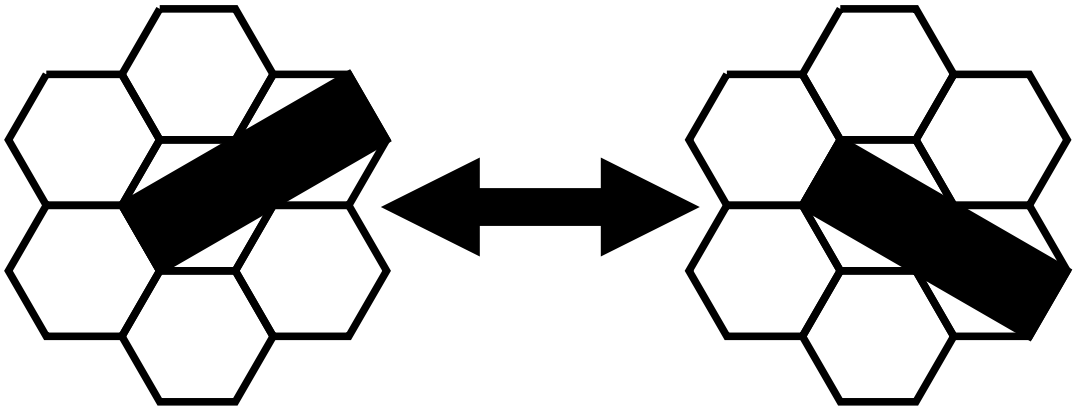
CSE101 Homework 1 Solutions

Winter 2015

Question 1 (Moving, 20 points)**.** *Alice is attempting to move a table from one side of her apartment to the other. The apartment is laid out on a hexagonal grid. The table is large enough that it takes up two adjacent hexagons at any given time. Alice can move the table by leaving one end fixed and rotating the other end either clockwise or counterclockwise one unit as shown below:*



Unfortunately, Alice's apartment has several immovable obstacles, each taking up a full hexagon so that the table cannot be moved to overlap any obstacle. Produce an efficient algorithm that given the layout of Alice's apartment, the location of obstacles and the desired starting and ending configurations of the table, determines whether or not it is possible for Alice to move the table from the starting configuration to the ending one.

Solution 1. *The problem of rearranging Alice's table can be abstracted as an undirected graph reachability problem. Each node in the graph, which we'll call $G_T = (V_T, E_T)$, corresponds to a unique table position, and an edge between two nodes means that the two positions can be reached from each other by a single 60-degree rotation of the table. (Since any move can be undone, the edges are undirected.) Since the table occupies two adjacent cells, each pair of adjacent cells in the apartment gets a unique node V_T, unless one of those cells contains an immovable object, in which case the node is not included. (Alternatively, we could leave the node in V_T, but since the table cannot occupy that space, it would have no edges with any other table positions.)*

Once we have constructed G_T, we can simply run a partial depth-first-search on the graph, starting at the node corresponding to the table's starting position (call it s), and ending when s is popped off of the stack. (It is a "partial" search because it is not guaranteed to visit all nodes or edges, only the ones reachable from s.) If the destination configuration (t) is visited before this partial depth-first-search terminates, then there is a way to move the table to the destination. Otherwise, there is no configuration path which moves the table from s to t.

So we need an efficient algorithm that, given Alice's apartment layout, generates the graph G_T. We'll assume that Alice's apartment is given as a graph $G_A = (V_A, E_A)$, where each node in V_A is a cell, and an edge $\{x, y\}$ indicates that cells x and y are adjacent. Assume we have G_A in adjacency list form, so that $A(x)$ denotes the set of vertices adjacent to vertex x. The following algorithm has time complexity $O(|V_A|)$:

```
procedure makeGT(G_A):

  (V_A, E_A) <- G_A // parse G_A as V_A, E_A tuple
  V_T = E_A // every edge in G_A is a node in G_T
  E_T = {}

  for u in V_A:
    if u contains immovable object, skip u
    // for each pair of nodes adjacent to cell u
    for v in A(u):
      if v contains immovable object, skip v
      for w in A(u):
        if w contains immovable object, skip w

        if w in A(v): // can rotate table around u
          E_T <- E_T union { {u, v}, {u, w} } // constant time in Adjacency-list form

  return G_T = (V_T, E_T)
```

Even though there are several nested loops in this algorithm, we end up doing a constant amount of work per node u. This is because each cell is adjacent to at most 6 cells, and each table configuration is adjacent to at most 4 other configurations (2 per side of the table; a clockwise rotation and a counterclockwise one). We could technically improve this algorithm, since it visits each edge in E_A multiple times, but this will not improve its big-O complexity.

