Caio Brighenti
COSC 302 - Analysis of Algorithms – Spring 2019
Assignment 5

1. **Dijkstra's**

   We provide an algorithm for finding the edge $e' \in E'$ that when added to graph G minimizes the distance s, t, where s and t are the two cities to be considered. Adding an edge u, v that minimizes this path will create a path s, u, v, t such that this path is the smallest possible path starting at s and ending at t. Thus, we look more formally for a path that minimizes the distance of the path s, u, the distance of the path u, v,, and path v, t. We leverage Dijkstra's to provide the following algorithm.

```
1  addRoad(G,E'){
2
3  S[] = Dijkstra(G,s)
4  T[] = Dijkstra(G,t)
5
6  min_path = ∞
7  for edge e' = u,v in E':
8    min_path = min(min_path, S[u] + l_e' + T[v])
9    min_path = min(min_path, S[v] + l_e' + T[u])
10
11 return min_path
12 }
```

   This algorithm leverages Dijkstra's to calculate the shortest path from s and t to each other city in the network. Since any path s, t passing through any edge $e' = u, v$ must pass through u and v, then the shortest path *must* take the shortest distance s, u and v, t respectively. Thus by applying this principle on each iteration, and keeping track of the overall minimum, this algorithm finds the best edge to add. The problem requires that we give an efficient algorithm, so we show that this runs efficiently. First, this algorithm runs Dijkstra's twice. From this we have $O(2(E+V)\log(E))$, as this is the running time of Dijkstra's multiplied by two. Then, we have a for loop that runs E' times, each time performing a constant operation. Thus this adds a $O(E')$ term to the overall running time. Thus, the overall running time is $O(2(E+V)\log(E)) + O(E') \in O((E+V)\log(E))$. This is an efficient algorithm.

2. **Let T be an MST of G. Given a connected subgraph $H \in G$, show that $T \cap H$ is contained in some MST of H.**

   We proceed by contradiction. Assume that $T \cap H$ is not in a MST of H. Then, there must exist two partitions S and V such that $V = H - S$, where the lightest edge e crossing S, V where $e \in T$ is *not* the minimum edge in H that crosses S, V. By consequence, there must be another edge e' that crosses S, V and such that e' is lighter than e, which can thus replace e in a MST. Then, T cannot be an MST of G, as it does not contain e'. This violates what we know of T, therefore T cannot be an MST of G but not of H.

3. **Algorithm for MST using heaviest edge in cycle property**

   (a) Prove this property

   We proceed by a direct proof via the cut property. Let C be a cycle in G. Let S and V be two partitions of G such that $V = G - S$. Let e be an edge that crosses the cut, and is the heaviest edge in C. As C is a cycle, there must be another edge e' such that e' also crosses the cut and is lighter than e. This is because in any cycle, if there is an edge $e = u, v$, then removing e from the graph does not disconnect u and v. Thus, e cannot be in an MST given that there will always be an edge e' that also crosses the cut and is lighter.

(b) Prove this algorithm

Consider a graph G that has no cycles. Then, G must be a tree, as there is only one path to each node. Thus, G is by definition its own MST. This is because there is only a single way to connect all nodes, as no edges are redundant. Let G' be a graph with cycles. Let S be a set of edges such that $S \in G.E$. Let T be an MST of G. Additionally, let $T = G.E - S$. Thus, we can reach a minimum spanning tree T of G by removing each edge in S. If S is definitionally the set of edges *not* in the MST, then it is clear how removing each of those edges from G will produce an MST. It is left to show how the algorithm identifies each edge in S.
Applying the property shown in a) will reveal if an edge is in S. By repeatedly applying this property, each time removing the identified edge $s \in S$, the algorithm will eventually remove every cycle from the graph. When this happens, we will be left with the tree T, as we have removed each edge in S. Thus, by leveraging the fact that a graph with no cycles must be an MST, and the property shown in a), the algorithm appropriately finds an MST of G.

