CS2040S: Data Structures and Algorithms

Recitation Plan for Week March 30–April 3

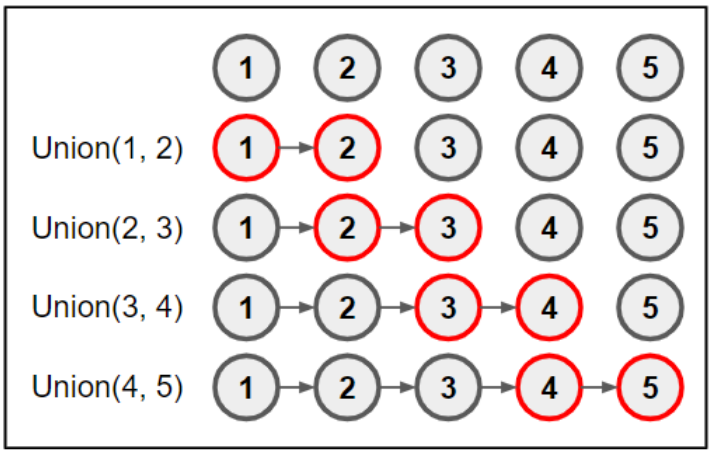
For: Week 11

Goals:

- Union-Find review
- Shortest paths
- Dijkstra's Algorithm

Problem 1. (Union-Find Review)

Problem 1.a. What is the worst-case running time of 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 1.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, re-label 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 j in list[k2]: set id[j] = k1
       postpend 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 postpend one linked list on to another in $O(1)$ time. (How would you do that?)

Solution: Find operations are obviously cost $O(1)$. For m union operations, the cost is $m \log n$. The only expensive part is updating the ids in `list[k2]`. And notice that, as in union by weight, each time we update the ids, the size of the list at least doubles. So each item can be updated at most $\log n$ times. Of course, notice that any one operation can be expensive. For example, the last union operation might be combining two sets of size $n/2$ and hence have cost n. However, if there are m union operations, then the biggest set is of size m (why?), and so each item was updated at most $\log m$ times.
The appending of one linked list to the end of another is pretty easily done through manipulation of the head pointers.

Problem 1.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 two operations:

- `getNextTask(name)` that returns the next task for the corporation with the specified name.
- `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 $O(n(\log^2(n) + \log(n)\log(m))$.
You might also observe that there is a better solution. For example, you can implement each of the corporations as a $(2, 3)$ tree, and there is an efficient algorithm for merging two $(2, 3)$ trees of size n in time $O(\log n)$. This, however, requires first designing mergeable $(2, 3)$-trees. Alternatively, you may use heap structures which allow for cheap merging (such as a Binomial Heap or Fibonacci Heap).

Problem 2. Elephant Encounters
Related Kattis Problems:

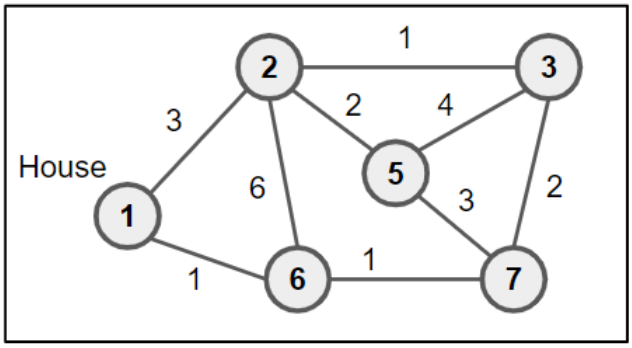
- https://open.kattis.com/problems/arbitrage
- https://open.kattis.com/problems/getshorty

Travelling can be a risky proposition: you never know when you will meet a large pink elephant. Luckily, you have been given a map where each road segment is labelled with the exact probability that someone driving on that road will encounter a large pink elephant. You want to drive from New York to Los Angeles. What route should you take so as to have the smallest chance of meeting a large pink elephant?

