Assignment 3 Solutions

COMP 3804, Fall 2018
Michael Kuang 101000485

November 19, 2018

1 Guidelines

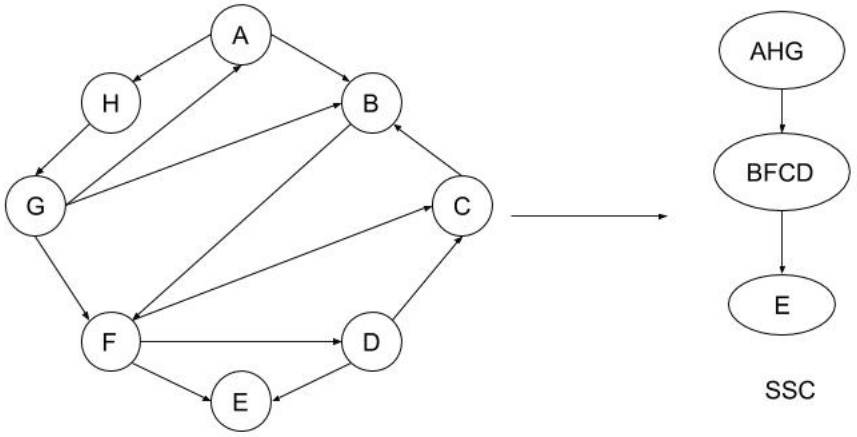
General guidelines are as follows:

1. Since we are only accepting assignments via CU-Learn, no late submissions will be entertained after the cut-off time & date.
2. Please write clearly and answer questions precisely. It is your responsibility to ensure that what is uploaded is clearly readable. If we can't read, we can't mark!
3. Please cite all the references (including web-sites, names of friends, etc.) which you used/consulted as the source of information for each of the questions.
4. All questions/problems carry equal marks.
5. When a question asks you to design an algorithm - it **requires** you to
   (a) Clearly spell out the **steps** of your algorithm in pseudo code.
   (b) **Prove** that your algorithm is correct
   (c) **Analyze** the running time.
6. You can assume that a graph $G = (V, E)$ uses adjacency list representation.

2 Problems

1. Find strongly connected components of the graph in the Figure.

   Solution



2. Let $G = (V, E)$ be a directed graph. We say that G is *semi-connected* if for every pair of distinct vertices $u, v \in V$, we have that there is a directed path from u to v or there is a directed path from v to u in G. Given G in the adjacency list representation, design an algorithm running in $O(|V| + |E|)$ time to determine whether G is semi-connected. Note that this is similar to the question which we had in the mid-term, where we needed to check whether a directed acyclic graph is semi-connected. As a hint, again design an algorithm for the directed acyclic graphs. Think of a general directed graph, in terms of directed acyclic graphs of its strongly connected components. [Consult Section 3.4 of the text-book.] Is the graph in the above Figure semi-connected?

   Solution

   The SCC() function returns the strongly connected components of the graph. The DFS() function does a depth-first search in the graph, labelling the post numbers for each vertex. The TopologicalSort() function topologically sorts and returns the vertices starting from the largest post number in decreasing order.

Algorithm 1 Determining if G is semi-connected

```
1: procedure IS-SEMICONNECTED(G)
2:     G' ← SCC(G)                          ▷ Obtain a SCC from graph G
3:     DFS(G')                              ▷ DFS G', labelling the post numbers
4:     VSorted ← TopologicalSort(G')        ▷ Return topologically sorted array of vertices
5:     for i = 0 to length(VSorted) - 2 do
6:         if (VSorted[i], VSorted[i + 1]) ∉ G.E then
7:             Return False
8:     Return True
```

Proof of Correctness

Suppose that the SCC(), DFS() and TopologicalSort() are correct. In order to determine if the graph is *semi-connected*, we observe that there needs to be a single path that goes through all vertices, or in other words there is an edge between every consecutive pair of vertices. We can do this by topological sort. For each vertex in the topologically sorted array, it must have a directed edge to the next vertex in that sorted array. So, assume that we have a topologically sorted array $VT = (v_1, v_2, v_3, ...v_n)$ of graph G, then the edge(v_i, v_{i+1}) must be present in $G.E$ for the graph to be *semi-connected*, otherwise it is not. In the algorithm, we first find the strongly connected components so we then we have a DAG. This is important because the SCC has no cycles, and we know that there are pathes from u to v in cycles. Then we depth-first search the strongly connect components, labelling the pre and post values of each component, so now we can topologically sort the components. Line 5-7 loops through the topologically sorted array of vertices and checks if edges $(v_i, v_{i+1}) \in G.E$ for all vertices in $G.V$ except the last vertex. Therefore this algorithm is correct.

