Homework 6 for EECS 340

Yu Mi,yxm319

April 24, 2018

1 Warm-Up

a R-15.1

Draw a simple, connected, undirected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Illustrate the execution of Kruskal's algorithm on this graph. (Note that there is only one minimum spanning tree for this graph.)

Answer: The sequence of Kruskal algorithm is shown as Fig.1 through Fig.8.

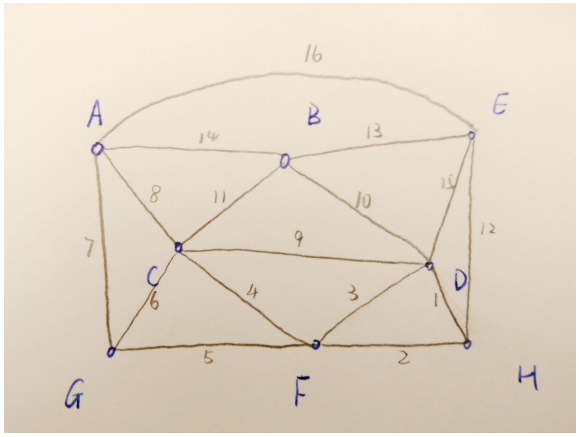

Figure 1: Initial Figure

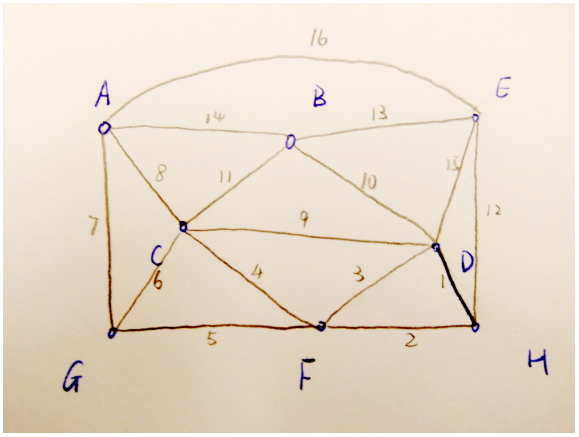

Figure 2: Add (D, H) to the tree

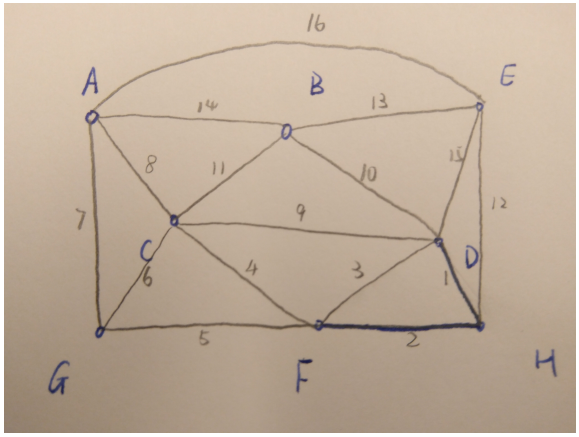

Figure 3: Add (F, H) to the tree

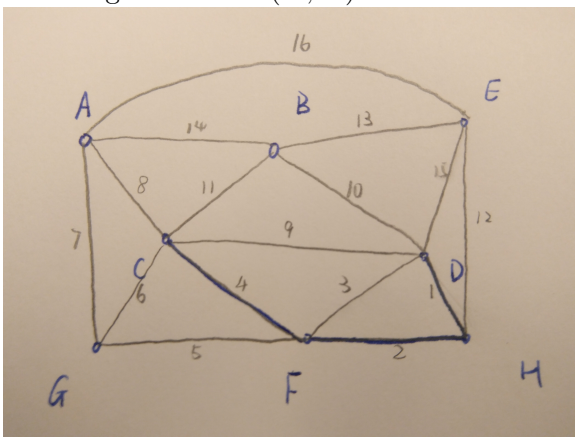

Figure 4: Add (C, F) to the tree

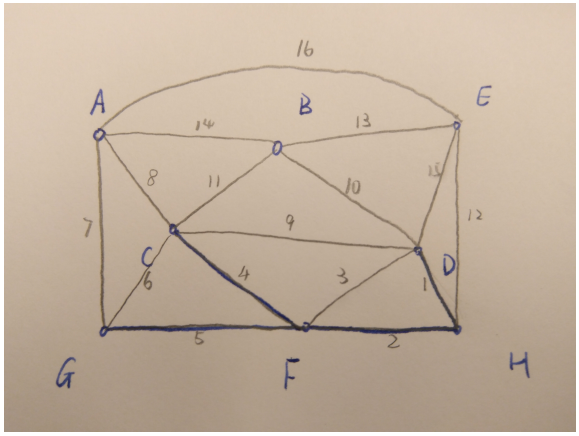

Figure 5: Add (G, H) to the tree

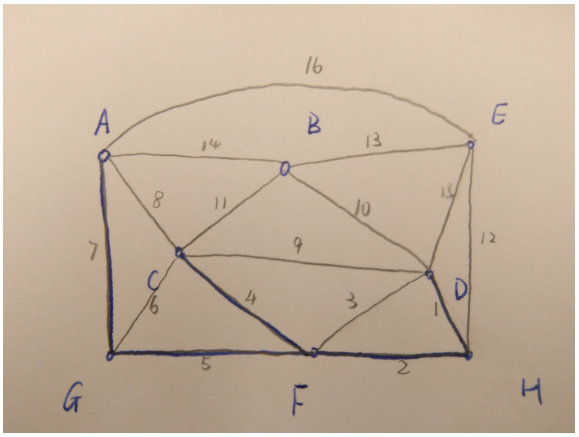

Figure 6: Add (A, G) to the tree

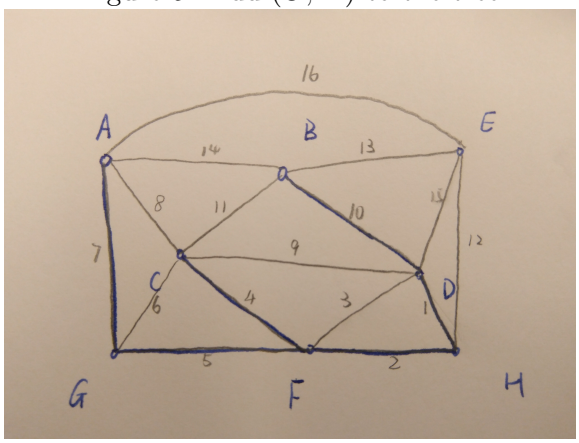