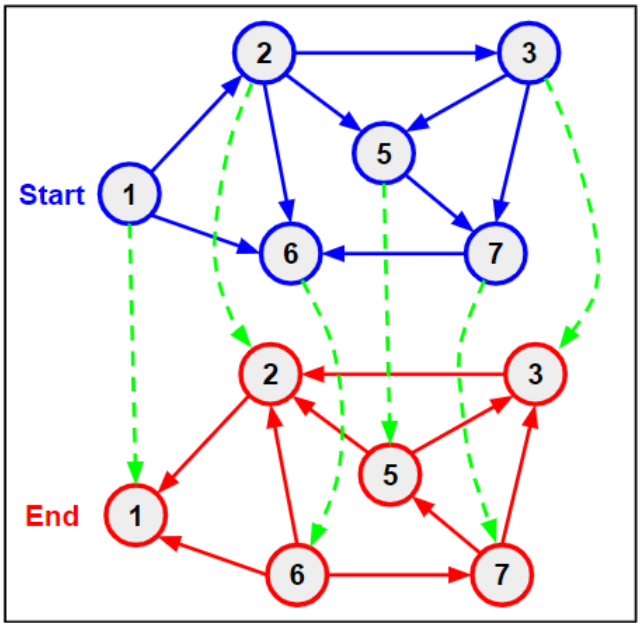
More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The safety of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t.

Solution: This is just a shortest path problem, where instead of summing the edge costs you multiply the probabilities. (This is very similar to the currency conversion problem that we have already seen in recitation.)

Problem 3. Running Trails

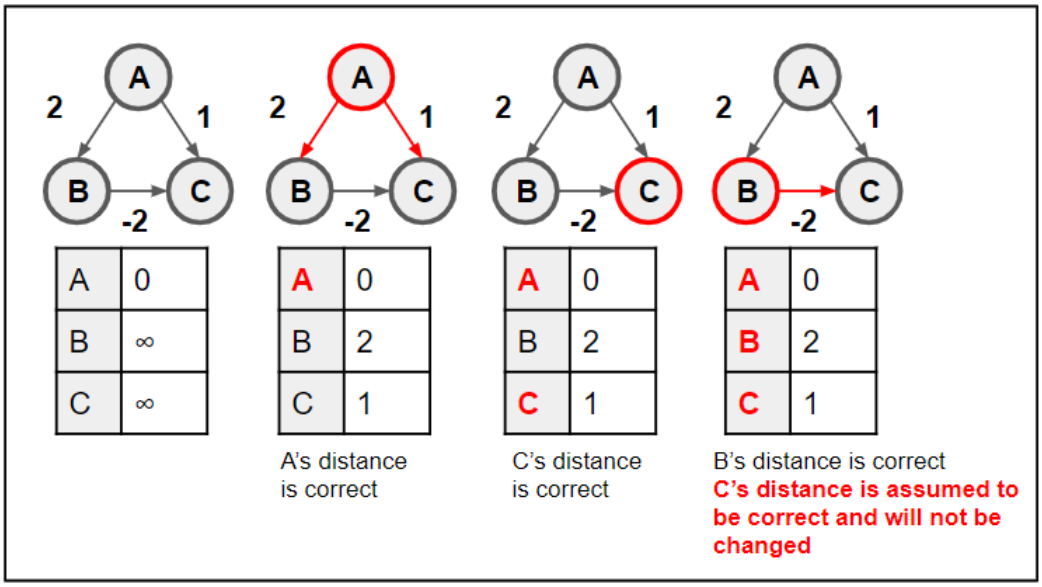


I want to go for a run. I want to go for a long run, starting from my home and ending at my home. And I want the first part of the run to be only uphill, and the second part of the run to be only downhill. I have a trail map of the nearby national park, where each location is represented as a node and each trail segment as an edge. For each node, I have the elevation (value shown in the node). Find me the longest possible run that goes first uphill and then downhill, with only one change of direction.



