CS483 - Practice Problems 2 (due September 29th)

Jana Košecká

Graph, Greedy algorithms

1. (5) When an adjacency-matrix representation is used, most graph algorithms require time $\Omega(V^2)$, but there are some exceptions. Show that determining whether a directed graph G contains a universal sink a vertex with in-degree $|V| - 1$ and out-degree 0 can be determined in time $O(V)$, given an adjacency matrix for G.

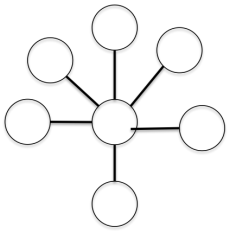
   Solution:

   Universal sink is a vertex that has out degree zero, i.e. there are no edges going out of that node and all vertices have edges to it. That corresponding row of that vertex in the adjacency matrix will have all zeros and the column of that vertex has all one's expect at the diagonal. If we encounter the row with all zero's, we can then check by traversing the first column in $O(V)$ time and see if it has all 1s. If so then node 1 is a universal sink, otherwise the graph has no universal sink. The algorithm terminates once we find a row of all zeros whether that row represents a universal sink or not, thus guaranteeing $O(V)$ running time.

   Now suppose that we start traversing the adjacency matrix with the first row. If there are no 1's encountered that we check the first column and if it has all 1's then the vertex 1 is universal sink. Otherwise we stop at first 1 we encounter, say at k position, then we know that the vertices 1 to $k-1$ cannot be universal sinks because vertex 1 has no edge to them, this takes $O(k)$ steps. Then we can continue on row j and do the same as for the first row.

   We begin examining the next available row j and follow a procedure similar to the first row. If no 1 is encountered, we test the j^{th} column for 1's and terminate as explained above. If a 1 is encountered after m zeros then we can eliminate these m nodes from being universal sink. Thus, it can be claimed that we have eliminated $k+m$ nodes in $O(k+m)$ time. Repeating the above procedure, examining one row at a time from vertices that have not yet been eliminated, we can find whether a universal sink exist or not. Thus, finding whether a graph has a universal sink or not, can be determined in $O(V)$ time.

2. (3) We covered two routines for graph traversal - DFS(G) and BFS(G,s) - where G is a graph and s is any node in G. These two procedures will create a DFS tree and a BFS tree respectively. If G = (V,E) is a connected, undirected graph then the height of DFS(G) tree is always larger than or equal to the height of any of the BFS trees created by BFS(G, s).



   Answer:

   If we assume that DFS can start from any arbitrary node, we pick the center node. For BFS we pick any of the boundary nodes. In this case the height of DFS tree is 1, whereas the height of BFS tree is 2. The example above is an counter example to the statement.

   For the part when we assume that they star tat the same node, we can proof the statement as follows.

Let $h_{\mathrm{BFS(G,s)}}(v)$ be the height of a node in a tree created by breadth first search of the graph G starting at node s. And let $h_{\mathrm{DFS(G,s)}}(v)$ be the height of a node in a tree created by depth first search of G starting at s. We start with $h_{\mathrm{BFS(G,s)}}(s) = h_{\mathrm{DFS(G,s)}}(s) = 0$. In either search, whenever a node p is used to discover a node c, we have $h(c) = h(p) + 1$.

One way to approach the proof is to first show that $h_{\mathrm{BFS(G,s)}}(v) = s\mathit{hortest}(s, v)$ is the length of the shortest path from s to v. Then, since $h_{\mathrm{DFS(G,s)}}(v)$ is the length of some path from s to v, $h_{\mathrm{DFS(G,s)}}(v) \geq h_{\mathrm{BFS(G,s)}}(v)$.

To show that BFS gives the shortest path, consider, as a potential contradiction, that there exists a node for which this is not true. Let a be a node such that $h_{\mathrm{BFS(G,s)}}(a) > s\mathit{hortest}(s, a)$. (Note the $>$. We do not need to consider the $<$ case as h cannot be smaller than the shortest path.) Furthermore, let a be the closest node to s for which this holds, where closest means having the smallest length $s\mathit{hortest}(s, a)$. Now consider a path of length $s\mathit{hortest}(s, a)$ from s to a, and consider the node n just before s in the path, so that the last edge of the path to a is from n to a. We know that $s\mathit{hortest}(s, n) + 1 = s\mathit{hortest}(s, a)$. Consider the possibilities for a when n is explored by BFS.

(a) If a was not yet discovered, then it is discovered by n and $h_{\mathrm{BFS(G,s)}}(a) = h_{\mathrm{BFS(G,s)}}(n) + 1 = s\mathit{hortest}(s, n) + 1$ contradicting $h_{\mathrm{BFS(G,s)}}(a) > s\mathit{hortest}(s, a)$.

(b) If a was discovered but not yet explored, then there it was discovered by some node m that was discovered and explored before n and so we would have $h_{\mathrm{BFS(G,s)}}(m) \leq h_{\mathrm{BFS(G,s)}}(n) = s\mathit{hortest}(s, n)$ and $h_{\mathrm{BFS(G,s)}}(a) \leq h_{\mathrm{BFS(G,s)}}(m)+1 \leq s\mathit{hortest}(s, n)+1$, contradicting $h_{\mathrm{BFS(G,s)}}(a) > s\mathit{hortest}(s, a)$.

(c) If a was discovered and explored before n then $s\mathit{hortest}(s, n) = h_{\mathrm{BFS(G,s)}}(n) \geq h_{\mathrm{BFS(G,s)}}(a)$ contradicting $h_{\mathrm{BFS(G,s)}}(a) > s\mathit{hortest}(s, a)$.

