COMPSCI 311: Introduction to Algorithms

First Midterm Exam, October 3, 2018

Name: _____ ID: _____

- Answer the questions directly on the exam pages.
- Show all your work for each question. More detail including comments and explanations can help with assignment of partial credit.
- Questions have different point values. Move on to another question if you get stuck.
- You need not write pseudocode for BFS/DFS, but if you adapt them, state clearly how.
- If you need extra space, use the back of a page.
- No books, notes, or electronic devices are allowed. Any cheating will result in a grade of 0.
- If you have questions during the exam, raise your hand.

Question	Value	Points Earned
1	4	
2	3	
3	2	
4	3	
5	4	

Question	Value	Points Earned
6	6	
7	2	
8	8	
9	2	
10	6	
Total	40	

1. (4p) Construct a stable matching example where the Gale-Shapley algorithm can have two runs with different numbers of accepted proposals, depending on the order in which colleges propose. Count the executions of the outer loop and explain why the numbers are the same or different.

Solution: Suppose there are two colleges A and B, and two students X and Y. The preference lists are: $A = B = (X > Y)$, while $X = (A > B)$ and $Y = (B > A)$. Consider the two following runs of the Gale-Shapley algorithm:

$A : X$ *accepts* (AX)	$B : X$ *accepts* (BX)
$B : X$ *rejects* (AX)	$A : X$ *accepts* (AX)
$B : Y$ *accepts* (AX, BY)	$B : Y$ *accepts* (AX, BY)

So it follows that even though the *number of iterations* of the algorithm is the same over the two runs, the number of (possibly temporary) acceptances is **different** (it is two for the first run, but three for the second). You can try to prove that the number of iterations of the algorithm will always be the same for the same instance, no matter what choices the algorithm will make at each iteration. ■

2. (3p) Let $f(n) = n!$ and $g(n) = n^n$. Prove for each of the following whether it is true or false: a) $g(n) = \Theta(f(n))$, b) $g(n) = \Omega(f(n))$, c) $g(n) = O(f(n))$.

Solution: Clearly, $g(n) > f(n)$ for all $n > 1$ (why?), and so $g(n) = \Omega(f(n))$ trivially. Now note that if $g(n)$ were to be $O(f(n))$, then $\lim_{n\to\infty}(g(n)/f(n))$ would be bounded. But

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^n}{n!} = \lim_{n\to\infty} \frac{n}{n} \cdot \frac{n}{n-1} \cdot \frac{n}{n-2} \cdots \frac{n}{2} \cdot \frac{n}{1} = \infty,$$

since all terms in the product are at least 1, and *some* term (in particular the last one) is at least n. Therefore, $g(n)$ is not $O(f(n))$, and so $g(n)$ is not $\Theta(f(n))$ either. ■

3. (2p) Does $\log f(n) = \Theta(\log g(n))$ imply $f(n) = \Theta(g(n))$? Prove or give a counterexample.

Solution: No. For instance, you can take $f(n) = 3^n$ and $g(n) = 2^n$. Clearly, $\log f(n) = n \log 3 = \Theta(n) = \Theta(\log 2^n) = \Theta(\log g(n))$. However, $f(n)/g(n) = (3/2)^n$, which grows to infinity in the limit $n \to \infty$, and so $f(n) \neq \Theta(g(n))$. Interestingly, you could also use the previous problem to solve this by taking logarithms of both sides, though this is significantly harder. ■

4. (3p) Let v be any node in an undirected graph that contains a cycle. Prove that there is some node whose depth in the DFS tree from v is larger than in the BFS tree from v.

Solution: Suppose we have the graph G with a cycle, and we run DFS from some vertex v. Since G has a cycle, there is some edge e of G that is not in the BFS tree T. Suppose this edge $e = \{u_1, u_2\}$. We know that e is a non-T edge, and so one of u_1 and u_2 is the ancestor of the other (why?). Suppose u_1 is the ancestor of u_2. Note that u_1 cannot be the parent of u_2 in T (since then there would be more than one edge between u_1 and u_2), and so the distance between u_1 and u_2 in T is at least 2. This means in particular that the $\text{depth}_{DFS}(u_2) \geq \text{depth}_{DFS}(u_1) + 2$. But note that we have some path from v to u_2 in G that goes to u_1 and then directly goes to u_2 via the edge e, so that $\text{dist}_G(v, u_2) \leq \text{dist}_G(v, u_1) + 1$. Therefore,

$$\text{depth}_{BFS}(u_2) \leq \text{dist}_G(v, u_1) + 1 \leq \text{depth}_{DFS}(u_1) + 1 > \text{depth}_{DFS}(u_2),$$

and we are done. ■

5. (4p) Given a connected undirected graph G, give an $O(|V| + |E|)$ algorithm to find an edge that can be removed while still leaving the graph connected, or answers that no such edge exists.