Solution: First, we want to model this as a more useful graph. Create two copies of the map graph, where in the first each edge is directed uphill (blue graph) and in the second each edge is directed downhill (red graph). Connect each node in the first copy with a directed edge to the corresponding node in the second copy (green edges). Edges connecting two nodes with the same height are omitted.
The problem now reduces to finding the longest route from the source (your home) in the first copy to the destination (your home) in the second copy.
In general, longest path is a hard problem. Luckily, there is a special property here: this is a directed acyclic graph. Notice there cannot be a cycle in the graph. If there were a cycle in the first copy, that would imply a cycle of only uphill edges, which is impossible. Similarly, there can't be a cycle in the second copy. And there are no edges from the second copy back to the first copy. So we can solve this problem by using the algorithm for finding the longest path in a directed acyclic graph, i.e., negate the weights, find a topological order of the graph, and relax the outgoing edges of each node in order.

Problem 4. (Bad Dijkstra)
Give an example of a graph where Dijkstra's Algorithm returns the wrong answer.



Solution: The idea here is to create a graph with *negative* weights so that if you run Dijkstra's Algorithm it fails. The graph shown above is one possible way to construct such a graph.

Problem 5. (Costly Cycles)

Given a weighted directed graph $G = (V, E)$ in which each edge has a weight between 0 and 1: we say that a cycle with c edges is *costly* if the sum of the weights is $> c - 1$.

Problem 5.a. Give an $O(V^3 \log V)$ algorithm for finding the minimum cost (directed) cycle in G.

Solution: For every v, run single-source shortest paths in time $O(E \log V)$ for a total time of $O(V^3 \log V)$. For every pair of nodes v and w, check the shortest path from v to w and the shortest path from w to v—giving the shortest cycle containing both v and w. The minimum over all pairs yields the minimum cost directed cycle.

Problem 5.b. Give an $O(V^3 \log V)$ algorithm for determining whether G has a *costly* cycle. (Hint: use part (a) on a graph with modified edge weights such that a costly cycle has a small total weight.)

Solution: Assign each edge (v, w) the weight `1 - weight(v,w)`. If some cycle with c edges has weight $> c - 1$, then the resulting graph has a cycle with weight < 1. Find the minimum cost cycle in the new graph, and return true if it is < 1 and false otherwise.

Problem 6. (A Random Problem with a dude called Dan)

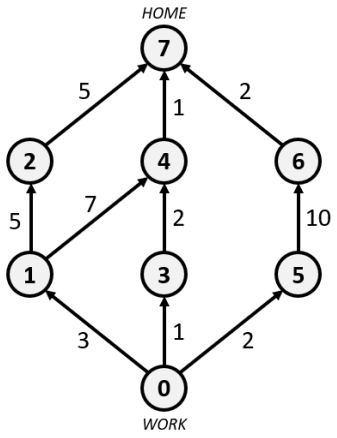


Figure 1: Example city

Dan is on his way home from work. The city he lives in is made up of N locations, labelled from 0 to $(N-1)$. His workplace is at location 0 and his home is at location $(N-1)$. These locations are connected by M **directed** roads, each with an associated *(non-negative)* cost. To go through a road, Dan will need to pay the cost associated with that road. Usually, Dan would try to take the cheapest path home.

The thing is, Dan has just received his salary! For reasons unknown, he wants to flaunt his wealth by going through a *really expensive road.* However, he still needs to be able to make it back home with the money he has. Given that Dan can afford to spend up to D dollars on transportation, help him find **the cost of the most expensive road that he can afford to go through** on his journey back home.

Take note than Dan only cares about the most expensive road in his journey; the rest of the journey can be really cheap, or just as expensive, so long as the entire journey fits within his budget of D dollars. He is also completely focused on this goal and does not mind visiting the same location multiple times, or going through the same road multiple times.

