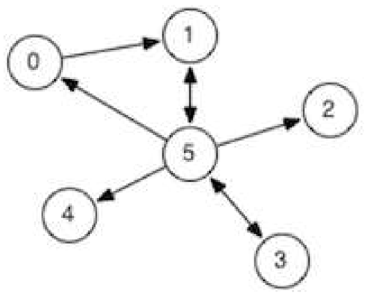
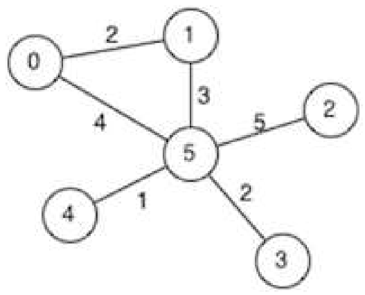
COMP9024 18x1 Data Structures and Algorithms

Week 05 Problem Set
Minimum Spanning Trees, Shortest Paths

1. (Graph representations)

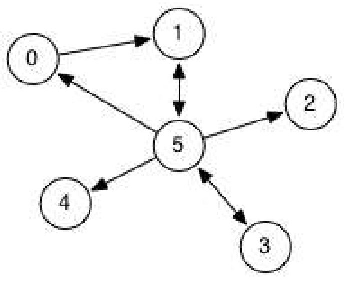
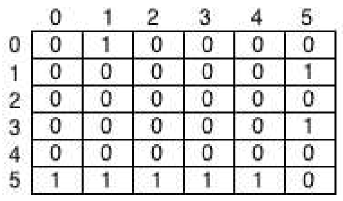
   For each of the following graphs:

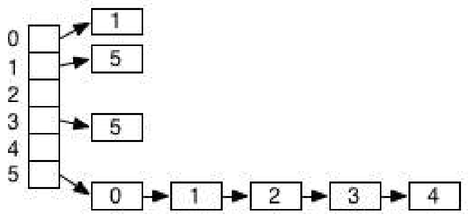
   Show the concrete data structures if the graph was implemented via:

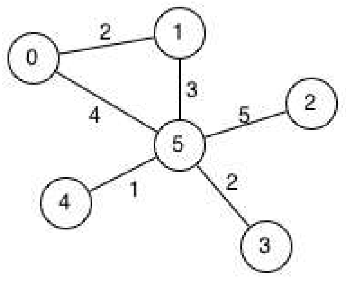
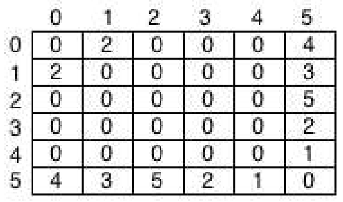
   a. adjacency matrix representation (assume full V×V matrix)
   b. adjacency list representation (if non-directional, include both (v,w) and (w,v))

   Answer:

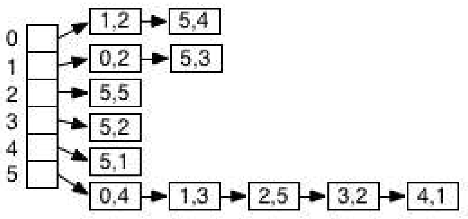



	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	0	0
3	0	0	0	0	0	1
4	0	0	0	0	0	0
5	1	1	1	1	1	0



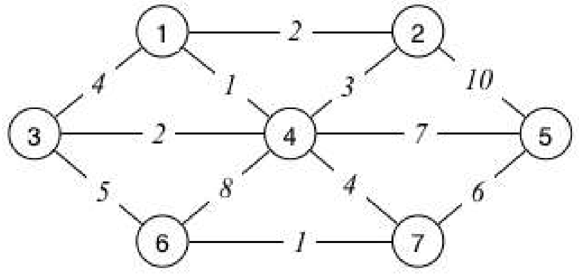



	0	1	2	3	4	5
0	0	2	0	0	0	4
1	2	0	0	0	0	3
2	0	0	0	0	0	5
3	0	0	0	0	0	2
4	0	0	0	0	0	1
5	4	3	5	2	1	0



2. (MST – Kruskal's algorithm)

   a. Identify a minimum spanning tree in the following graph (without applying any of the algorithms from the lecture):



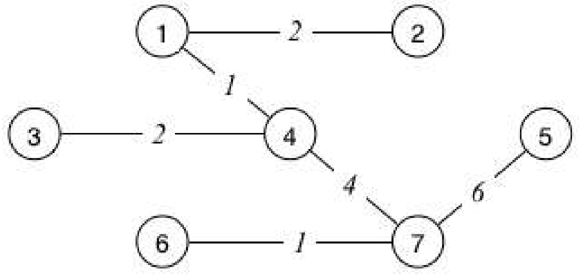
What approach did you use in determining the MST?

b. Show how Kruskal's algorithm would construct the MST for the above graph. How many edges do you have to consider?

c. For a graph *G=(V,E)*, what is the least number of edges that might need to be considered by Kruskal's algorithm, and what is the most number of edges? Add one vertex and edge to the above graph to force Kruskal's algorithm to the worst case.

