CS3000: Algorithms & Data — Summer 2023 — Laney Strange

Exam 2 Practice Problems

- These practice problems are to help you prepare for Exam 2. We'll release the solutions on Tuesday (June 13) so you can go over them and ask questions in your recitation.
- Exam 2 takes places during lecture on June 15, 2023.
- The exam will be due at the end of lecture (11:30am). You'll hand it in on paper, but we'll scan it later for grading on gradescope.
- You may bring one 8.5x11-inch paper as a cheat sheet, with anything written or typed on it (one side only). You will submit this cheat sheet along with your exam, and you will not be permitted to use any other materials or notes during the exams.
- If you have a DRC accommodation for exams, please arrange to take the exam at their center. Make sure you schedule that time ASAP if you haven't yet!

Pseudocode For Reference

For the real exam, we'll copy-paste relevant pseudocode that you might need for reference, or that we ask you to modify. For these practice problems, we'll just link instead. Any of the following might come in handy while you work through the problems.

- BFS
- Topo and DFS
- Kruskal
- Prim
- Dijkstra's
- Floyd Warshall

Problem 1. *Greedy*

Humpty Dumpty has fallen off the wall, and broken into n different pieces. All the king's horses are trying to put Humpty back together, but they need to make sure they have enough tape to repair him.

Humpty broke into n fragments of various sizes $s_1, s_2, \ldots, s_n$. Since bigger fragments are heavier, they need more tape to stick together. When we put two fragments with sizes s_i and s_j back together, we replace the two fragments with a single fragment of size $s_k = s_i + s_j$, and we use $s_i + s_j$ pieces of tape. We would like to completely put Humpty Dumpty back together, i.e. end with a single fragment, using the least amount of tape possible.

(a) Assume for this part that Humpty is size 38, and split into $n = 4$ pieces with sizes $[4, 5, 7, 12]$. What is the least amount of tape we can use to put Humpty together, and in what order do you connect the fragments?

Solution:

(b) For the general case, if you tackle this problem with a greedy approach, what would your greedy choice be at each step?

Solution:

(c) Give pseudocode for your greedy strategy. Your algorithm will take as input the value $n > 0$ and a min-heap of positive integers $[s_1, ..., s_n]$, with each integer representing the size of a fragment. Your algorithm should return the value of an optimal solution (i.e., the total amount of tape used for a Humpty of size n).

Solution:

Problem 2. *Amortized Analysis*

In lecture we saw that when inserting into an unsorted array which is full we can create a new array with double the capacity and copy the old array's contents into a new array in order to obtain $O(1)$ amortized cost for insertion. For this problem, we'll add a delete operation into the mix.

You may assume that we will always insert and delete the rightmost element of the array.

1. As a first approach, we can halve the array any time the capacity would not be exceeded. For example, if an array with capacity $2n$ has its $(n+1)$-th element removed, we can create a new array with capacity n and copy the remaining n elements of the old array.

   However, this approach doesn't give us an amortized $O(1)$ cost per operation. Give a counterexample of a sequence of k operations that would result in $O(k^2)$ run-time.

   Solution:

2. Let's try a better approach. Now, we halve the capacity of the array any time the number of elements does not exceed a quarter of the total capacity. For example, if an array with capacity $4n$ has its $(n+1)$-th element removed, we can create a new array with capacity $2n$ and copy the remaining n elements of the old array.

   Argue that this results in $O(1)$ amortized cost over the operations, by showing that any sequence of k insert/delete operations has an aggregate time at most $O(k)$. (*Hint:* Anytime we complete an "expensive" insert/delete, we get an array with roughly double the capacity of its contents. Argue that before the next expensive operation, we must have performed at least $O(k)$ "cheap" operations.)

   Solution:

Problem 3. *Which Graph Algo*

For each problem on the left, give the best matching algorithm that could solve the problem from the right.

(1) The best way to lay out train tracks, if n cities need to have a station and you want to minimize the total amount of track used.

(2) An order in which you can take classes with pre-requisites.

(3) The fastest route to get from school to home.

