CSE 373 19su: Final, Part 2 Solutions

Name:

UW email address:

Instructions

- **Do not start the exam until told to do so; immediately stop writing when time is called.**
- No electronic devices (including calculators and cell phones) are allowed.
- You are allowed one sheet of 8.5" x 11" paper (both sides) of notes.
- You have 60 minutes to complete the exam.
- Write your answers neatly in the provided space. Be sure to leave some room in the margins: we will be scanning your answers.
- If you need extra space, you may use the back of a sheet, **but you must clearly indicate in each subproblem where you want us to read the back**
- If you have a question, raise your hand to ask the course staff for clarification.
- Closed-forms of some summations, some logarithm identities, and Master Theorem are on a separate sheet.

Advice

- Write down your assumptions, thoughts and intermediate steps (this makes it easier to award partial credit).
- Clearly circle your final answer.
- The questions are not necessarily in order of difficulty; skip around!

Unless otherwise noted, all big-$\mathcal{O}$ and big-Ω analyses must be tight, and all $\mathcal{O}, \Omega, \Theta$ expressions must be simplified.

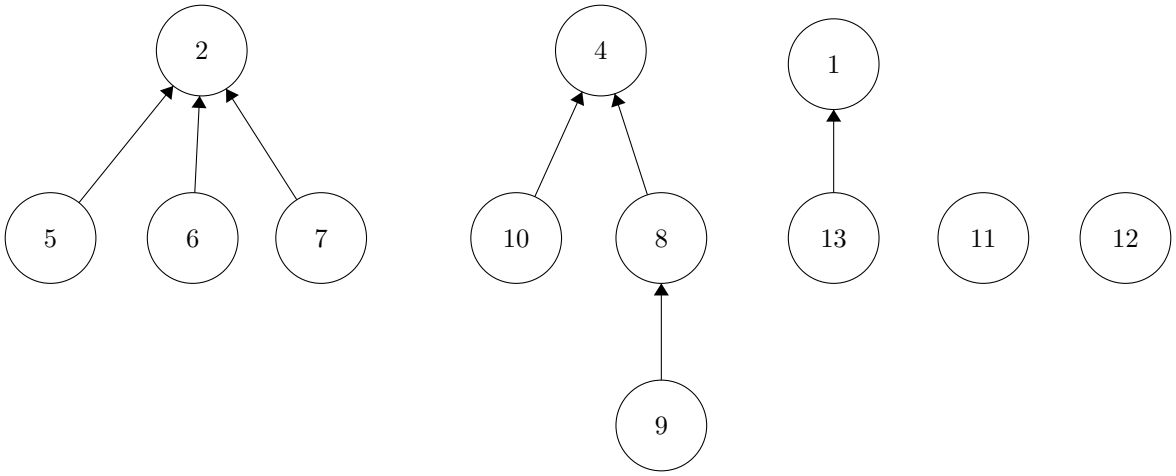
Any question asking for an algorithm or implementation should be as efficient as possible in big-$\mathcal{O}$ terms.

The following table gives **approximate** point values per problem. The values may change slightly during grading; these numbers are intended to help you prioritize what problems to spend time on.

Question	Max points
Union-Find	3
Graph Modling IA	8
Graph Modeling IB	8
Graph Modeling II	8
Sorting and Graphs	14
Design Decisions: Sorting	6
Graphs Short Answer	9
Total	**56**

1. Union-Find

Below is an initial state of a disjoint-sets data structure (using the implementation from Project 4). Draw the final state of the data structure after all of the following commands have been executed. Use both the union-by-rank and path compression optimizations. If a union call could choose either old root for the new root, select the smaller number to be the new root.