Figure 7: Add (B, D) to the tree

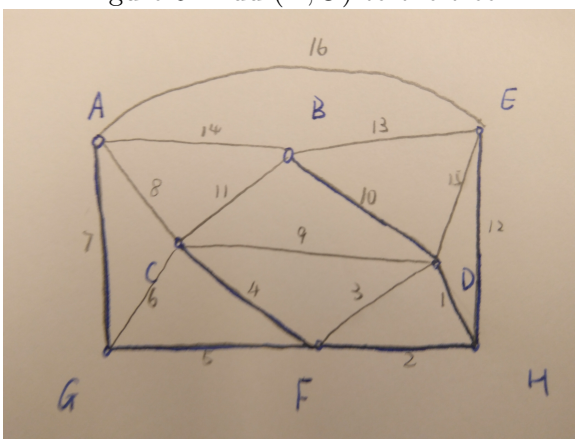

Figure 8: Add (E, H) to the tree

b R-18.3

Give an example of a graph G with at least 10 vertices such that the greedy 2-approximation algorithm for VERTEX-COVER given above is guaranteed to produce a suboptimal vertex cover.

Answer: The initial graph is shown in Fig.9, which contains 10 vertice, and the process of 2-approximation algorithm is shown in Fig9 through 12. As we can see in Fig.12, to cover edge(v_3, v_4), we must either add v_3 or v_4 to the set C, which must at least contain 7 elements. However, we can simply conclude from the initial graph that set $C = \{v_2, v_3, v_5, v_6, v_7, v_9\}$ can be a better VERTEX COVER for the initial graph which have 6 vertices. Such example proved that this greedy approximation algorithm will produce a suboptimal result.