(c) **Give a linear algorithm for finding if there is a cycle in G through edge e**

Let $e = u, v$ be an edge in G. If e belongs to a cycle, then there exists a path between u and v that does not include e. We leverage this fact to create a DFS-based algorithm to determine whether e is in a cycle as follows. The explore function called in the algorithm is assumed to be a simple DFS, which collects seen nodes in the array $visited[]$.

```
1 inCycle(G,e){
2   explore(G-e,u)
3   return visited[v]
4 }
```

This algorithm relies on the very simple principle elaborated above. If we remove $e = u, v$ from G, and there still exists a path from u to v, then edge e must be in a cycle. Thus, the algorithm does exactly this: it removes e from G, performs a DFS starting at u, and returns true if the search reached v. The running time of this is just the running time of DFS, which is $O(E + V)$. This is therefore a linear time algorithm.

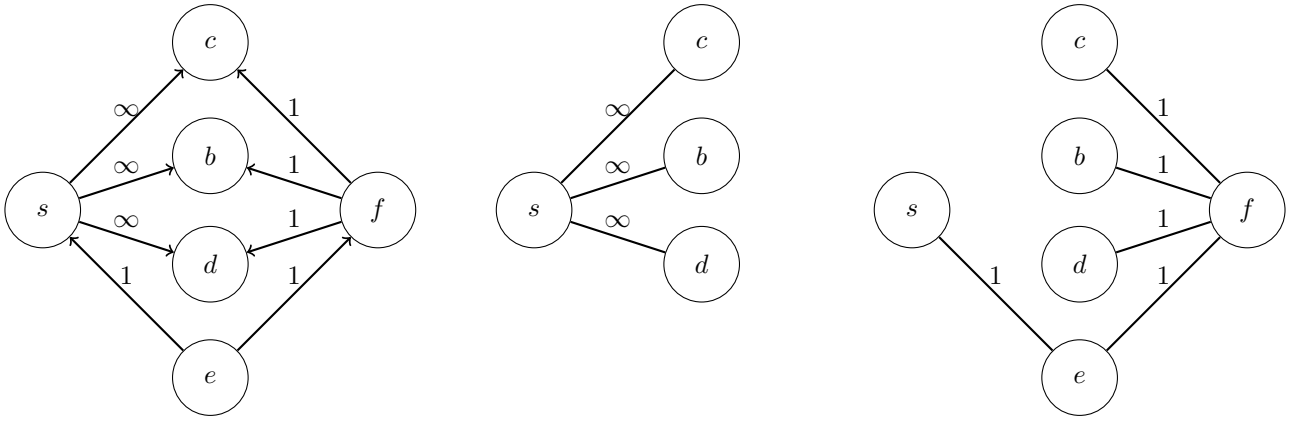
(d) **Show the running time of the algorithm in terms of E**

This algorithm has two main parts: the sorting component, and the for loop. We show each invidually, and then add the results. Firstly, the sorting depends on the algorithm used. We assume an efficient implementation with running time $O(E \log(E))$, as this is typically the best we can obtain using algorithms such as quicksort.
We now show the running time of the for loop. Clearly, the loop runs E times, as it is a loop over the number of edges. In each loop, we look for whether a cycle exists using our algorithm provided in part c. As this algorithm has $O(E+V)$ complexity, the overall complexity of the loop is $O(E(E + V))$. This is because all other operations are constant.
We combine the running time of the sort component with the for loop to obtain $O(E \log E) + O(E(E + V))$. As this is a connected graph, we also know that $V \leq E - 1$, meaning we can replace V with E. Thus, we have $O(E \log E) + O(E(E + V)) = O(E \log E) + O(E(2E)) = O(E \log E) + O(E^2) \in O(E^2)$. This is the overall running time of this algorithm.

4. **Is it possible for a tree of shortest paths from s in G not to share any edges with MST T?**

The answer for this question is different for directed and undirected graphs. We begin with the undirected case.
In the undirected case, this is impossible. Let T be an MST of G and S be the tree of shortest paths from s. S must contain an edge from s to each neighbor of s. One of these edges *must* also be in T, otherwise T is not an MST as it would not connect s, and thus not span the entirety of the graph. Thus, this is not possible in the undirected case.
However, we can have this scenario in the directed case. We provide an example graph, followed by

depictions of S and T.



Respectively, these diagrams are an example of a directed graph G, the tree S of shortest paths from s, and the unique MST T of G. We can clearly see that none of the edges in S are present in the MST T, despite being the shortest possible paths from s. This is because each node reachable from s is also reachable in a shorter path from another node. Thus, this scenario is clearly possible for directed graphs.