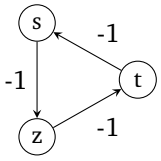
Section 08: Solutions

1. Limitations and properties of shortest path algorithms

(a) Draw an example of a directed graph where (a) there exists a path between two vertices s and t but (b) there *is no shortest path* between s and t.

Solution:

If the graph has a negative-cost cycle, there's always a way to get from s to t, but there's no shortest way: we can basically loop through the negative cycle an infinite number of times to bring the cost down as much as we want.



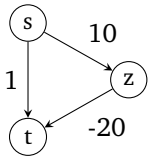
(b) Draw an example of a directed graph where there does exist a *shortest path* between two vertices s and t but if we try running Dijkstra's algorithm on s, it fails to find that shortest path and returns an incorrect result.

Solution:

Dijkstra's algorithm is not guaranteed to work correctly if there are negative cost edges.

For example, in this problem, if we start on s, Dijkstra's algorithm will initially assign t and z a cost of 1 and 100 respectively. Dijkstra's will then pick the t node next – remember, Dijkstra's algorithm is a *greedy* algorithm and will always pick the pending node that has the smallest cost.

This turned out to be a poor decision though, because if we were more patient and tried exploring z first, we would have discovered a more cost-efficient route.



(c) Given some arbitrary graph, how would you determine if it contained a cycle?

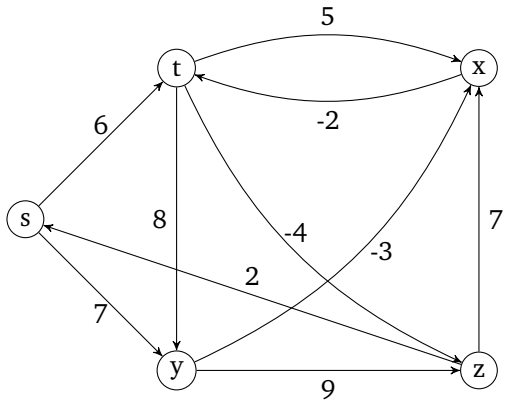
Solution:

Run any graph-traversal algorithm (e.g. BFS or DFS) and see if we visit a node we were already on.

The main edge case we must handle is if the graph contains unconnected components: after running the traversal algorithm, we basically need to check if there are any nodes we haven't visited yet, and try running the traversal algorithm again.

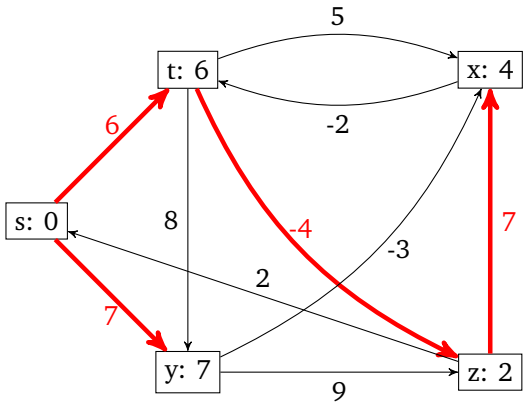
2. Simulating Dijkstra's

(a) Consider the following graph:

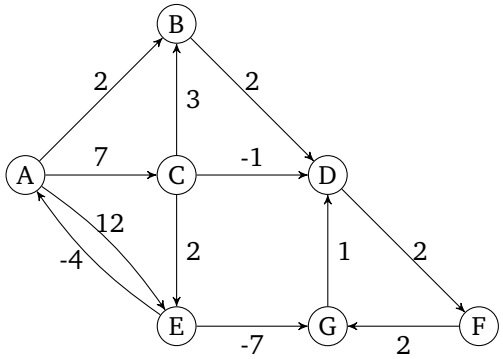


Suppose we run Dijkstra's algorithm on this graph starting with vertex s. (Note: we will very likely get meaningless results due to the negative edges, but this is still a good way to practice running the algorithm.) What are the final costs of each vertex and the shortest paths from s to each vertex?

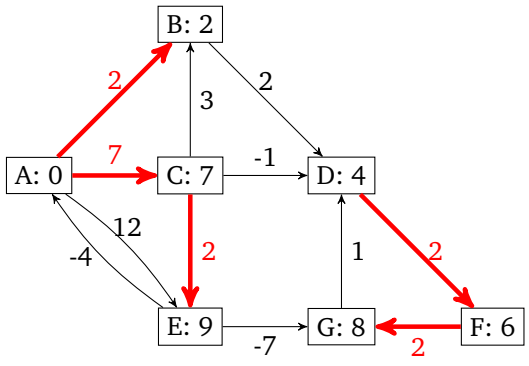
Solution:



(b) Here is another graph. What are the final costs and shortest paths if we run Dijkstra's starting on node A?