Once we run this procedure and get G_T, we run the partial DFS beginning at s. Ordinarily, we would say that DFS has running time $O(|V_T| + |E_T|)$, but ultimately we want to get the time complexity as a function of $|V_A|$ and $|E_A|$. This will take a few steps.

First, we observe that in a given configuration, the table can be moved to at most 4 neighboring configurations. This means that the degree of each vertex in V_T is at most 4. Therefore, the sum of the degrees of each node in V_T is less than $4|V_T|$, so that $|E_T|$ is bound by $2|V_T|$, and therefore the DFS has time complexity $O(|V_T| + 2|V_T|) = O(|V_T|)$.

Next, we note that by the definition of G_T, $V_T = E_A$, since each configuration of the table can be mapped to an edge in G_A, i.e., a pair of adjacent cells in the apartment. Therefore the DFS has time complexity $O(|E_A|)$.

Finally, we note that $|E_A|$ is bound by the number of cells in the apartment, since each cell is adjacent to at most 6 other cells. Therefore, the DFS has time complexity $O(|V_A|)$.

So, generating G_T has complexity $O(|V_A|)$, and running the partial DFS takes time $O(|V_A|)$, so our overall algorithm has complexity $O(|V_A|)$.

Question 2 (DAG Detection, 20 points)**.** *Give a linear time algorithm that given a directed graph determines whether or not it is a DAG.*

Solution 2. *The key question we must answer about the given graph is whether or not it contains a cycle. To answer this question, we can modify the standard depth-first-search algorithm to assign pre and post numbers to each node, as described in class. During the depth-first-search, if an edge (s, p) is found such that s is the node currently at the top of the stack, and p a node which has been assigned a pre number but no post number, then the algorithm reports a cycle. If the depth-first-search completes without detecting such an edge, then it reports that there are no cycles, and therefore the graph is a DAG.*

This algorithm is guaranteed to find a cycle if one exists in the graph. One way to see this is that if there is a cycle, then each node in the cycle is reachable from itself. Therefore, if at some point the depth-first-search is exploring the paths out of a node s which is in a cycle, eventually the algorithm will visit s again before s is popped off the stack, i.e., before s has been given a post number. One subtle point

is that this event is only guaranteed to be occur if s is the first node in the cycle that has been visited, but since depth-first-search eventually visits all nodes, there is guaranteed to be one node in a cycle which is visited first.

If on the other hand there is no cycle, then there is no way for a node to be visited a second time before it is popped off the stack, i.e., before a post number is affixed to it. Therefore, our algorithm reports a cycle exactly when there is a cycle, without false positives or false negatives.

Since DFS (with pre/post number assignment) is $O(|V|+|E|)$, and checking a node to see if it has a pre or post number is a constant amount of work per edge, this algorithm has time complexity $O(|V|+|E|)$.

Alternate Solution: *Another way to do it is as follows. Run DFS on the graph computing pre- and post- numbers, then check whether or not $\text{post}(u) > \text{post(w)}$ for each edge $(u,w) \in E$. We claim that this holds for all edges if and only if the graph is a DAG. On the one hand, if G is a DAG, then we know from class, that the above relation holds for all edges. On the other hand, if all edges go from vertices with larger post numbers to smaller post numbers, there cannot be a cycle, because the vertex in the cycle with the smallest post number could not have an edge leading to another vertex in the cycle. The runtime of the algorithm is $O(|V|+|E|)$ for the DFS and then $O(|E|)$ to run the check over all edges, for a total runtime of $O(|V|+|E|)$.*

Essentially, what we are doing here is pretending that our graph is a DAG, running topological sort, and then seeing if we found an actual linear order.

