**CS2040S: Data Structures and Algorithms**

Discussion Group Problems for Week 12

*For: April 7–April 11*

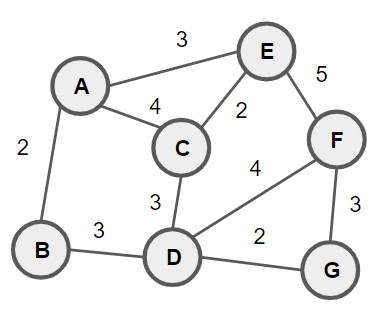
*Goals:*

* Priority Queue
* Union-Find review
* MST

**Problem 1. MST Review**

**Note to tutors:** 10 - 15 minutes.

**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?

**Ans:** E(A, B), E(D, G), E(G, F), E(C, E), E(B, D), E(C, E)

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

**Ans:**

Prim’s – E(A, B), E(A, E), E(E, C), E(C, D), E(D, G), E(G, F)

Kruskal’s – E(A, B), E(C, E), E(D, G), E(B, D), E(C, D), E(G, F)

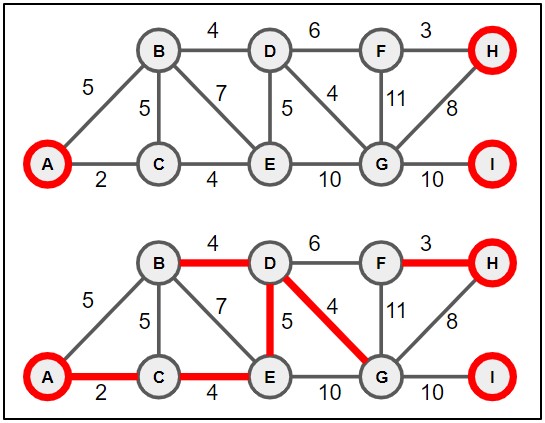
**Problem 1.b.** Henry The Hacker has some ideas for a faster MST algorithm! Recently, he read about *Fibonacci Heaps*. A Fibonacci heap is a priority queue that implements insert, delete, and decreaseKey in *O*(1) amortized time, and implements extractMin in *O*(log*n*) amortized time, assuming *n* 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?

**Ans:** Firstly, we insert all nodes into the priority queue with their priority being infinite except for the starting node which has a priority of 0. As there are *V* nodes, this would have a time complexity of O(*V*). Since we need to iterate through every node, there would be a total of *V* iterations. For each iteration, we perform an extractMin operation to retrieve the next node to add to our MST. This would have a time complexity of O(). From the extracted node, we retrieve its neighbours, and proceed to relax the edges connecting to their neighbour by using the decreaseKey operation to decrease the priority of the neighbour nodes. Since there are a total of *E* edges we can perform decreaseKey on, this would have a time complexity of O(*E*). We do this for every single node until we obtain our MST. Thus the total time complexity is O() = O().

**Problem 2. Power Plant**

**Note to tutors:** 10 - 15 minutes.

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.

**Ans:** We can use Kruskal’s Algorithm to solve this problem. We start by sorting all the different edges in ascending order. Originally, all the nodes (both power plants and houses) are not connected. Then, we can iterate through all the sorted edges and connect the two nodes together via the current iterated edge, only if their respective subtrees are not connected with one another or if the house node is currently not connected to a powerplant.

**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 *ki* kilograms.

The *i*th horse has a strength *si*—the maximum weight, in kilograms, it can buck—and can be hired for *di* 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.

**Ans:** If *n* < *m*, since every horse can buck at most one bale of hay, it would not be possible to buck all *m* bales of hay. Additionally, if the heaviest bale of hay is too heavy for the strongest horse, then it would also be impossible to buck all *m* bales of hay.

We can sort all *m* bales of hay by their weight (from heaviest to lightest) and all *n* horses by their strength (from the highest weight to the lowest weight). Then we can use a min-heap priority queue to keep track of the available horses’ cost. Initialise total\_cost and horse\_index to 0.

For each bale in the sorted bales array, add all horses with strength >= the current bale of hay to the min-heap. We increment the horse\_index for each horse added. Once we have added all the horses that are able to carry the current bale of hay, if the heap is empty, then it is not possible to buck all *m* bales of hay. Else, we simply extract the cheapest horse from the min-heap and add its cost to our total\_cost.

**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)?

**Ans:** O()

**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): // O(1)

return (id[i] == id[j])

Union(i, j): // O(n)

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?)

**Ans:** Each Find operation has a time complexity of O(1) whereas each Union operation has a time complexity of O(*n*). Thus, the running time for performing *m* Union and Find operations is at most O(*mn*).

We can append one linked list on to another in O(1) time by appending the head of one linked list to the tail of the other linked list.

**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 *Tj*[1*...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.

**Ans:** We need a priority queue that supports efficient merging. A suitable data structure for this would be Fibonacci heap, where it has a time complexity of O(1) for find-max, O() for extract-max and O(1) for merging. Additionally, we need a hash table to map the corporation’s name to their respective priority queue.

For getNextTask(name), we simply look up the corporation’s priority queue using the corporation name from the hash table, and perform a peek operation on the priority queue. This would have a time complexity of O(1).

For executeNextTask(name), likewise, we look up the corporation’s priority queue using their name from the hash table, and perform a extractMax operation to retrieve and remove the task with the highest priority from the priority queue. This would have a time complexity of O(1 + ) = O().

Lastly, for merge(name1, name2, name3), we can look up both priority queues for name1 and name2 and merge their priority queues together. We then add the mapping of corporation name3 to the newly merged priority queue into our hash table. This would have a time complexity of O(1).