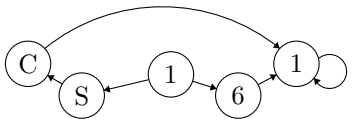
CS 161 **Problem Set 5 with Solutions**
Spring 2017 **Due:** May 19, 2017, 3pm

Please answer each of the following problems. Refer to the course webpage for the **collaboration policy,** as well as for **helpful advice** for how to write up your solutions.

Note: For all problems, if you include pseudocode in your solution, please also include a brief English description of what the pseudocode does.

Drawing graphs: You might try `http://madebyevan.com/fsm/` which allows you to draw graphs with your mouse and convert it into LaTeX code:

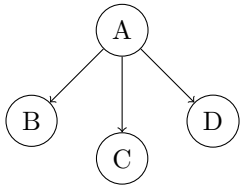


---

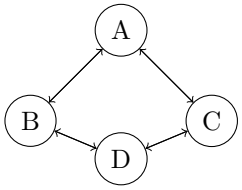
1. **(Examples.)** (4 points) Give one example of a directed graph on four vertices, A, B, C, and D, so that both depth-first search and breadth-first search discover the vertices in the same order when started at A. Give one example of an directed graph where BFS and DFS discover the vertices in a different order when started at A. Above, *discover* means the time that the algorithm first reaches the vertex. Assume that both DFS and BFS iterate over outgoing neighbors in alphabetical order.

   [We are expecting: a drawing of your graphs and an ordered list of vertices discovered by BFS and DFS.]

   SOLUTION: Our graphs are as follows; all searches start at A.



G



G'

   For G, both DFS and BFS, when started at A, recover vertices in order ABCD. For G', DFS starting at A discovers vertices in order ABDC, while BFS starting at A does it in order ABCD. In G', the edges drawn like $\leftrightarrow$ are all meant to represent two edges, on in each direction.

2. **(In-order traversal)** (6 points) In class, we stated the fact that Depth-First Search can be used to do in-order traversal of a binary search tree. That is, given a binary search tree T containing n distinct elements $a_1 < a_2 < \cdots < a_n$, DFS can be used to return an array $[a_1, a_2, \ldots, a_n]$ containing these elements in sorted order. Call this algorithm `inOrderTraversal`. Work out the details for this algorithm, and answer the following questions:

   (a) (3 pts) Give pseudocode for `inOrderTraversal`. For your pseudocode, assume that each node in T has a key value (one of the a_i) and pointers to its left and right children, and that if a vertex has no child this is represented as a `NIL` child.

(b) (2 pts) What is the asymptotic running time of `inOrderTraversal`, in terms of n? We're looking for a statement of the form "the running time is $\Theta(\_\_\_)$."

(c) (1 pt) Does the algorithm need to look at the values $a_1, \ldots, a_n$ (other than to return the values)?

[We are expecting: just the answers to these questions, no justification needed.]

SOLUTION:

(a) The pseudocode is as follows:

```
inOrderTraversal(T):
     return inOrderTraversal_helper(T.root)
inOrderTraversal_helper(node):
     if node is NIL:
          return
     inOrderTraversal_helper(node.leftChild)
     print node.key
     inOrderTraversal_helper(node.rightChild)
```

That is, we do DFS. At each node, we output first the results of the left DFS subtree, then we output the node itself, then we output the results of the right DFS subtree.

(b) The running time is $\Theta(n)$. No argument is required, but here are two: **Argument 1:** The running time is the same as that of DFS, $\Theta(n+m)$; but in a tree, $m = n-1$, and so this is $\Theta(n)$. **Argument 2:** The number of times that `inOrderTraversal` is called is exactly n, once for every element in the tree. In each call, $\Theta(1)$ work is done, so the total running time is $\Theta(n)$.

(c) No, the algorithm does not need to look at the values; the BST property is enough.

3. **(Too good to be true)** (6 points) Your friend claims to have built a new kind of binary search tree with $O(1)$ (deterministic) insertion time (better than the $O(\log(n))$ of red-black trees), which can handle arbitrary comparable objects. How do you know they are wrong?

[We are expecting: a short but formal proof that your friend is wrong.]

