CS 161: Homework 6

Kenny Smith

May 29, 2015

1.
```
def minGraphTree(A):
  G <- Graph with size[A] vertices
  for i in (1 to length(A)):
    G[i].x <- A[i][1]
    G[i].y <- A[i][2]
    G[i].z <- A[i][3]
  X <- MERGESORT(G, by=x)
  Y <- MERGESORT(G, by=y)
  Z <- MERGESORT(G, by=z)
  for i in (1 to (length(A) - 1)):
    G.ADD_EDGE(X[i], X[i+1])
    G.ADD_EDGE(Y[i], Y[i+1])
    G.ADD_EDGE(Z[i], Z[i+1])
  return KRUSKAL(G)
```

Correctness: Although an edge can theoretically exist between any two nodes, the algorithm considers putting an edge between two nodes i and j if and only if there it is not true that there exists nodes a, b, and c such that:

- $x_i < x_a < x_j$ or $x_j < x_a < x_i$
- $y_i < y_b < y_j$ or $y_j < y_b < y_i$
- $z_i < z_c < z_j$ or $z_j < y_c < z_i$

Say that the minimum spanning tree has an edge between i and j. We can split the MST into two supernodes, that which contains nodes connected to i, which we will call t_i and that which contains j, which we will call t_j. Since these nodes contain the entire MST (otherwise the graph would be unconnected), it follows that given a node k, k must be in one of these two supernodes. Assuming that k is in t_i, we can assume there is a unique edge which connects k to r, part of a supernode t_r containing i but not k and its associated supernode t_k, such that $t_r \cup t_k = t_i$ because if there was not, the graph would be unconnected, and if there were multiple, a cycle would be created, contradiction the no-cycles property of MSTs. Thus, by applying the following transformation, we will get a spanning tree whose edges have a smaller sum: (1) Create an edge between i and k, and one between j and k (2) delete the edge between k and r as well as the edge between i and j. i and j maintain their connectivity through the edge between i and k and that between j and k. k and i maintain their connectivity directly through an edge. There are no cycles created, because there remains exactly one connection between t_j and t_k and one between t_k and t_r, and we have not manipulated any of the internal workings of the supernodes.

Let us assume without loss of generality that $|x_j - x_i| < |y_j - y_i| < |z_j - z_i|$. Thus, $\ell(i,j) = |x_j - x_i|$. The statement $x_i < x_a < x_j$ OR $x_j < x_a < x_i$ is equivalent to the statement $|x_j - x_i| = |x_j - x_a| + |x_a - x_i|$.

Say that the weight of our original MST was w. The weight of our new tree is $w-\ell(i,j)-\ell(r,a)+\ell(i,a)+\ell(j,a) = w - |x_j - x_i| + |x_a - x_i| + |x_j - x_a| - \ell(r,a) = w - \ell(r,a)$. Thus, since $\ell(r,a)$ must be positive nonzero, we conclude that our original tree is not the minimum spanning tree, and therefore we should only consider edges that satisfy the above conditions. From the correctness of Kruskal's algorithm, we can conclude that we have the correct minimum spanning tree after having narrowed down to the correct set of edges.

Runtime: We have n vertices and roughly $m = 3n$ edges in our graph. `MERGESORT` is $O(n \log n)$, with n representing the number of vertices in the graph. `KRUSKAL` runs in $O(m \log n) = O(3n \log n) = O(n \log n)$. The `for` loop has n iterations which contain $O(1)$ operations, assuming that the `Graph` structure is optimized for up to 6 edges on a vertex, so the loop is $O(n)$. Thus, we have no operation that costs more than $O(n \log n)$ time.

2. (a) This algorithm is more or less the same as Dijkstra's algorithm. At the end of the algorithm, all vertices will have an attribute `capacity` which will say the maximum amount of capacity to that vertex.

```
def getCapacities(G, root):
  pq <- PriorityQueue()
  for vertex in G:
    vertex.capacity <- 0
  c <- root
  c.capacity <- infinity
  pq.enqueue(c, -c.capacity) // highest capacity goes first
  while true:
    c <- pq.dequeue()
    for e in c.edges:
      v <- e.vertexthatisnt(c)
      if v.status == FINISHED:
        continue
      if min(e.capacity, c.capacity) > v.capacity
        v.capacity <- min(e.capacity, c.capacity)
        pq.enqueue(v, -v.capacity)
    c.status <- FINISHED
    if pq.size == 0:
      break
```

Correctness: Our base case is that at the depot, there is no capacity limit. Say we have run n iterations of the algorithm and we are considering a vertex A with calculated maximum capacity s and correct capacity c. Say that $c < s$. This is easily contradicted since we can trace back the algorithm to determine the best path, so no incorrect paths turn up. Now say $c > s$. This means that A has at least one neighbor B_1 that has a capacity equal to c or greater. This B_1 must be connected to a set of nodes $B_2, B_3, \ldots$ that ultimately leads back to the root. Say that the last `FINISHED` node of these is B_n. We know that B_{n-1} has a capacity greater than or equal to s, and since all the attached nodes are visited and enqueued during the "finishing" process, this means we would have had to have dequeued and "finished" the node B_{n-1} before A, contradicting that B_n is the last node that was visited. Thus, we conclude that $s = c$.

Runtime: Each vertex is considered at most once because it is not enqueued if its status is `FINISHED`, and this is not reversible. Each edge is considered at most twice in the context of each unique vertex. Thus, the runtime is $O(n + m)$.

