dollars.

Bucking hay is a very physically demanding task, so to avoid the risk of injury every horse can buck at most one bale of hay.

Determine, in $O(n \log n + m \log m)$, the minimum amount, in dollars, needed to buck all bales of hay. If it is not possible to buck them all, determine that as well.

Solution: Let's sort the bales of hay by non-increasing weight, and similarly sort the horses by non-increasing strength. Assume from this point that $k_1 \geq k_2 \geq \cdots \geq k_m$ and $s_1 \geq s_2 \geq \cdots \geq s_n$.

So here's the high level idea, consider the bales of hay in order of decreasing weights. For the very first one with weight k_1, figure out the initial set of horses that can buck it. This can be done by figuring out the largest index j for which $s_j \geq k_1$. Add these into a min-priority queue where the priority is the price. Then remove the cheapest horse.
Now, while there are bales of hay remaining (similar to as before):

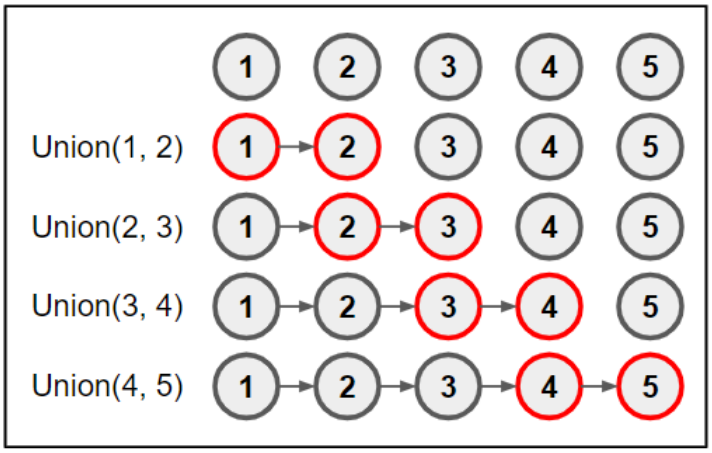
1. Consider the next bale of hay k_i, find the next largest index j of horses for which $s_j \geq k_i$.
2. Add these horses into the min-priority queue (priority based on the price).
3. Remove the cheapest horse from the priority queue.

If at any point, we had an empty priority queue but still at least one bale of hay to buck, then we can output IMPOSSIBLE.
Analysis: $O(n\log(n) + m\log(m))$ for sorting the hay and horses. $O(mlogm)$ for running at most m insertions and n extracts from a priority queue of size m.
But why is this algorithm correct? Here's the proof:

1. Assume that the set S of horses our algorithm picks does not cost the least (is not optimal).
2. There must have been an optimal set $T \neq S$ of horses such that we can buck all the hay **and it costs the least**.
3. Since $T \neq S$ and $|S| = |T| = n$, there must exist some horse $h \in S \wedge h \notin T$.
4. Our algorithm must have assigned this horse $h = (s, d)$ to buck some bale of hay k_i (for some i) for cost d.
5. On the other hand, from the set T, there must have been some other horse $h' = (s', d')$ to buck the same bale of hay k_i.
6. So both horses h and h' are such that they're both strong enough to buck the bale of hay k_i.
7. Now, by definition of our algorithm, when considering bale of hay k_i, both horses h and h' should be in the priority queue.
8. So the reason our algorithm chose horse h over h' was because $d < d'$.
9. Now let's consider the set of horses $T' = T \setminus \{h'\} \cup \{h\}$. All we have done is swap horse h' for horse h to buck the i^{th} bale of hay.
10. Notice that we can still buck all m bales of hay.
11. Furthermore, $cost(T) - d' + d = cost(T')$, where $cost(T)$ is the cost of using horses in set T, and T' is the cost of using horses in set T'.
12. But since $d < d'$, we know this means $d - d' < 0$, so $cost(T) > cost(T')$. So using the horses in T' costs less, but we had assumed that using the horses in set T cost the least. Contradiction!