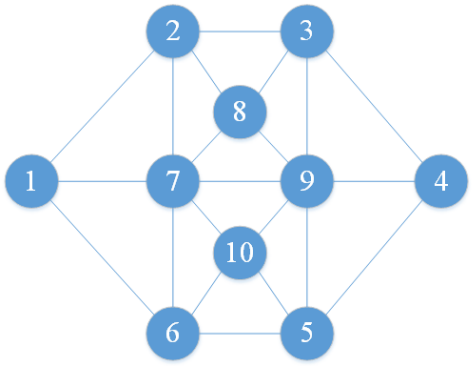

Figure 9: Initial Graph

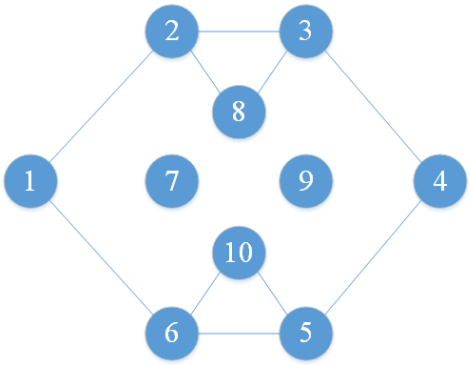

Figure 10: Add v_7 and v_9 to the C

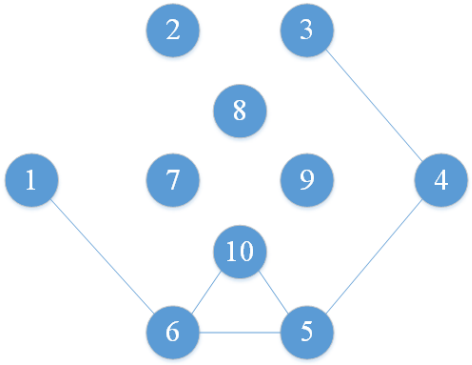

Figure 11: Add v_2 and v_8 to the C

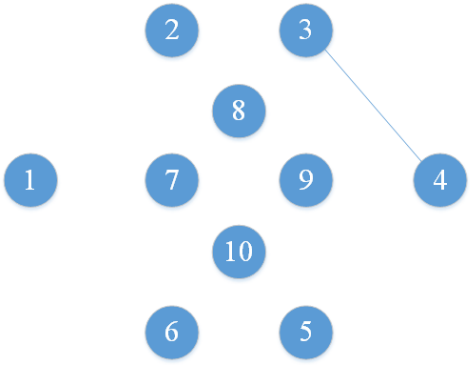

Figure 12: Add v_6 and v_5 to the C

2 Dijkstra Revisited

There is an alternative way of implementing Dijkstra's algorithm that avoids use of the locator pattern but increases the space used for the priority queue, Q, from $O(n)$ to $O(m)$ for a weighted graph, G, with n vertices and m edges. The main idea of this approach is simply to insert a new key-value pair, $(D[v], v)$, each time the $D[v]$ value for a vertex, v, changes, without ever removing the old key-value pair for v. This approach still works, even with multiple copies of each vertex being stored in Q, since the first copy of a vertex that is removed from Q is the copy with the smallest key. Describe the other changes that would be needed to the description of Dijsktra's algorithm for this approach to work. Also, what is the running time of Dijkstra's algorithm in this approach if we implement the priority queue, Q, with a heap?

Answer: The algorithm is shown as follows:

Algorithm Dijkstra(G, v)
Input: A connected graph G, a root vertex v
Output: the list of shortest distance from the root vertex to all the other vertices.
$Q \leftarrow$ an empty priority queue that stores (w, v) pair
$P \leftarrow$ an empty list
$visited \leftarrow v$ ▷ a list that records visited vertices.
$P.push(v)$
while P is not $NULL$ **do**
  $c \leftarrow P.popMinimum(Q, visited)$ ▷ Like a BFS
  $visited.push(c)$
  for n in $neighbor(c)$ **do**
    $P.push(n)$
    if not $Q.contain(n)$ or $weight(c, n) + Q.getWeight(c) < Q.getWeight(n)$ **then**

$Q.push((weight(c,n) + Q.getWeight(c)), n)$
end if
end for
end while
$D \leftarrow$an empty list
for i in V **do** ▷ V is the set of all the vertices in G
$D[i] \leftarrow Q.getWeight(v)$
end for

Note in the algorithm, $Q.getWeight(n)$ returns the weight of vertice n in queue Q. $weight(n, c)$ returns the weight of edge between n and c, $Q.contain(n)$ returns whether there is n in the queue Q. The $P.popMinimum()$ returns the element in P which has the smallest distance from the starting point v as is recorded in Q, and the returning element is not visited as is recorded in *visited*.

In this algorithm, it will go over a process like BFS, which costs $O(|E| + |V|)$ time, and the popping method of Q also costs $O(\log n)$ time when Q has a size of n. Thus the total runtime would be $O(|E| + |V| \log |V|)$.

3 The Struggle of Commuter Students

Given an undirected weighted graph $G = (V, E)$ with non-negative edge weights, and two vertices $s, t \in V$, let G' be G, but where every edge has its weight increased by 1. Prove that G and G' have the same set of shortest paths from s to t, or provide a counterexample.

