CS 577: Introduction to Algorithms — Graphs Review Solutions

Solutions

1. (a) Every tree T contains a leaf node v, i.e., a vertex v that has a degree of 1, and $T \setminus v$ is also a tree.

   Now we proceed by induction on the number n of vertices of a tree T. For $n = 1$, i.e., the base case where the tree consists of just a single vertex, the tree is a bipartite graph. Suppose that any tree T containing less than n vertices is bipartite.

   Let v be a leaf node in T and let u be its only neighbor, i.e., its parent in T. We know that $T \setminus v$ is a tree with $n - 1$ vertices, so by our inductive hypothesis, $T \setminus v$ is a bipartite graph.

   Let L and R be a bipartition of $T \setminus v$. If $u \in L$, then L and $R \cup v$ are a bipartition of T. However, if $u \in R$, then R and $L \cup v$ is a bipartition of T.

   (b) A bipartite graph is equivalently 2-colorable. Therefore, an algorithm that determines if a graph is 2-colorable will suit our needs.

   Algorithm We pick any vertex and run a modified BFS from it. Our modified BFS will assign vertices in odd-numbered levels the color C_1 and vertices in even-numbered levels the color C_2. During search, if we encounter an adjacent vertex that has the same color as the vertex we are searching out of, then the graph is not 2-colorable, so we indicate that it is not bipartite. If the search finishes, then the graph is bipartite.

   Correctness We simply need to prove that the algorithm correctly determines whether a graph is 2-colorable. For any vertex v, all of v's neighbors need to be a different color than v. At any given point in the search, v's neighbors would be its parent (of course, if v is the root, it doesn't have a parent), all unexplored vertices adjacent to v, and all vertices adjacent to v that have already been encountered in the search and are thus colored already. If v is currently being explored out of, then its color is already set, and its parent (if it exists) was already explored out of and will be of the opposite color. When exploring unexplored vertices from v, they need to be the opposite color of v in order for the coloring to be consistent; these vertices would be in the level below v in the BFS tree, so our algorithm colors them correctly. When searching out if v, if we encounter a vertex that was already colored by the search, and its color matches that of v, then a coloring is impossible; correspondingly, our algorithm indicates the graph is not bipartite.

   Running Time BFS runs in $O(|V| + |E|)$ time. The only changes we introduced to BFS were the assignment of colors to nodes, which does not affect our running time, and a new termination condition, which also does not affect our running time. Thus, our algorithm runs in $O(|V| + |E|)$ time.

   Note: We aren't limited to just BFS. Any traversal algorithm will work, provided the appropriate modifications are made. For example, DFS would also work. We just used BFS as an example.

2. We will prove two claims before moving on to the main proof. First, we will show that if T is a depth-first spanning tree of G, and (u, v) is an edge in G, then either u is an ancestor of v in T, or v is ancestor of u in T.

   Without loss of generality, suppose u is visited first by DFS. Once u is visited, DFS recursively visits each unvisited neighbor of u. Since an edge connects u and v, one of two things will happen:

   - (u, v) is traversed during DFS, and v is thus visited.
   - Some other neighbor of u, w, is visited first, and during recursive DFS calls on unvisited neighbors of w, v is visited.

   In either case, v is visited during the recursive DFS calls on each unvisited neighbor of u.

   Now, we will show that if T is a breadth-first spanning tree of G, and (u, v) is an edge in G, then the depth of u differs from the depth of v by at most 1.

Without loss of generality, suppose u is visited first by BFS. Recall that BFS manages a queue of nodes, and when a node is visited, it is added to the queue. What this means is u will be removed from the queue before v is because u was visited first. When u is removed, one of two things will happen:

- v is not yet in the queue; since it is an unvisited neighbor of u, it will be added to the queue and is the child of u in T.
- v is already in the queue; this could mean one of two things:
  - u and v are in the same level of T, and since u was visited first, it gets removed before v.
  - v is in the level directly below u; v was added to the queue when a node in the same level as u was removed from the queue before u and had all of its unvisited neighbors visited and subsequently added to the queue.

  It is impossible for v to be in any level above u because if that were the case, v would have been added to the queue before u because BFS visits all nodes in a single level before moving on to the next. It is also impossible for v to be more than one level beyond u because this implies that a node w in the level below u was removed from the queue before u; this cannot happen because u is visited before w since u is in a higher level, so u is to be removed from the queue before w, and v is then added to the queue as the child of u.

Either way, the depth of u differs from the depth of v by at most 1.

With these two claims, we now proceed to the main proof.

Given that the depth-first spanning tree of G matches the breadth-first spanning tree of G (assuming both share the same root node), it follows that for any edge (u, v) in G, either u is the parent of v in the DFS/BFS tree T or v is the parent of u in T; this is because one must be the ancestor of the other in T, and their depth in T differs by at most 1. Therefore, G has to be a tree.

