Problem Set Eight

Zoe Farmer
Jeremy Granger
Ryan Roden

February 26, 2024

```
## Error in library("qgraph"):  there is no package called 'qgraph'
```

1. First, 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

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

```
## Error in qgraph(edges, esize = 5, gray = TRUE): could not find function "qgraph"
```

   (a) Draw the depth-first spanning forest, including and identifying all tree, 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.)

      i. Please reference Table 1 for a list of edge types.

```
## Error in qgraph(edges, esize = 5, gray = TRUE): could not find function "qgraph"
```

1	2	Forward
1	4	Cross
1	8	Forward
2	3	Forward
3	1	Back
4	8	Back
5	2	Back
5	6	Forward
6	2	Back
6	3	Back
6	5	Back
7	4	Forward
8	7	Forward

Table 1: Edge Types

(b) List all the strong components in G.

i. A strong component is a strongly connected subgraph of G. There are three strong components in G:

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

2. On an overnight camping trip at the Weathertop National Park you and your hobbit friend Samwise are woken from a restless sleep by a scream. Crawling out of your tent to investigate, you see a terrified park ranger running out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. Reaching your tent, he gasps "Get out... while... you... ", thrusts the paper into your hands and falls to the ground, dead.
Looking at the crumpled paper, you recognize a map of the park, drawn as an undirected graph, where vertices represent landmarks in the park, and edges represent trails between those landmarks. (Trails start and end at landmarks and do not cross.) Coincidentally, you recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.
On closer examination, you notice that someone (perhaps the dead ranger) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on the back of the map indicates that a number next to a vertex or edge is the probability of encountering a deadly ringwraith along the corresponding trail or landmark. The note warns you that stepping off the marked trails will surely result in death. You glance down at the corpse at your feet. His death certainly looked painful. On closer examination, you realize that the ranger is not dead at all, or rather, is turning into a wraith who will surely devour you. After burning the undead ranger's body, you wisely decide to leave the park immediately.

(a) Give a (small) example G such that the path from your current location to the EXIT node that minimizes the *expected number* of encountered ringwraiths is different from the path that minimizes the *probability of encountering any* ringwraiths at all. Explain why, in general, these two criteria lead to difference answers.

i. If we let X be the random variable indicating how ringwraiths are encountered, we can defined its expected value to be

$$E(X) = \sum_{i=0}^{l} x_i \cdot p(x_i)$$

Where x_i is the edge's index in the given path, and $p(x_i)$ is the probability of the corresponding edge. This probability distribution function is defined as

$$f(X = k; l) = \begin{cases} p^k \cdot (1-p)^{l-k} & \rightarrow k \in \{0, 1, \ldots, l\} \\ 0 & \rightarrow Otherwise \end{cases}$$

Where l is the length of the chosen path. Also known is the formula to determine the probability of encountering zero ringwraiths on the journey, which is defined as

$$P(X = 0) = \prod_{i=0}^{l} (1 - p_i)$$

Where l is again the length of the path and p_i is the probability of encountering a ringwraith on that leg of the journey. We can now define a small network that demonstrates the requirement.

```
## Error in qgraph(edges, esize = 5, node.height = 2, node.width = 2, labels
= nodes, :  could not find function "qgraph"
```

We can now look at all paths and identify the minimums and maximums.

Path	$P(X = 0)$	$E(X)$
`START -> 0.48 -> EXIT`	0.0835364	1.5658373
`START -> 0.32 -> EXIT`	0.0789408	1.5079654
`START -> 0.48 -> 0.32 -> EXIT`	0.012702	2.5602729
`START -> 0.32 -> 0.48 -> EXIT`	0.1225848	1.6791986

Table 2: Path Probabilities

In Table 2 we see that the optimal path for encountering no ringwraiths is `START -> 0.32 -> 0.48 -> EXIT`, while the optimal path for encountering a minimal number of expected ringwraiths is `START -> 0.32 -> EXIT`.

