1 Homework 4 Due May 13, 11:59 in canvas Disclose Internet Sources

Classical Greedy Graph Algorithms for MST and SSSP

The Minimum-Weight Spanning Tree problem is abbreviated as MST. Prim's algorithm builds an undirected tree, also called a free tree, starting at the designated source vertex.

The Single-Source Shortest Path problem is abbreviated as SSSP. The Dijkstra (pronounced "dike-stra") algorithm builds a directed tree, rooted at the source vertex, whether the underlying graph is directed or undirected.

These two algorithms are perhaps the best known for solving a graph problem **optimally** with a greedy algorithm.

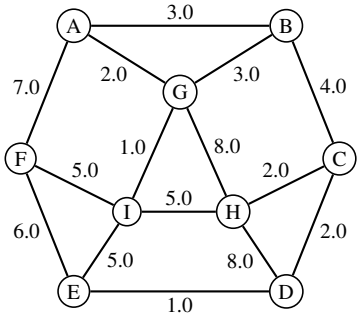
Kruskal's MST algorithm can also be cast as a greedy algorithm. It requires considerably more algorithmic sophistication to achieve its best efficiency.

Text-book references on reserve in the Science Library include Baase and Van Gelder, ch. 8; Cormen *et al.* 2nd ed, sects. 23.2 and 24.3; Kleinberg and Tardos sects. 4.4 and 4.5. Students are encouraged to look at these sources, and citing them for this homework is not required.

Homework 4.01 (20 points)

Carry out the Prim MST algorithm on the weighted undirected graph below. Use A as the source vertex.

The algorithm was described in lecture and is shown in scanned handouts. Also see the last two pages of this homework.



1. **(10 points)** Show the priority queue table just before each edge is added, then indicate which edge is chosen from that table to be a tree edge, and continue with the updated table in a new diagram, and the next chosen edge, and so on, until the tree is complete.

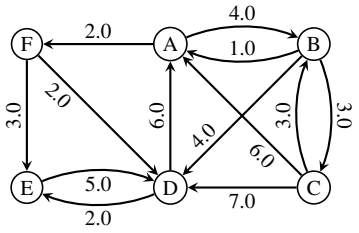
   Your new diagram for each step should show all the data for fringe vertices and for vertices whose status changed. To save time and clutter you do not need to repeat data for vertices in intermediate diagrams that did not change since the prior diagram. But make the final diagram complete.

2. **(5 points)** How does one use the final table to find the total weight of the minimum spanning tree that was found?

3. **(5 points)** Illustrate (partly) how the MST that was found obeys the **Minimum Spanning Tree Property** using the heaviest edge not in the MST and the lightest edge not in the MST.

Homework 4.02 (20 points)

Carry out the Dijkstra SSSP algorithm described in lecture and shown in scanned handouts on the weighted directed graph below. Use A as the source vertex and assume adjacency lists are in alphabetical order (A, B, C, D, E, F).



1. **(10 points)** Show the priority queue table just before each edge is added, then indicate which edge is chosen from that table to be a tree edge, and continue with the updated table in a new diagram, and the next chosen edge, and so on, until the tree is complete. Be careful to update the intermediate tables correctly.
2. **(4 points)** How does one use the **final** table to find the total distance of an arbitrary vertex (B–F) from the source vertex? Illustrate your answer with E.
3. **(6 points)** How does one use the final table to find the actual shortest path from the source vertex to an arbitrary vertex? Illustrate your answer with D.

As a "sanity check" it is recommended that you inspect the graph to see whether your answers look correct.

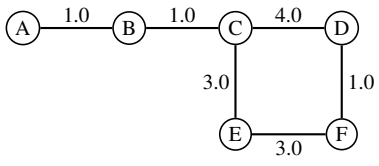
Homework 4.03 (10 points)

Give a weighted directed graph and a source vertex such that neither the depth-first search tree nor the breadth-first search tree is a shortest-path tree, regardless of how the adjacency lists are ordered.

To justify your example show all possible depth-first search trees and all possible breadth-first search trees, including the total distance to the tree leaves from the designated source vertex. You must consider **all orders for adjacency lists**, so try to make your example have as little branching as you can.

Homework 4.04 (10 points)

For the graph below, indicate which edges would be in the minimum spanning tree constructed by Prim's MST algorithm and which would be in the tree constructed by Dijkstra's shortest-path algorithm, both using A as the source.



Asymptotic Analysis

Homework 4.05 (10 points)

Let $G = (V, E)$ be a directed graph with n vertices and m edges. What is the worst time (big-O with parameters n and/or m) for Dijkstra's SSSP algorithm in its simplest implementation (as originally published)? Explain your answer in terms of priority-queue operations.

Now suppose the priority queue is implemented as a binary heap (as in Heapsort, excepting minimizing). What is the worst time for Dijkstra's SSSP algorithm now?

Many geographic graphs are planar. What property of planar graphs influences the choice of implementation if they the predominant type of graphs for which shortest paths are desired? Which of the above two implementations is likely to be fastest? Explain your answer briefly.

Kruskal's MST Algorithm

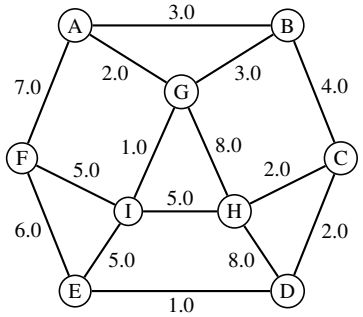
Homework 4.06 (10 points)

How much work must Kruskal's MST algorithm do **before** it starts choosing edges for its MST? Assume the undirected graph has n vertices and m edges. Explain the necessary preliminary work and its big-O cost if done efficiently.

