1. **Applying Algorithms**

Start with #2

(a) Given a directed graph G and a node v in the graph, devise an algorithm that returns whether or not v is part of a cycle in G.

Run BFS, starting from v
if v is added to the queue again, return true (cycle exists)
else once all nodes are seen, return false (no cycle)

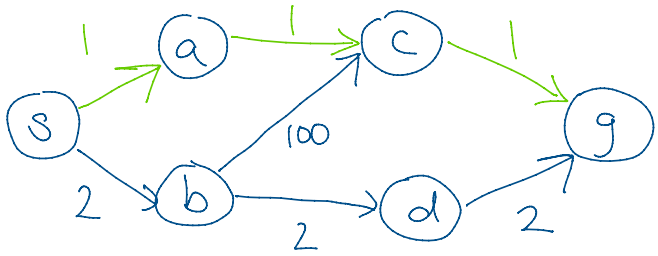
(b) If **True**, give a (short) explanation why. If **False**, give a counterexample:

Assume a connected graph G that we have run Dijkstra on, with start s and goal g. This means each node has its predecessor set correctly.

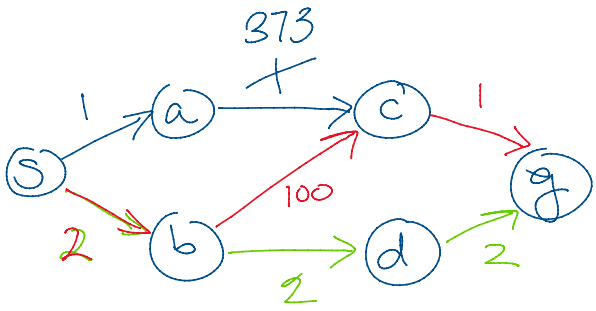
After this, I increase the weight of some edge $u \to v$ on the shortest path between s and g (you don't know by how much). Although my shortest path may no longer be valid, I can look at all of v's incoming edges, select the cheapest one, and set v's new predecessor to be the other endpoint of that cheapest edge.

This sets all of the predecessors correctly, and I have repaired my shortest path despite the edge change.

False



The shortest path from s to g is 3



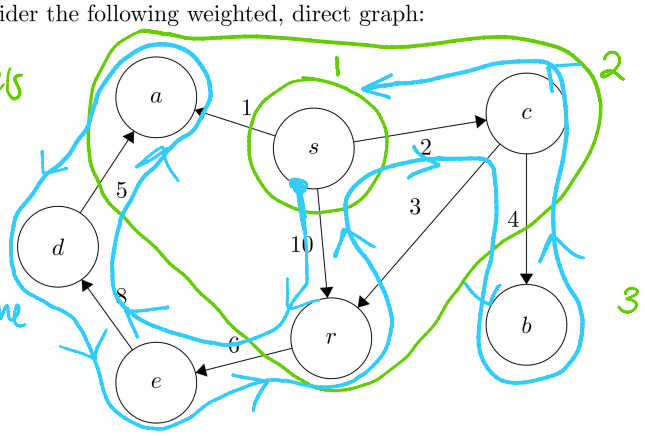
Using the procedure described above results in the red path.
The correct path is the green path.

Therefore, this procedure does not work. The best way to deal with weight changes is to rerun Dijkstra's.

2. **Mechanical Questions** Consider the following weighted, direct graph:

BFS – visits in levels from start

DFS – goes as deep as possible/ traverses outline



(a) Starting from s, what is the path traversed when we add children alphabetically to the Queue using BFS? Ignore the edge weights.

"r" added second time because not visited yet

Queue front: ~~s~~ ~~a~~ ~~c~~ ~~r~~ ~~b~~ ~~e~~ ~~d~~

Visited: s a c r b e d

"a" not added again because already visited

(b) Starting from s, what is the path traversed when we add children alphabetically to the Stack using DFS? Ignore the edge weights.

"a" added second time because not visited yet

Stack as we proceed:

	~~s~~	~~r~~ c a	~~e~~ c a	~~d~~ c a	~~a~~ c a	~~c~~ a	~~b~~ a	~~a~~

Visited: s r e d a c b

"r" not added again because already visited

(c) Starting from s, use Dijkstra's to find the shortest distances to all the other nodes.

Node	Distance	Predecessor	Processed?
s	0	–	✓ 1
a	~~∞~~ 1	s	✓ 2
b	~~∞~~ 6	c	✓ 5
c	~~∞~~ 2	s	✓ 3
d	~~∞~~ 19	e	✓ 7
e	~~∞~~ 11	r	✓ 6
r	~~∞~~ ~~10~~ 5	s/c	✓ 4

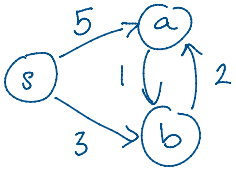
The path to "r" from "c" is only 5 compared to 10 from "s" to "r"

(d) Can we use Prim's or Kruskal's algorithm on this graph? Why or why not?

No, because this graph is weighted. Prim's and Kruskal's do not work on directed graphs.

An MST is defined such that at least 1 node in the MST has a path to all the other nodes in the MST. With directed graphs, adding shortest edges might not be able to accomplish this.

eg.