Answer: G and G' may not have the same set of shortest paths from s to t. As is shown in Fig.13, the path $s \to a \to c \to d \to t$ has the least cost which is 11. However, when all the edges have their weight increased by 1, the former path would cost 15 while another path $s \to b \to t$ would cost 14. Thus this case becomes a counterexample.

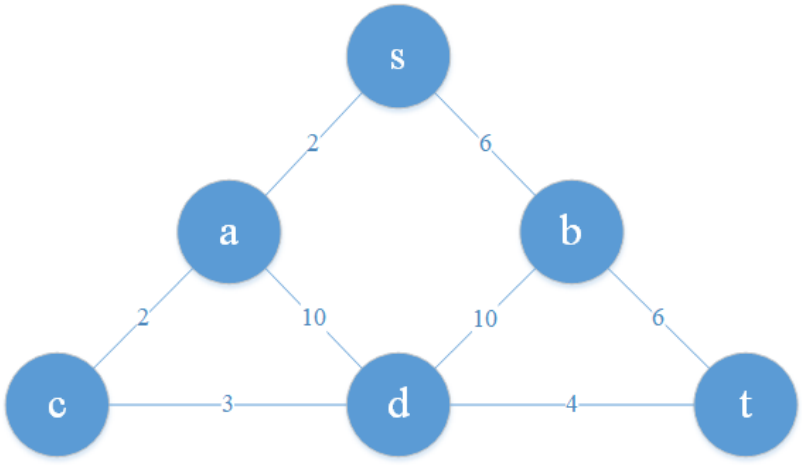

Figure 13: A counterexample, number on an edge is the edge's cost

4 Unique Bonsai

Suppose G is an undirected, connected, weighted graph such that the edges in G have distinct edge weights. Show that the minimum spanning tree for G is unique.

Answer: Let's assume there **are** two *distinct* minimum spanning trees, T_1 and T_2. Then the edge of minimum weight among all the edges that are contained in one of T_1 and T_2. Let's say this edge appears only in T_1 and we call it e_1. Then $T_2 \cup \{e_1\}$ must contain a cycle, and one of the edges of this cycle e_2 is not in T_1, otherwise T_2 would have a cycle. Since e_2 is an edge different from e_1 and is contained in exactly

one of T_1 or T_2, it must be $w(e_1) < w(e_2)$. Since $T = T_2 \cup \{e_1\} - \{e_2\}$ is a spanning tree. The total weight of T is smaller than the total weight of T_2, but this is a contradiction with T_2 is a minimum spanning tree. The proof above denied the possibility of having two *distinct* minimum spanning trees, which makes the minimum spanning tree for G unique.

5 Bonsai No More

a C-15.9

Suppose G is a weighted, connected, undirected graph with each edge having a unique integer weight, which may be either positive or negative. Let G' be the same graph as G, but with each edge, e, in G' having weight that is 1 greater than e's weight in G. Show that G and G' have the same minimum spanning tree.
Answer: Suppose we have a minimum spanning tree of the graph G, if every edge weight of G is increased by 1, the new graph G' will still have the same minimum spanning tree as G. If G has n vertices, then any spanning tree of G has $n-1$ edges. Therefore incrementing each edge weight by 1 increase the cost of every spanning tree by a constant. Thus in every step of selecting the edge to add them into the spanning tree, the edge to be selected does not change since the order of weight of edge does not change. So any spanning tree with minimal cost in the original graph also has minimal cost in the new graph. Another thing to note is as proved in Question 4, since we have each edge having unique integer weight, the minimum spanning tree is unique. So that the minimum spanning tree in G and G' must be the same.

b C-15.9 Addition

In the setting of C-15.9, suppose that instead of adding 1 to the weight of every edge, we multiply the weight of every edge by 2. Are the minimum spanning trees of G and G' still the same? Prove, or provide a counterexample.
Answer: As is described in the former question, the order of the edges does not change when the edges' weight got multiplied by 2. Thus the minimum spanning trees of G' is also the same as G.

6 Burma Shave

Answer: This problem is the same as the salesperson problem described in the textbook. To solve this problem, we can simply do the following:

1. Calculate each pair of strings' S_i and S_j distance , with method *TimeBetween(S_i,S_j)*;
2. Let string S_1 be the starting and ending point of the transition;
3. Construct minimum spanning tree from S_1 as root using Kruskal's algorithm;
4. Do a preorder transversal of the constructed MST and add S_1 at the end.