(b) Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the *total expected number* of ringwraiths encountered along the path is as small as possible. Be sure to account for both the vertex probabilities and the edge probabilities.
Gandalf's Hint: This is clearly and SSSP problem, but you must identify how to reduce the input G to a form that can be solved by SSSP. Remember to include the cost of this transformation in your running-time analysis.

i. First, let us define our network as a series of nodes in an adjacency matrix, where row or column indicates source or terminal node, and the value represents edge weight. We can now use Dijkstra's algorithm to determine SSSP. This algorithm examines our graph, starting with the source node, and looks at edge weights of all its adjacent edges. Let V_i indicate our current vertex, with adjacent vertices A_i. For each adjacent vertex, examine the cost of traveling to that vertex through the current one (this cost is equal to the current node's cost, plus the cost of the edge between the current and adjacent vertices, plus the cost of the adjacent vertex) and if this new cost is less than the adjacent vertex's previous cost, update with the smaller. After all adjacent vertices have been examined, travel to the least costly adjacent vertex. Repeat until desired EXIT node is located.

Since this derivation of Dijkstra's original algorithm is almost identical, it also has $O(|V|)$ running time if we internally store vertices in a queue.

(c) Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the *probability of encountering any* ringwraiths at all is minimized.

i. This is also a derivation of Dijkstra's algorithm, however our method of calculating the cost of a path has changed. For this algorithm, the cost of a path is now defined as the current node's cost, times the edge cost, times the next node's cost. We still select the lowest cost path after each analysis.

Again, since this derivation of Dijkstra's original algorithm is almost identical, it also has $O(|V|)$ running time if we internally store vertices in a queue.

3. In a late night algorithms study session you and Golumn argue about the conditions under which a minimum spanning tree is unique. You agree that if all edges in G have unique weights the MST is also unique, but you disagree about how to relax this assumption. Let $w(e)$ be a function that returns the weight of some $e \in E$.

(a) Give an example of a (small) weighted graph that has both a unique MST and some e and e' such that $w(e) = w(e') = x$.

i. Demonstrated below.

```
## Error in qgraph(edges1, esize = 5, node.height = 2, node.width = 2, labels
= nodes, :  could not find function "qgraph"
## Error in qgraph(edges2, esize = 5, node.height = 2, node.width = 2, labels
= nodes, :  could not find function "qgraph"
```

If we use Kruskal's algorithm we can order all edges by their weights from smallest to largest. An edge is included as long as it does not make a cycle. If there are two minimum edge weights such that $w(e) = w(e')$, then we know that they will both be selected for our MST because the first two edges cannot make a cycle. Therefore since all other edges are distinct, you will end up with one minimum spanning tree. That is, whatever order of selecting the two minimum edges, the tree would be the same.

(b) Golumn claims that the following is true. Prove via (small) counter examples that it is false.
Golumn's Claim: G has a unique MST if and only if (i) for any partition of the vertices of G into two subsets the minimum-weight edge with one endpoint in each subset is unique and (ii) the maximum-weight edge in any cycle of G is unique.

i. If we partition our example graph (Figure 1) through edges (V_4, V_5) down to (V_1, V_3) we see that, although the edge (V_1, V_3) is the minimum weight edge with one endpoint in each subset, it is not unique ($(V_1, V_3) = (V_1, V_2)$), yet the Minumum Spanning Tree for our graph is still unique.

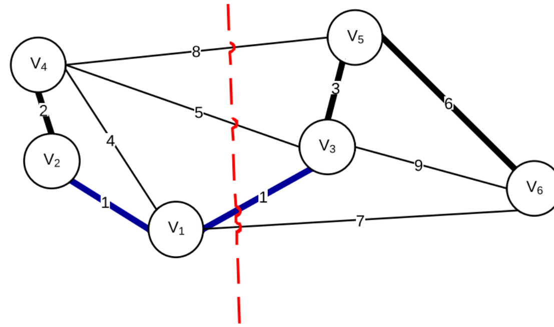

Figure 1: G Divided

We see that Figure 5 still only has one unqiue Minumum Spanning Tree, yet the maximum weight edges (V_4, V_5) and (V_3, V_4) are *not* unique. Thus, it is possible that a graph with non-unique minimum weight edges and/or non-unique maximum weight edges has a unique MST, and Gollumn is incorrect.