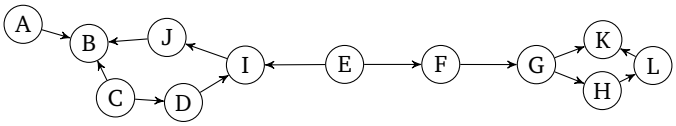


Solution:



3. Practicing topological sort

Find a topological sort of the following graph:



Solution:

One possible solution: E, F, G, H, L, K, A, C, D, I, J, B

4. Graph representations

(a) Suppose we have a directed graph represented in *adjacency list* form. Design an algorithm that produces a new graph (also in adjacency list form) where each edge is reversed.

Solution:

Here is one possible implementation:

```
def reverse(graph):
    output = new empty graph
    for ((v, neighbors) : graph):
        for (u : neighbors):
            if (!output.containsKey(u)):
                output.put(u, new set or list)
            output.get(u).add(v)
    return output
```

(b) Implement an algorithm that does the same thing, except that we now accept and return a graph in *adjacency matrix* form.

Solution:

Here is one possible implementation:

```
def reverse(graph):
    output = new 2d array
    for (i : 0 to graph.numRows()):
        for (j : 0 to graph.numCols()):
            output[j][i] = graph[i][j]
    return output
```

5. Designing algorithms: Applying algorithms

(a) Explain how you would implement DFS recursively, rather then iteratively. How much memory does the recursive algorithm use vs the iterative one?

Solution:

Here is one way we could implement this algorithm recursively:

```
def dfs(Vertex v, Set<Vertex> visited):
    if (v not in visited):
        visited.add(v)
        for (u : v.neighbors()):
            dfs(u, visited)
```

Note that both the iterative and recursive versions use the same amount of memory. With the iterative one, we store pending nodes on an explicit stack data structure. With the recursive one, we store them implicitly on the function call stack.

(b) Suppose you want to model how a tweet gets re-tweeted by followers on Twitter. You have data on who the users of Twitter are and all the followers of each user. Given a source, a tweet can be re-tweeted by any follower of that source.

Suppose a particular user makes a tweet. Design an algorithm that designs which users could have seen this tweet, assuming the tweet was re-tweeted at most k times.

Solution:

Run BFS or DFS to a depth of $k + 1$ (the extra iteration is to account for the people on the final layer who may have seen the tweet but don't retweet it).

(c) Suppose we have graph where the edges are unweighted, but the vertices have numerical weights. Explain how you would modify one of the algorithms we've studied in class to find the shortest path between two vertices.

Solution:

Here, we can run Dijkstra's algorithm almost unmodified. The only change we'd need to make is that instead of updating adjacent nodes using the sum of the current vertex cost and the edge weight, we just add the current vertex cost to the other vertex's existing cost.

(d) **Challenge:** Suppose you are trying to find the most strenuous hiking trail between two points: the path with the most elevation gain or loss per mile. We model this problem as a connected undirected graph $G = (V, E)$. Each vertex is associated with some height $h(v)$ and each edge is associated with some positive distance $d(e)$.

Let a and b be our starting and ending points. We assume $h(a) = h(b) = 0$.

Now, we define an "up-down path" as a path where there exists some vertex t such that all vertices we visit from a to t have increasing heights, and all vertices we visit from t to b have decreasing heights. Metaphorically, an "up-down path" represents a hill, where t is the very top of the hill.

Next, we define the "difficulty" of the up-down path has:

$$\text{difficulty} = \frac{\text{h(t)}}{\text{sum of the lengths of the edges in path}}$$

That is, the difficulty is the height of the hill divided by the total horizontal length we end up walking.

Design an algorithm that finds an up-down walk with the maximum possible difficulty between two nodes a and b. If there does not exist an up-down path, throw an exception.

Solution:

- Construct a directed graph $G' = (V, E')$ where for each $e \in E$ where $e = (u, v)$, there is a directed edge $e' = (u, v)$ in E' only if $h(u) < h(v)$.
- Run Dijkstra's algorithm starting on a within graph G'. Let d_1 and π_1 be the shortest path distances and predecessors.
- Run Dijkstra's algorithm starting on b within graph G'. Let d_2 and π_2 be the shortest path distances and predecessors.
- If $d_1(v) = \infty$ or $d_2(v) = \infty$ for all $v \in V$ (ignoring a and b, throw an exception
- Otherwise, let t' be whatever node that maximizes the quantity $h(t)d_1(h) + d_2(v)$.
- Construct the shortest paths p_1 and p_2 using π_1 and π_2 respectively.
- Return p_1 concatenated with the reverse of p_2.

6. Designing algorithms: Pathfinding in mazes

(a) Suppose we are trying to design a maze within a 2d top-down video-game. The world is represented as a grid, where each tile is either an impassable wall, an open space a player can pass through, or a *wormhole*. On each turn, the player may move one space on the grid to any adjacent open tile. If the player is standing on a wormhole, they can instead use their turn to teleport themselves to the other end of the wormhole, which is located somewhere else on the map.

Now, suppose the there are several coins scattered throughout the map. Your goal is to design an algorithm that finds a path between the player and some coin in the fewest number of turns possible.