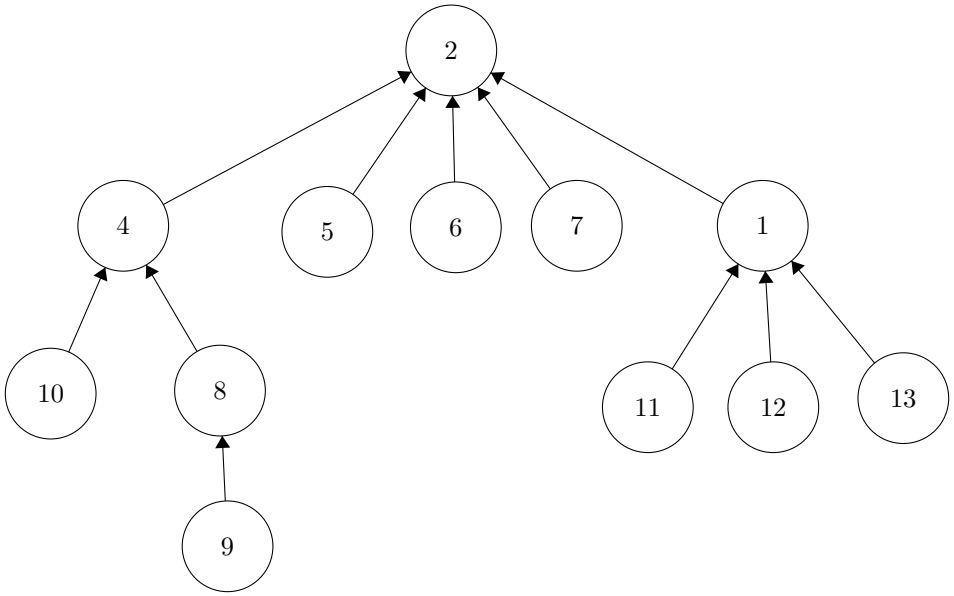
The initial ranks of the trees, from left to right, are $2, 2, 1, 0, 0$.

Execute the following commands:

```
union(4,2)
union(11,12)
union(1,11)
union(12,5)
```

Draw the final data structure here. Remember to show the rank(s) of the final tree(s).

Solution: The tree is rank 3.



2. Graph Modeling I

2.1. IA: A first pass

After finishing your final, you decide to celebrate with a trip downtown; unfortunately your U-PASS has expired so the only way to travel is with LemonBikes. Every LemonBike has a battery-powered motor which will make biking easier for 500 feet. But after 500 feet, the battery dies and the motor cuts out.

With the LemonBike app, you can see the location of every LemonBike available (they are all fully-charged). Your goal is to get downtown, by only biking **on bikes whose motors are still running** (you are too exhausted to walk or bike without battery-assistance). Also, **you want to use as few bikes as possible on your trip**, because LemonBikes charge per unlocking (and not distance or time). You will leave from your apartment (there is, conveniently, a LemonBike right outside).

(a) Explain how you would model this scenario as a graph. Answer the following questions in bullet points or short sentences.

(i) What do your vertices represent? If you store extra information in each vertex, what is it?

Solution: We have one vertex for each bike (or the position of each bike). And one extra vertex for our destination downtown.

(ii) What do your edges represent (your answer may be a real-world object, or an abstract description of what edges will exist)? If you store extra information in each edge, what is it?

Solution: There is an edge between two vertices if they are within 500 feet (i.e. if you can get from one bike to the other on battery power. An edge between (u, v) exists if it is possible to reach v from u with a powered bike.

(iii) Is your graph directed? Briefly explain (1 sentence).

Solution: No. If we can bike from node u to node v, then the reverse is also true.

(iv) Is your graph weighted? If so what are the weights? Briefly explain (1 sentence).

Solution: No. We don't care about how far apart two bikes are, as long as we can get from one to the other on battery. We only care about the number of bikes (i.e. edges) used.

(b) Call the vertex for your apartment s and your desired destination downtown t. Describe an algorithm to run to plan your trip downtown. You may use any algorithm from class (just tell us what the inputs are). If you wish to modify an algorithm from class, you may—briefly describe your changes in English (not pseudocode).

Solution: Run BFS-Shortest-Paths from s, with destination t. We can trace back through predecessors to find the path itself. (Note that Dijksra's also produces the correct answer, but will be slower.)

(c) What is the tight, simplified big-$\mathcal{O}$ for the worst-case running time of your algorithm (if your graph has n vertices and m edges)?