Time Complexity Analysis

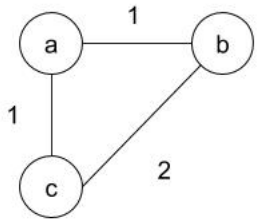
Line 2 is $O(V + E)$, line 3 is $O(V + E)$ and line 3 is $(O(V + E)$. The for loop for lines 5-7 runs at $O(V)$ because the worst case is that the graph has no cycles meaning each component is a vertex in the $G.V$. Therefore the final time complexity of the algorithm is $O(V + E)$.

3. This question is based on the *cut lemma* for minimum spanning trees. Let $G = (V, E)$ be a connected graph where each edge has a positive weight. If for any cut of G, there is a unique edge in the cut of minimum weight then show that minimum spanning tree of G is unique. Show that the converse may not be true using an example. I.e., construct a graph G that has a unique minimum spanning tree, but there are cut(s) in G containing multiple edges having the minimum weight. However, if the cut in G contains multiple edges having minimum weights, then this is not true as shown in the following figures.

   Solution

   We will prove by contradiction. Let's suppose the contradiction that graph G has two unique minimum spanning trees T and T'. Let (u, v) be an edge in T but not in T'. Removing edge (u, v) cuts the tree T into two components and let T_u and T_v be the

vertices in the component containing u and v respectively. Consider the cut (T_u, T_v) and let (x, y) be the unique minimum edge crossing the cut. If $(x, y) \neq (u, v)$ then $w(x, y) < w(u, v)$ and the spanning tree $T - \{(u, v)\} \bigcup \{(x, y)\}$ has a smaller sum of edge weights than T which leads to a contradiction because if T was originally an MST, then by definition the edge (u, v) should be the minimum edge that is part of T. Therefore, T must equal T' so there is only one unique spanning tree.



Conversely, if we consider the graph above, there is a unique MST that contains edges (a, b) and (a, c), however the cut $(\{a\}, \{b, c\})$ doesn't have a unique minimum edge crossing the cut.

4. Consider a connected graph $G = (V, E)$ where each edge has a non-zero positive weight. Furthermore, assume that all edge weights are distinct. Using the cut property, first show that that for each vertex $v \in V$, the edge incident to v with minimum weight belongs to a Minimum Spanning Tree (MST). Can you use this to devise an algorithm for MST - the above step identifies at least $|V|/2$ edges in MST - you can collapse these edges, by identifying the vertices and then recursively apply the same technique - the graph in the next step has at most half of the vertices that you started with - and so on. What is the running time of your algorithm?
Note that for an edge $e = uv$ in the graph $G = (V, E)$, *identifying* vertex u with v or *collapsing* e is the following operation: Replace the vertices u and v by a new vertex, say u'. Remove the edge between u and v. If there was an edge from u (respectively, v) to any vertex w ($w \neq u$ and $w \neq v$), then we add an edge (with the same weight as of edge uw (respectively, vw)), between the vertices u' and w. This transforms graph G to a new graph $G' = (V', E')$, where $|V'| < |V|$ and $|E'| < |E|$. Note that G' may be a multigraph (i.e., between a pair of vertices, there may be more than one edge). For example, if uv, uw, and vw are edges in G, then G' will have two edges between u' and w when we identify u with v. We can transform G' to a simple graph by keeping the edge with the lower weight among uw and vw as the representative for $u'w$ for the computation of MST.

Solution

Assume that $X \subseteq T$ and T is some MST of G. We make a cut $(S, V - S)$ that respects X. Let's assume that $S = (\{v\})$ where $v \in V$, then the cut we make is $(\{v\}, \{V - v\})$. Let e be the minimum weighted edge crossing the cut. By cuts property, edge e belongs to an MST of G.

Algorithm 2 Finding MST

```
1:  procedure MST(G)
2:      F ← G.V ▷ Initialize F as a set of one-vertex trees, one for each vertex of the graph
3:      while F has more than 1 component do
4:          Determine the connected components of F and label each vertex of G by its component
5:          Initialize the minEdge for each component of F to be ∞
6:          for each edge(u, v) in G do
7:              if ComponentLabel(u) ≠ ComponentLabel(v) then
8:                  if edge(u, v) < the minEdge of component of u then
9:                      Set edge(u, v) as minEdge for the component of u
10:                 if edge(u, v) < the minEdge of component of v then
11:                     Set edge(u, v) as minEdge for the component of v
12:         for each component in F do
13:             Add minEdge(component) to F
14:     Return F
```

Proof of Correctness
Earlier we proved that that for each vertex $v \in V$, the edge incident to v with minimum weight belongs to a minimum spanning tree. The algorithm above builds upon this idea. We start off with a set of components, one vertex for each component of the graph which is seen on line 2. Inside the while loop, we first determine all the connected components, then for each connected component we find the minimum edge incident to. This while loop terminates once we have only 1 component, and this component is therefore an MST for G.