Answer:

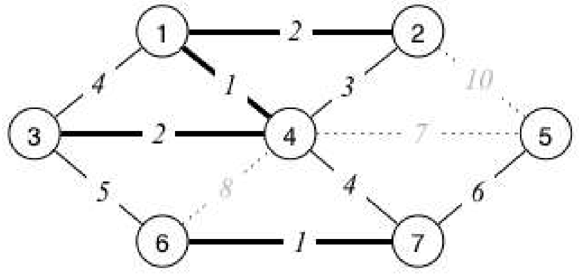
a. The following is the unique MST for the given graph:



I suspect (no proof) that most people would use a strategy like:

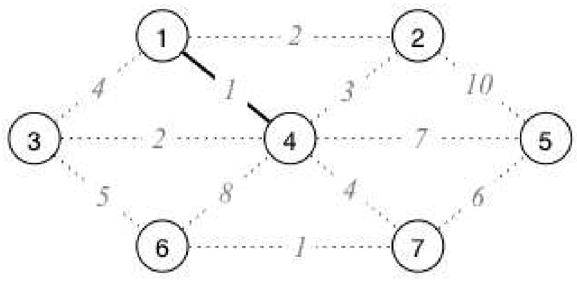
- discount all the high-cost edges to simplify the problem; this might remove edges with weights 7, 8 and 10
- include all of the low-cost edges to make a start on the solution; this might include edges with weights 1 and 2

This would get you to a partial solution like:

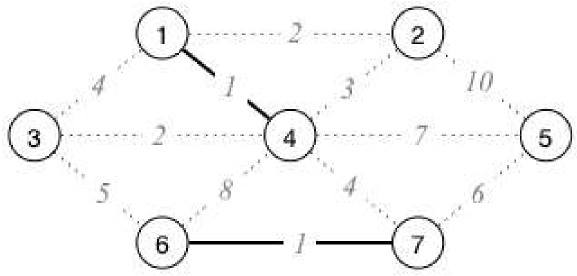


Now there are much fewer choices to consider. You could quickly reject the edge 1-3 with weight 4; vertices 1 and 3 are already included, using lower cost edges. Similarly, you could reject the edge 2-4 with weight 3, because vertex 2 is already included using a lower-cost edge. That leaves you with only a choice between the 3-6 edge and the 4-7 edge, to connect the {6,7} part of the graph to the {1,2,3,4} part. For that, you'd choose the lower-cost 4-7 edge. Finally, you need to connect vertex 5 and there's now only one choice: the 5-7 edge with weight 6.

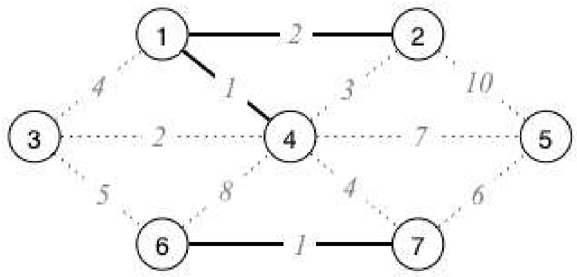
b. In the first iteration of Kruskal's algorithm, we could choose either 1-4 or 6-7, since both edges have weight 1. Assume we choose 1-4. Since its inclusion produces no cycles, we add it to the MST (non-existent edges are indicated by dotted lines):



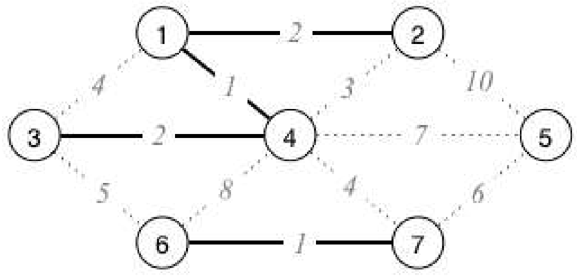
In the next iteration, we choose 6-7. Its inclusion produces no cycles, so we add it to the MST:



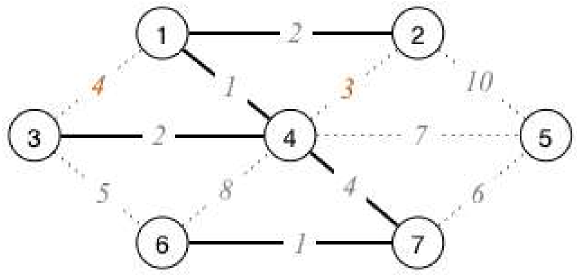
In the next iteration, we could choose either 1-2 or 3-4, since both edges have weight 2. Assume we choose 1-2. Since its inclusion produces no cycles, we add it to the MST:



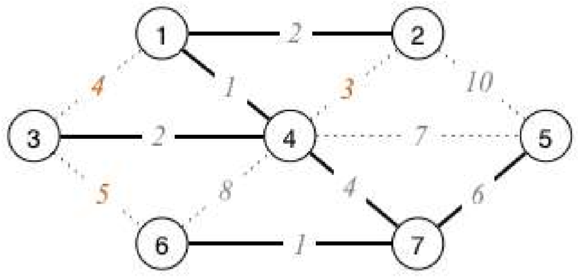
In the next iteration, we choose 3-4. Its inclusion produces no cycles, so we add it to the MST:



In the next iteration, we would first consider the lowest-cost unused edge. This is 2-4, but its inclusion would produce a cycle, so we ignore it. We then consider 1-3 and 4-7 which both have weight 4. If we choose 1-3, that produces a cycle so we ignore that edge. If we add 4-7 to the MST, there is no cycle and so we include it:



Now the lowest-cost unused edge is 3-6, but its inclusion would produce a cycle, so we ignore it. We then consider 5-7. If we add 5-7 to the MST, there is no cycle and so we include it:



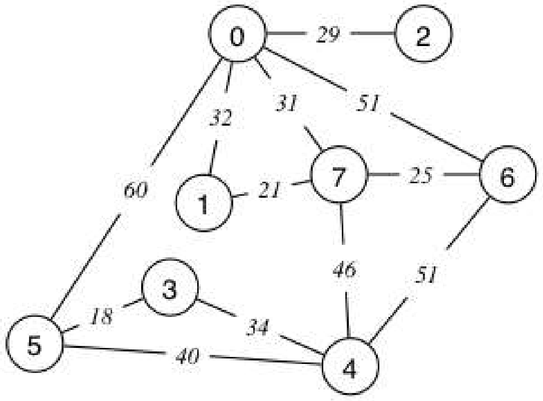
At this stage, all vertices are connected and we have a MST.

For this graph, we considered 9 of the 12 possible edges in determining the MST.

c. For a graph with *V* vertices and *E* edges, the best case would be when the first *V-1* edges we consider are the lowest cost edges and none of these edges leads to a cycle. The worst case would be when we had to consider all *E* edges. If we added a vertex 8 to the above graph, and connected it to vertex 5 with edge cost 11 (or any cost larger than all the other edge costs in the graph), we would need to consider all edges to construct the MST.

3. (MST – Prim's algorithm)

Trace the execution of Prim's algorithm to compute a minimum spanning tree on the following graph:



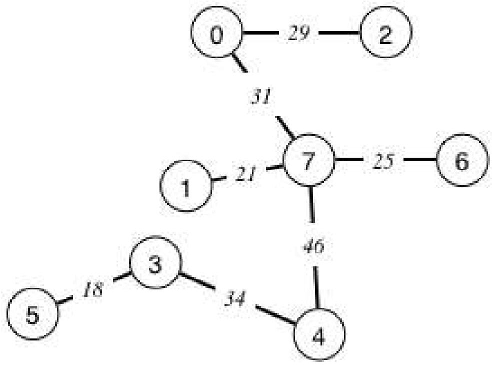
Choose a random vertex to start with. Draw the resulting minimum spanning tree.

Answer:

If we start at node 5, for example, edges would be found in the following order:

- 5-3
- 3-4
- 4-7
- 1-7
- 6-7
- 0-7
- 0-2

The minimum spanning tree:



4. (Priority queue for Dijkstra's algorithm)

   Assume that a priority queue for Dijkstra's algorithm is represented by a global variable PQueue of the following structure:

```
#define MAX_NODES 1000
typedef struct {
   Vertex item[MAX_NODES];  // array of vertices currently in queue
   int    length;           // #values currently stored in item[] array
} PQueueT;

PQueueT PQueue;
```

   Further assume that vertices are stored in the `item[]` array in the priority queue in *no* particular order. Rather, the `item[]` array acts like a set, and the priority is determined by reference to a `priority[0..nV-1]` array.