The correct MST is s → b → a but using Kruskal's we get s → a → b because the first edge chosen is a → b.

3. **Design Decisions**

For each situation, describe a graph that can be used to model it and a graph algorithm that can be used. What are the nodes? What are the edges? Is it weighted or directed?

*There are many possible answers

(a) You are in Bellevue Square looking at the map. Because you had a very large bubble tea earlier in the day, you need to find the closest restroom to your current location and get there as fast as possible.

Nodes: locations → eg corners, ends of hallways
Edges: routes → eg. hallways, escalators
Weighted? Yes → either by time or distance
Directed? Yes if there are one-way hallways / up and down only escalators
No if everything can be accessed either direction
Algorithm: Dijkstra's because it is weighted and we are looking for shortest path

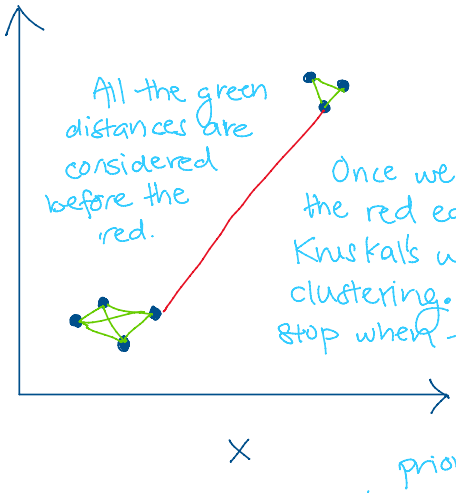
(b) John Wick is at the Continental Hotel and he only has a pencil. He must take down all his enemies before they take him down. John doesn't care how far he has to walk or who he gets to first.

Nodes: Enemies (+ John if he is the starting node)
Edges: Paths to enemies
Weighted? No, he doesn't care about distance
Directed? No
Algorithm: DFS, BFS → traverses all the nodes
not Prim's or Kruskal's because graph is not weighted

(c) Given a set of (x, y) points that visually form distinctly separated clusters, how do you cluster them into these groups using a modified graph algorithm? We might be able to see the groups by eye, but computers are unable to do so unless we tell them how to 'see' using the algorithm.

Nodes: points
Edges: Pair-wise distances
Weighted? Yes → distance
Directed? No
Algorithm: Kruskal's
but modified so that we stop adding new edges if
1) we reach "k" clusters
and/or
2) The edges are longer than the threshold distance.

This clustering method is frequently used for analyzing gene expression data. Cells can be grouped together based on expressed genes so that healthy cells and cancer cells can be separated into groups.



All the green distances are considered before the red.

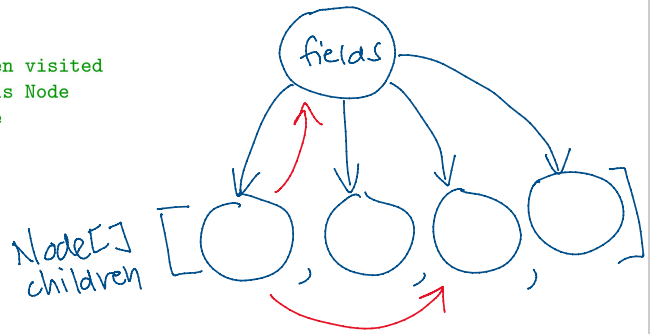
Once we reach the red edge in Kruskal's we can stop clustering. Or we can stop when there are 2 clusters, but that requires prior knowledge of how many clusters there should be.

4. **Runtime Analysis**

Consider this recursive code:

```
1   // The Node class
2   public Node {
3       public boolean visited; // True if this node has been visited
4       public final Node[] children; // The children of this Node
5       public final value; // The value stored in this node
6   }
7
8   // The recursive method
9   public static boolean mystery(Node node, int val) {
10      node.visited = true;
11      if (node.value == val) {
12          return true;
13      }
14      Node[] children = node.children;
15      boolean b;
16      for (int i = 0; i < children.length; i++) {
17          if (children[i].visited != true && mystery(children[i], val)) {
18              return true;
19          }
20      }
21      return false;
22  }
```

int



Although this looks like a tree, it might not be because the Node[] can contain references to ancestors, siblings, or grandchildren.

↳ adjacency list

(a) How is this graph represented? What advantages/disadvantages does it have to using a Dictionary implementation?

Similar to how LinkedLists or trees are represented.

+ saves space (still has adjacency list as Node[] but no empty space in hash table)

− hard to change graph (eg. insert/delete edges and nodes)

(b) What is the recursive method doing?

Checking if the graph contains a value

(c) Which graph algorithm is this most similar to and why?

DFS

↳ recursion is called on each child of a node, so we traverse a far as possible for one child before moving on to the next child.

(d) What is the worst-case and its runtime?

Not there or last node searched → O(V+E)

where V is the number of nodes
E is the number of edges.

(e) What is the best-case and its runtime?

First node is the value → O(1)

(f) Is this version more or less efficient (time and/or memory) compared to a non-recursive version using the Dictionary implementation of graphs? Why? (Hint: how would you implement this method if this graph was a Dictionary?)

Dictionary implementation has hashing so nodes can be directly found in O(1) time. This implementation has to traverse the graph node by node until it is found.