Time Complexity Analysis The initialization of F on line 2 takes O(V) time. The while loop on line 3 will loop $log(V)$ times because the size of the components is at worst case halved each time. The inner for loop runs E times so the total running time for the for loop takes $O(E\ logV)$. Therefore the final time complexity for this algorithm is $O(E\ logV)$.

5. Suppose you are given a set S of n distinct points in the plane. Let A and B represents a partition of S, i.e. $A \subset S$, $B \subset S$, $S = A \cup B$, and $A \cap B = \emptyset$. Define the distance between A and B, denoted by $d(A, B)$, as the minimum among Euclidean distances between pair of points, where one point is from A and the other from B, i.e.

$$d(A, B) = \min_{a \in A, b \in B} |ab|$$

Our task here is to find a partition of S into two non-empty sets A and B that maximizes $d(A, B)$. For this, we define a complete graph $G = (V, E)$ on n vertices and $\binom{n}{2}$ edges on these points as follows. Each vertex in V represents a distinct point of S, and there is an edge between every pair of (distinct) vertices, where the weight of an edge $e = (u, v)$ is Euclidean distance between the points corresponding to u and v. Consider a minimum spanning tree T of G. Let e be the most expensive edge in T (i.e. e is the last edge added to T by Kruskal's algorithm). Let V_1 and V_2 be the two sets of vertices in the connected components obtained after the removal of e from T. Show that the points corresponding to V_1 and V_2 forms the required partitioning of S. (Recall that Euclidean distance between two points $a = (3, 5)$ and $b = (4, 2)$ is $|ab| = \sqrt{(3-4)^2 + (5-2)^2} = \sqrt{10}$.)

Solution

Let's assume edge e is the last edge added to T, an MST of G, by Kruskal's algorithm and the removal of edge e gives two vertex sets A and B which maximizes $d(A, B)$. Let's assume a contradiction that there is an edge e' that satisfies $d(A', B')$ where A' and B' is some vertex set and $d(A', B') > d(A, B)$. We know that Kruskal's algorithm adds edges that are part of an MST in non-decreasing order and edge e is the last edge added to T to form an MST. So then, we know that there must be some edge in T that crosses A' and B' and this edge will be called e'', which causes a contradiction. This is because we said earlier that $d(A', B') > d(A, B)$, or for simplicity, the weight of $e' > e$. However, we know that $e'', e \in T$ and e is added last in T, this means $e'' \leq e$

and so $e'' < e'$. Therefore, $d(A', B') \leq d(A, B)$ and so the last edge added in Kruskal's algorithm, e, maximizes the distance.

6. Let $G = (V, E)$ be a connected simple graph, where each edge has a weight of 3. Devise an algorithm, running in $O(|V|+|E|)$ time, for computing shortest path distances from a specific vertex $s \in V$ to all other vertices of G.

   Solution

Algorithm 3 Shortest path distances from s to all other vertices

```
1: procedure SSD(G, s)
2:     BFS(G, S)        ▷ breadth-first search G, starting at s, labelling the depth for each vertex
3:     for each vertex v ∈ G.V do
4:         ∂(s, v) ← v.depth * 3
```

Proof of Correctness
Because each edge weight is the same, 3, we can use breadth first search (BFS) to find all the shortest paths from a specific vertex to all other vertices in G. We know this because the shortest path between two vertices is always the one with the fewest edges required to have a path between them as all edges have the same weight, and so the shortest path distance is defined by the depth of the vertex from the source in a BFS times 3 and this algorithm does exactly that. Therefore this algorithm is correct.
Time Complexity Analysis
The time complexity of this algorithm is $O(V + E)$ because we visit each vertice and edge once.

7. Prove that the distance values extracted from the heap (priority queue) over the entire execution of Dijkstra's single source shortest path algorithm, in a directed connected graph with positive edge weights, is a NON-Decreasing sequence. Where is this fact used in the correctness of the algorithm?

   Solution Referring to the class notes on Dijkstra's single shortest path algorithm, let $u \in Q$ with minimum $d(u)$ value in an iteration of the while-loop. Before the for-loop is executed, $d(u) \leq d(v)$ for all vertices in Q as $d(u)$ is selected as the minimum. We see that in the for-loop, some of the $d(v)$ values may decrease from its current value to $d(v) = d(u) + weight(edge(u, v)) \geq d(u)$. So we know that $d(v) \geq d(u)$ for all $v \in Q$. This proves the claim 3 that the minimum d-value in Q never decreases and as such the distance values in the heap over the entire execution of Dijkstra's single source shortest path algorithm is a non-decreasing sequence. This fact is used for each iteration in $v_1, v_2, ..., v_k$ to show that we keep the vertex that is larger than the previous vertex in the queue, Q. For example in v_1, we see that $d(v_2) > d(v_1)$, so therefore $v_2 \in Q$.