(4) How can an airline guarantee that they fly to all the places they say in their ads, but with as few flights as possible.

(5) The fewest number of clicks to get from one web page to another.

(6) The cheapest flight you can take, including possible stopovers, to get from Boston to Vancouver.

(a) Breadth-First Search

(b) Topological Sort

(c) Minimum Spanning Tree

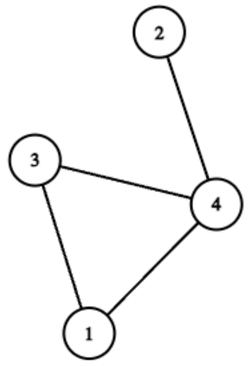
(d) Dijkstra's Algorithm

(e) APSP Algorithm

Solution:

Problem 4. *Breadth First Search*

This problem is concerned with the unweighted, undirected graph below.



(a) What would the d and π values be of the vertices after the BFS algorithm runs, assuming we use 1 as the starting vertex?

vertex	1	2	3	4
d				
π				

Solution:

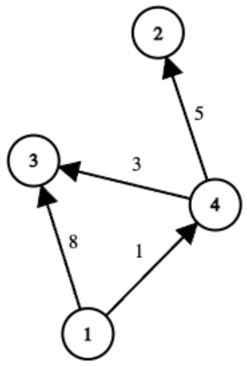
(b) Give pseudocode for a function that, given a graph G, starting vertex s, and destination vertex v, would print out the shortest path from s to v using the vertices' d and π values. Your function should print "no path exists" if there is no path from s to v. (*Hint*: we recommend a recursive algorithm!)

Solution:

Problem 5. *DFS and SSSP*

Recall that Dijkstra's algorithm works on a directed, weighted graph with non-negative weight edges. We can find a solution to SSSP more simply than Dijkstra's if we know that our graph is directed and acyclic, by changing the order in which we explore the vertices.

(a) What are the DFS finish times on the DAG below, assuming we start with vertex 1 and break ties in numerical order?



vertex	1	2	3	4
finish time				

Solution:

(b) Finish the pseudocode below, so that it Relaxes the edges of the graph above, once you've found the DFS finish times. We can assume vertex 1 is the source.

```
DAG-RELAX($G, w, s$)
1  sort the vertices of $G$ by descending DFS finish time
```

Solution:

(c) What would the d and π values be for each vertex in the above graph after your algorithm runs?

vertex	1	2	3	4
d				
π				

Solution:

(d) Has your algorithm correctly found the shortest path from vertex 1 to all other reachable vertices?

Solution:

(e) Would your algorithm correctly find the shortest path from vertex 1 to all other vertices, if the graph had negative-weight edges?

Solution:

Problem 6. *SSSP*

You are given a connected, undirected graph G and a starting vertex s, and you want to compute a SSSP solution. However, the weights of this graph are on vertices rather than edges; the weight function $w(u)$ returns the weight of a vertex u. The length of a path is the sum of the weights on the vertices comprising the path. The vertex weights are non-negative.

Suppose you construct a new directed graph G' with the same vertices as $G.V$ and two edges (u,v) and (v,u) for every undirected edge in G. You define the edge-weight function w in G' as $w(u,v) = w(v)$ from the original graph G. You want to run Dijkstra's on G' to find the shortest paths in G.

(a) If you have a shortest path from s to some vertex v in G, what would be the equivalent shortest path from s to v in G'?

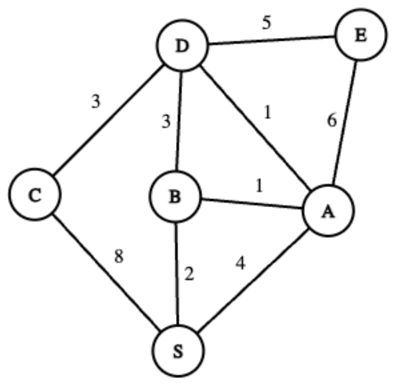
Solution:

(b) If the weight of a shortest path from s to some vertex v in G' is w_p, what would be the weight of the equivalent shortest path from s to v in G?