Question 3 (Longest Path, 20 points)**.** *Find an efficient algorithm that given a DAG, G, finds the length of the longest path in G. Hint: topologically sort and compute the lengths of the longest path starting from v for each v in some order.*

Solution 3. *To compute the longest path in a DAG, we will keep track of the length of the longest path from each vertex in the graph. After each of these path lengths have been computed, the maximum value can be output.*

Specifically, let `lenpath`$[\cdot]$ *be an array of length $n = |V|$ with an index corresponding to each node in the graph. Define the array by* `lenpath`$[\text{v}] =$ *the length of a longest path starting at v. Then each entry can be computed by the following:*

$$\texttt{lenpath}[\text{v}] = 1 + \max_{u \in V \,:\, (v,u) \in E} \texttt{lenpath}[\text{u}].$$

First, it is clear that the second vertex on any path from v of non-zero length must be a neighbor vertex u such that $(v,u) \in E$. Let u_0 be a vertex which maximizes the above expression (i.e. u_0 is the destination of some edge from v such that `lenpath`$[\text{u}]$ *is maximum among all such u's). Then the length of a longest path from v is exactly* `lenpath`$[\text{u}_0] + 1$ *for the edge (v,u_0). We will show this by a simple contradiction: assume there is a path from v of length greater than* `lenpath`$[\text{u}_0] + 1$*. Then the next vertex on this path must be some other neighbor u' of v, and the length of the path would be the length of a longest path from u' plus 1 for the edge (v,u'). But this is exactly* `lenpath`$[\text{u}'] + 1$*. Therefore it must be that* `lenpath`$[\text{u}'] >$ `lenpath`$[\text{u}_0]$*, which contradicts u_0 being the vertex that maximizes* `lenpaths`*.*

Now, since computing `lenpath`$[\text{v}]$ *involves the values for vertices that are later in a topological ordering of the DAG, we will fill in the array in a reverse topological order. The base cases will be vertices with no out-going edges, for whom the longest path is necessarily of length 0. Below is the full algorithm:*

```
LongestPath(G):
//input: DAG G=(V,E) in adjacency list format
//output: the length of a longest path in G

let n = |V|
initialize an array lenpath[] of length n

find a topological ordering of G
for every v in V in reverse topological order:
    if v has no outgoing edges:
        lenpath[v] = 0
    else:
        for every u in V such that (v,u) in E:
```

```
            if lenpath[v] < lenpath[u] + 1:
                lenpath[v] = lenpath[u] + 1

return max(lenpath)
```

Topological sort takes $O(|V|+|E|)$ time. Testing if a vertex has no outgoing edges is a simple look-up in the adjacency list, and then for every edge in the graph we do an addition and a comparison. Thus the total running time is $O(|V|+|E|)$.

Question 4 (Cheapest Reachability, 20 points)**.** *Bob is planning a trip to Digraphia. Digraphia has many cities, but for unfathomable reasons travel between them is restricted, with certain pairs of cities connected by one-way roads. Bob has determined the cheapest available flights into and out of each city, and the available travel routes. Provide an efficient algorithm to determine the pair of cities s and t so that t is reachable from s and so that the cost of flying into s plus the cost of flying out of t is minimized. If there are n cities and m accessible one-way roads, you should aim for a runtime of $O(n\log(n)+m)$ or better. Hint: Try running depth first search exploring vertices in increasing order of entry cost.*

Can you do better if the roads are all bi-directional?

Solution 4. *Assume that the land of Digraphia is abstracted adjacency list form, where nodes are cities and directed edges are one-way roads. Our algorithm is as follows:*

```
procedure find_min_cost(G):

    sorted_nodes = [sort nodes in ascending order of fly-in cost] // O(nlogn)
    min_cost = infinity

    for n in sorted nodes:
        n.visited = false

    for n in sorted_nodes:
        min_out = modified_explore(n)
        min_cost = minimum( (n.in_cost + min_out), min_cost)

    return min_cost



procedure modified_explore(n):

    if n.visited:
        return infinity
    n.visited = true

    min_out = n.out_cost
    for x in A(n): // for immediate reachable neighbors
        min_out = minimum(min_out, modified_explore(x) )

    return min_out
```

Sorting the nodes requires time $O(n\log n)$, and running the modified depth first search requires time $O(n+m)$, so the overall algorithm has complexity $O((n\log n)+n+m) = O(n(\log n+1)+m) = O(n\log n+m)$.

Proving the correctness of this algorithm is a little tricky, because it does not explore all possible pairs (s,t) such that t is reachable from s. We must show that the pairs ignored by our algorithm cannot produce a minimum-cost pair of flights.