SOLUTION: Suppose your friend were right. Then we could use their binary search tree to give a $O(n)$-time comparison-based sorting algorithm. Our algorithm is

```
tooFastSort(A):
    T = myFriend'sBST()
    for a in A:
        T.insert(a)
    return inOrderTraversal(T.root).
```

Above, `inOrderTraversal` is as in the previous problem (except modified to output a sorted array). This runs in time $O(n)$, and it's also comparison-based. Indeed, anything that `myFriendsBST` does must be comparison-based, since it works on arbitrary comparison-based objects. And `inOrderTraversal()` is also comparison-based, because it doesn't even look at the input. So this contradicts the lower bound on comparison-based sorting algorithms.

4. **(Dijkstra and negative edge weights)** (15 points) Let $G = (V, E)$ be a weighted directed graph. For the rest of this problem, assume that $s, t \in V$ and that **there is a directed path from s to t.** The weights on G could be anything: **negative, zero, or positive.**

   For the rest of this problem, refer to the implementation of Dijkstra's algorithm given by the pseudocode below.

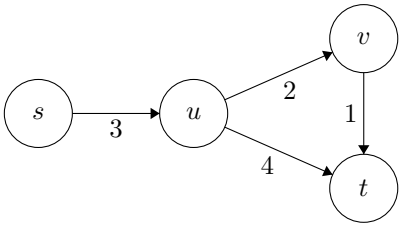
```
Dijkstra_st_path(G, s, t):
    for all v in V, set d[v] = Infinity
    for all v in V, set p[v] = None
    // we will use the information p[v] to reconstruct the path at the end.
    d[s] = 0
    F = V
    D = []
    while F isn't empty:
        x = a vertex v in F so that d[v] is minimized
        for y in x.outgoing_neighbors:
            d[y] = min( d[y], d[x] + weight(x,y) )
            if d[y] was changed in the previous line, set p[y] = x
        F.remove(x)
        D.add(x)

    // use the information in p to reconstruct the shortest path:
    path = [t]
    current = t
    while current != s:
        current = p[current]
        add current to the front of the path
    return path, d[t]
```

   Notice that the pseudocode above differs from the pseudocode in the notes: first it runs Dijkstra's algorithm (as presented in the notes), and then it uses this to compute the shortest path from s to t. It returns the shortest path and the cost of that path.

   (a) (2 points) Step through `Dijkstra_st_path`(G, s, t) on the graph G shown below. Complete the table below to show what the arrays `d` and `p` are at each step of the algorithm, and indicate what path is returned and what its cost is.

G



   [We are expecting: the table below filled out, as well as the final shortest path and its cost. No further justification is required.]

	d[s]	d[u]	d[v]	d[t]	p[s]	p[u]	p[v]	p[t]
When entering the first while loop for the first time, the state is:	0	∞	∞	∞	None	None	None	None
Immediately after the first element of D is added, the state is:	0	3	∞	∞	None	s	None	None
Immediately after the second element of D is added, the state is:								
Immediately after the third element of D is added, the state is:								
Immediately after the fourth element of D is added, the state is:								

(b) (3 points) **Prove or disprove**: In every such graph G, the shortest path from s to t exists. Here, a *path* from s to t is formally defined as a sequence of edges

$$(u_0, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_{M-1}, u_M)$$

so that $u_0 = s$, $u_M = t$, and $(u_i, u_{i+1}) \in E$ for all $i = 0, \ldots, M-1$. A *shortest path* is a path $((u_0, u_1), \ldots, (u_{M-1}, u_M))$ so that

$$\sum_{i=0}^{M-1} \text{weight}(u_i, u_{i+1}) \leq \sum_{i=0}^{M'-1} \text{weight}(u'_i, u'_{i+1})$$

for all paths $((u'_0, u'_1), \ldots, (u'_{M'-1}, u'_{M'}))$.

(c) (3 points) **Prove or disprove**: In every such graph G in which the shortest path from s to t exists, `Dijkstra_st_path`(G, s, t) returns a shortest path between s and t in G.

(d) (3 points) **Prove or disprove**: In every such graph G in which there is a negative-weight edge, and for all s and t, `Dijkstra_st_path`(G, s, t) does not return a shortest path between s and t in G.