For example, suppose Dan's budget is $D = 13$ dollars. Consider the city given in Figure 1, consisting of $N = 8$ locations and $M = 10$ roads.

The path that Dan will take is $0 \to 1 \to 4 \to 7$. In this journey, the total cost is 11 dollars and the most expensive road has a cost of 7 dollars - the road from locations 1 to 4. Therefore, the expected output for this example would be "7".

Note that this path is neither the cheapest path ($0 \to 3 \to 4 \to 7$), nor is it the most expensive path that fits within his budget of 13 dollars ($0 \to 1 \to 2 \to 7$).

There is also a more expensive road within this city - the road from locations 5 to 6 with a cost of 10 dollars. However, the only path that goes through this road, $0 \to 5 \to 6 \to 7$, has a total cost of 14 dollars which exceeds Dan's budget.

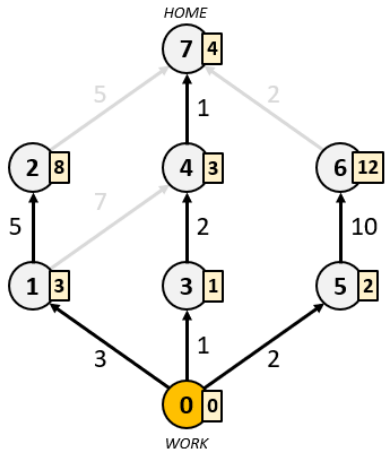
Solution: There are two possible solutions, both with differing approaches.
The first solution tests every edge ($u \to$ v) and checks if the sum of the following does not exceed D:

- The (weight of the) shortest path from vertex 0 to u
- The weight of the edge $u \to v$
- The (weight of the) shortest path from v to vertex $N-1$

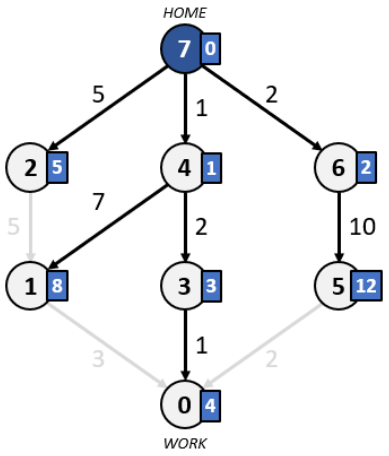
If (and only if) that sum does not exceed D, then Dan can afford to go through the edge $(u \to v)$ on his trip home. Therefore, a simple algorithm would be to iterate through all the edges and find the edge with the maximum weight that satisfies the aforementioned property.
To perform the above check efficiently, we will need to perform some pre-processing. In particular, we need an efficient way to obtain the following:

- The shortest path from vertex 0 to any vertex.
- The shortest path from any vertex to vertex $N-1$.

The former is achieved by performing Dijkstra's algorithm from vertex 0. This will get us the shortest path from vertex 0 to all other vertices in the graph.
The latter is achieved by performing Dijkstra's algorithm from vertex $N-1$ on the transpose graph (i.e. the same graph, but with all the edge directions reversed). For any vertex u, the shortest path from vertex $N-1$ to u in the transpose graph represents the shortest path from u to vertex $N-1$ in the original graph. Therefore, this will get us the shortest path from all vertices in the graph to vertex $N-1$.



Original Graph



Transpose Graph

Now we can test each edge in $O(1)$ time. Including the time taken during the pre-processing, the overall time complexity is $O(M \log N + N) = O(M \log N)$.

Solution: For the second solution, we will define the following property:
Let P_K denote the property that there exists a path *(with possibly repeating edges)* from vertices 0 to $N-1$ such that:

- The path goes through an edge of weight at least K.
- The total weight of the path does not exceed D.

For example, in the given example, P_4 is True because the path $(0 \to 1 \to 2 \to 7)$ satisfies the above conditions. However, P_9 is False because there is no such path.