To see that $h_{\mathrm{DFS(G,s)}}(v)$ is the length of a path from s to v, construct a path back from v to the node that discovered it during DFS, and continue adding the node that discovered that node until reaching s. This path is at least as long as the shortest path, and $h_{\mathrm{DFS(G,s)}}(v)$ is the length of this path, so $h_{\mathrm{DFS(G,s)}}(v) \geq s\mathit{hortest}(s, v) = h_{\mathrm{BFS(G,s)}}(v)$.

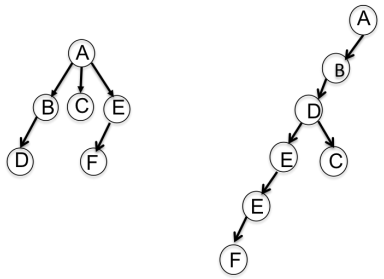
3. (9) For the following problem, use the directed unweighted graph given by the following adjacency list. Be sure to consider the edges in the given order.

A: C E B
B: E D
C: E
D: C F E
E: F
F:

(a) For the source vertex s = A what is the order in which the vertices are visited by BFS (breadth first search)? Also, show the breadth-first search tree that you obtain.
(b) What is the order in which the vertices are visited by DFS (depth first search)? You should assume that the top-level DFS procedure visits the vertices in alphabetical order. Set up a global counter which gets incremented every-time when the vertex if first explored or is finished being explored. For each vertex give the discovery and finishing time.
(c) Suppose that this graph is a precedence graph. Using your work above either give a valid order in which to perform the tasks (call them task A, task B, . . ., task F) or prove that there is no valid order.

BFS and DFS trees are below:

(b)The order in which the vertices are visited in DFS is as follows A (1/12, B(2/11), D(3/10), E(4/7), F(5/6), C(8/9), where in the parenthesis are the times when the vertex was first explored / when it finished exploring.



(c) The valid order is the order of the vertices is the order using reverse finishing times. The valid precedence order can be also determined using topological sort and is A, B, D, C, E, F.

4. (5pt) Chapter 3.2 (discuss the solution using one the graph traversal algorithms BFS or DFS). Assume that the graph is connected. Starting from arbitrary node s, obtain BFS tree T. If every edge go G that appears in the tree, then G = T, so G contains no cycles. Otherwise, there is some edge $e = (u, w)$ that belongs to G but not T connecting the nodes within the same level or to some of the ancestors of u. If such edge exists there is a cycle. The cycle can be printed by tracing all the ancestors of that node along the path in the BFS tree. To do that you may need to associate an additional field `node.parent` with each node. This can be done in $O(m + n)$ time.

5. Given a set $x_1 \leq x_2 \leq ... \leq x_n$ of points on the real line, give an algorithm to determine the smallest set of unit-length closed intervals that contains all of the points. A closed interval includes both its endpoints; for example, the interval $[1.25, 2.25]$ includes all x_i such that $1.25 \leq x_i \leq 2.25$.

   Answer:
   Here is a greedy algorithm for this problem: place the 1st interval at $[x_1, x_1 + 1]$ and remove all points in $[x_1, x_1 + 1]$, and then repeat this process on the remaining points, x_i is the left most point not contained in any interval, then the next interval is $[x_i, x_i + 1]$. This algorithm is clearly $O(n)$. We now prove it correct.

   Let P be any optimal solution and suppose it places its leftmost interval at at $[x, x + 1]$. It must be that $x \leq x_1$, since any feasible solution covers the leftmost point x_1. Let be P^* the solution obtained by replacing this first interval with $[x_1, x_1 + 1]$. The new interval still covers every point between x and $x + 1$ since there are no points to the left of x_1 and $x_1 + 1 > x + 1$. Hence, the new solution remains feasible and it uses the same number of intervals as P so is still optimal. After picking the interval $[x_1; x_1 + 1]$ and removing the points it covers, we are left with subproblem P' and an optimal solution that covers all points to the right of $x_1 + 1$. Any solution to P' can be feasibly combined with the greedy choice because all points to the left of $x_1 + 1$ are already covered.

6. (5pt) You are given a sequence of n songs, where the i-th song is l_i minutes long. You want to place all of the songs onto a collection of CDs, each of which can hold m minutes. Furthermore, (i) The songs must be recorded in the order given: song 1, song 2, . . . , song n. (ii) All songs must be included. (iii) No song may be split across CDs. Give an algorithm to place songs on CDs so as to minimize the number of CDs needed.

   Answer:
   Put as many songs as possible onto one CD without exceeding the m- minute limit. Then, close this CD, start a new one, and repeat the process for the remaining songs. The decision to close a CD can be made in constant time per song if we keep a running total of how much time we have used on the CD so far; hence, the algorithm takes $O(n)$ total time. We now prove it correct:

   Let S be an optimal solution in which the 1st CD holds the first k songs. Suppose the greedy algorithm puts the 1st g songs on this CD. If $g = k$, we are done; otherwise, $g > k$, and we modify S to produce a solution S' by moving songs from the second and later CD's to the rest of CD's until it contains

g songs. If the greedy algorithm could put the 1st g songs on a CD, these songs must take t in m minutes, and no other CD's total time increases; hence, the solution is still feasible.

Moreover, the number of CD's used by S' is at most as many as for S, so the solution is still optimal. After we close CD 1, we have the subproblem P0 of putting the remaining songs, from $g+1$ to n, on as few CD's as possible. Any feasible solution to the subproblem is compatible with our choice for the 1st CD, hence the choice is optimal.