   If the priority queue PQueue is initialised as follows:

```
void PQueueInit() {
   PQueue.length = 0;
}
```

   then give the implementation of the following functions in C:

```
// insert vertex v into priority queue PQueue
// no effect if v is already in the queue
void joinPQueue(Vertex v) { ... }

// remove the highest priority vertex from PQueue
// remember: highest priority = lowest value priority[v]
// returns the removed vertex
Vertex leavePQueue(int priority[]) { ... }
```

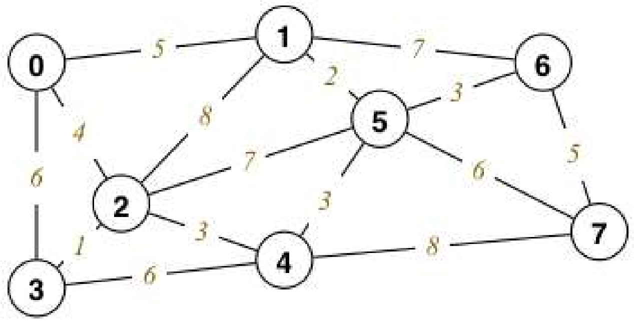
   Answer:

   The following are possible implementations for joining and leaving the priority queue:

```
void joinPQueue(Vertex v) {
   assert(PQueue.length < MAX_NODES);
   int i = 0;
   while (i < PQueue.length && PQueue.item[i] != v)  // check if v already in queue
      i++;
   if (i == PQueue.length) {                         // v not found => add it at the end
      PQueue.item[PQueue.length] = v;
      PQueue.length++;
   }
}

#define VERY_HIGH_VALUE 999999

Vertex leavePQueue(int priority[]) {
    assert(PQueue.length > 0);

    int i, bestVertex, bestIndex, bestWeight = VERY_HIGH_VALUE;
    for (i = 0; i < PQueue.length; i++) {               // find i with min priority[item[i]]
       if (priority[PQueue.item[i]] < bestWeight) {
          bestIndex = i;
          bestWeight = priority[PQueue.item[i]];
          bestVertex = PQueue.item[i];                   // vertex with lowest value so far
       }
    }
    PQueue.length--;
    PQueue.item[bestIndex] = PQueue.item[PQueue.length];  // replace dequeued node
                                                          // by last element in array
    return bestVertex;
 }
```

5. (Dijkstra's algorithm)

   a. Trace the execution of Dijkstra's algorithm on the following graph to compute the minimum distances from source node 0 to all other vertices:



   Show the values of `vSet`, `dist[]` and `pred[]` after each iteration.

   b. Implement Dijkstra's algorithm in C using your priority queue functions for Exercise 4 and the Weighted Graph ADT (`WGraph.h`, `WGraph.c`) from the lecture. Your program should

   - prompt the user for the number of nodes in a graph,
   - prompt the user for a source node,
   - build a weighted undirected graph from user input,
   - compute and output the distance and a shortest path from the source to every vertex of the graph.

   An example of the program executing is shown below. The input graph consists of a simple triangle along with a fourth, disconnected node.

```
prompt$ ./dijkstra
Enter the number of vertices: 4
Enter the source node: 0
Enter an edge (from): 0
Enter an edge (to): 1
Enter the weight: 42
Enter an edge (from): 0
Enter an edge (to): 2
Enter the weight: 25
Enter an edge (from): 1
Enter an edge (to): 2
Enter the weight: 14
Enter an edge (from): done
Finished.
0: distance = 0, shortest path: 0
1: distance = 39, shortest path: 0-2-1
```

```
2: distance = 25, shortest path: 0-2
3: no path
```

Note that any non-numeric data can be used to 'finish' the interaction. Test your algorithm with the graph from Exercise 5a.