What are the best case and worst case for Kruskal's MST algorithm with parameters n and/or m? Explain your answer.

Homework 4.07 (20 points)

Carry out the Kruskal MST algorithm on the weighted undirected graph below.

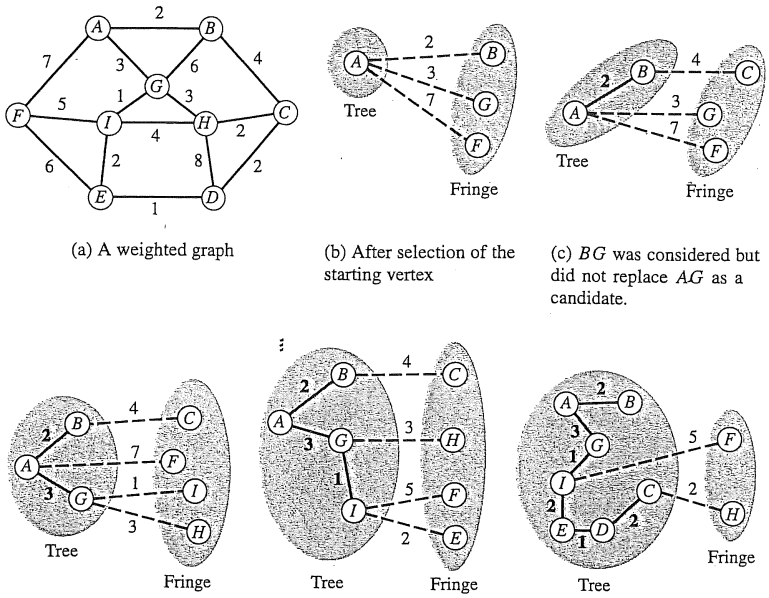


1. **(10 points)** Show the **disjoint sets of vertices** just before each edge is considered. Do not forget to include "sets" with only one vertex. Use circles or ovals around the vertices in one set, and list the vertices **with the leader first** as done in lecture and sections.

   Indicate whether each edge is taken to be part of the eventual MST or is discarded. In case of ties on weight, use alphabetical vertex order. Continue with the updated disjoint sets in a new diagram, and the next chosen edge, and so on, until the MST is complete.

2. **(5 points)** Explain how the algorithm can be opportunistic by not considering some of the edges in some cases. Were there some edges it did not need to consider in this problem? What is the smallest number of edges it **must** consider?

3. **(5 points)** Illustrate (partly) how the MST that was found obeys the **Minimum Spanning Tree Property** using the two lightest edges not in the MST.



(a) A weighted graph

(b) After selection of the starting vertex

(c) BG was considered but did not replace AG as a candidate.

(d) After AG is selected and fringe and candidates are updated

(e) IF has replaced AF as a candidate.

(f) After several more passes: The two candidate edges will be put in the tree.

Figure 8.4 An example for Prim's minimum spanning tree algorithm.

Exercise 8.9, where we discover that the worst case is worse than $\Theta(n^2)$. Can we do any better?

If we are to improve upon the heap time in general, clearly we need to consider implementations for which decreaseKey runs faster than $\Theta(\log n)$. However, we can afford to make getMin and deleteMin *slower* than $\Theta(\log n)$ in the trade-off. Can we think of an implementation for which decreaseKey is $O(1)$ and the others are no worse than $O(n)$? Then Equation (8.1) would evaluate to $\Theta(n^2 + m) = \Theta(n^2)$. Readers are invited to consider alternatives before continuing.

▪ ▪ ▪

The answer is so simple, we are likely to overlook it. Simply store the information in one or more arrays, indexed by vertex number. That is, we can use a separate array for each field, or we can collect fields in an organizer class and have one array whose entries are objects in this class. We will proceed with separate arrays because it simplifies the syntax

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*
fringeWgt	0	2	4			7	3	3	1
parent	−1	*A*	*B*			*A*	*A*	*G*	*G*
status	tree	tree	fringe	unseen	unseen	fringe	tree	fringe	fringe

Figure 8.5 Minimum spanning tree data structure for the situation in Figure 8.4(d): Adjacency lists are not shown. Vertices are assumed to be in alphabetical order within each list.

```
void primMST(EdgeList[] adjInfo, int n, int s, int[] parent, float[] fringeWgt)
    int[] status = new int[n+1];
    MinPQ pq = create(n, status, parent, fringeWgt);

    insert(pq, s, -1, 0);
    while (isEmpty(pq) == false)
        int v = getMin(pq);
        deleteMin(pq);
        updateFringe(pq, adjInfo[v], v);
    return;

/** See if a better connection is found to any vertex in
 * the list adjInfoOfV, and decreaseKey if so.
 * For a new connection, insert the vertex. */
void updateFringe(MinPQ pq, EdgeList adjInfoOfV, int v)
    EdgeList remAdj;
    remAdj = adjInfoOfV;
    while (remAdj ≠ nil)
        EdgeInfo wInfo = first(remAdj);
        int w = wInfo.to;
        float newWgt = wInfo.weight;
        if (pq.status[w] == unseen)
            insert(pq, w, v, newWgt);
        else if (pq.status[w] == fringe)
            if (newWgt < getPriority(pq, w))
                decreaseKey(pq, w, v, newWgt);
        remAdj = rest(remAdj);
    return;
```