3. (a) No. Suppose we have a graph $G = (V, E)$ where $V = \{a, b, c\}$ and $E = \{ab, bc, ac\}$, where edge costs are $c_{ab} = c_{bc} = -1$ and $c_{ac} = 1$. We would like to find the shortest path from a to c. Let's define G' using the same graph as G but with edge costs increased by 100; so $c_{ab} = c_{bc} = 99$ and $c_{ac} = 101$. The shortest path in G is $a \to b \to c$, but when Dijkstra's algorithm is ran on G', the algorithm returns $a \to c$ as being the shortest path, rather than $a \to b \to c$. Adding a large constant to each edge does not work because in general, not all paths have the same number of edges. In our example, the shortest path has two edges, so by adding 100 to each edge, the cost of the shortest path increases by 200, whereas the cost of the other longer path increases by just 100.

   (b) Dijkstra's algorithm works when all of the negative weight edges are those that leave s, and s has no incoming edges. The proof for Dijkstra's algorithm still holds when only the edges outgoing from s are allowed to be negatives.

4. For clarity, we will define the grid square (*NOT* the number in the square, actually the square itself) in the ith row and jth column of the number maze as $M[i, j]$.

   Algorithm Define a graph $G = (V, E)$ using the following rules:

   - For each square $M[i, j]$, add a corresponding vertex to V.
   - For each vertex u corresponding to $M[i, j]$, add an edge e_{uv} to E if the square corresponding to v is reachable from $M[i, j]$. Let k be the number on $M[i, j]$. v can correspond to the following squares (of course, assuming these squares actually exist, i.e., they're not out of bounds):
     - $M[i - k, j]$ (square above $M[i, j]$)
     - $M[i + k, j]$ (square below $M[i, j]$)
     - $M[i, j - k]$ (square to the left of $M[i, j]$)
     - $M[i, j + k]$ (square to the right of $M[i, j]$)

Succinctly, for each square $M[i, j]$, we're defining a vertex in V and at most four edges in E corresponding to each square reachable from $M[i, j]$. Let s be the vertex corresponding to $M[1, 1]$ and f be the vertex corresponding to $M[n, n]$. We run BFS starting from s, terminating when f is reached, or when all reachable vertices have been visited with f not being one of them. Recalling that BFS is capable of returning the level of vertices in the BFS spanning tree, if f was reached, we can take the level of f in the BFS spanning tree as being the minimum number of moves needed to reach $M[n, n]$ from $M[1, 1]$. Otherwise, if f was not reached, we indicate that the maze has no solution.

Correctness The defined graph G is a faithful representation of the number maze, so its construction will not be the focus of the proof. What is more important is showing that applying BFS is indeed correct. Indeed, for any graph G, the BFS spanning tree T rooted at some vertex $v \in G$ is a shortest-path tree, i.e., the path distance from v to any other vertex u in T is the shortest path distance from v to u in G. Additionally, for any vertex u, u's level in T is the shortest path length from v to u in G. Therefore, upon running BFS from s until f is reached, we can take f's level in the BFS tree as being the shortest path length from s to f in G, and our algorithm does just that.

Running Time The algorithm requires two steps: building the graph G and running BFS on G. For each square $M[i, j]$, we add a vertex to V and at most four edges to E. Thus, building the graph entails a constant amount of work per square. Recalling that the grid has n^2 squares, building G runs in $O(n^2)$ time.

Recall that BFS on a graph $G = (V, E)$ runs in $O(|V| + |E|)$ time. It is clear that $|V| = n^2$, and since each square induces at most four edges, $|E| \leq 4n^2$, thus BFS on G runs in $O(|V| + |E|) = O(n^2)$ time.

Overall, the algorithm runs in $O(n^2)$ time.

5. (a) *Proof.* Use induction to prove that for n pigeons, a pecking order can be established. In the base case, we have two pigeons, so we put the pecker on the right. Our inductive hypothesis is that for k pigeons, a pecking order can be established. Using this hypothesis, we will show that a pecking order for $k + 1$ pigeons can be established as well. Let's pull one pigeon p from the set of $k + 1$ pigeons and establish a pecking order for the remaining k pigeons (which can be done by our hypothesis); let's call these k pigeons the set S. We have three scenarios in front of us:

   - Each pigeon in S pecks p, so we can put p in the leftmost position.
   - p can peck every pigeon in S, so we can put p in the rightmost position.
   - When we arrange the pigeons in S in a pecking order, there exists a consecutive pair of pigeons such that p pecks one, and gets pecked by the other; place p in between these two.

   In each of these scenarios, we were able to find a position for p to reside in. Therefore, the existence of a pecking order for $k + 1$ pigeons is proven. □

   (b) We will use our result from part (a) to develop an algorithm.

   Algorithm Remove a vertex v from the graph. Find a pecking order for the rest of the $n - 1$ vertices using a recursive call. Insert v into the pecking order by trying out all possible locations.

   Correctness We assume that the recursive call on $n - 1$ vertices works and indeed returns a pecking order. By part (a), there has to be some location in that pecking order such that we can insert v and still have a proper pecking order.

   Running Time We have the recurrence $T(n) = T(n - 1) + O(n)$. Thus, the overall algorithm runs in $O(n^2)$ time.