COMP 3804 — Solutions Assignment 3

Algorithm DFS(G):
for each vertex v
do $visited(v) = false$
endfor;
$clock = 1$;
for each vertex v
do if $visited(v) = false$
  then EXPLORE(v)
  endif
endfor

Algorithm EXPLORE(v):
$visited(v) = true$;
$pre(v) = clock$;
$clock = clock + 1$;
for each edge (v, u)
do if $visited(u) = false$
  then EXPLORE(u)
  endif
endfor;
$post(v) = clock$;
$clock = clock + 1$

Question 1: Write your name and student number.

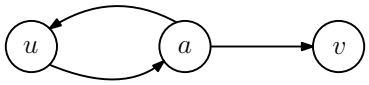
Solution: Salma Paralluelo, 7

Question 2: Let $G = (V, E)$ be a directed graph. After algorithm DFS(G) has terminated, each vertex has a *pre*- and *post*-number. Let u and v be two distinct vertices in V. Assume that the following are true:

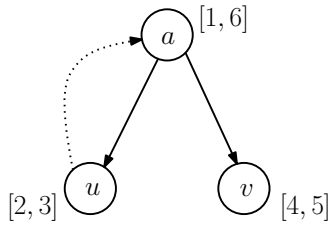
- There is a directed path in G from u to v.
- $pre(u) < pre(v)$.

Professor Lionel Messi claims that, in the DFS-forest, v must be in the subtree of u. Is Professor Messi's claim correct? As always, justify your answer.

Solution: Professor Messi is wrong. Consider the following graph; the adjacency list of a stores, in this order, u and v.



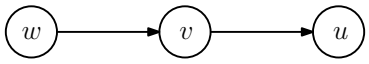
When running DFS on this graph, we consider the vertices in the for-loop in alphabetical order. This gives the following DFS-forest; the dotted edge is a back edge.



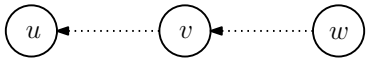
Question 3: Give an example of a directed graph $G = (V, E)$ that contains a vertex v having the following properties:

- v has at least one incoming edge and at least one outgoing edge.
- Consider the DFS-forest after algorithm DFS(G) has terminated. This forest contains a tree containing only the vertex v.

Solution: Here is an example:



When running DFS on this graph, we consider the vertices in the for-loop in alphabetical order. This gives the following DFS-forest consisting of three trees, each tree having one single vertex. Both edges are cross edges.



Question 4: Let $G = (V, E)$ be a directed graph that is given to you using adjacency lists. The vertices of V are numbered arbitrarily as $v_1, v_2, \ldots, v_n$.

For each vertex u of V, let $R(u)$ be the set of all vertices v such that there exists a directed path in G from u to v, and let $Min(u)$ be the smallest index of all vertices in $R(u)$. Thus,

$$Min(u) = \min\{i : v_i \in R(u)\}.$$

Give an algorithm that computes, in $O(|V| + |E|)$ time, the value of $Min(u)$ for all vertices u in V. As always, justify your answer.
Hint: Consider the graph G' obtained from G by reversing the direction of all edges in G. Algorithm DFS is useful for this question.

Solution: We follow the hint and consider the graph G'. We rephrase the problem on G to a problem on G'. For each vertex u of V, let $I(u)$ be the set of all vertices v such that there exists a directed path in G' from v to u. Observe that $v \in R(u)$ if and only if $v \in I(u)$. Therefore,

$$Min(u) = \min\{i : v_i \in R(u)\} = \min\{i : v_i \in I(u)\}.$$

In words, our task is to compute, for each vertex u, the value of $Min(u)$ as the smallest index of all vertices v_i that can reach, in G', the vertex u.

Here is our approach (from now on, we only use the graph G'): We run EXPLORE(v_1). This results in a DFS-tree T_1 rooted at v_1 that contains all vertices u that can be reached from v_1, i.e., $v_1 \in I(u)$. It is clear that for each vertex u in T_1, $Min(u) = 1$. (Note that this is also true for $u = v_1$.)

Consider the smallest index k, such that vertex v_k has not been visited during the call to EXPLORE(v_1). In other words, k is the smallest index such that v_k is not a vertex of T_1. Note that $v_1, v_2, \ldots, v_{k-1}$ are all in T_1, and T_1 may contain vertices having an index larger than k. We run EXPLORE(v_k).

- This results in a DFS-tree T_k rooted at v_k.
- Each vertex u in T_k can be reached from v_k, but not from any vertex in T_1. In particular, u cannot be reached from any vertex among $v_1, v_2, \ldots, v_{k-1}$. Therefore, $Min(u) = k$.
- There can be vertices u in T_1 that can be reached, through cross edges, from v_k. However, for each such vertex u in T_1, we already know that $Min(u) = 1$.
- Conclusion: For each vertex u in the DFS-tree rooted at v_k, we have $Min(u) = k$.