The method listed above has its' runtime mainly determined by the Kruskal's algorithm, which has the time complexity $O(|E| \log |V|)$, where $|E|$ denotes the number of edges and $|V|$ denotes the number of edges. Since $|E| \leq |V|^2$, thus we have time complexity $O(|E| \log |V|) \leq O(|V|^2 \log |V|) \leq O(|V|^3)$, which works in polynomial time. Since the values of $TimeBetween$ satisfy the triangle inequality, the approxmiation algorithm listed above also have the solution's weight not higher than 2 times more heavier than the optimal solution as is proved in the textbook in section 18.1.

7 FedUP, Regulated Federally

Suppose you work for a major package shipping company, FedUP, as in the previous exercise, but suppose there is a new law that requires every truck to carry no more than M pounds, even if it has room for more boxes. Now the optimization problem is to use the fewest number of trucks possible to carry the n boxes across the country such that each truck is carrying at most M pounds. Describe a simple greedy algorithm for assigning boxes to trucks and show that your algorithm uses a number of trucks that is within a factor of 2 of the optimal number of trucks. You may assume that no box weighs more than M pounds.

Answer: In our simple greedy algorithm, we can grab the first j boxes which their total weight not exceeding M, but another additional box will exceed the M, which is $b_i + b_{i+1} + ... + b_{i+j-1} \leq M \leq b_i + b_{i+1} + ... + b_{i+j}$, and put them into a truck. We can do this operation till all the boxes are on board.

Then we prove this algorithm is within a factor of 2 of the optimal number of trucks. First, we can consider the worst situation I, where all the boxes appear in a repeated pattern $\{b_i, b_{i+1}\}$ where $b_i + b_{i+1} > M$. Thus we have to put all the boxes separately into the trucks. In this case, the average free space in all the trucks are $space = \frac{M-b_i+M-b_{i+1}}{2} = \frac{2M-(b_i+b_{i+1})}{2}$. Since we have $b_i + b_{i+1} > M$, we can conclude that $space < \frac{M}{2}$, which means the average free space in all the trucks does not exceed half of the capacity of each truck.

In the most optimal algorithm (if there is), the trucks are all filled up with no free space. Which means the truck needed is only $OPT(I) = \frac{\sum_{i=1}^{n} b_i}{M}$, but in the worst situation, we need also to 'carry' spaces which is $space < \frac{M}{2}$ for each truck. This means the available capacity of a truck is reduced to a little bit more than half of its total capacity. $A(I) = \frac{\sum_{i=1}^{n} b_i + A(I) \cdot space}{M} < \frac{\sum_{i=1}^{n} b_i + A(I) \cdot \frac{M}{2}}{M} = \frac{\sum_{i=1}^{n} b_i}{M} + \frac{A(I)}{2}$, thus we have $A(I) < 2OPT(I)$. ■

8 EECS454 only

a R-17.3

Show that the problem SAT, which takes an arbitrary Boolean formula S as input and asks whether S is satisfiable, is ***NP***-complete.

Answer: A language L is said to be ***NP***-complete if $L \in \mathbf{NP}$, and every language in ***NP*** can be polynomial time reducible to L.

First, we examine the problem is ***NP***: let's say the S have n boolean variables as input, thus the SAT problem can be examined in 2^n time since there is only 2^n possible assignment, and they can be checked in polynomial time. Such that this problem is in ***NP***.

Second, let's consider the CIRCUIT-SAT problem, which is a ***NP***-complete problem. We can assert that for every boolean variables as input for S, there can be a wire input corresponding to it in an instance I for the CIRCUIT-SAT problem, and for each boolean operator in S, there can be a corresponding logical gate in CIRCUIT-SAT problem. Thus, an arbitrary instance of CIRCUIT-SAT problem can be reduced to an instance of SAT in polynomial time.

As is discussed above, the SAT problem is ***NP***-complete.

b R-17.14

Show that the SET-COVER problem is in ***NP***.

Answer: To prove the SET-COVER problem is in ***NP***, we only need to prove that for any instance of the problem, we can verify the solution in polynomial time whether the solution is acceptable.

To prove this, we can simply develop an algorithm solving it:

Algorithm Examine(L, U)
Input: A list of sets L of the provided solution, the 'universe' set U.
Output: Whether the L is acceptable in the SET-COVER problem.

$S \leftarrow$ an empty set
for s in L **do**
    $S.union(s)$ ▷ Add s to the S
end for
if $|S| = |U|$ **then**
    return $True$ ▷ L can cover all the elements in U.
else
    return $False$ ▷ L cannot cover all the elements in U.
end if

As is shown above, the time complexity of the algorithm is $O(n^2)$ when there are totally n elements. This shows the algorithm can verify a solution in polynomial time.