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to the problem set and week. It expects to find a program named* `dijkstra.c` *in the current directory. You can use dryrun as follows:*

```
prompt$ ~cs9024/bin/dryrun prob05
```

Answer:

a. Initialisation:

```
vSet = { 0, 1, 2, 3, 4, 5, 6, 7 }
dist = [ 0, ∞, ∞, ∞, ∞, ∞, ∞, ∞ ]
pred = [ -1, -1, -1, -1, -1, -1, -1, -1 ]
```

The vertex in vSet with minimum dist[] is 0. Relaxation along the edges (0,1,5), (0,2,4) and (0,3,6) results in:

```
vSet = { 1, 2, 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 6, ∞, ∞, ∞, ∞ ]
pred = [ -1, 0, 0, 0, -1, -1, -1, -1 ]
```

Now the vertex in vSet with minimum dist[] is 2. Considering all edges from 2 to nodes still in vSet:

- relaxation along (2,1,8) does not give us a shorter distance to node 1
- relaxation along (2,3,1) yields a smaller value (4+1 = 5) for dist[3], and pred[3] is updated to 2
- relaxation along (2,4,3) yields a smaller value (4+3 = 7) for dist[4], and pred[4] is updated to 2
- relaxation along (2,5,7) yields a smaller value (4+7 = 11) for dist[5], and pred[5] is updated to 2

```
vSet = { 1, 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 11, ∞, ∞ ]
pred = [ -1, 0, 0, 2, 2, 2, -1, -1 ]
```

Next, we could choose either 1 or 3, since both vertices have minimum distance 5. Suppose we choose 1. Relaxation along (1,5,2) and (1,6,7) results in new values for nodes 5 and 6:

```
vSet = { 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, ∞ ]
pred = [ -1, 0, 0, 2, 2, 1, 1, -1 ]
```

Now we consider vertex 3. The only adjacent node still in vSet is 4, but there is no shorter path to 4 through 3. Hence no update to dist[] or pred[]:

```
vSet = { 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, ∞ ]
pred = [ -1, 0, 0, 2, 2, 1, 1, -1 ]
```

Next we could choose either vertex 4 or 5. Suppose we choose 4. Edge (4,7,8) is the only one that leads to an update:

```
vSet = { 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, 15 ]
pred = [ -1, 0, 0, 2, 2, 1, 1, 4 ]
```

Vertex 5 is next. Relaxation along edges (5,6,3) and (5,7,6) results in:

```
vSet = { 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 10, 13 ]
pred = [ -1, 0, 0, 2, 2, 1, 5, 5 ]
```

Of the two vertices left in vSet, 6 has the shorter distance. Edge (6,7,5) does not update the values for node 7 since dist[7]=13<dist[6]+5=15. Hence:

```
vSet = { 7 }
dist = [ 0, 5, 4, 5, 7, 7, 10, 13 ]
pred = [ -1, 0, 0, 2, 2, 1, 5, 5 ]
```

Processing the last remaining vertex in vSet will obviously not change anything. The values in pred[] determine shortest paths to all nodes as follows:

```
0: distance = 0, shortest path: 0
1: distance = 5, shortest path: 0-1
2: distance = 4, shortest path: 0-2
3: distance = 5, shortest path: 0-2-3
4: distance = 7, shortest path: 0-2-4
5: distance = 7, shortest path: 0-1-5
6: distance = 10, shortest path: 0-1-5-6
7: distance = 13, shortest path: 0-1-5-7
```

b. Sample implementation of Dijkstra's algorithm, including a recursive function to display a path via `pred[]`:

```
void showPath(int v, int pred[]) {
   if (pred[v] == -1) {
      printf("%d", v);
   } else {
      showPath(pred[v], pred);
      printf("-%d", v);
   }
}

void dijkstraSSSP(Graph g, int nV, Vertex source) {
   int  dist[MAX_NODES];
   int  pred[MAX_NODES];
   bool vSet[MAX_NODES];  // vSet[v] = true <=> v has not been processed
   int s, t;

   PQueueInit();
   for (s = 0; s < nV; s++) {
      joinPQueue(s);
      dist[s] = VERY_HIGH_VALUE;
      pred[s] = -1;
      vSet[s] = true;
   }
   dist[source] = 0;
   while (!PQueueIsEmpty()) {
      s = leavePQueue(dist);
      vSet[s] = false;
      for (t = 0; t < nV; t++) {
         if (vSet[t]) {
            int weight = adjacent(g,s,t);
            if (weight > 0 && dist[s]+weight < dist[t]) {  // relax along (s,t,weight)
               dist[t] = dist[s] + weight;
               pred[t] = s;
            }
         }
      }
   }
   for (s = 0; s < nV; s++) {
      printf("%d: ", s);
      if (dist[s] < VERY_HIGH_VALUE) {
         printf("distance = %d, shortest path: ", dist[s]);
         showPath(s, pred);
         putchar('\n');
      } else {
         printf("no path\n");
      }
```

```
    }
}
```