We can prove this through a crude kind of induction. We explore nodes in order of increasing fly-in cost, so the first node we explore, n_1, has the cheapest fly-in cost. Let R_x denote the set of nodes reachable from node x. Our algorithm will explore all pairs (n_1,t) such that $t \in R_{n_1}$, and find the minimum

fly-out cost for each of these, $o_{\min} = \min_{m \in R_{n_1}} \{m.\text{outCost}\}$.

In the simplest case (the "base" case), all nodes in the graph are in R_{n_1}. *In this case, the minimum cost and solution to the algorithm will be* $n_1.inCost + o_{\min}$, *since* $o_{\min}$ *is the cheapest fly-out cost in the entire graph, and* n_1 *has the cheapest fly-in cost, so no better solution is possible.*

In the "inductive" case, some nodes have already been explored, but others have not. Since we explore the nodes in order of increasing fly-in cost, the next node we (non-trivially) explore, n_k *will have the smallest fly-in cost of all unvisited nodes. Let* T *be the set of nodes which have already been visited. This exploration will visit the nodes* $R_{n_k} - T$, *the nodes reachable from* n_k *which have not already been visited. Thus, we now have* $o_{\min} = \min_{m \in R_{n_k} - T} \{m.\text{outCost}\}$. *So* $o_{\min}$ *is no longer guaranteed to be the minimum fly-out cost of all nodes reachable from* n_k, *since it is possible that the node with the smallest fly-out cost,* $n_{\min}$, *has already been visited. However, if* $n_{\min}$ *has already been visited, then that means that it was reachable from a node with a smaller fly-in cost than* $n_k.inCost$, *so the pair* $(n_k, n_{\min})$ *cannot be the optimal solution.*

Therefore, the pairs of nodes which are not examined by our algorithm cannot produce a minimum flight cost, and so our algorithm finds the optimal flight cost.

If the graph is undirected, things are easier. Then we only need to compute the connected component, and compare for each component the minimum fly-in cost plus the minimum fly-out cost. Since any city in the component is reachable from any other city in the component, this cost is both achievable and optimal. This algorithm takes $O(|V| + |E|)$ *time to compute the connected components of the graph. For each component* C, *it takes* $O(|V_C|)$ *time to find the minimum fly-in and fly-out costs in the component, so the total time over all components is* $O(|V|)$. *Thus, the total runtime is only* $O(n + m)$.

Question 5 (Pre- and Post- Orderings, 20 points)**.** *For each of the following tables either demonstrate a graph so that the given values could be the pre- and post- numbers for the vertices of that graph under a depth first search, or show that no such graph exists.*

(a)

Vertex	*Pre*	*Post*
A	*1*	*10*
B	*2*	*4*
C	*3*	*5*
D	*6*	*7*
E	*8*	*9*

(b)

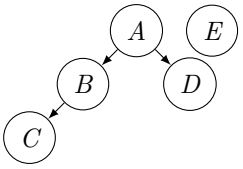
Vertex	*Pre*	*Post*
A	*1*	*8*
B	*2*	*5*
C	*3*	*4*
D	*6*	*7*
E	*9*	*10*

(c)

Vertex	*Pre*	*Post*
A	*1*	*10*
B	*2*	*3*
C	*4*	*8*
D	*5*	*7*

Solution 5. *(a) Since vertex* B *has a pre- value of* 2 *and vertex* C *has a pre-value of* 3, *it must be that* `explore(G, C)` *was called from* `explore(G, B)`, *which in turn means that* (B, C) *is an edge in the graph. However,* $pre(B) < pre(C) < post(B) < post(C)$ *is an impossible relationship since the pre- and post- numbers are interleaved. Therefore no graph could be given the pre- and post- values.*

(b) The following is an example of a graph whose vertices could be given the pre- and post- values during a depth-first search:



(c) No graph can be given the pre- and post- value. In particular, $pre(D) = 5, post(D) = 7$ is impossible. Consider the following two cases: either we explore a vertex from D or we don't. If we explore a vertex, call it U, then $pre(U) = pre(D) + 1 = 6$ and the clock must be updated at least once more for the post-visit of U, so $post(U) \geq 7$, but $post(D) = 7$ so this is a contradiction. If we don't explore a vertex from U, immediately after we pre-visit D, we increment the clock and post-visit D, so $post(D) = pre(D) + 1 = 6$, another contradiction.

Question 6 (Extra credit, 1 point)**.** *Approximately how much time did you spend working on this homework?*