Describe how you would represent this scenario as a graph (what are the vertices and edges? Is this a weighted or unwighted graph? Directed or undirected?). Then, describe how you would implement an algorithm to complete this task.

Solution:

We can represent this as an undirected, unweighted graph where each tile is a vertex. Edges connect tiles we can travel between. When we have a wormhole, we add an extra edge connecting that wormhole tile to the corresponding end of the wormhole.

Because it takes only one turn to travel to each adjacent tile, there is actually no need to store edge weights: it costs an equal amount to move to the next vertex.

All paths are bidirectional, so we can also use an undirected graph. (If there are paths or wormholes that are one-way, we can switch to using a directed graph).

To find the shortest path, we can run BFS starting with the player and stop the moment we hit a coin.

(We can use other algorithms like DFS or Dijkstra's algorithm if we're careful, but those would be less efficient.)

(b) **Challenge:** Suppose the map now stuffed with a huge number of players. We now want an algorithm that finds the shortest path between every single player and the closest coin.

A naive way of doing this would be to run the same algorithm from the previous part on every single player. However, this would be prohibitively expensive. Design a more efficient algorithm that does the same thing.

When designing your algorithm, you may assume that the map is relatively small/that there are only a few coins, relative to the number of players.

Solution:

One idea is to start at each coin and run BFS or DFS radiating outwards. Every time we visit a tile, save the distance from it to the coin along with a backpointer that leads towards the coin.

If we visit a tile that already has a distance associated with it, see if that distance is higher or lower then the current one. If it's higher, replace it with the shorter distance/the new backpointer. Otherwise, leave it alone.

Now, once we've computed a distance and backpointer for every tile, we can simply just loop through each player, check the backpointer for the tile they're standing on, and move them in that direction.

We now need to run BFS/DFS just once over the entire map, rather then per each player.

Once a player find a coin, we can re-run BFS/DFS again and update the backpointers.

(c) **Challenge:** Now, suppose our map contains just a single player again. The player now wants to collect *all* the coins in the fewest number of turns possible. Design an algorithm to do this efficiently.

Solution:

This is basically the traveling salesman problem, which is NP-COMPLETE. We'll discuss what this means in the last week of class, but basically, if you can find an efficient algorithm to solve this problem which doesn't resort to brute force, you should basically be teaching this class/will be instantly rich and famous/will have your name immortalized in history books forever...

Good luck lol

7. Designing algorithms: Planning course prerequisites

(a) Suppose you just got into the department of your choice and are trying to plan out the courses you need to take in order to graduate. You have a list of all available courses, their prerequisites, and whether or not that course is optional.

Now suppose you want to all of the available classes. Design an algorithm that prints out one possible order in which you can take these classes. What is the worst-case runtime of your algorithm? Note: if a class has a prerequisite, make sure you take the prerequisites first.

For the sake of simplicity, you may assume you will take only one class per quarter.

Solution:

Run toposort – this will take $\mathcal{O}(|V| + |E|)$ time.

However, we also know our schedule will contain no duplicate edges or self-loops. This means our graph is actually a simple graph – so $|E| \in \mathcal{O}\left(|V|^2\right)$. This means we can express our runtime as $\mathcal{O}\left(|V| + |V|^2\right) = \mathcal{O}\left(|V|^2\right)$.

That said, in practice, each course is likely to have only a small, constant number of pre-reqs. This means our graph is sparse – it's highly likely that $|E| \in \mathcal{O}(|V|)$. Therefore, our runtime is (most likely) $\mathcal{O}(|V|)$ assuming the department's course listing isn't insane.

(b) What if you only want to take some of the classes? Modify your algorithm so that it accepts a list of all classes you want to take and prints out a schedule where you take only those courses (or prerequisites to those courses). What is the worst-case runtime of your algorithm?

Solution:

There are several possible solutions.

One possible solution is to scan through the graph and retain only the courses we want to take (and their pre-reqs). One way we can do this is to start at each mandatory node and run DFS, following the edges backwards. We then save each node and edge we visit into our new graph.

We then run toposort on this graph.

We visit each node and edge a constant number of times to do the initial filtering, then once again to run toposort. So the final runtime is the same as before.

(c) **Challenge:** What if you can take up to k classes per quarter? Adjust your algorithm so it prints out a schedule that lets you graduate as soon as possible. What is the runtime of your algorithm?

Solution:

It turns out this is also an example of an NP-COMPLETE problem when $k \geq 3$! This means that the best we can do boils down to brute force, if you can figure out a non-brute-force algorithm you deserve fame and glory, etc.

That said, there are also many different *heuristics* for solving this problem – many algorithms that attempt to approximate a reasonable schedule.

It turns out this problem is a fairly important one, and shows up in many different places in computer science. For example, suppose we are trying to schedule tasks on a computer that has 4 cores, where each task may depend on the completion of other pending tasks. In what order should we assign tasks to which processor?

Ideally, we'd want to make use of all four processors as frequently as possible help make our computer speedy.