(e) (4 points) Your friend offers the following way to patch up Dijkstra's algorithm to deal with negative edge weights. Let G be a weighted graph, and let w^* be the smallest weight that appears in G. (Notice that w^* may be negative). Consider a graph $G' = (V, E')$ with the same vertices, and so that E' is as follows: for every edge $e \in E$ with weight w, there is an edge $e' \in E'$ with weight $w - w^*$. Now all of the weights in G' are non-negative, so we can apply Dijkstra's algorithm to that:

```
modifiedDijkstra(G,s,t):
     Construct G' from G as above.
     return Dijkstra_st_path(G',s,t)
```

Prove or disprove: Your friend's approach will always correctly return a shortest path between s and t if it exists.

[We are expecting: for each "prove or disprove," either a proof or a (small) counterexample. In the previous sentence, "(small)" means no more than 5 vertices.]

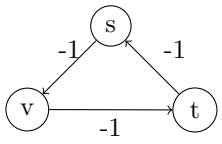
SOLUTION:

(a) Our table is below:

	d[s]	d[u]	d[v]	d[t]	p[s]	p[u]	p[v]	p[t]
When entering the first while loop for the first time, the state is:	0	∞	∞	∞	None	None	None	None
Immediately after the first element of D is added, the state is:	0	3	∞	∞	None	s	None	None
Immediately after the second element of D is added, the state is:	0	3	5	7	None	s	u	u
Immediately after the third element of D is added, the state is:	0	3	5	6	None	s	u	v
Immediately after the fourth element of D is added, the state is:	0	3	5	6	None	s	u	v

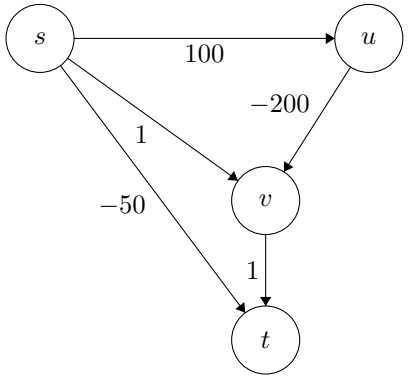
The path returned is $s \to u \to v \to t$ and the cost is $d[t] = 6$.

(b) This statement is not true. For example:



Then there are paths of arbitrarily small negative cost (looping many times around the cycle), and a minimum-cost path does not exist.

(c) This statement is not true. For example:



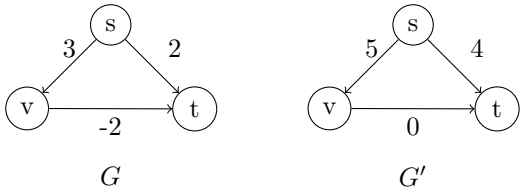
On this graph G, `Dijkstra_st_path`(G, s, t) returns the path $s \to t$ which has cost -50, while the shortest path (which does exist) is $s \to u \to v \to t$ which has cost -99. To check this, we can trace through the functionality of the pseudo-code.

First, we will update from s, and set $d[t] = -50$ and $p[t] = s, d[v] = 1, p[v] = s$, and $d[u] = 100$, $p[u] = s$. Next we will choose t and then v to update; neither of these will change anything. Finally, we will choose u; this updates $d[v] = -100$ and $p[v] = u$. However, at the end we have $d[t] = -50$ and $p[t] = s$, and so Dijkstra's algorithm will return that the shortest path goes from s to t (with a cost of -50), which is not correct.

Note: This problem is a bit tricky! If you don't include this edge from s to t, for example, then Dijkstra's algorithm (as above) will return the wrong value, but it will return the correct path.

(d) This statement is also false. Consider the two node graph consisting only of s and t, with a single weight -1 edge from s to t. Then Dijkstra's algorithm correctly returns the shortest path from s to t.

(e) Your friend's logic is pretty shaky. Underlying their approach is the assumption that the shortest path G is the same as the shortest path in G'. This is not true, as the following counter-example shows.



G G'

In G, the shortest path from s to t is through v, with cost 1. In G', the shortest path is the edge directly from s to t, with cost 4. Thus, even though there is a shortest path from s to t in G, your friend's algorithm would fail and return the wrong path.

5. **(Social engineering)** (13 points) Suppose we have a community of n people.[1] We can create a directed graph from this community as follows: the vertices are people, and there is a directed edge from person A to person B if A would forward a rumor to B. Assume that if there is an edge from A to B, then A will always forward any rumor they hear to B. Notice that this relationship isn't symmetric: A might gossip to B but not vice versa. Suppose there are m directed edges total, so $G = (V, E)$ is a graph with n vertices and m edges.