Solution: Do a depth-first search. If a node is visited again and it's *not the parent* of the currently explored node, we have a back edge which closes a cycle and can be removed. Otherwise, report no such edge exists (the graph is a tree).

```
function DFSCycleEdge(parent, crt)
crt.visited = true
for all edges (crt, nxt) do
  if nxt.visited = false then
    if DFSCycleEdge(crt, nxt) then // edge found, already printed
      return true
  else if nxt != parent then
    print edge (crt, nxt)
    return true   // edge found
return false   // edge not found

algorithm FindCycleEdge
if not DFSCycleEdge(null, start) then
  print can't remove an edge
```

This runs the same traversal as a regular DFS, but it may finish early (if an edge is found). All extra work in the foreach loop is $O(1)$ for every edge, so the time is still $O(|V| + |E|)$. ■

6. (6p) Given a directed graph, write pseudocode for an $O(|V| + |E|)$ algorithm that outputs a cycle if one exists in the graph, otherwise produces a topological ordering of the graph.

Solution: We run the topological sorting algorithm done in class. If it stops early, all remaining nodes have incoming edges. We follow edges *backwards* from any remaining node – it can always progress since all nodes have incoming edges, and eventually closes a cycle since the nodes are finite.

```
Q = Z = ∅.
forall v in V compute v.indegree; if v.indegree = 0 add v to Z; end for;
while Z ≠ ∅ do
  remove some node v from Z and V, add v to Q
  for all edges v → w do
    remove edge v → w
    if --w.indegree = 0 then
      add w to Z
if V = ∅ then // no nodes left
  print Q (topological ordering)
else // find cycle
  choose v from V; S = ∅; // stack of cycle nodes
  repeat
    v.marked = true; S.push(v);
    choose edge w → v; v = w
  until v.marked    // cycle found
  repeat
    print S.pop();
  until S.top() = v // closing node
```

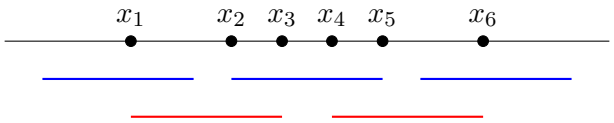
The topological sorting algorithm is unchanged, it runs in $O(|V| + |E|)$ (and might not reach all nodes and edges). The backward loop to find a cycle runs in $O(|V|)$ since it reaches each vertex at most once until it stops; printing the cycle is also $O(|V|)$. ■

7. (2p) We transform a directed graph as follows: each strongly connected component C becomes a single node $n(C)$; if there is any edge from a node in SCC C_1 to SCC C_2 we draw a single edge between nodes $n(C_1)$ and $n(C_2)$. Argue whether the resulting directed graph has a cycle or not.

Solution: Suppose the original directed graph was G and the resulting directed graph is G'. Suppose G' has some directed cycle, say $(n(C_1), n(C_2), \ldots, n(C_k) = n(C_1))$ for some k. By definition, this means that G' has directed edges between $n(C_i)$ and $n(C_{i+1})$ for all i, and so G had directed edges between C_i and C_{i+1} for all i. Because each C_i was strongly connected, this means that we could then construct a directed cycle C' going through each connected component C_i in this cycle in order. Now, take some $v_i \in C_i$ and some $v_{i'} \in C_{i'} \neq C_i$. We can now find some directed path from v_i to $v_{i'}$ and also one from $v_{i'}$ to v_i, using the strong connectivity of the C_i and the edges between these components in C'. However, these two directed paths would imply that v_i and $v_{i'}$ would have to be in the same connected component, which contradicts $C_i \neq C_{i'}$. It follows that we cannot have a directed cycle, and so G' is a DAG. ■

8. (8p) Given points $x_1 < x_2 < ... < x_n$, we want to find the smallest set of unit length closed intervals that contain all these points. Here are two greedy algorithms trying to find a smallest set:
a) Pick the interval $[x_1, x_1 + 1]$, remove all covered points, and continue until all points are covered.
b) Pick an interval $[y, y + 1]$ that maximizes the number of points covered, remove these points, and continue until all points have been covered.
Prove that one suggestion works and analyze its complexity. Give a counterexample for the other.

