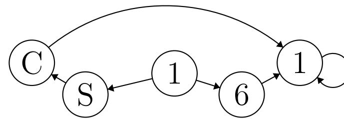
CS 161 **Problem Set 4**
Summer 2017 **Due:** July 28, 2017, 11:59 p.m.

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. **Warmup with DFS/BFS.** (4 points)

   Give one example of a directed graph on four vertices, A, B, C, and D, such that both depth-first search and breadth-first search discover the vertices in the same order when started at A. Give one example of a directed graph where DFS and BFS discover the vertices in a different order when started at A. "Discover" means the time that the algorithm first reaches the vertex, referred to as `start_time` during lecture. 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 DFS and BFS.]

2. **Warmup with Dijkstra.** (9 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 exists 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.

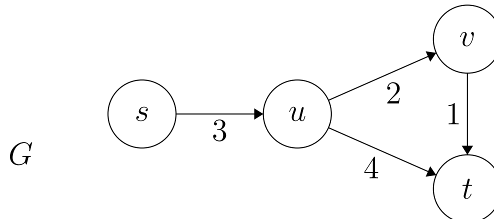
```
dijkstra_st_path(G, s, t):
  for all v in V, set d[v] = Infinity
  for all v in V, set p[v] = None

  // we will use p to reconstruct the shortest s-t path at the end
  d[s] = 0
  F = V
  D = []
  while F isn't empty:
    x = vertex v in F such 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 p to reconstruct the shortest s-t 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. The variable `p` maintains the "**p**arents" of the vertices in the shortest s-t path, so it can be reconstructed at the end.

(a) (1 point) 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.



[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) (1 point) **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

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

such 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 $((u_0, u_1), \ldots, (u_{M-1}, u_M))$ such 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) (2 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) (2 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) (3 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 such 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:

```
modified_dijkstra(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.]

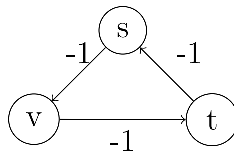
SOLUTION:

(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

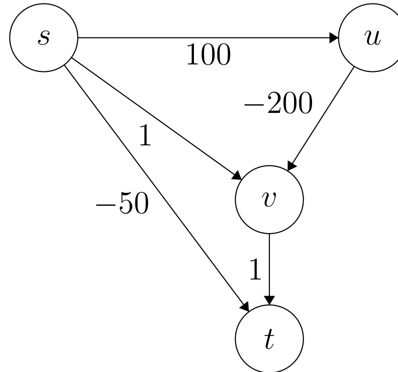
The path returned is $s \to u \to v \to t$ and the cost is $d[t] = 6$.

(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:



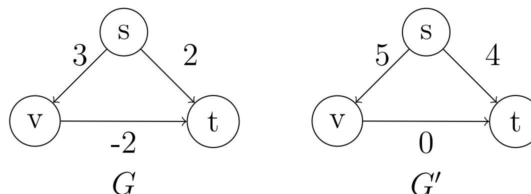
On this graph G, `Dijkstra_st_path`(G, s, t) returns the path $s \to t$ which has cost -50, while the shortest path (which does exist) is $s \to u \to v \to t$ which has cost -99. To check this, we can trace through the functionality of the pseudo-code.

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.



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.

3. **Fun with Reductions.** (3 points)

(a) (3 points) Suppose the economies of the world use a set of currencies $C_1, \ldots, C_n$; think of these as dollars, pounds, Bitcoin, etc. Your bank allows you to trade each currency C_i for any other currency C_j, and finds some way to charge you for this service. Suppose that for each ordered pair of currencies (C_i, C_j), the bank charges a flat fee of $f_{ij} > 0$ dollars to exchange C_i for C_j (regardless of the quantity of currency being exchanged).

Devise an efficient algorithm which, given a starting currency C_s, a target currency C_t, and a list of fees f_{ij} for all $i, j \in \{1, \ldots, n\}$, computes the cheapest way (that is, incurring the least in fees) to exchange all of our currency in C_s into currency C_t. Also, justify the correctness of your algorithm and its runtime.

[**We are expecting a description or pseudocode of your algorithm as well as a brief justification of its correctness and runtime.**]

(b) (2 points extra credit) Consider a robot that can take steps of k distinct lengths $s_1, s_2, \ldots, s_k \in \mathbb{Z}$ in feet. The robot starts at position p_0 on a circular track with circumference length $T \in \mathbb{N}$ in feet and it starts walking counter-clockwise around the track to calibrate itself. It takes one step for each of the lengths $s_1, s_2, \ldots, s_k$, bringing it to position p_1.

Describe an $O(T + kT)$-time algorithm that finds the fewest number of steps that the robot needs to take to get back to position p_0, assuming the robot can't turn around (all steps must be taken in the counter-clockwise direction).

[**We aren't expecting anything; this is a bonus problem.**]

SOLUTION.

(a) **Algorithm:** Build the complete graph on the currencies with weights equal to the corresponding fees; i.e. $G = (V, E, w)$ where $V = C_1, ..., C_n$, $E = (C_i, C_j) : i \neq j$, and $w(C_i, C_j) = f_{ij}$. Run Dijkstras algorithm from C_s; return a shortest $C_s \to C_t$ path.

Correctness: Notice that any path from C_s to C_t in G indicates a sequence of exchanges, and that its total weight is precisely the sum of the fees needed to perform those exchanges. Thus it suffices to find a $C_s \to C_t$ path in G of minimum weight. Note that the f_{ij}s are all positive, so this is a valid input to Dijkstras algorithm.