We continue by taking the smallest index ℓ, such that vertex v_ℓ has not been visited during the calls to EXPLORE(v_1) and EXPLORE(v_k). In other words, ℓ is the smallest index such that v_ℓ is not in T_1 and not in T_k. Each of the vertices $v_1, v_2, \ldots, v_{\ell-1}$ is in T_1 or in T_k. We run EXPLORE(v_ℓ). This results in a DFS-tree T_ℓ rooted at v_ℓ. By the same reasoning as above, for each vertex u in T_ℓ, we have $Min(u) = \ell$.

Based on this, we get the following algorithm.

Step 1: Construct the graph G' by reversing all edges in G. In one of the tutorials, it was shown that this can be done in $O(|V| + |E|)$ time.

Step 2: Run algorithm DFS(G'). In the for-loop of this algorithm, visit the vertices in increasing order of their indices. We have seen in class that this can be done in $O(|V| + |E|)$ time.

Technical detail: It may happen that the set of vertices is not given in sorted order of their indices. If this is the case, then we use bucket-sort to obtain the sorted order in $O(|V|)$ time: Initialize an array $A[1 \ldots |V|]$. For each vertex v_i, set $A[i] = v_i$.

Step 3: For each tree T in the DFS-forest:

- Let v_k be the root of T.
- Traverse T and, for each vertex u in T, set $Min(u) = k$.

The time for Step 3 is $O(|V|)$.

Question 5: Let $G = (V, E)$ be an undirected connected graph in which each edge $\{u, v\}$ has a weight $wt(u, v)$. Consider the following algorithm:

- Let $G' = G$.
- While G' contains a cycle:
  - Let C be an arbitrary cycle in G'.
  - Let e be an edge of C whose weight is maximum.
  - Delete the edge e from G'.
- Return the graph G'.

Prove that the output of this algorithm is a minimum spanning tree of the input graph G.

Solution: The graph G' that is returned is connected and does not have any cycles. Therefore, G' is a tree. From now on, we denote this tree by T. Thus, we have to show that T is a minimum spanning tree of G.

Below, we assume that all edge weights are distinct.

Let T' be an arbitrary spanning tree of G with $T \neq T'$. We will show that T' is not a minimum spanning tree of G.

Let e be an edge of T that is not in T'. Since e is an edge in T, it is not a maximum-weight edge of any cycle in G. (Otherwise, e would have been removed and would not be in T.)

We add e to T', which creates a cycle C'. Let e' be an edge of C' whose weight is maximum. Note that $e' \neq e$. We delete e' (but keep e), resulting in a spanning tree T''. Observe that

$$wt(T'') = wt(T') + wt(e) - wt(e') < wt(T').$$

Therefore, T' is not a minimum spanning tree of G.

We have shown: Any tree not equal to T is not a minimum spanning tree. Therefore, T is a minimum spanning tree.

Question 6: Let $G = (V, E)$ be a directed connected graph in which each edge (u, v) has a weight $wt(u, v)$. Let s be an arbitrary source vertex. Assume that the following are true:

- Edges that leave s may have negative weights.
- All other edges have positive weights.
- There is no cycle with negative weight.

Professor Justin Bieber claims that Dijkstra's algorithm correctly computes, for each vertex v, the length of a shortest path from s to v.

Is Professor Bieber's claim correct? As always, justify your answer.

Solution: Professor Bieber is correct. If all edges leaving s have positive weights, then the claim follows from the correctness of Dijkstra's algorithm. From now on, we assume that there is at least one edge leaving s that has a negative weight.

Let W be a sufficiently large number such that for each edge (s, v), $wt(s, v) + W > 0$. Let $G' = (V, E)$ be the graph in which each edge (s, v) has weight $wt'(s, v) = wt(s, v) + W$. For each other edge (u, v), $wt'(u, v) = wt(u, v)$. Note that all edge weights wt' are positive. Thus, Dijkstra's algorithm computes, for each vertex v, the length $\delta'(s, v)$ of a shortest path, using the weights wt', from s to v.

For any vertex v, let $\delta(s, v)$ be the length of a shortest path, using the weights wt, from s to v. Since the original graph G does not have cycles with negative weight, such a shortest path contains exactly one edge leaving s. This implies that

$$\delta'(s, v) = \delta(s, v) + W.$$

Run two versions of Dijkstra's algorithm in parallel, one version uses the weights wt, whereas the other version uses the weights wt'. The first version uses variables $d(v)$, and the second version uses variables $d'(v)$. Then, $d(s) = d'(s) = 0$. For every vertex $v \neq s$, we have, at any moment, $d'(v) = d(v) = \infty$ or $d'(v) = d(v) + W$. In other words, except for the factor W, both versions do exactly the same.

