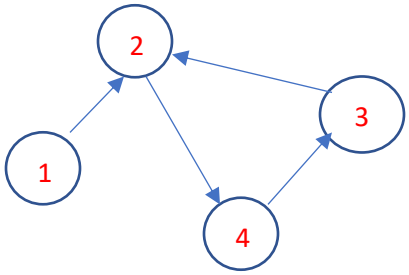
Poonam Gupta

COT 5407 – Introduction to Algorithms **Homework Assignment #5**

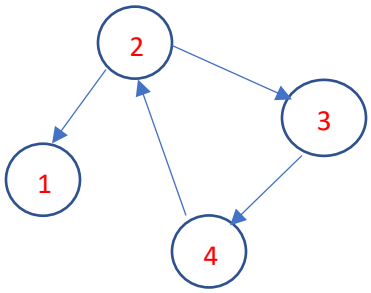
Problem 1 (1.5 points) Transpose of a Directed Graph

The transpose of a directed graph $G = (V, E)$ is a graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V: (u, v) \in E\}$. Thus, G^T is G with all the edges reversed. Give an efficient algorithm to compute G^T from G using the adjacency list representation and analyze its running time.

G



G^T



G^T is the result of reversing all edges in G. A new adjacency list must be constructed for G^T. Every list in adjacency list of G is scanned. While scanning adjacency list of v, if we encounter u, we put v in adjacency list of u.

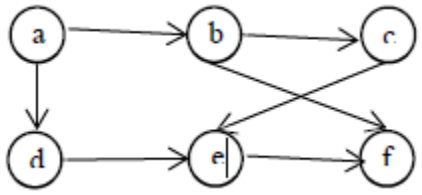
Transpose(G)
for u = 1 to V(G)
    for each element $v \in$ adj[u]
    insert u into the front of adj[v]

Complexity: we are traversing lists of all vertices of graph G that takes O(V+E) time and to create the linked list of G^T takes O (V+ E) time. Thus, the overall time complexity of this algorithm is O(V+E).

Problem 2 (3 points) Reachability

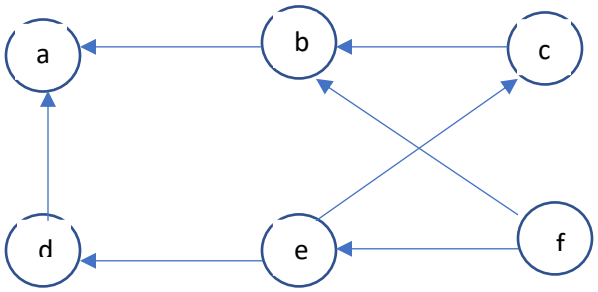
Let $G = (V, E)$ be a directed graph in which each vertex in $u \in V$ is labeled with a unique integer $L(u)$ from the set {1, 2, …, $|V|$}. For each vertex $u \in V$, let $R(u) = \{v \in V: u \rightsquigarrow v\}$ be the set of vertices that are reachable from u (a node can always reach itself by default). Let $\min(u)$ be the vertex in $R(u)$ with the minimum label, i.e. $\min(u)$ is vertex v such that $L(v)$ is the smallest among all other nodes in $R(u)$. Describe an $O(V + E)$ time algorithm that computes $\min(u)$ for all vertices $u \in V$, given the nodes are already sorted in increasing order of their labels. Illustrate your algorithm on this diagram, i.e. give $\min(u)$ for every node, where $L(a) = 2$, $L(b) = 4$, $L(c) = 1$, $L(d) = 5$, $L(e) = 3$, $L(f) = 6$.

//Hint: Depth-First Search is used.



Begin by locating the element v of minimal label. We would like to make u.min = v.label for all u such that $u \rightsquigarrow v$. Equivalently, this is the set of vertices u which are reachable from v in G^T. We can implement the algorithm as follows, assuming the u.min is initially set equal to Nil for all vertices $u \in V$, and simply call the algorithm on G^T .