Running Time: $O(n^2)$ total. Since all exchange pairs are possible, we have $m = \binom{n}{2} = \Theta(n^2)$ edges—this is also how long it takes to build G. Dijkstras algorithm therefore takes $O(n^2 + n \log n) = O(n^2)$ time using the Fibonacci heap implementation. Other implementations are possible; for example, using a red-black tree or binary heap leads to $((m + n) \log n) = O(n^2 \log n)$.

(b) **Algorithm:**

4. **Another topological sort algorithm.** (7 points)

Consider a **complete directed acyclic graph** $G = (V, E)$ where $|V| = n$ and $|E| = \frac{n(n-1)}{2}$. Since the graph is complete, there exists an edge between each pair of the vertices $u, v \in V$ such that either $(u, v) \in E$ or $(v, u) \in E$ but not both (since the graph is acyclic).

Suppose we define a function `contains(u,v)` which returns `True` if $(u, v) \in E$ and `False` if $(v, u) \in E$. Assume that we do not have access to the list of edges and can only access the graph through the `contains` function. (We do have access to the list of vertices though.)

(a) (1 point) Give an asymptotically tight lower bound on the worst-case number of calls to `contains(u,v)` required to find a topological sort of G.

[We are expecting an asymptotic tight lower bound like $\Omega(\ldots)$.]

(b) (3 points) Prove that the bound is tight, i.e., that no algorithm can find a topological sort of G with fewer than that many calls to `contains(u,v)` in the worst-case.

[We expecting a convincing argument.]

(c) (3 points) Describe an algorithm that achieves this bound.

[We are expecting a description or pseudocode of your algorithm.]

SOLUTION.

(a) $\Omega(n \log n)$

(b) The key insight is that a topological sort produces a total order on the vertices, and we can say that vertex 1 is "less than" vertex 2 if there is an edge pointing from vertex 1 to vertex 2. Since the `contains` function essentially makes comparisons, an algorithm that sorts vertices using only information from the `contains` function should be subject to the comparison sort lower bound. This bound is $\Omega(n \log n)$.

We may argue more formally by creating a reduction from the sorting problem to the topological sort problem. Suppose we have an algorithm M that topologically sorts a complete directed acyclic graph using no information about the graph other than the vertex labels (which are used purely for labeling purposes) and the outputs of fewer than $O(n \log n)$ calls to `contains`. Let A be an unsorted array, and define an interface to the corresponding graph G as follows:

```
let the vertex labels be the elements of A
def contains(A[i], A[j]):
    return A[i] < A[j]
```

(Note that we do not have to explicitly construct G to run our algorithm M, since M only interacts with G through the `contains` function. In fact, constructing G would take $\Theta(n^2)$ comparisons, so we want to avoid that if possible.)

Now we can build a comparison sorting algorithm as follows:

```
Construct our interface to G
Run M using our interface, which takes fewer than O(n log n) comparisons,
    and uses no other information about A other than the names of the elements
M outputs an ordered list of vertex labels
Copy this list into the output array
```

This sorts the input array using fewer then $O(n \log n)$ comparisons and no other information about the elements in the array (other than the labels of the elements), which violates the comparison sort lower bound. Therefore, M cannot exist.

(c) Let A be an array containing the vertices (producing this array costs $O(n)$ if it does not exist already). Then we can run merge sort on this array, except whenever two elements u and v are supposed to be compared, we replace it with a call to `contains(u, v)`. That is, our new Merge routine is

```
Merge(L, R):
    m = length(L) + length(R)
    S = empty array of size m
    i = 1; j = 1
    for k = 1 to m:
        if contains(L[i], R[j]) is true:
            S[k] = L[i]
            i = i + 1
        else:
            S[k] = R[j]
            j = j + 1
    return S
```

Since `contains` takes constant time (just like a comparison), this new algorithm takes $O(n \log n)$.

5. **Social engineering.** (9 points)

Suppose we have a community of n people. 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 an influential person P, eventually the rumor will reach everybody. You have a rumor that you'd like to spread, 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) (1 point) 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) (5 points) Suppose that an influential person exists. Give an algorithm that, given G, finds an influential person in time $O(n + m)$.

[We are expecting a description or pseudocode of your algorithm, a proof of correctness, and a short argument about the runtime.]

(c) (3 points) 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 a description or pseudocode or your algorithm.]

Solutions

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's a path from v' to v to u.

b There's (at least) 3 different good solutions.

Solution 1.

`Run DFS and keep track of finishing times.`

`Return the vertex with the largest finishing time.` The runtime is just 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 $C_1, ..., C_r$ 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.

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^*.finish > C.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.finish > C'.finish. This implies that if there is a path from C the C' in

the SCC DAG, that C.finish > C'.finish, even if there is no edge directly from C to C'. Finally, Claim1 implies that C^*.finish > C.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 V.
Proof. By definition, C^*.finish is the maximum value of v.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 $w^* \in C^*$ and 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

```
Run the SCC-finding algorithm from class to obtain DAG G'
        on strongly connected components of G.
Run the topolotical 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 v in C.
```

Both SCC and toposort are $O(n+m)$, so this is as well.
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.

```
findInfluentialPerson(G):
    L = V
    while L is not empty:
        v = any vertex in L
        VISITED = []
```

```
        DFS_truncated(VISITED< L, v)
        remove all vertices in VISITED from L
    return v

// 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 doesnt 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 $u_1, ..., u_m$ 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 $u_1, ..., u_m$. This is because there is no path from any of theu_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.

6. **How is the course so far?** (0.5 points extra credit)

   Please complete the poll at `https://goo.gl/forms/8cKzJ2u08evIc6EM2`.