Solution: We claim that the first algorithm works, while the second does not. To see the second does not work optimally, consider the values $(x_1, x_2, x_3, x_4, x_5, x_6)$ located at the points $(-2/3, 0, 1/3, 2/3, 1, 5/3)$ respectively. Then the second algorithm would first greedily pick the middle interval $[0, 1]$ since it is the only closed interval that contains four points. After that, it would need two other closed intervals to cover the remaining points x_1 and x_6, giving a total of three intervals. However, the optimal solution is clearly to pick $[-2/3, 1/3]$ and $[2/3, 5/3]$, since between them they cover all the x_i. So the second algorithm is not optimal. We show this situation here, with blue representing the intervals picked by the second algorithm, and red the optimal (two) intervals; note that the first algorithm does precisely pick the red intervals as well.

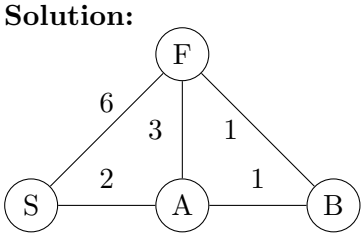


To see the first algorithm performs optimally, take some optimal solution S^*, and suppose its leftmost interval is placed at some $[x, x + 1]$, for some real number x. By the way our algorithm is designed, we must have $x \leq x_1$ (since we will only start our first interval at x_1). However, clearly, $x_1 \in [x, x + 1]$, as it is the leftmost point. We now claim that all points in $[x, x + 1]$ is covered by $[x_1, x_1 + 1]$. Otherwise, there would have to be some point in the half-open interval $[x, x_1)$, which is impossible (why?). So now, consider the set of intervals $S' = S^* - \{[x, x + 1]\} \cup \{[x_1, x_1 + 1]\}$; it follows that S' would cover at least all the points covered by S^*, meaning it does cover all the points, and furthermore, it has the same size as the optimal solution, so S' is optimal itself. We can now recursively continue this process, adding subsequent intervals from our greedy algorithm to our optimal solution via an exchange argument, to transform the entire solution into our algorithm's solution, without sacrificing optimality. It follows that the algorithm in part (a) is optimal.

Obviously, since we are already given a sorted list of points, our algorithm only has to scan the line from left to right, giving a total running time of $O(n)$. Note that we do not in fact have to binary search within each interval: we can continue the scan strictly to the right and account for points that get covered by intervals. ■

9. (2p) Construct a graph where the tentative shortest path to a node would be updated three times while running Dijkstra's algorithm. Hint: updates happen when a new neighbor is explored.

Solution:



The steps (choice of nodes and updates) are as follows:
Initially: d(S) = 0, d(A) = d(B) = d(F) = ∞
Choose S (d = 0). d(A) = 2, d(F) = 6.
Choose A (d = 2). d(B) = 3, d(F) = 5.
Choose B (d = 3). d(F) = 4.
Choose F (d = 4), done. ■

10. (6p) Give an algorithm that finds a *maximum* spanning tree for a connected graph G with distinct edge weights, prove its correctness and analyze its running time.

Solution: Suppose the maximum weight of any edge in G is W. We make a new graph G' by taking the same vertex set and adjacencies as G, but by redefining the edge weights as $w'(e) = W - w(e)$ for all e, where w was the original weight function in G. First of all, notice that all weights in G' are non-negative (this is technically not needed for Kruskal or Prim's algorithms, but it is a technicality that you might be more comfortable with). We claim that a minimum spanning tree in G' corresponds precisely to a maximum spanning tree in G. This is clear, because for a spanning tree T of G or G',

$$w'(T) = \sum_{e \in T} w'(e) = \sum_{e \in T} (W - w(e)) = W(n-1) - \sum_{e \in T} w(e) = W(n-1) - w(T),$$

and since $W(n-1)$ is constant, we maximize $w(T)$ precisely when we minimize $w'(T)$. Therefore, we can just run either Kruskal or Prim on G', find an MST, and output that same tree as a maximum spanning tree in G. ■