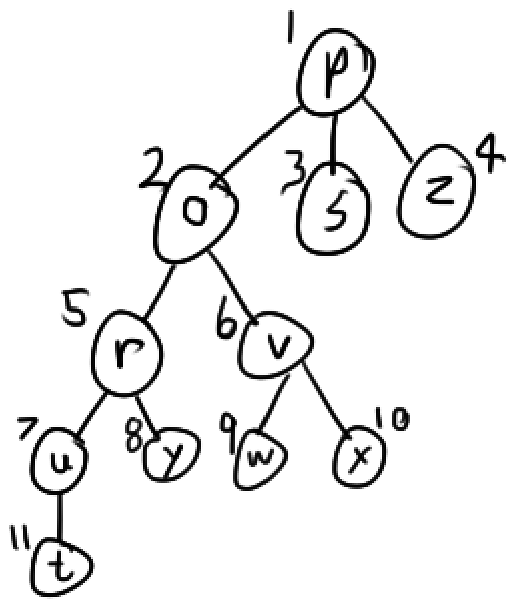
CSC263, Assignment #6

Brendan Neal[1], Filip Tomin[2]

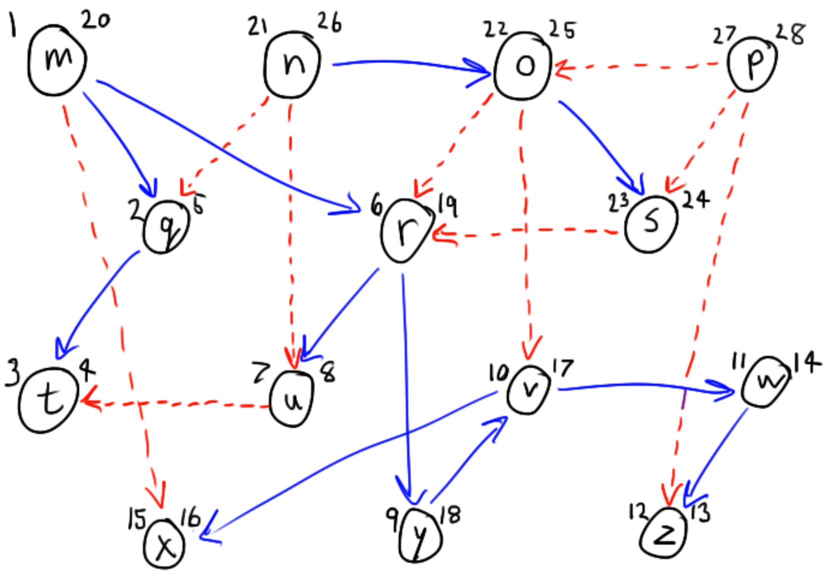
[1]Section L0201 Student number: 1001160236

[2]Section L0201 Student number: 1001329984

1. (a) Here is the result of BFS(p). The number outside the node corresponds to its discovery value, $d[a]$.



   (b) Here is the result of DFS(G). The numbers outside the node correspond to the node's discovery value $d[a]$ on the left and its explored value $f[a]$ on the right. Any blue lines are edges while any red lines are dotted edges, where the node found was not white.



(c) For an edge (a, b) in the DFS graph G, the edge is a back edge only when it completes a cycle (it is the last node in a cycle). Given $d[a], d[b], f[a]$, and $f[b]$, this means that the edge (a, b) is a back edge when a is an ancestor of b, or:

$$d[b] < d[a] < f[a] < f[b]$$

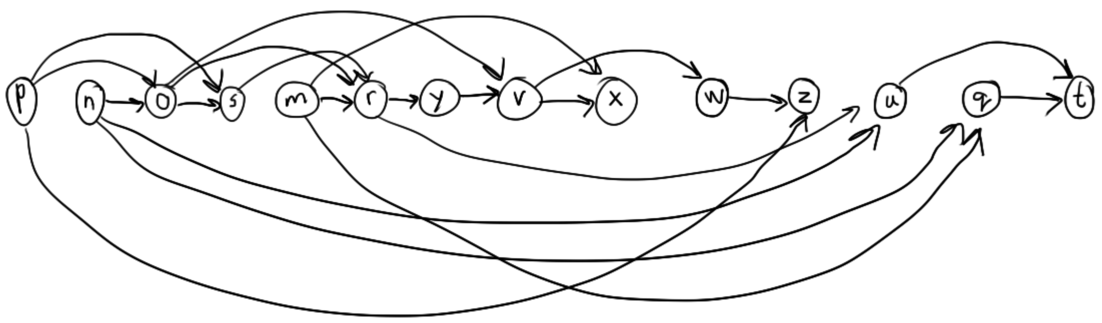
When given a cycle of an arbitrary size and running a DFS on a node, let's call it b, the DFS must loop through each edge within the cycle, incrementing the discovery value along each node, thus $d[b] < d[x]$ for any other node x in the cycle. This continues until we hit the last node in the cycle, node a, which has an edge to b, since b is not white, a is then fully explored, and thus $d[x] < d[a] < f[a]$. The DFS then moves back through the cycle incrementing the explored value along the way, such that $f[a] < f[x]$ for any other node x in the cycle. We end up back at b, where it has now been fully explored and thus has the last explored value, $f[b]$. Then $f[x] < f[b]$.
By the rules stated above, where $d[b] < d[x] < d[a]$, $d[a] < f[a]$, and $f[a] < f[x] < f[b]$, we can concatenate and shorten these to obtain $d[b] < d[a] < f[a] < f[b]$. This is true for any arbitrary cycle of any size, regardless of whether or not a or b have additional nodes attached to them, as that only delays $f[a]$ and $f[b]$ which still must satisfy $f[a] < f[b]$.

(d) We will use Lemma 22.11 from the textbook, which states: A directed graph G is acyclic iff a DFS of G yields no back edges. In a DFS, any back edge would show up as a dotted edge, so we can list every dotted edge in the DFS above and show that none of them follow the rule stated in (c).

Edge (a, b)	$d[b]$	$d[a]$	$f[a]$	$f[b]$	Fail State
(m, x)	15	1	20	16	$d[b] > d[a]$
(n, q)	2	21	26	5	$f[a] > f[b]$
(n, u)	7	21	26	8	$f[a] > f[b]$
(o, r)	6	22	25	19	$f[a] > f[b]$
(o, v)	10	22	25	17	$f[a] > f[b]$
(p, o)	22	27	28	25	$f[a] > f[b]$
(p, s)	23	27	28	24	$f[a] > f[b]$
(p, z)	12	27	28	13	$f[a] > f[b]$
(s, r)	6	23	24	19	$f[a] > f[b]$
(u, t)	3	7	8	4	$f[a] > f[b]$

As denoted by the table above, none of the dotted edges in the DFS of G follow the rule stated in (c), thus there are no back edges in the DFS. By the lemma, since there are no back edges, the graph is acyclic and thus has no cycles.

(e) Doing a topological sort on the DFS of G shown above results in:



2. (a) By the definition of $d(S, \overline{S})$, $d(S, \overline{S})$ is the shortest distance between two separated components of a graph G, S and $\overline{S}$. This minimum distance would be the weight of some edge e which connects some node i in S and some node j in $\overline{S}$.
Let's say T is the MST of this graph G. By the definition of a MST, each edge e in T must have the least weight attached to it, or the least distance.
If we have a complete MST T, and remove some edge e to form two components S and $\overline{S}$, then by the definition of an MST, this edge would have the least distance between S and $\overline{S}$, as it would not have existed in the MST if it was not the minimum distance required to connect the two components. As such, $w(e)$ would be the smallest distance between S and $\overline{S}$, since $d(S, \overline{S}) = min_{i \in S, j \in \overline{S}} D[i, j]$, and $min_{i \in S, j \in \overline{S}} D[i, j] = w(e)$, then $d(S, \overline{S}) = w(e)$.
Since $w(e)$ is defined as $w(e) = D[i, j]$, then $d(S, \overline{S}) = w(e) = D[i, j]$.