(b) The algorithm returns a table with the max capacity between to vertices searchable based on the

vertices' respective numbers.

```
def getAllCapacities(G):
  getv <- Array(n) of vertex pointers
  capacities <- Matrix(n,n) of NaNs
  for i in (1 -> n):
    capacities[i][i] <- infinity
  groups <- Matrix(n,n) of NaNs
  for i in (1 -> n):
    groups[i][1] <- i
  edges <- MERGESORT(G.edges) \\ largest capacity to smallest
  i <- 0
  for v in G.vertices:
    i++
    v.number <- i
    v.group <- i
    getv[i] <- &v
  for e in G.edges:
    g1 <- groups[e.v1.group]
    g2 <- groups[e.v2.group]
    i <- 0
    while i < n and g1[++i] != NaN:
      j <- 0
      while j < n and g2[++j] != NaN:
        capacities[g1[i]][g2[j]] <- e.capacity
        capacities[g2[j]][g1[i]] <- e.capacity
    j <- 0
    i--
    while j < n and g2[++j] != NaN:
      g1[i+j] <- g2[j]
      getv[g2[j]].group <- e.v1.group
  return capacities
```

Correctness: This algorithm is correct because say we are combining two supernodes ("groups") and saying that the maximum capacity c between vertices of the first node and the second node is the same as that of the newly connecting edge s. Say that in reality $c > s$. That means we should have already considered all edges of size c, so that means that the supernodes should have already been connected into one supernode. Say that $c < s$. That contradicts that c is the weight of the maximum-length path, since we know that s's status as a path can be verified by a traceback. The only remaining option is that $c = s$.

Runtime: We have $O(n)$ operations in `for` loops of size n and m (we know $m < n^2$ since there can only be one edge between any two vertices). As for the remaining operations in while loops, we know they constitute no more than $O(n^2)$ time because there is a bijection between $O(1)$ operations and turning `NaN` matrix cells into non-`NaN` matrix cells in `groups` and `capacities`. Both these matrices have an $O(n^2)$ size, so that means that these constitute $O(n^2)$ time only. Thus, the algorithm is $O(n^2)$.

3. Prove. Let S be the set of all possible edge weights for an MST of G. Let s be the least-valued item in S such that the number of edges in any MST that have weight s can vary if one draws different MSTs. Now let k equal the maximum possible number of edges in an MST that have weight s. Now let us construct an MST $M \subset G$ such that there are k edges of weight s. Let us pick any edge $e \in M$ with weight s,

connecting two supernodes t_1 and t_2. Say that there is an edge connecting these two supernodes with less weight than e. That contradicts the cut property which would say that that edge has to be in the MST. Thus, by cut property, e or an edge of the same size has to be in the MST. Thus, it is necessary to have at least one edge in the MST for each of the k edges we delineated earlier, so therefore we can conclude that there must be exactly k edges of weight s in any MST. This contradicts that the number of edges with size s is not constant. It follows that $\forall s \in S, \exists k$ such that every MST of G contains exactly k edges of size s.

Having established this fact, say we are given an MST of G as in the problem. We are guaranteed that given G and a size s, the MST contains exactly k edges of size s. Say there are p edges of size s in the entire graph. All we have to do is to treat the k edges that we want as having smaller values than the remaining $p-k$. As the algorithm considers each edge, it will only reject an edge if accepting that edge would create a cycle. Since we know our MST does not contain cycles, the algorithm will not reject any of these edges. Thus, we will get the correct MST if we treat the k edges we want as having less weights for all s.

4. (a) The two algorithms do the same thing. They pick an edge uniformly at random. They join the two supernodes, and then pick another random edge provided it is not inside a supernode. The randomness is supplied by the random weighting in MST-Karger. This process is repeated until there are two supernodes. In MST-Karger, the edge that connects the last two supernodes is the same as the max-weight edge of the MST. We know that MST-Karger has a uniform random shuffling algorithm, but in order to show it for the Karger algorithm, we must inductively show that there is an equal likelihood of all permutations in selecting first an edge, removing it from the pool, and selecting another random edge etc. The base case is that if we have 1 edge possible, all 1! permutations are trivially equally likely. Say we have $n+1$ edges, we have an equal probability $\frac{1}{n+1}$ of picking any edge as the first edge. From there, we have an equal probability of picking (inductive assumption) any of the $n!$ possible combinations of the remaining edges. Thus, the probability is $\frac{1}{(n+1)!}$ for any of the remaining permutations of edges. Thus, the outputted probability will be the same.

   (b) From lecture, we know that Karger and hence MST-Karger require $n^2 \log n$ runs to ensure that the minimum is encountered at least once with probability $1 - 1/n$, and $cn^2 \log n$ runs to get a success probability of $1 - 1/n^c$. The `foreach` loop requires m $O(1)$ operations to be done. Kruskal's algorithm requires $O(m \log n)$ of time. Finding the maximum from the tree requires $O(n)$ amount of time because there are $n-1$ edges whose weights need to be checked while keeping a running maximum. Finally, selecting each node in the MST and classifying it as being in either S or T requires $O(n)$ operations. Thus, the algorithm is $O(m + m \log n + n)$. Since the graph is connected, we know that $m >= n - 1$ and therefore we can reduce the runtime equation to $O(m \log n)$. Multiplying this by the required $n^2 \log n$ iterations, we see that our algorithm is $O(mn^2 \log^2 n)$.