CSE 332: Data Structures and Parallelism

Section 9: Graphs & Connectivity Solutions

In lecture, we solved the **CONN** problem. That is,

CONN	
Input(s):	Graph G
Output:	`true` iff G is connected

We used a `WorkList` algorithm:

```
1  isConnected(G) {
2     V, E = G
3     worklist = first(V);
4     seen = {v};
5     while (worklist.hasWork()) {
6        v = worklist.next();
7        for (w : v.neighbors()) {
8           if (w ∉ seen) {
9              worklist.add(w);
10             seen.add(w);
11          }
12       }
13    }
14    return seen == V;
15 }
```

This algorithm has several distinct names based on which type of WorkList we use. If we use a `FIFOQueue`, it's called Breadth-First Search (BFS), and if we use a `Stack`, it's called Depth-First Search (DFS).

0. Recursively, Now!

Although we've implemented it here and in lecture iteratively, we could implement **DFS** recursively as well. Write Pseudo-code for a DFS that uses recursion.

Solution:

```
1  dfs(G) {
2     dfs(first(G), {})
3  }
4
5  dfs(v, seen) {
6     if (v ∈ seen) {
7        return;
8     }
9     for (w : v.neighbors()) {
10       seen = seen ∪ {w}
11       dfs(w, seen)
12    }
13 }
```

1. Two-Coloring

A graph G is two-colorable if and only if we can make a function $f : V \to \{\text{red}, \text{black}\}$ such that $\{u, v\} \in G \to f(u) \neq f(v)$. That is "adjacent vertices have different colors". Write an algorithm to solve the **2-COLOR** problem.

Solution:

Instead of storing the vertices in a "seen" set, store them in a dictionary from $V \to \{\text{red}, \text{black}\}$. Then, attempt to make a coloring (start with, say black) using a DFS (starting from any vertex). If we try to color a vertex two different colors, there is no two-coloring. If we finish, it works.

2. A Social Networking Event

Suppose you have social network data for some people (including yourself and a famous person). Explain how to use a graph algorithm to find the answers to the following questions:

(a) Find the person with the *most friends* in the data?

Solution:

Instead of storing the vertices in a "seen" set, store them in a dictionary from $V \to \texttt{int}$. Then, do a BFS/DFS (starting from any vertex), where we store the number of neighbors when we remove a vertex. Additionally, as we continue, also store the largest number of neighbors we've seen so far (and the vertex who had those neighbors).

(b) Find *the length of the shortest path* from yourself to the famous person.

Solution:

Do a BFS starting at yourself. In addition to the normal stopping condition (we've seen this vertex before), stop when we hit the famous person. In the case where we fail to find the famous person, return ∞. Otherwise, return the number of vertices we've seen so far.

Notice that a DFS *does not work here*! A DFS will return *some path*, but not necessarily the shortest one.

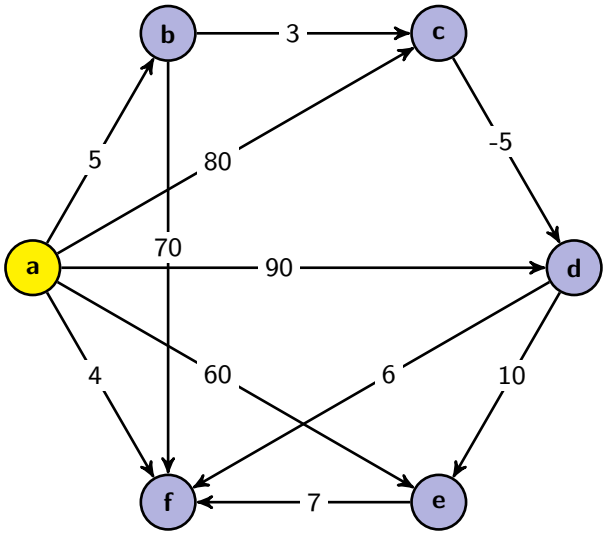
(c) Find *the number of people* who do not know anyone you know.

Solution:

Let S be the set of all people who within distance 2 of yourself. That is, S is the set containing yourself, your neighbors, and your neighbors' neighbors. You can find this by doing a modified BFS/DFS that keeps track of the distance explored as an additional parameter. The answer is then $|V| - |S|$.

3. Better Find the Shortest Path Before It Catches You!

Consider the following graph:



(a) Use Dijkstra's Algorithm to find the **lengths** of the shortest paths from **a** to each of the other vertices. For full credit, you must show the worklist at every step, but how you show it is up to you.

Solution:

Vertex	Cost	Done?
a	0	✓
b	~~∞~~ 05	✓
c	~~∞~~ ~~80~~ 08	✓
d	~~∞~~ ~~90~~ 03	✓
e	~~∞~~ ~~60~~ 13	✓
f	~~∞~~ 04	✓

(b) Are any of the lengths you computed using Dijkstra's Algorithm in part (a) incorrect? Why or why not?

Solution:

In this case, no. In general, Dijkstra's Algorithm does not necessarily work correctly with negative edge weights, but here, it actually returns the right result.

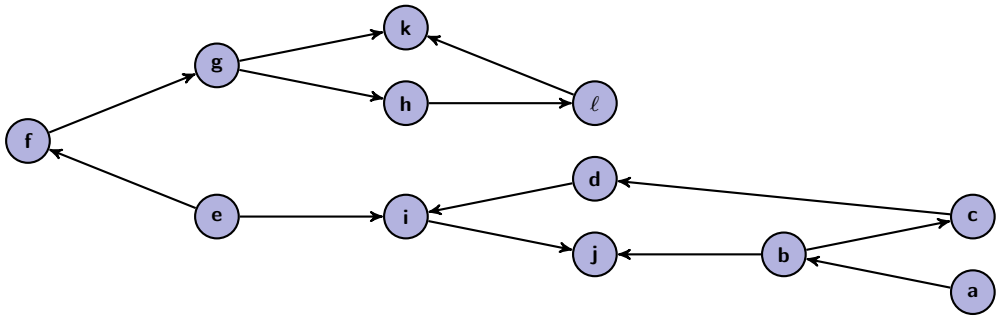
(c) Explain how you would use Dijkstra's Algorithm to recover the actual paths (rather than just the lengths).

Solution:

Keep an extra dictionary which maps nodes to their predecessors. (A predecessor is the node we took an edge from to get to the node.) Then, walk from the target vertex back toward the source vertex using the predecessor map.

4. It Rhymes with Flopological Sort

Consider the following graph:



(a) Does this graph have a topological sort? Explain why or why not. If you answered that it does not, remove the **MINIMUM** number of edges from the graph necessary for there to be a topological sort and carefully mark the edge(s) you are removing. Otherwise, just move on to the next part.

Solution:

Yes, it does. This is a DAG (i.e., it has no cycles).

For the remaining parts, work with this (potentially) new version of the graph.

(b) Find a topological sort of the graph. Do not bother showing intermediary work.

Solution:

There are many. One example is e, f, g, h, l, k, a, b, c, j, d, i.

(c) If this graph represented various tasks in a `ForkJoin` algorithm, what would the work of the algorithm be assuming each individual task is $\Theta(1)$.

Hint: There are 12 tasks...

Solution:

There are a constant number of tasks; so, the work is $\Theta(1)$.