Question 7: Let $S = (a_1, a_2, \ldots, a_n)$ be a sequence of n positive numbers. A subsequence T of S is called *awesome*, if for every i with $1 \leq i \leq n-1$, a_i and a_{i+1} are not both in T. In other words, whenever a number is in T, none of its neighbors in S is in T. The weight of the subsequence T is the sum of all numbers in T.

Give a dynamic programming algorithm that computes, in $O(n)$ time, the maximum weight of any awesome subsequence of T.

As always, justify your answer. Follow the three dynamic programming steps that we have seen in class.

Solution: We want to apply dynamic programming, so we have to go through the three steps, as we did in class.

Step 1: Show that there is optimal substructure.

Assume we know the optimal solution for the entire problem.

- Assume that a_n is not included in the optimal solution. Then the weight of the optimal solution is the same as the weight of the optimal solution for the numbers $a_1, a_2, \ldots, a_{n-1}$.
- Assume that a_n is included in the optimal solution. Then the optimal solution does not contain a_{n-1}. The value of the optimal solution is equal to a_n plus the weight of the optimal solution for the numbers $a_1, a_2, \ldots, a_{n-2}$.
- Since we do not know which of these two cases holds, we take the larger of them.

Step 2: Set up a recurrence relation.

For $i = 0, 1, 2, \ldots, n$, let $W(i)$ be the weight of the optimal solution for the numbers $a_1, a_2, \ldots, a_i$.

We want to compute the value of $W(n)$.

- $W(0) = 0$.
- $W(1) = a_1$. (Here we use the fact that $a_1 > 0$.)
- For $i = 2, 3, \ldots, n$, $W(i) = \max(W(i-1), W(i-2) + a_i)$.

Step 3: Solve the recurrence, in a bottom-up order.

$W(0) = 0$;
$W(1) = a_1$;
for $i = 2, 3, \ldots, n$
do $W(i) = \max(W(i-1), W(i-2) + a_i)$
endfor;
return $W(n)$

It is clear that the running time is $O(n)$.

Question 8: Tyler is not only your friendly TA, he is also the CEO of *Tyler Enterprises*. This company buys long copper wires, cuts them into subwires, and then sells these subwires. Tyler Enterprises only buys long copper wires having integer lengths and cuts such that each subwire has an integer length.

Let n be a large integer and let $p_1, p_2, \ldots, p_n$ be a sequence of positive numbers. For each i with $1 \leq i \leq n$, Tyler Enterprises sells a subwire of length i for p_i dollars.

Consider a copper wire of length n. Give a dynamic programming algorithm that computes, in $O(n^2)$ time, the maximum revenue that can be obtained by cutting the length-n wire into subwires.

As always, justify your answer. Follow the three dynamic programming steps that we have seen in class.

For example, let $n = 4$. Here are the different options to cut a length-4 wire:

- The wire is not cut. Then the revenue is p_4.
- The wire is cut into one subwire of length 1 and one subwire of length 3. Then the revenue is $p_1 + p_3$.
- The wire is cut into two subwires of length 1 and one subwire of length 2. Then the revenue is $2 \cdot p_1 + p_2$.
- The wire is cut into two subwires of length 2. Then the revenue is $2 \cdot p_2$.
- The wire is cut into four subwires of length 1. Then the revenue is $4 \cdot p_1$.

Solution: We want to apply dynamic programming, so we have to go through the three steps, as we did in class.

Step 1: Show that there is optimal substructure.

Assume we know the optimal solution for the entire problem.

- Consider the last subwire in the optimal cutting. Let i be the length of this last subwire. Note that $1 \leq i \leq n$. Then the optimal revenue for the lenght-n wire is equal to p_i plus the optimal revenue for a wire of length $n - i$.
- Since we do not know the value of i, we consider all possible values for i, and take the largest revenue.

Step 2: Set up a recurrence relation.

For $m = 0, 1, 2, \ldots, n$, let $R(m)$ be the optimal revenue for a wire of length m.
We want to compute the value of $R(n)$.

- $R(0) = 0$.
- For $m = 1, 2, \ldots, n$,
$$R(m) = \max_{1 \leq i \leq m} (p_i + R(m - i)).$$

Step 3: Solve the recurrence, in a bottom-up order.

$R(0) = 0$;
for $m = 1, 2, \ldots, n$
do $R(m) = -\infty$;
  for $i = 1, 2, \ldots, m$
  do $R(m) = \max(R(m), p_i + R(m - i))$
  endfor
endfor;
return $R(n)$

The outer for-loop makes n iterations. For each such iteration, the inner for-loop makes at most n iterations. Therefore, the total running time is $O(n^2)$.