G^T



Adj(c) -> {b}
Adj(a) -> {}
Adj(e) -> {d, c}
Adj(b) -> {a}
Adj(d) -> {a}
Adj(f) -> {b, e}

Assuming vertices are already sorted by their label in increasing order. In the example above it is: $L(c)$ = 1, $L(a)$ = 2, $L(e)$ = 3, $L(b)$ = 4, $L(d)$ = 5, $L(f)$ = 6

Reachability(G)

1. For each vertex $u \in V$ do
2. If u.min== nil then
3. Reachability-visit (u, u.label)
4. End if
5. End

Reachability-visit (u, labelofvertex)

1. u.min = labelofvertex
2. For ($v \in$ G.adj[u] do
3. If v.min == nil then
4. Reachability-visit (v, labelofvertex)
5. End if
6. end

As, we process the above algorithm we obtain the following results for u.min as min(c) = 1, min(b) = 1, min(a) = 1, min(e) = 3, min(d) = 3, min(f) = 6.

Thus, it takes O(V+E) to do the graph traversal (DFS).

Problem 3 (6 points) Second-Best Minimum Spanning Tree

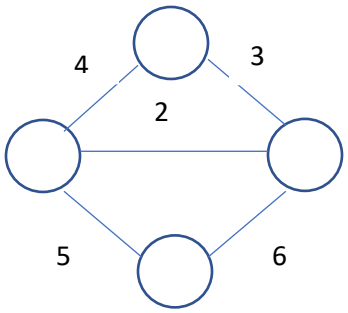
Let G = (V, E) be an undirected, connected graph whose weight function is w: E → $\Re$, wand suppose |E| ≥ |V| and all edge weights are distinct. Let Γ be the set

of all spanning trees of G, and let T be a minimum spanning tree of G, a second-best minimum spanning tree is a spanning tree T' such that $w(T') = \min_{T'' \in \Gamma - \{T\}} \{w(T'')\}$, i.e. T' is a minimum spanning tree after removing T.

(1) (1 point) Show that the minimum spanning tree is unique (//provide logical arguments),

Let's assume T and T' are two different minimum spanning trees.
There is at least one edge $(u, v) \in T'$ and $(u, v) \notin T$.
Then $T \cup (u, v)$ must form a cycle including (u, v).
Let (x, y) be the heaviest edge in the cycle.
According to the cycle property, (x, y) can't be contained in any MST.
If (x, y) $\in T$ (in T' resp), then T (T' resp) is not an MST, which contradicts to our assumption. Therefore, uniqueness of MST is proved.

but the second-best minimum spanning tree need not be unique (//give a simple example);



In the above graph, the best MST consists of edges of weights (2,3,5) and total is 10.
There are two second- best MST consists of edges having weights (2,4,5) and (2,3,6) total is 11.

(2) (1 point) Let T be the minimum spanning tree of G. Prove that G contains edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G (hint: use logical arguments to reason that a second-best minimum spanning tree can only be obtained by replacing no more than one edge);

Any spanning tree has exactly $|V|$-1 edges, any second-best minimum spanning tree must have at least one edge that is not in the best minimum spanning tree.

If a second-best minimum spanning tree has exactly one edge, i.e. (x, y), that is not in the minimum spanning tree, then it has the same set of edges as the minimum spanning tree, except that (x, y) replaces some edge, say (u, v),of the minimum spanning tree. In this case $T' = T - \{(u, v)\} \cup \{(x, y)\}$.
Here we will show that, if we replace two or more edges of the minimum spanning tree, we cannot obtain a second-best minimum spanning tree.

Let T be the minimum spanning tree of G.
T' be the second best minimum spanning tree differ from T by two or more edges.
There are at least two edges in $T - T'$ and let (u, v) be the edge in $T - T'$ with minimum weight. If we were to add (u, v) to T, we would get a cycle c. This cycle contains some edge (x, y) in $T' - T$ (since otherwise, T would contain a cycle).
We claim that w (x, y)> w (u, v).
Let's assume w (x, y) < w (u, v) //weights are distinct.
If we add (x, y) to T, we get a cycle c', which contains some edges (u', v') in $T' - T$.
Therefore, the set of edges $T'' = T$ - $\{(u', v')\} \cup \{(x, y)\}$ forms a spanning tree, and we must also have w (u', v') < w (x, y),since otherwise T'' would be a spanning tree with weight less than w(T).
Thus, w (u', v') < w (u, v), which contradicts our choice of (u, v) as the edge in $T - T'$ of minimum weight.
Since the edges(u, v) and (x, y) would be on a common cycle c if we were to add (u, v) to T', the set of edges T' - $\{(u, v)\} \cup \{(x, y)\}$ is a spanning tree, and its weight is less that w(T'). Moreover, it differs from T. Thus, we have formed a spanning tree whose weight is less than w(T') but is not T. Hence, T' was not a second-best minimum spanning tree.