Solution: $\mathcal{O}(m+n)$.

2.2. IB: A better plan

Oh no! You run your algorithm and realize you can't make it downtown on only powered bikes. You decide you'll have to just use unpowered bikes part of the way. You want to plan a trip downtown, which minimizes the **number of feet** you have to ride on bikes **after their batteries have run out**. Note: you no longer care about minimizing the number of bikes you use.

(a) Explain how you would model this scenario as a graph. Answer the following questions in bullet points or short sentences.

(i) What do your vertices represent? If you store extra information in each vertex, what is it?

Solution: The position of each bike, plus one extra node for downtown.

(ii) What do your edges represent (your answer may be a real-world object, or an abstract description of what edges will exist)? If you store extra information in each edge, what is it?

Solution: There is an edge between every pair of vertices. Because all the positions are reachable via unpowered ride.

(iii) Is your graph directed? Briefly explain (1 sentence).

Solution: No. The weight of (u, v) and (v, u) are the same.

(iv) Is your graph weighted? If so what are the weights? Briefly explain (1 sentence).

Solution: Yes. It represents the distance we travel without battery-assistance. To be precise, each edge has a weight which is 0 if the distance between them is smaller than 500 feet, or the distance between the two vertices minus 500 otherwise.

(b) Call the vertex for your apartment s and your desired destination downtown t. Describe an algorithm to run to plan your trip downtown. You may use any algorithm from class (just tell us what the inputs are). If you wish to modify an algorithm from class, you may—briefly describe your changes in English (not pseudocode).

Solution: Run Dijkstra algorithm to find the shortest path between s and t. We can trace back through predecessors to find the path itself.

(c) What is the tight, simplified big-$\mathcal{O}$ for the worst-case running time of your algorithm (if your graph has n vertices and m edges)?

Solution: $\mathcal{O}\left((m+n)\log n\right)$.

3. Graph Modeling II

UW has decided it is too bright outside for people used to Seattle weather. They plan to build canopies (temproary coverings) over certain sidewalks. The undersides of the canopies will be painted with clouds, so that everyone can be more comfortable while walking through campus. UW wants to ensure that everyone can make it from any building to any other without needing to be hit by direct sunlight, but they don't want to spend any more money than necessary. They hired a contractor, who has told you how much it would cost to cover any particular sidewalk. You must decide which sidewalks to cover.

(a) Explain how you would model this scenario as a graph. Answer the following questions in bullet points or short sentences.

(i) What do your vertices represent? If you store extra information in each vertex, what is it?

Solution: Each vertex represents a building. No extra information.

(ii) What do your edges represent (your answer may be a real-world object, or an abstract description of what edges will exist)? If you store extra information in each edge, what is it?

Solution: Each edge represents the sidewalk between 2 buildings.

(iii) Is your graph directed? Briefly explain (1 sentence).

Solution: No. Sidewalks are bi-directional.

(iv) Is your graph weighted? If so what are the weights? Briefly explain (1 sentence).

Solution: Yes. It is the cost of building canopies on the sidewalk.

(b) Describe an algorithm to run to plan where to put up the canopies. You may use any algorithm from class (just tell us what the inputs are). If you wish to modify an algorithm from class, you may—briefly describe your changes in English (not pseudocode).

Solution: Run either Prim's algorithm or Kruskal's algorithm on the graph to find the minimum spanning tree, and build canopies on the sidewalks that are contained in the MST.

(c) What is the tight, simplified big-$\mathcal{O}$ for the worst-case running time of your algorithm (if your graph has n vertices and m edges)?

