Jared Rafola HW3

Q1) Given: Undirected graph → G
Edge → e

Goal: Find if G has a cycle containing e in linear time

Depth First Search (DFS)

↳ Visits all vertexes and marks them as not visited

↳ Visits all vertexes again and if vertex v is not visited, explore the vertex, checking all adjacent points reachable from v.

Step 1: Run DFS normally from u to v
If v isn't visited, but e is, e ~~Ems~~ is contained in a cycle.

Step 2: If step one was false, run DFS in reverse, meaning from v to u.
If u isn't visited, but e is, e is contained in a cycle. If not, e is not contained in a cycle.

Runtime: $O(|V|+|E|)$
The runtime of DFS is $O(|V|+|E|)$ and we perform DFS twice.
$O(|V|)$ checks all vertices in the visited is false step.
$O(|E|)$ handles the explore portion.

Justification: This works, because we are checking for back edges which are necessary to create a cycle.

Jared Rotola HW3

Q2) Given: graph of the city → $G = (V, E)$
V is the intersections
E is one way roads

a) Since the streets are one way we have a directed graph. The claim is that there is a way to legally drive from any intersection to another. This falls under the relation for strongly connected component: Two nodes u and v ~~of~~ of a directed graph are connected if there is a path from u to v and v to u. This can be checked in linear time by using DFS (Depth First Search)

b) Using the same directed graph, treating the townhall as s, we can perform DFS again for an intersection (vertex) v and repeat for all vertices. The property will hold true if s cannot reach a vertex with a different connected component. If the vertex has a different connected component, it cannot make it back to s.

Runtime: $O(|V| + |E|)$ → DFS is run for both situations.

Justification: Directed graphs are connected if there is a path from u to v and from v to u.

Jared Rafala HW3

Q3)

Given: Graph of roads → $G = (V, E)$

V → Cities

E → Roads

l_e → length of Road E

E' → list of city pairs

Goal: Find the road that creates maximum decrease in driving time

* Dijkstras *

↳ Takes input of graph, vertices and edge lengths

↳ Finds shortest path from starting point to every vertex by attempting every path and exiting once the distance becomes greater.

Step 1: Run Dijkstras using s as starting point

Step 2: Run Dijkstras using t as starting point.

* Now we have all possible distances between s and t *

Step 3: Cycle through E' and choose the path that creates the shortest distance between s and t

Runtime: $O(|E| + |E| \log |V|)$

$O(|E|)$ → Searching E'

$O(|E| \log |V|)$ → Dijkstras

dijkstra's

Justification: We run dijkstra's twice for the undirected graph to precompute the shortest distances from s and t so we can search for the closest pairs of cities from E' to s and t.

Scanned with CamScanner

Jared Raiola HW3

Q4) Given: Weighted graph → $G = (V, E)$
positive weights → c_i for $i \in E$

Goal: Decide if input $e = (v, v) \in E$ with weight c_e is part of an MST of G in linear time.

Step 1: Remove all weighted edges having a greater weight than c_e

Step 2: With the new graph, run DFS (Depth First Search) to look for a cycle.
* Cycles break tree properties, so if a cycle exists, it cannot be a tree *

Step 3: If no cycle is found: v cannot be reached from v with max weight being c_e. There is a minimum spanning tree containing e.

Runtime: $O(|E| + |V|)$
$O(|E|)$ → Traversing all edges to remove those with higher weight than c_e.
$O(|E| + |V|)$ → DFS

Justification: Since we do not have to find the MST, just the existance of one, we only have to prove that a cycle does not exist to uphold the properties of trees being acyclic. This means we can use DFS to check for cycles.