(3) (2 points) Let T be a spanning tree of G and, for any two vertices $u, v \in V$, let max [u, v] denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$- time algorithm that, given T, computes max [u, v] for all $u, v \in V$ (hint: either BFS or DFS can be used to compute max);

Here we are using BFS from each of $|V|$ vertices. We used table max itself to keep track of visited vertices instead of coloring.

Compute-max (T, w)

```
1.  For each vertex u ∈ T.V
2.      For each vertex v ∈ T.V
3.          max [u, v] = nil
4.      Q = ∅
5.      Enqueue (Q, u)
6.      While Q ≠ ∅
7.          X = Dequeue(Q)
8.          For each vertex v ∈ T.adj[x]
9.              If max[u, v] = nil and u ≠v
10.                 If x= u or w(x, v) > max[u,x]
11.                     max[u, v] = (x, v)
12.                 else
13.                     max[u, v] = max[u,x]
14.                 Enqueue (Q, v)
15. return max
```

Algorithm runs a BFS, and thus takes O(V(V+E)) time. observing that a spanning tree has exactly $|V|$-1 edges, we can restate the running time as $O(V^2)$.

(4) (2 points) Give an efficient algorithm to compute the second-best minimum spanning tree of G.

We can obtain a second-best minimum spanning tree by replacing exactly one edge of the minimum spanning tree T by some edge $(u, v) \notin T$.

If we create spanning T' by replacing edge $(x, y) \in T$ with edge $(u, v) \notin$, then $w(T') = w(T) - w(x, y) + w(u, v)$.

For a given edge (u, v), the edge $(x, y) \in T$ that minimize $w(T')$ is the edge of maximum weight on the unique path between u and v in T, that edge is max $[u, v]$.

Thus, our algorithm needs to determine an edge $(u, v) \notin T$ for which w (max $[u, v]$) – w (u, v) is minimum:

Second-best-MST(G,w)

1. T = MST-PRIM(G,w)
2. max = compute-max(T,w)
3. minedge = nil
4. minvalue = ∞
5. for each edge (u, v) ∈ G.E and (u,v) ∉ T.E
6. value = w (max $[u, v]$) – w (u, v)
7. if value < minvalue
8. minedge = (u, v)
9. minvalue = value
10. return T – {max[minedge]} U {minedge}

Analysis: MST-PRIM takes O (E log(v)) time, compute takes O(V^2), for loop take O(E). For a dense graph with $|E|$close to $|V^2|$, algorithm takes O (V^2 log V) time.

The running time of MST-PRIM can be improved to O (E + V lg V) by using Fibonacci heaps. This brings the running time of SECOND-BESTMST down to O(V^2).

Problem 4 (4.5 points) Single Source Shortest Path

A d-dimensional box with dimensions (x_1, x_2,…, x_d) nests within another box with dimensions (y_1, y_2,…, y_d) if there is a permutation π on {1,2,…, d} such that $x_{\pi 1} < y_1$, $x\pi_2 < y_2$,…, $x\pi d < y_d$.

(1) (1 point) Argue that nesting relation is transitive;

Let the boxes are x = (x_1, ..., x_d), y = (y_1, ..., y_d), z = (z_1, ..., z_d)
If x nests inside y and y nests inside z, there exist permutations π_1 and π_2 such that:

$x_{\pi 1 (i)} < y_i$ for i = 1, ..., d
$y_{\pi 2 (i)} < z_i$ for i = 1, ..., d

Then, for permutation π_3 (i) = π_2 (π_1 (i))

$x_{\pi 3 (i)} < y_{\pi 2 (i)} < z_i$ for i = 1, ..., d

So, x nests inside z.