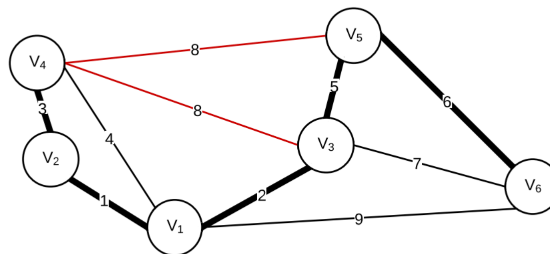

Figure 2: G's MST

(c) Golumn now demands that you produce the correct relaxed condition, which you claim is the following. Prove that you are correct.
Your Claim: An edge-weighted graph G has a unique MST T_{mst} if and only if the following conditions hold: (i) for any bipartation of the vertices induced by removing some edge $e \in T_{mst}$ the minimum-weight edge with one endpoint in each subset is unique, and (ii) the maximum-weight edge of any cycle constructed by adding on edge f to T_{mst}, where $f \notin T_{mst}$ is unique.
Gandalf's Hist: Note that for any spanning tree T on G, removing some edge $e \in T$ induces a bipartation of the vertices. Consider the edges that span this cut.

i. For this claim, we can prove using contradiction, making use of Professor Clauset's graph from his Lecture 23-25 notes.

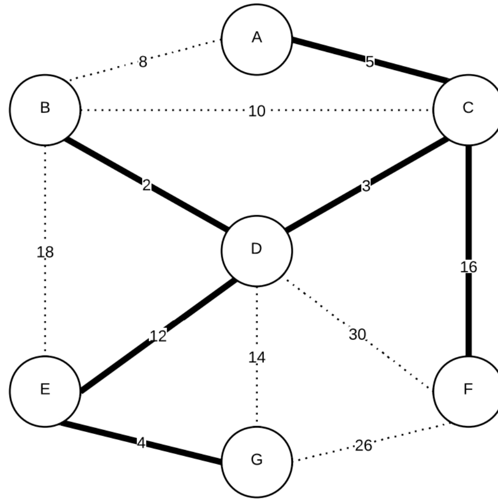

Figure 3: Clauset's Graph

Examining part i), if we suppose that we were to remove edge (C, D), and that edges (A, B) and (G, F) have the same weight, $w = 8$.

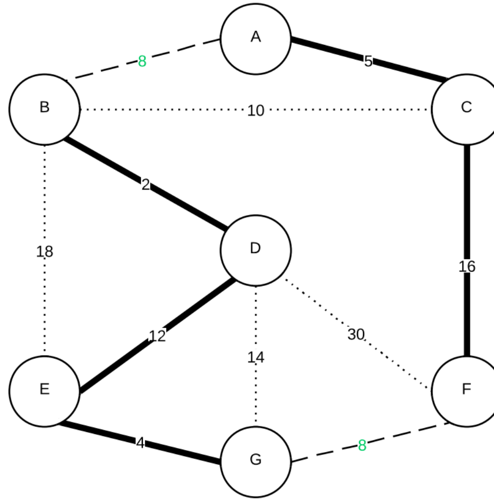

Figure 4: Adjusted Clauset's Graph

Since edges (A, B) and (F, G) are *both* minimum weight edges, either edge could be used as the connector with one endpoint in each of our subtrees. The simple fact that we have a choice and that, in choosing one edge over the other, we have the same total weighted tree $w(G)$ either way, means that graph G *does not* have a unique minimum spanning tree, and that in order to have one, graph G must have unique edges for any cut we decide to make.

For part ii) of our claim, suppose we have a graph with non-unique maximum edges with which to make a cycle.

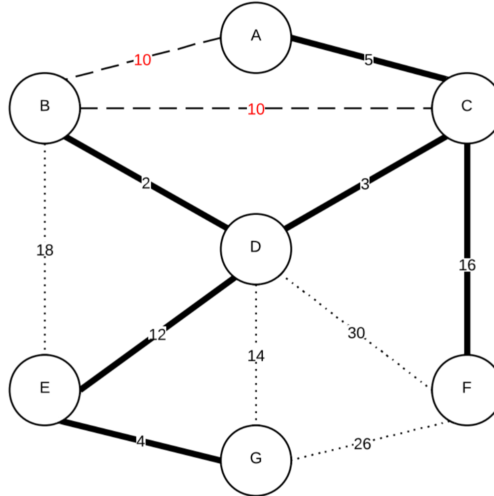

Figure 5: Mondified Clauset's Graph for Part ii

Since edges (A, B) and (B, C) are maximums, both with the same weight $w = 10$, the cycle induced by adding either of these edges to G's MST would equate to the same weighted value of the MST. Therefore the MST would *not* be unique. Any edges spanned by a cut in G must be unique for the MST to be unique.