   Define a person P to be *influential* if for all other people A in the community, there is a directed path from P to A in G. Thus, if you tell a rumor to P, eventually it will reach everybody. You have a rumor that you'd like to spread,[2] but you don't have time to tell more than one person, so you'd like to find an influential person to tell the rumor to.

   In the following questions, assume that G is the directed graph representing the community, and that you have access to G as an array of adjacency lists: for each vertex v, in $O(1)$ time you can get a pointer to the head of the linked lists v.`outgoing_neighbors` and v.`incoming_neighbors`. Notice that G is not necessarily acyclic. In your answers, you may appeal to any statements we have seen in class, in the notes, or in CLRS.

   (a) (2 pts) Show that all influential people in G are in the same strongly connected component, and that everyone in this strongly connected component is influential.
   [We are expecting: a short but formal proof.]

   (b) (8 pts) Suppose that an influential person exists. Give an algorithm that, given G, finds an influential person in time $O(n + m)$.
   [We are expecting: pseudocode, a proof of correctness, and a short argument about the runtime.]

   (c) (3 pts) Suppose that you don't know whether or not an influential person exists. Use your algorithm from part (b) to give an algorithm that, given G, either finds an influential person in time $O(n + m)$ if there is one, or else returns "no influential person."
   [We are expecting: pseudocode, and a short argument for both correctness and runtime. You do not need to re-write your algorithm from part (b), you can just call it.]

   SOLUTION:

   (a) If v and v' are influential, there is a path from v to v' and from v' to v. Thus v and v' are in the same strongly connected component, by definition. Similarly, if v is influential and v' is in the same SCC as v, then v' is influential, because for all $u \in V$, there is a path from v' to v to u.

   (b) There are (at least) three different good solutions to this problem.
   SOLUTION 1: The algorithm is:

```
findInfluentialPerson(G):
Run DFS and keep track of finishing times.
    Return the vertex with the largest finishing time.
```

   The runtime is just that of DFS, which is $O(n + m)$.
   To analyze this, we will follow our analysis of using DFS twice to identify strongly connected components (SCCs). (However, our solution only runs DFS once). Suppose that $C_1, \ldots, C_r$ are the strongly connected components of G. Define the finishing times of a vertex and an SCC as we did in class, and denote then v.`finish` and C.`finish`, respectively.
   As we have seen in class, the induced graph on SCCs is a DAG. By part (a) we know that there is some unique strongly connected component C^* that contains all of the influential people.
   We will go through the proof with a series of claims.

[1] For example, students in CS161.
[2] For example, the vicious rumor that "there's a tpyo in Problem 5."

- **Claim 1:** For all SCCs $C \neq C^*$, there is a path in the SCC DAG from C^* to C.
  Proof. Let P be an influential person in C^*, and let A be any person in C. There is a path from P to A, by the definition of influential, so there is a path from C^* to C in the SCC DAG.
- **Claim 2.** $C^*.\texttt{finish} > C.\texttt{finish}$ for all SCCs $C \neq C^*$.
  Proof. We saw in class that if there is an edge from C to C' in the SCC DAG, then $C.\texttt{finish} > C'.\texttt{finish}$. This implies that if there is a path from C the C' in the SCC DAG, that $C.\texttt{finish} \geq C'.\texttt{finish}$, even if there is no edge directly from C to C'. Finally, **Claim 1** implies that $C^*.\texttt{finish} > C.\texttt{finish}$ for all SCCs $C \neq C^*$.
- **Claim 3** Let v^* be the vertex with the largest finish time, which we return. Then there is a path from v^* to every other vertex $v \in V$.
  Proof. By definition, $C^*.\texttt{finish}$ is the maximum value of $v.\texttt{finish}$ for all $v \in C^*$. Thus (using **Claim 2**) the vertex in C^* with the maximum finish time is the vertex in V with the maximum finish time, which is v^*.
  Let $v \in V$ be any vertex, and suppose that it is in an SCC C. By **Claim 1** (aka, the definition of C^*), there is a path in the SCC DAG from C^* to C. This means there is a path P_1 from some $w^* \in C^*$ to some $w \in C$. Because C^* and C are strongly connected, there is a path P_0 from v^* to w^* and a path P_2 from w to v. Putting these paths together, we find a path $P = P_0P_1P_2$ from v^* to v. This proves **Claim 3.**

Finally, Claim 3 was what we wanted to show: the vertex v^* with the largest finish time (which is what we returned in our algorithm) can indeed reach every other vertex v in the graph. So the person corresponding to this vertex is the person we should tell our rumor to.