Problem 4. (Union-Find Review)

Problem 4.a. What is the worst-case running time of the find operation in Union-Find with path compression (but no weighted union)?



Solution: The worst-case is $\Theta(n)$. You can construct a linear length tree quite easily by just doing Union(1,2), Union(2,3), Union(3,4), ..., Union(n-1, n) as shown above.

Problem 4.b. Here's another algorithm for Union-Find based on a linked list. Each set is represented by a linked list of objects, and each object is labelled (e.g., in a hash table) with a set identifier that identifies which set it is in. Also, keep track of the size of each set (e.g., using a hash table). Whenever two sets are merged, relabel the objects in the smaller set and merge the linked lists. What is the running time for performing m Union and Find operations, if there are initially n objects each in their own set?

More precisely, there is: (i) an array id where $id[j]$ is the set identifier for object j; (ii) an array $size$ where $size[k]$ is the size of the set with identifier k; (iii) an array $list$ where $list[k]$ is a linked list containing all the objects in set k.

```
Find(i, j):
    return (id[i] == id[j])

Union(i, j):
    if size[i] < size[j] then Union(j,i)
    else // size[i] >= size[j]
        k1 = id[i]
        k2 = id[j]
        for every item m in list[k2]:
            set id[m] = k1
        append list[k2] on the end of list[k1] and set list[k2] to null
        size[k1] = size[k1] + size[k2]
```

```
size[k2] = 0
```

Assume for the purpose of this problem that you can append one linked list on to another in $O(1)$ time. (How would you do that?)

Solution: Find operations obviously cost $O(1)$. For m union operations, the cost is $m \log n$. The only expensive part is relabelling the objects in `list[k2]`. And notice that, just like in Weighted Union, each time we union two sets, the size of the smaller set at least doubles. So each object can be relabelled at most $logn$ times (as we can double the size of a set at most $logn$ times). Note that since there are m union operations, the biggest set after those operations is of size $O(m)$, and as each object in that set was updated at most $logn$ times, the total cost is $m \log n$. Of course, notice that any one operation can be expensive (each union operation have different costs). For example, the last union operation might be combining two sets of size $m/2$ and hence have cost m, while the first union operation would have a cost of $O(1)$.
The appending of one linked list to the end of another is pretty easily done in $O(1)$ through manipulation of the head and tail pointers.

Problem 4.c. Imagine we have a set of n corporations, each of which has a (string) name. In order to make a good profit, each corporation has a set of jobs it needs to do, e.g., corporation j has tasks $T^j[1 \ldots m]$. (Each corporation has at most m tasks.) Each task has a priority, i.e., an integer, and tasks must be done in priority order: corporation j must complete higher priority tasks before lower priority tasks.

Since we live in a capitalist society, every so often corporations decide to merge. Whenever that happens, two corporations merge into a new (larger) corporation. Whenever that happens, their tasks merge as well.

Design a data structure that supports three operations:

- `getNextTask(name)` that returns the next task for the corporation with the specified name.
- `executeNextTask(name)` that returns the next task for the corporation with the specified name and removes it from the set of tasks that corporation does.
- `merge(name1, name2, name3)` that merges corporation with names `name1` and `name2` into a new corporation with `name3`.

Give an efficient algorithm for solving this problem.

Solution: This can be solved using the same technique as the previous one, but now instead of using linked lists, you can use a heap to implement a priority queue for each corporation. To get the next task, just use the usual heap operation of extract-max. To merge two corporations, take the smaller remaining set of tasks and add them all to the heap for the corporation with the larger set of tasks. (Use, e.g., a hash table to store a pointer to the proper heap for each corporation.) Each time corporations merge, each item in the smaller heap pays $\log nm$ to be inserted into the new heap. (There are at most nm tasks in total.) Using the same argument as before, each item only has to be copied into a new heap at most $\log n$ times, and so the total cost of operations is $O(nm(\log n)(\log nm)) = O(nm(\log^2(n) + \log(n)\log(m))$.