Solution: $\mathcal{O}(m \log m)$ for Kruskal's. $\mathcal{O}(m \log n)$ for Prim's

4. Sorting and Graphs

4.1. Fixing Bob's sort

Bob just learned about topological sort on graphs and decided to use topological sort to implement a comparison sort. His idea is to have a vertex for each object, and use edges to represent which objects are less than others. He shows you his algorithm.

```
1  public Object[] topoSortSort(Object[] input) {
2      int k = input.length;
3      DirectedGraph g = new DirectedGraph();
4      for (int i = 0; i < k; i++) {
5          // the new `Vertex` object stores input[i] in its `value` field
6          g.addVertex(new Vertex(input[i]));
7      }
8      for (Vertex u : g.vertices) {       // there's a bug in this loop
9          for (Vertex v : g.vertices) {
10             // `equals()` checks "reference equality" (whether two variables point
11             //     to the same object), not whether the `value` fields are equal
12             // `<=` would be implemented with a `compareTo()` call in real Java code
13             if (!u.equals(v) && u.value <= v.value) {
14                 g.addEdge(u, v); // adds a directed edge from u to v
15             }
16         }
17     }                                   // there's a bug in this loop
18
19     List<Vertex> sorted = g.topoSort(); // returns vertices in topological-sorted order
20
21     Object[] toReturn = new Object[k];
22     int i = 0;
23     for (Vertex v : sorted) {
24         toReturn[i] = v.value;
25         i++;
26     }
27     return toReturn;
28 }
```

(a) Bob's code has a bug which will cause line 19 to fail on some inputs. Describe an input on which it would fail, and why the `topoSort` runs into a problem. (The issue is in the loop between lines 8-17.)

Solution: Recall that `topoSort` fails exactly when the graph has a cycle. One input that generates a cycle is $[1, 1, 1]$. Every possible edge (in both directions) will appear in this graph, so we will have cycles and topological sort fails.

(b) Describe how to fix the bug. One potential fix changes only one line of code. You can answer in either a few lines of pseudocode or 1-2 sentences of English.

Solution: Change line 13 from $\leq$ to $<$.

4.2. Analyzing the final version

Now that we've fixed `topoSortSort`, let's analyze it. Bob's original code is reprinted at the bottom of this page for your convenience, but note that the first two parts ask about possible modifications, not Bob's original version.

(a) Can you modify `topoSortSort` to be a stable sort? If yes, briefly describe how to make it stable, if not describe why it must be unstable. In either case your description should be 1–2 sentences of English (not pseudocode).

Solution: Yes. If two vertices have equal values, add only the edge from the vertex which appeared first in the original array to the one that appears second in the original array (in actual code, the easiest way to write this part is to use explicit iterators in place of for-each loops, which is why we didn't ask you to implement this change).

(b) Is your sort in-place? Briefly (1–2 sentences) explain why.

Solution: No. The graph is definitely more than $\mathcal{O}(1)$ extra memory.

(c) Let's try to find the worst-case running time of `topoSortSort`. Give your answers in terms of k, the length of the `input` array.

(i) What is the simplified, tight big-$\mathcal{O}$ for the worst-case running time of line 19? (Assume that the method call does not fail.)

Solution: $\mathcal{O}\left(k^2\right)$. The graph can contain $\Omega\left(k^2\right)$ edges and $\Omega(k)$ vertices in the worst-case (if all values are distinct, for example). Note that $\mathcal{O}\left(k^2 + k\right)$ simplifies to $\mathcal{O}\left(k^2\right)$.

(ii) What is the simplified, tight big-$\mathcal{O}$ for the worst-case running time of `topoSortSort`? You may assume that any method call other than `topoSort` on line 19 takes $\mathcal{O}(1)$ time.

Solution: $\mathcal{O}\left(k^2\right)$.

5. Design Decisions: Sorting

For each of the following scenarios, choose the most appropriate sorting algorithm from the following list, then briefly (1–2 sentences) explain why your choice is best.

Selection sort, Insertion sort, Heap sort, Quick sort, Merge sort

(a) You must sort an absolutely enormous dataset—it takes up more than half of the memory on your system. Your boss will closely watch the performance of the algorithm on many datasets, and gets upset if any of them go too slowly.

Solution: Heap sort. Because we are using so much memory to store the dataset, we need an in-place sort, and because performance is being watched for consistency, we need worst case $\mathcal{O}(n \log n)$. Heap sort meets both of these conditions (none of our other sorts do).