Solution:

Problem 7. *MST - Kruskal*

This problem is concerned with the graph below. Recall that Kruskal's algorithm sorts edges by weight in increasing order. It adds an edge (u, v) to the MST A if (u, v) is safe, i.e., u and v are not in the same set.



(a) List the edges that would be contained in the final set A in Kruskal's algorithm, in the order in which they are added to A.

Solution:

(b) What is the total weight of the MST generated by Kruskal's algorithm?

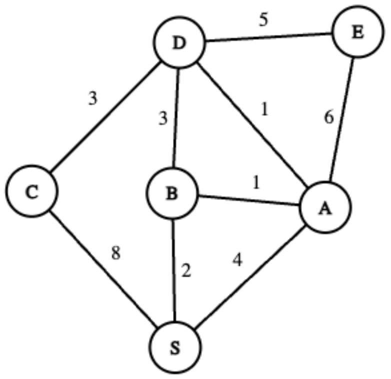
Solution:

(c) Suppose all the edge weights in the graph above are inverted, i.e., the new weight w' would be $w'(u, v) = -1 \cdot w(u, v)$. Now we have all negative-weight edges. Would Kruskal's correctly computed an MST without any changes to the algorithm? What total weight would you get from running Kruskal on this new graph?

Solution:

Problem 8. *MST - Prim*

This problem is concerned with the graph below. Recall that Prim's algorithm adds edges to an MST such that, at any step, it has built a tree. It maintains a *key* (weight) and π (predecessor) value for each vertex.



(a) List the *key* and π values that would be assigned to each of the vertices in the above graph, after Prim's algorithm has completed, assuming it starts at source vertex s.

	S	A	B	C	D	E
π	NIL					
key						

Solution:

(b) What is the total weight of the MST generated by Prims's algorithm?

Solution:

(c) Suppose each undirected edge (u, v) in the above graph is replaced by a directed edge (u, v), such that the directed edge (u, v) exists if v is later alphabetically than u. All the weights stay the same. For example, (c, d) would be a directed edge with weight 3. (b, s) would be a directed edge with weight 2, etc. Would your solution from Part B still be *optimal* in the sense that we could touch all the vertices (with an incoming and/or outgoing edge) for the same total weight?

Solution:

(d) Would Prim's algorithm still correctly compute that solution without any changes to the algorithm, if we start at vertex s? Justify your answer by describing how the algorithm would assign *key* and/or π values to each vertex. **Solution:**.

Problem 9. *Sink Vertex*

A directed graph G is said to contain a universal sink if there exists some vertex v with (1) no outgoing edges, and (2) incoming edges from every vertex except itself.

(a) Give pseudocode for an algorithm that determine whether a particular vertex, i, is a universal sink. Your algorithmm should accept the graph G as an input, along with i. It should return a boolean indicating whether i is a universal sink. Assume that your graph is stored as an adjacency matrix.

Solution:

(b) What would the run-time be for the entire graph if you call your function above once per vertex?

Solution:

(c) TRUE or FALSE: If your adjacency matrix has $A[i, j] = 1$ for some i, j, can we conclude that the ith vertex cannot be a sink.

Solution:

(d) TRUE or FALSE: If your adjacency matrix has $A[i, j] = 0$ for some i, j, can we conclude that the ith vertex cannot be a sink.

Solution:

Problem 10. *APSP*

The Floyd-Warshall algorithm has space complexity $\Theta(V^3)$. Write a new version of the pseudocode below that requires only $\Theta(V^2)$ space. The run-time will still be cubic, but the space requirements are improved. (You can assume the first part of the code, where we initialize $D^{(0)}$, is the same.)

Solution:

Problem 11. *Max Flow*

One greedy approach to Max Flow would repeatedly find a path from the source s to the sink t. If the path exists, then find the minimum-capacity edge on that path, increase flow on all edges of the path, and then remove those edges from consideration.

(a) Give an example of a directed graph, with capacities on each edge, for which that Greedy choice would not find an optimal solution. **Solution:**

(b) In your graph above, what solution would the Greedy approach give you?

Solution:

(c) What would the optimal solution be for your example?

Solution: