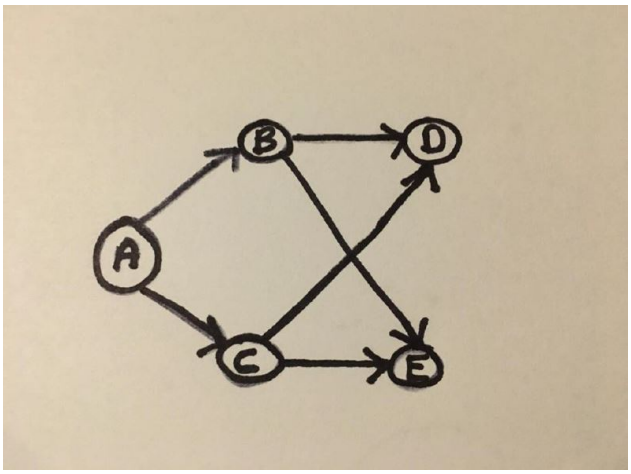
CSCI 3104 Spring 2017 **Problem Set 7**

Matthew Niemiec (06/24)
Mehmet Karaoglu (07/30)

1. (10 pts) *Hermione needs your help with her wizardly homework. She's trying to come up with an example of a directed graph $G = (V, E)$, a source vertex $s \in V$ and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique path in the graph (V, E_π) from s to v is a shortest path in G, yet the set of edges E_π cannot be produced by running a breadth-first search on G, no matter how the vertices are ordered in each adjacency list. Include an explanation of why your example satisfies the requirements.*



The graph shown above is an example of a graph that contains a set of tree edges $E_\pi \subseteq E$ such that for each $v \in V$, the path from s to v is a shortest path in G. The particular edges in E_π are:

$(A, B), (A, C), (C, E), (B, D)$

The tree that contains these edges will give us the shortest path from s to any of the other vertices because we simply can't find any other path that is shorter for each vertex. The distance to B and C is 1 and the distance to D and E is 2 (both are shortest distances). Yet when we run Breadth-First on G, we can *never* produce the unique edges in E_π. The idea of the Breadth-First search algorithm is that we enqueue

into our queue every single adjacent vertex of the vertex which we're currently on, and dequeue vertices from the queue in a FIFO(first in, first out) fashion. That is, we mark as visited every vertex at a distance d from the source vertex before moving on to vertices located at distance $d + 1$.

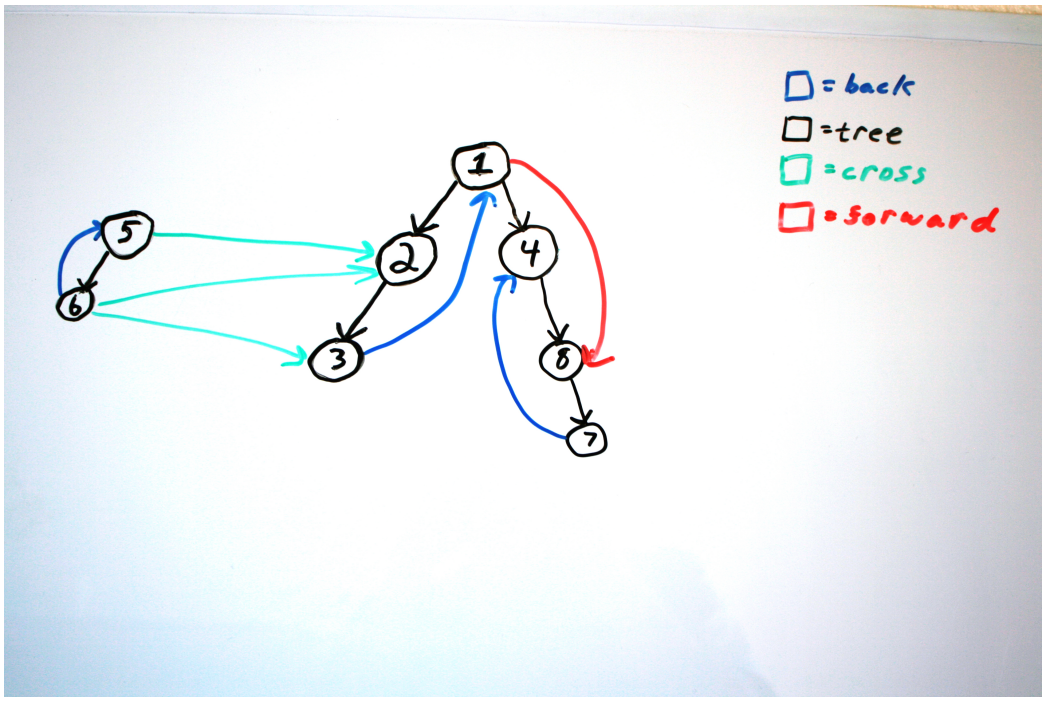
For our graph, we begin at our source vertex A, and after we dequeue it, we discover vertices B and C, and the edges (A, B) and (A, C), which are included in E_π. Next, BFS can either visit the adjacent vertices of either B or C, depending on how the vertices are ordered in the adjacency list. If it visits the vertices connected to B, it will discover vertices D and E, and add edges (B, D) and (B, E) to the tree. After D and E are marked visited and C is dequeued, we don't have any more unvisited vertices in the graph, so BFS terminates. This means that edge (C, E) will not be produced by BFS in this case. The same idea also applies if instead C is dequeued first: BFS will still visit both D and E, so when it dequeues B after, it will not produce the edge (B, D). The conclusion that we draw is that edges (B, D) and (C, E), which are in E_π, can never be produced in the *same* running of BFS since D and E are adjacent to both B and C.

2. (10 pts) *Review the material on depth-first spanning forests in Chapter 22.3 of the textbook. Then, consider the directed graph G defined by the edge list*

$$G = \{(1,2),(1,4),(1,8),(2,3),(3,1),(4,8),(5,2),(5,6),(6,2),(6,3),(6,5),(7,4),(8,7)\}$$

(a) *Draw the depth-first spanning forest, including and identifying all trees, back, forward, and cross edges. (If you prefer, you can identify the forward, back, and cross edges in separate lists instead of trying to draw and label them.)*

If you draw out the graph and run DFS on it starting with 1, you get the following graph. Note how nodes 5 and 6 are in their own tree linked by cross edges. They are in their own tree because a tree DFS will not find them - there is no path from 1, the starting point, to either 5 or 6. Next, (4,8) is a tree edge because when running DFS, the algorithm will go from 4 to 8 before it checks (1,8). Therefore, when that edge is checked, it is a forward edge.



(b) *List all the strong components in G.*

5 and 6 is a two-node cycle with in-degrees 0, so 5 and 6 are a strong component. 1, 2, and 3 form a cycle, so they are also a strongly connected component. Finally, 4, 8, and 7 are a cycle with no way out, so they are a strongly connected component.

3. (30 pts) *Prof. Snape claims that when an* adjacency matrix *representation is used, most graph algorithms take $\Omega(V^2)$ time, but there are some exceptions.*

   One such exception, Snape claims, is determining whether a directed graph G contains a black hole, which is defined as a vertex with in-degree $|V|-1$ (i.e., every other vertex points to this one) and out-degree 0. Snape sourly claims that this determination can be made in time $O(V)$, given an adjacency matrix for G. Prove that Snape is correct.

   Hint: This is a "proof by algorithm" question. That is, describe and analyze an algorithm that yields the correct answer on every type of input ("yes" when a black hole exists and "no" when it does not). Think about the boundary cases, where it is the most difficult to distinguish a correct "yes" from a correct "no."

   The first important thing to note is that there can only be one black hole in the graph. Thus, whenever we compare any two edges $Mat[i][j]$ and $Mat[j][i]$ we can eliminate at least one node for being the black hole. There are three cases - one where $Mat[i][j] == Mat[j][i] == 0$, where $Mat[i][j] == 1$ and $Mat[j][i] == 0$, and where $Mat[i][j] == Mat[j][i] == 1$. Thus, if we keep track of which matrices we know aren't the black hole, then we will have to make at most $|V|-1$ comparisons to determine if there is a black hole or not. Consider the following pseudocode:

```
boolean isBlackHole(A)
    boolean holeVector[A.length] = [true]*A.length
    int current = 1
    // Check if every node is a black hole
    while current <= A.length
        // If node was previously marked, there is no need to check it
        if holeVector[current] == false
            current++
            continue
        // Compare the current node with every other node
        // This is the only way to test for a black hole
        for i=1 to A.length
            // Case where neither is black hole
            if A[current][i] == A[i][current]
                holeVector[current] = holeVector[i] = false
                break
            // Case where current goes into i
            // Current is not the black hole
```

```
            if A[current][i] == true
                holeVector[current] = false
                break
        if holeVector[current]
           return true  // There is a black hole, because it didn't get marked
    return false  // All nodes were checked and they were all false
```

Here the algorithm is $O(V)$ because we compare any node at most n times. In the worst case, imagine that for current=1, we have to check every single node, only we find out that it isn't the black hole on the last comparison. This is fine - we then only have to check one more node, because we falsified all of the rest of the nodes, and this one will run O(V) operations. Thus $O(V) + O(V) = O(V)$. In the other worst case, imagine that we made a single comparison for each node. Then we have made $O(V)$ comparisons, which obviously runs in $O(V)$ time. Therefore it is possible to check for a black hole on an adjacency matrix in $O(V)$ time.

4. (30 pts) *Deep in the heart of the Hogwarts School of Witchcraft and Wizardry, there lies a magical Sphinx that demands that any challenger efficiently convert directed multigraphs into undirected simple graphs. If the wizard can correctly solve a series of arbitrary instances of this problem, the Sphinx will unlock a secret passageway.*

   Let $G = (E, V)$ denote a directed multigraph. An undirected simple graph is a $G' = (V, E')$, such that E' is derived from the edges in E so that (i) every directed multi-edge, e.g., $\{(u, v), (u, v)\}$ or even simply $\{(u, v)\}$, has been replaced by a single pair of directed edges $\{(u, v), (v, u)\}$ and (ii) all self-loops (u, u) have been removed.

   Describe and analyze an algorithm (explain how it works, give pseudocode if necessary, derive its running time and space usage, and prove its correctness) that takes $O(V + E)$ time and space to convert G into G', and thereby will solve any of the Sphinx's questions. Assume both G and G' are stored as adjacency lists.

   Hermione's hints: Don't assume adjacencies `Adj[u]` *are ordered in any particular way, and remember that you can add edges to the list and then remove ones you don't need.*

   Our algorithm works as follows: First, we go through each adjacency list in the adjacency list representation of G. For each `Adj[u]`, we visit every adjacent vertex v, and add the vertex u to the list `Adj[v]`, since the adjacency lists of *undirected* graphs include representations of both the edges (u, v) and (v, u). Then, we check for repetitions of edges by creating an array of size $|V|$. We set all of the elements of the array to 0, and then we go through the vertices, incrementing the value stored in the element that corresponds to the vertex by one, and when we see a value in the array that is greater than 1, we remove it from the adjacency list because that vertex is a repeated vertex. We do this for each list. When we have finished removing all the repeated edges, we are left with the adjacency representation of G', our undirected simple graph.

This is the pseudocode for our algorithm, which takes input the adjacency matrix data structure that represents the graph G:

```
def convert(Adj):
    for u in Adj:
        for v in Adj[u]:
            Adj[v].append(u)
    allocate Array A of size = (number of vertices)
    for i in A.length:
        A[i] = 0
    for u in Adj:
        for v in Adj[u]:
            A[v]++
            if A[v] > 1:
               Adj[u].delete
               A[v]--
    return (Adj)
```

Space complexity: The only auxiliary space needed to be allocated for our algorithm is an array of size $|V|$ used for counting repeated edges, which means that our space complexity is $\underline{\Theta(V)}$.

Time complexity: The first part of our algorithm, when we go through each adjacency list and append new edges to the lists, takes $V + cE$ (where c is some small constant) or $O(V+E)$ time because we visit each vertex u and every edge of the graph plus the added edges, which is some constant multiplied by number of original edges. Initializing the array takes $O(E)$ time, and deleting the repeated edges takes $O(V+E)$ time, which gives us $\underline{O(V+E)}$ time in total.

Proof of algorithm: Our algorithm works because we go through each adjacency list of the graph, and add the edge into the graph in the reverse direction, since undirected graphs contain edges that go in both directions. After we have added them, we remove all the repeated edges, and make sure that each particular edge in a list doesn't appear more than once. Once we have done these, our resulting graph meets all the requirements for an undirected simple graph.

5. (20 pts) *Crabbe and Goyle are at it again. Crabbe thinks min-heaps are for muggles and instead wants to use a max-heap to run the SSSP algorithm. He writes the following code on the whiteboard and bounces up and down in excitement, because he thinks he's going to get an algorithm named after himself.*

```
CrabbeSSSP(G, s) :
     # G is an adjacency list with n nodes and m directed edges
     H = empty max-heap
     for each node i {
        d[i] = INF
        insert i into H with key d[i]
     }
     d[s] = 0
     insert s into H with key d[s]
     while (H is empty) {
        v = extractMin(H)
        for each edge (v,u) {
           if (v,u) is tense {
              Relax(v,u)
              insert v into H with key d[v]
           }
     }}
```

Hermione snorts, saying this is hardly an efficient algorithm, and besides, it has one or more bugs in it (she won't say which!).

(a) *Identify the bug(s) in Crabbe's algorithm, explain why they would lead to an incorrect algorithm, and give the corrected pseudocode.*

The first bug is that it says `while (H is empty)`, however, to begin with we insert all of the nodes into H, so the `while` loop will never run. The second bug is that all of the nodes are inserted into the heap in the initialization - instead of the pred node being set to NULL. In fact, this algorithm doesn't keep track of pred nodes at all, so when it gets weights, it won't be able to extract the path. Finally, it inserts v, which is already in the heap, to be checked again, rather than inserting u, v's neighbor.

Below is the corrected code:

```
CrabbeSSSP(G, s) :
     # G is an adjacency list with n nodes and m directed edges
     H = empty max-heap
     for each node i {
        d[i] = INF
        pred(i) = NULL
     }
     d[s] = 0
     insert s into H with key d[s]

     while (H is not empty) {
        v = extractMin(H)
        for each edge (v,u) {
           if (v,u) is tense {
              Relax(v,u)
              insert u into H with key d[u]
           }
     }}
```

(b) *Analyze the corrected version of Crabbe's algorithm to determine is running time, and comment on why Crabbe will probably not get an algorithm named after him.*

Extracting the smallest element from a max heap requires scanning every element in the bottom of the array. There can be up to $n/2$ elements on the bottom, so `extractMin(H)` runs in $O(n/2) = O(n)$ time. Since the size of the heap is $\leq |V|$, then `extractMin(H)` runs in $O(V)$ time. Other than that it runs the same as Dijkstra's algorithm, which means that in the worst case, we will extract the min from the heap V times, and heap updates still take $O(\log V)$ time, which we will have to run $O(E)$ time. Thus the runtime is $O(V^2 + E \log V)$. This is significantly worse, or at best, asymptotically no better, than Dijkstra's Algorithm. Sorry, Crabbe, not today.

6. (15 pts extra credit) *Professor Snape has provided the young wizard Hermione with three magical batteries whose sizes are 12, 7, and 6 morts, respectively. (A mort is a unit of wizard energy.) The 7-mort and 6-mort batteries are fully charged (containing 7 and 6 morts of energy, respectively), while the 12-mort battery is empty, with 0 morts. Snape says that Hermione is only allowed to use, repeatedly, if necessary, the mort transfer spell when working with these batteries. This spell transfers all the morts in one battery to another battery, and it halts the transfer either when the source battery has no morts remaining or when the destination battery is fully charged.*

   Snape condescendingly challenges Hermione to determine whether there exists a sequence of mort-transfer spells that leaves exactly 2 morts either in the 7-mort or in the 6-mort battery.

   (a) *Hermione knows this is actually a graph problem. Give a precise definition of how to model this problem as a graph, and state the specific question about this graph that must be answered.*
   I don't know

   (b) *What algorithm should Hermione apply to solve the graph problem?*
   I don't know

   (c) *Apply that algorithm to Snape's question. Report and justify your answer.*
   I don't know