(b) We have verified in 2(a) that $d(S, \overline{S}) = w(e) = D[i, j]$ when an MST is "split" over the edge connecting nodes i and j. We also know that $d(S, \overline{S})$ is equal to the minimum possible distance between a point in S and a point in $\overline{S}$. Our algorithm, expressed in pseudocode, is as follows:

```
An edge e that connects nodes i and j with weight w will be such that
e.first = i, e.second = j, e.weight = w. Such an edge can be created
with new_edge(first = i, second = j, weight = w).

remove_edge uses the forest implementation of the disjoint-set ADT

T.remove_edge(e = [i,j]):
    T.edges.remove(e)
    edges = T.edges
    clusters = {T.nodes}
    for node in clusters:
        MakeSet(node)
    for edge in edges:
        x = FindSet(edge.first)
        y = FindSet(edge.second)
        Union(x, y)
    S = clusters[0]
    S’ = clusters[1]
    return (S, S’)

1 cluster(G):
2   edges = []
3   vertices = []
4   for i from 1 to (G.width - 1):
5       vertices.append(i)
6       for j from 1 to i:
7           edges.append(new_edge(i, j, G[i, j]))
8   MST = Prim’s_Algorithm(vertices, edges)
9   biggest_edge = NULL
10  for edge in MST.edges:
11      if edge.weight > biggest_edge.weight:
12          biggest_edge = edge
13  (S, S’) = MST.remove_edge(biggest_edge)
14  return (S, S’)
```

Essentially, our algorithm will first convert the given adjacency matrix into an adjacency list (lines 2-7). Next, it will use the adjacency list to build an MST using Prim’s algorithm (line 8). After doing this, the algorithm then looks through all the edges of the MST to determine the edge with the maximum weight/distance (lines 10-12). Finally, the algorithm removes that edge from the MST to create two separate sets of nodes/points using remove_edge (line 14).

Our algorithm is correct because according to 2(a), we know that the distance between two clusters of nodes (connected together by two parts of a minimum-spanning tree) is equal to the distance of the edge that connects the two parts of the MST. Since our algorithm is looking for the largest/longest edge of the MST and 'splits' the MST on that edge, the distance between the two clusters of nodes must be equal to the weight/distance of the largest/longest edge.

(c) The first step of our algorithm is to convert the adjacency matrix into an adjacency list. We know that the $n \times n$ adjacency matrix is symmetric and only has zeroes along it's main diagonal. Therefore, we can ignore n edges from the adjacency matrix because we know that their weights are 0. Since the matrix is symmetric, we only need to consider the edges that lie above (or below) the main diagonal. There are $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{n}$ edges in this region of the matrix. Thus, we only need to loop through $\frac{n(n-1)}{n} - n = \frac{n^2-3n}{2} \in O(n^2)$ slots in the adjacency matrix in order to collect all the information we need about the nodes and edges. This collection of relevant edges and all nodes is done in lines 2-7 of the algorithm's pseudo-code. By inspection, we see that we are executing two nested for loops for this step that run in $O(n)$ time each. Therefore, this section of the algorithm runs in $O(n^2)$ time.

The next step of the algorithm is to use Prim's Algorithm to build an MST from the adjacency list created above. According to CLRS Chapter 23.2, Pg. 633, Prim's algorithm produces an MST in $O(E + V \log V)$ time (using a Fibonacci heap to implement a priority queue of edges), where E is the number of edges and V is the number of vertices/nodes. We know from the above that the number of edges is on the order of n^2 and the number of vertices is on the order of n. Therefore, using Prim's algorithm to produce the MST from the adjacency list and list of vertices should result in a worst-case run-time in $O(n^2 + n \log n)$.

The third part of the algorithm then looks through all the edges of the MST produced from the previous step to determine the edge of maximum weight. Since an MST has $n-1$ edges (because a tree with n nodes has $n-1$ edges), we only need to execute linear search over this set of edges. Since linear search's run-time is $O(n)$ for an input of size n, this step will take $O(n-1) \in O(n)$ time.

The final part of this algorithm is to split the MST into two separate sets of nodes. This is accomplished by calling remove_edge, which uses the forest implementation of the Disjoint-Set ADT to group together the two separated groups of nodes. We verified in a previous assignment (Assignment 4) and in tutorial that using the forest implementation of the disjoint set ADT to group together the nodes of a graph with m edges and n nodes takes $O(n + m \log^* n)$ time. Since the number of edges we are feeding into remove_edge is $n-1-1=n-2 \in O(n)$, and there are $n \in O(n)$ nodes in the MST, remove_edge will take $O(n+n\log^* n)$ time.

Therefore, by summing the run-times of all the parts of the algorithm, we get $O(n^2)+O(n^2+n\log n)+O(n)+O(n+n\log^* n)$. Since $O(n^2+n\log n), O(n)$, and $O(n+n\log^* n)$ are all within $O(n^2)$, this algorithm will take $O(n^2)$ time in the worst-case, where n is the number of points. This algorithm can be said to be "efficient" because it runs in polynomial time.