(b) You just finished sorting a huge pile of (paper) exams, when 10 more people turn in the exam. You only have a small table, so you can't create additional piles. (This scenario is loosely based on a true story.)

Solution: Insertion sort. Note that the list is already sorted, so it runs much more quickly than any of the other sorts. Since we have no room for piles, we additionally require an in-place sort, which insertions sort also satisfies.

We also accepted observations relating to how easy/difficult sorts would be to implement by hand (insertion would likely be the easiest).

6. Graph Short Answer

6.1. BFS/DFS

Each of the following questions asks about applications of BFS/DFS. For "applications" we are expecting graph problems to solve; for example, "Finding a minimum spanning tree" is an application (but not a correct answer). No explanation/justification is required.

(a) Give an example of an application of BFS which cannot be (easily and efficiently) implemented with DFS.

Solution: finding shortest paths in an unweighted graph.

(b) Give an example of an application of DFS which cannot be (easily and efficiently) implemented with BFS.

Solution: Finding strongly connected components.

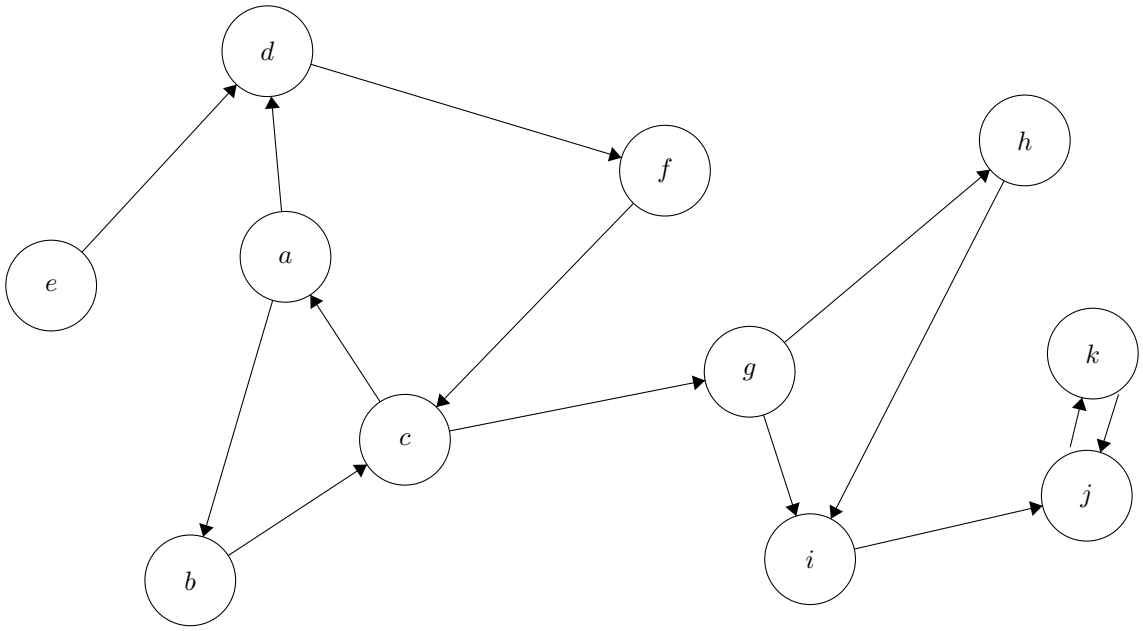
We also accepted topological sort – there is an alternative topological sort algorithm (not the one discussed in class) which uses DFS as its base, which could not be replicated with BFS.

(c) Give an example of an application where either BFS or DFS can be used.

Solution: Finding connected components in undirected graph (or the weakly connected components of a directed graph). The first step of topological sort (i.e. finding the out-degrees of each vertex) can also be done with either BFS or DFS.

6.2. Strongly Connected Components

Identify the strongly connected components of this graph.
You may either circle the components, or list them below the graph.



Solution: $\{a, b, c, d, f\}, \{k, j\}, \{e\}, \{g\}, \{h\}, \{i\}$

7. Extra Credit

Draw a picture of your TA in their natural habitat.