8. In the summer vacation, you decided to travel to various communities in Northern Canada by your favorite ATV (All-Terrain Vehicle). Each of the communities you want

to visit is represented as a vertex in your travel graph (a total of $|V|$ communities). Moreover, you are provided with distances between all pairs of communities. Think of your input graph as a complete graph (i.e. every pair of vertices are joined by an edge), and the weight of an edge, say $e = (uv)$ is the distance between the community u and v. Since this is in far North, and the routes between communities are not used that often, the gas stations are only located in communities (there are absolutely no gas stations which are outside a community). Furthermore, we can assume that each community has at least one gas station. Once you completely fill up the tank of your ATV, it has an upper limit, say of Δ kilometers, which it can travel, and to travel any further it needs to fill up (which means at that point it needs to be in a community!). You need to answer the following two questions

(a) First design a method, running in $O(|V| + |E|)$ time, which can answer whether is there some path which your ATV can take, so that you can travel between two particular communities, say s and t. It is obvious that if the distance between s and t is at most Δ, then you can travel directly without refuelling. Otherwise, you can travel between s and t, provided there are communities where we can refuel and proceed. [For fun you may like to see whether you can travel from La Loche (in Sask.) to Mandorah (in Northen Territories), when your ATV with full tank can travel at most 100 Kms.]

Solution

Algorithm 4 Finding a suitable path between s and t

1: **procedure** BFS($G, s, t, limit$) ▷ limit denotes the ATV's upper limit it can travel
2: Let G' be the same graph as G but with edges that are $\leq limit$
3: DFS(G', s) ▷ this labels the pre and post numbers for all vertices starting with s
4: **if** $s.pre \leq t.pre$ and $t.post \leq s.post$ **then**
5: Return True
6: Return False

Proof of Correctness

We assume that the DFS() algorithm is correct in that it will perform a depth-first search on graph G, starting on vertex s, and label all the pre and post values for each vertex. This algorithm begins by creating a new graph, G', with the same verticies as G, but with edges that are $\leq limit$. This way, we know that all edges in G' can be travelled on for the ATV. Then, we perform a DFS() on G' starting on s. We know that in order for there to be a path between s and t, the pre and post numbers for s must encompass the pre and post numbers for t. This means that $s.pre \leq t.pre$ and $t.post \leq s.post$ and this is exactly what line 4 checks for. Therefore this algorithm is correct.

Time Complexity Analysis

Line 2 takes $O(V + E)$, line 3 takes $O(V + E)$ and the rest takes constant time. So this algorithm as an overal run time of $O(V + E)$

(b) Design an algorithm running in $O(|E| \log |V|)$ time to determine the smallest value of Δ, which will enable you to travel from s to t. (Please present Pseudocode, correctness, analysis) and use the algorithms discussed in the class/book as black boxes).

Solution

Algorithm 5 Finding a smallest upper limit

```
1:  procedure SmallestBottleneck(G, s, t)  ▷ A modification to Dijkstra's algorithm, refer to class notes
2:      for v ∈ V do
3:          bottleneck(v) ← ∞
4:      bottleneck(s) ← 0; S ← ∅; Q ← G.V
5:      while Q ≠ ∅ do
6:          u ← Extract-Min(Q)
7:          DELETE u from Q
8:          INSERT u in S
9:          for each vertex v such that (u, v) ∈ E do
10:             if bottleneck(u) < wt(u, v) then
11:                 bottleneck(v) ← MIN(bottleneck(v), wt(u, v))
12:             else
13:                 bottleneck(v) ← MIN(bottleneck(u), bottleneck(v))
14:     Return bottleneck(t)
```

Proof of Correctness

The algorithm is a slight modification to Dijkstra's algorithm. We see that everything is the same other than instead of keeping track of the shortest path distance for each vertex, we keep track of the smallest bottleneck path for each vertex. So, like how we proved the correctness of Dijkstra's algorithm for single shortest path distance, the same proof can be applied for this algorithm as well. The modified lines are in the for loop, line 10-13. We see here for each v such that the edge $(u, v) \in E$, we first check if the previous bottleneck in the path which is $bottleneck(u)$ is smaller than the incoming weighted edge, $wt(u, v)$, to v. If that is the case, then we know that the bottleneck for v must be the minimum between it's current bottleneck value and the $wt(u, v)$. Otherwise if the incoming weight is smaller than $bottleneck(u)$, we take the minimum between $bottleneck(v)$ and $bottleneck(u)$. From this, we can see that the claims made in the proof for Dijkistra's algorithm can be made here and the rest can follow. Therefore this algorithm is correct.

Time Complexity Analysis

The running time for Dijkstra's algorithm is $O(E\ Log(V))$