SOLUTION 2: The algorithm is:

```
findInfluentialPerson(G):
    Run the SCC-finding algorithm from class to obtain a DAG G'
        on the strongly connected components of G.
    Run the topological sorting algorithm from class on G'.
    Let C be the SCC (vertex in G') that is first in the topological sort returned above.
    Return any vertex v in C.
```

Since both the SCC algorithm and the topological sorting algorithm run in time $O(n + m)$, this algorithm does as well.

(In fact, this is the same algorithm as in SOLUTION 1, just written in a different way).

To see that is correct, we have proven in class that G' is a DAG, so it makes sense to run `topologicalSort`. It suffices to show that if C^* is the SCC in G' that has all of the influential people in it, that this node will be first in the topological sort. Notice that there is no SCC $C \neq C^*$ in G' so that there is a directed edge from C to C^*. If there were, then C and C^* would be the same connected component. Thus, there is no C other than C^* that could come first in a topological sorting. This shows that C^* must come first in any topological sorting, and so the algorithm above is correct.

SOLUTION 3: The algorithm is:

```
findInfluentialPerson(G):
    L = V
    while L is not empty:
        v = any vertex in L
        VISITED = []
        DFS_truncated( VISITED, L, v )
        remove all the vertices in VISITED from L.
    return v
```

```
// DFS_truncated does DFS, but only explores vertices in L.
DFS_truncated(VISITED, L, s):
    if s == NIL or s is not in L:
        return
    VISITED.add(s)
    for v in s.neighbors:
        DFS_truncated(VISITED, L, v)
```

This algorithm does DFS starting at an arbitrary vertex; if DFS finds everything, then this vertex was influential and the algorithm returns true. If DFS doesn't find everything, then we throw away all of the nodes that it found and repeats on the next remaining vertex.

First we analyze the runtime. Each vertex is added to VISITED only once over all the `DFS_truncated` calls, since once it has been visited in a a call of `DFS_truncated`, it is added to VISITED and subtracted from L; it will never be visited by `DFS_truncated` again. Since `DFS_truncated` only traverses directed edges (u, v) if it adds u to VISITED, this means that each directed edge is traversed only once. Thus, the work done between all the calls of `DFS_truncated` is $O(n + m)$. Finally, the extra work in `findInfluentialPerson`, other than some $O(1)$-time bookkeeping, is to pass through VISITED and remove those vertices from L. This can be done in time $O(1)$ per vertex if L is stored as a binary array of length n (that is, $L[u] = 1$ if $u \in L$). Thus, the total cost of this step is also $O(n)$, because as above each vertex appears in VISITED only once.

Next we analyze the correctness. Suppose that v is the first influential vertex chosen in the line

`v = any vertex in L`,

and let $u_1, \ldots, u_m$ be all of the vertices previously picked in that line. We claim that v will be returned. First, we argue that we will not first have returned one of $u_1, \ldots, u_m$. This is because there is no path from any of the u_i to v; if there were, the u_i would have been influential too. So that means that v remains in L (in particular, L is not empty) until it is chosen. Next, we argue that we will return v when it is chosen. This is because it is influential, so everything in L is reachable from v. Thus, `DFS_truncated` will put all of L in VISITED, and L will be emptied. The while loop breaks, and we return v.

(c) We can verify whether or not a vertex is influential by running DFS or BFS. Thus, we can modify our algorithm from the previous part as follows:

```
findInfluentialPersonIfThereIsOne(G):
    v = findInfluentialPerson(G)
    if v is not a vertex of G:
         return "There is no influential person."
         // just in case the algorithm from part (b) returns garbage
         // if there is no influential person
    run DFS starting at v
    if the DFS run above visits all of V:
         return v
    return "There is no influential person."
```

The runtime is $O(m+n)$, because the runtime of DFS is $O(m+n)$, and the runtime of `findInfluentialPerson` is $O(n + m)$.

This algorithm is correct, because if there is an influential person, `findInfluentialPerson` will find them, and then the DFS search will verify that they were influential. On the other hand, if there is no influential person, then whatever v `findInfluentialPerson` returns, DFS will not reach everything starting from v.