(d) Describe and analyze an algorithm that will determine whether an input graph G has a unique MST in (effectively) $O(E \lg V)$ time.
Gandalf's Hint: Think about Kruskal's Algorithm.

i. This algorithm is quite simple since our goal is only to find out if a given graph has a unique MST. We must create vertex sets using two vertices at a time. Then we need to compare each edge to see if it is a maximum weight, and if so, it is unique. Also, if it is not a maximum edge, we need to check to see if it is a minimum weight edge. If so, we need to check if it is unique. If in either condition we find two minimum or maximum edges that have the same weight, we know that our graph cannot possible have a unique MST and we're done. Following is the pseudo-code.

```
1  def mst_unique(g):
2      unique = True
3      sort(Edge_Set_E).by_weight
4      for vertix v in Vertices V:
5          for each edge connected to vi:
6              compare safe_edges to current_edge:
7              if (v_is_min && w(v) &=& all w(vi) loop):
8                  unique = False
9                  break
10             elif (v_is_max && w(v) &=&  all w(vi) loop):
11                 unique = False
12                 break
13     return unique
```

Since this is a minor change from Kruskal's algorithm and navigates through all edges in the exact same way, the algorithm performs effectively in $O(E \log V)$ time. This is because we have only added a boolean value and comarison, which takes $O(1)$ time.

4. Implement Dijkstra's algorithm for the SSSP problem using a binary min-heap, and show (in a nicely structed figure) that its running time is $O(E \lg V)$ on the following type of graphs.
   Recall from problem set 7 that $G(n,p)$ denotes a simple (undirected, unweighted, no self-loops) random graph with n vertices in which each unordered pair (u,v), where $u \neq v$, is connected with probability $p = c/(n-1)$, for a constant expected degree c. Let $G_d(n,p)$ denote a directed generalization of this model in which we relax the undirected requirement by letting each ordered pair (u,v), where $u \neq v$, be connected with probability p. That is, instead of flipping $\binom{n}{2}$ coins for the upper triangle of the adjacency matrix, we flip $n^2 - n$ coins (because we still prohibit self loops) for the upper and lower triangles. Let $G_d(n,p)$ with $c = 5$ define the input graph for Dijkstra's algorithm.

   (a) The output graph looks as such:

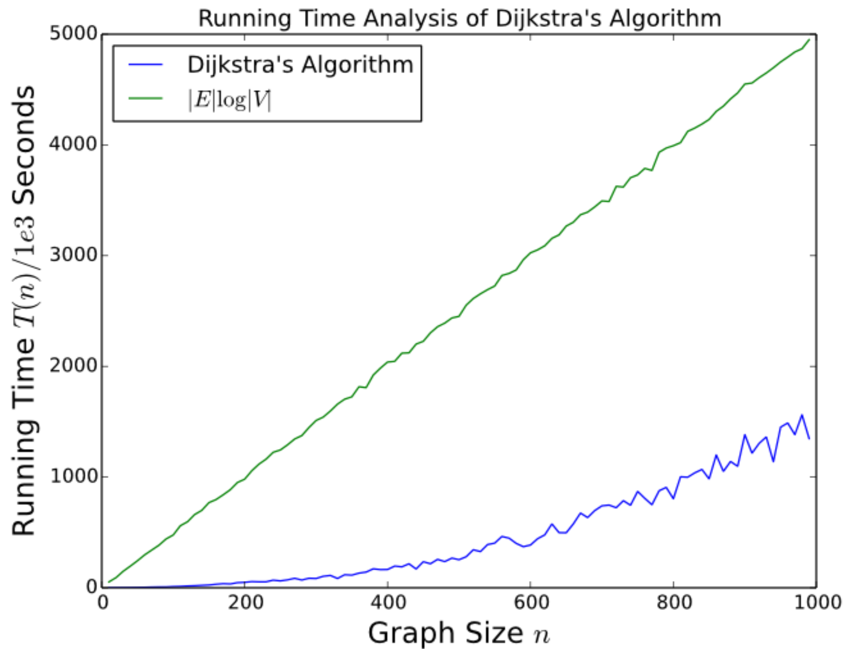

Figure 6: Runtime Analysis

5. Currency arbitrage is a form of financial trading that uses discrepancies in foreign currency exchange rates to transform one unit of some currency into more than one unit of the same currency. For instance, suppose 1 U.S. dollar bought 0.82 Euro, 1 Euro bought 129.7 Japanese Yen, 1 Japanese Yen bought 12 Turkish Lira and one Turkish Lira bought 0.0008 U.S. dollars. Then, by converting currencies, a trader could start with 1 U.S. dollar and buy $0.82 \times 129.7 \times 12 \times 0.0008 \approx 1.02 U.S.$ dollars, thus turning a 2% profit. Of course, this is not how real currency markets work because each transaction must pay a commission to a middle-man for making the deal.
   Suppose that we are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j . A traditional arbitrage opportunity is thus a cycle in the induced graph such that the product of the edge weights is greater than unity. That is, a sequence of currencies $\langle c_{i1}, c_{i2}, \ldots, c_{ik} \rangle$ such that $R[i_1, i_2] \times R[i_2, i_3] \times \cdots \times R[i_{k-1}, i_k] \times R[i_k, i_1] > 1$. Each transaction, however, must pay a commission, which is typically some α fraction of the transaction value, e.g., $\alpha = 0.01$ for a 1% rate.
   (a) Give an efficient algorithm to determine whether or not there exists such an arbitrage opportunity, given a commission rate α. Analyze the running time of your algorithm.
   Gandalf's hint: It is possible to solve this problem in $O(n^3)$. Recall that Bellman-Ford can be used to detect negative-weight cycles in a graph.
      i. The question asks to find a cycle such that

$$R[1,2] \cdot R[2,3] \cdot \cdots \cdot R[k-1,k] \cdot R[k,1] > 1$$

Inverting and taking the log of both sides gives us

$$\log\left(\frac{1}{R[1,2]} \cdot \frac{1}{R[2,3]} \cdot \cdots \cdot \frac{1}{R[k-1,k]} \cdot \frac{1}{R[k,1]}\right) < \log(1) or 0$$

Through logarithmic properties this is rewritten as

$$\log\left(\frac{1}{R[1,2]}\right) + \log\left(\frac{1}{R[2,3]}\right) + \cdots + \log\left(\frac{1}{R[k-1,k]}\right) + \log\left(\frac{1}{R[k,1]}\right) < 0$$

Changing the weights of each edge to the corresponding values above allows us to search for a negative cycle.
Since we are looking for a negative cycle, we use the Bellman-Ford algorithm to accomplish this. The problem asks for an efficient algorithm, and hints at Bellman-Ford which is $O(|V||E|) = O(|V|^3)$. The nested for loops that change the currency conversions to the appropriate values for B-F runs in $O(|E|) = O(|V|^2)$, so this does not change the order of the runtime

```
# Step 1: setup matrix to have proper values
#          for Bellman-Ford and negative cycles
matrix R = alpha*R

for i from 1 to n {
  for j from 1 to n {
    R[i][j] = log(1/R[i][j]);
  }
}

BellmanFord( matrix R, vertex source, currencies n ){
  for v from 1 to n{
    if v is source, then distance[v] = 0;
    else distance[v] = infinity;
    p[v] = null;
  }

  //relax edges
  for v from 1 to n - 1{
    for i from 1 to n{
      for j from 1 to n{
        if distance[i] + R[i][j] < distance[j]
          distance[j] = distance[i]+R[i][j]
          p[j] = i;
      }
    }
  }

  //check for neg-weight cycles
  for i from 1 to n {
    for j from 1 to n {
      if distance[i] + R[i][j] < distance[j]
        FOUND NEGATIVE CYCLE
    }
  }
}
```

(b) Explain what effect varying α has on the structure of the set of possible arbitrage opportunities your algorithm might identify.

i. Varying alpha will only change the amount of profit yielded from the arbitrage opportunity (as long as alpha is less than 100%, in which case there would be no negative cycle ever), so it does not change the size of the set of arbitrage opportunities.