(2) (1 point) Describe an efficient method to determine whether or not one d-dimensional box is nests inside another;

Box x = (x_1, ..., x_d) nests inside box y = (y_1, ..., y_d)
if and only if $x'_i < y'_i$ for i = 1, …, d
Where x' and y' are sorted dimensions of x and y respectively

Check-if-Nests (x, y, d)

```
1.  Sort(x)        // sort the dimension of x in ascending order
2.  Sort(y)
3.  for i =1 to d          // loop for checking each dimension
4.          If (x_i ≥ y_i )
5.          return false
6.  return true
```

Analysis: Sorting the dimension of boxes takes O (d log d) time, and comparing their elements takes O(d) time, so overall complexity is O (d log d)

(3) (2.5 points) Suppose that you are given a set of nd-dimensional boxes {B_1, B_2,…, B_n}. Give an efficient algorithm to find the longest sequence < Bi_1 , B_{i2} ,…, B_{ik} > of boxes such that Bi_j nests within Bi_{j+1} for j = 1,2,…, k − 1. Express the running time of your algorithm in terms of n and d.

We can break down the problem above into the following subproblems

a) Does a d-dimensional box B_x fits inside another d-dimensional Box_y (nesting)?
b) Use the solution to above to identify all possible nesting combinations for all boxes {$B1$, $B2$,…, Bn}
c) Now create a graph where boxes appear as vertices and the edge from B_a to B_b represent the Box B_a can be nested in Box B_b .
d) Then we add another node source, s, with edges connected to all vertices which creates a connected graph. Leveraging Single-source Shortest path problem, we assign weights of the edges such that the shortest path has the longest sequence of boxes.

As we observed in the solution to problem 4.1 and 4.2, we can use the transitive property and the Check-if-Nests () method defined above.

Then we leverage the following algorithm to find the longest sequence of boxes that can be nested.

```
CREATE-GRAPH(B,n)
    E = Ø, V = Ø                                  // Initialize empty sets
    for i = 1 to n – 1                            //Loop through n to n-1 boxes
        V = V U {i}
        for j = i + 1 to n
            if (Check-if-Nests (Bi, Bj,d))// Check if can be nested
                E = E U {(i, j)}
                w(i, j) = –1
    V = V U {0}                                   //Set up source node s
    for i = 1 to n
        E = E U {(0, i)}                          //Setup edge to each vertex
        w(0, i) = – 1                             //establish a weight as -1 for now
    G = (V, E)
    return G                                      //return the connected graph
DAG-SHORTEST-PATHS (G, w, s)
    for each vertex v Є V [G]                     // Initialize single source, s.
        do d[v] = ∞
            π[v] = NIL
        d[s] = 0
    for u = 0 to n                                //For each vertex in the DAG
        do for each vertex v such that (u, v) Є E
            do RELAX(u, v, w)                     //relax edges
PRINT-SEQUENCE (seq)                              //Use recursion to identify the order
                                                  of vertices from final to source
                                                  (ignoring source)
    if π[seq] = NIL
        return STACK
    else
        PUSH (seq)
        PRINT-SEQUENCE (π[seq])                   //Recursively call this method till
                                                  source
RELAX (u, v, w)                                   //Find shortest distance
    if d[v] > d[u] + w (u, v)                     //Use relaxation
        d[v] = d[u] + w (u, v)
        π[v] = u                                  //Set preceding vertex to π
```

NESTED-BOXES-SEQUENCE (B, n)

```
for i ← 1 to n
        SORT-BOX (B_i)                  // Sort the boxes by shorted edges
SORT-BOXES (B, n)                       // Topological ordering of boxes by
                                           shortest edge
G = CREATE-GRAPH(B,n)                   //Create graph
DAG-SHORTEST-PATHS (G, w, s)            // Use single source shortest paths
sequence = FIND-PATH (G)                // Return vertex with smallest dist
PRINT-SEQUENCE (sequence)               // Show the longest sequence
```

Analysis of the algorithm (NESTED-BOXES-SEQUENCE (B, n)):

Using heaps, since there are d edges to be sorted, each box can be sored in the time O (d log d).

The for-loop iterates n times, and sorting n boxes takes time in the order of O (nd log d). SORT-BOXES takes time in the order O (n logn).

CREATE-GRAPH takes time $O(n^2d)$. DAG-SHORTEST-PATHS takes time O (n^2).

Sequence and PRINT-SEQUENCE tales time O(n).

Hence the running time of the entire algorithm is:

O (n d log d + n logn+ n^2d + n^2 + n+ n) = O (n^2d + nd log d)