University of Houston

Final-Exam Review

COSC 3320
Algorithms and Data Structures

Gopal Pandurangan

This page intentionally left blank.

0 Previous Concepts

Concepts seen previously can be found in the mid-term review, located here.

1 Greedy Algorithms

A **Greedy Algorithm** will make the *locally optimal* choice at each step. A greedy algorithm is not always optimal, but in many cases is optimal or sufficiently close to the optimal solution to justify its use. Consider the **Unbounded Knapsack Problem**: given n items with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$, and a weight-limit W, determine the non-negative integers for each item, c_i, such that

$$\sum_{i=1}^{n} c_i v_i \text{ is maximal}$$
$$\sum_{i=1}^{n} c_i w_i \leq W$$

A greedy solution is to choose the "best" item at each step. Sort the items by their ratios of value to weight, $\frac{v_i}{w_i}$, and insert the maximum possible number of copies of the first item, then the second, and so on. This algorithm is not optimal: suppose our items, $\{n_1, n_2, n_3\}$ have weights $\{2, 3, 4\}$ and values $\{4, 6, 9\}$, with a maximum weight of 5. The algorithm will output $\{n_3\}$, which has a value of 9, when $\{n_2, n_1\}$ has a value of 10.

2 Graphs

A graph is a collection of vertices (also called nodes) and edges between them. There are two types of graphs: directed and undirected. In a directed graph, an edge has a direction or orientation. In an undirected graph, edges have no direction.

A cycle is a collection of vertices in a graph such that a vertex can be reached from itself. In an undirected graph with no cycle, there are at most $\binom{n}{2=\frac{(n)(n-1)}{2}}$. In a directed graph, this is doubled, and is simply $(n)(n-1)$.

2.1 Representing a Graph

A graph can be represented in three ways:

1. An **adjacency list**, in which each vertex stores a list of adjacent vertices.
2. An **adjacency matrix**, a matrix A where $A_{i,j} = 1$ if and only if there is an edge between the i^{th} and j^{th} vertices.

Let n denote the number of vertices and m the number of edges. The space and time complexity of the above representations are given in the following table:

	Adjacency List	Adjacency Matrix
Space Complexity	$O(n+m)$	$O(n^2)$
Add vertex	$O(1)$	$O(n^2)$
Add edge	$O(1)$	$O(1)$
Remove vertex	$O(m)$	$O(n^2)$
Remove edge	$O(n)$	$O(1)$
Check vertices are adjacent	$O(n)$	$O(1)$

2.2 Depth-First Search

A **Depth-First Search** (DFS) is a recursive algorithm for searching every vertex of a graph. Begin with some vertex, v, and perform the DFS on its unvisited neighbors. This has the following pseudocode:

Algorithm DFS(G, v): depth-first search of G starting at node v

```
1: procedure DFS(G, v)
2:     mark v as visited
3:     for all neighbors u of v do
4:         if u is not visited then
5:             call DFS(G, u)
6:         end if
7:     end for
8: end procedure
```

This algorithm runs in linear, $O(n + m)$ time, where n is the number of vertices and m is the number of edges. It uses $O(n)$ memory to store the stack of vertices in the worst case.

2.3 Breadth-First Search

A **Breadth-First Search** (BFS) is an iterative algorithm for searching every vertex of a graph. Begin with some vertex, v, and visit each of its unvisited neighbors. Repeat on the neighbors of the newly visited vertices, and repeat. This is typically done with a queue.

Algorithm BFS(G, v): breadth-first search of G starting at node v

```
1: procedure BFS(G, v)
2:     Q = queue()                          ▷ Q is a queue of which vertices to explore.
3:     enqueue v to Q
4:     while Q is not empty do
5:         a = dequeue(Q)                   ▷ Dequeue the queue and visit.
6:         for neighbors u of a do
7:             if u is unvisited then
8:                 enqueue u to Q
9:                 mark u as visited
10:            end if
11:        end for
12:    end while
13: end procedure
```

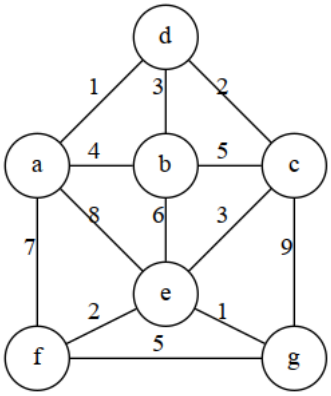
This algorithm runs in linear, $O(n + m)$ time, where n is the number of vertices and m is the number of edges. It uses $O(n)$ memory to store the stack of vertices in the worst case.

3 Graph Algorithms

3.1 Minimum Spanning Tree

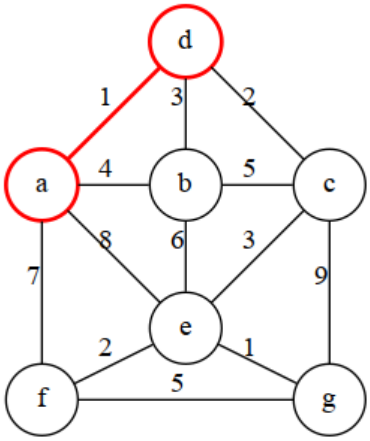
A **Minimum Spanning Tree** (MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all vertices and minimizes the total weight. In a case where the edge-weights are unique, the MST will be unique as well. A simple induction proof shows that the MST on a connected graph with n vertices will have $n - 1$ edges.

We present three algorithms to determine the MST and perform each on the following graph:

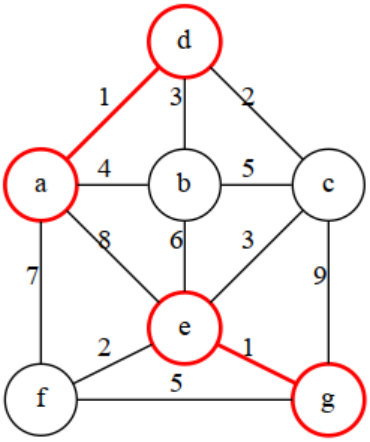


3.1.1 Kruskal's Algorithm

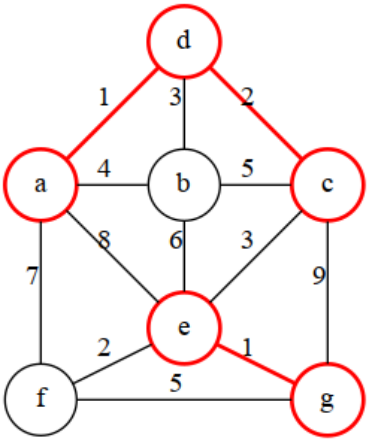
Begin with an empty tree. Sort the edges by weight and add the smallest remaining edge to tree *if it does not create a cycle*. Repeat until you have $n-1$ edges in your tree.



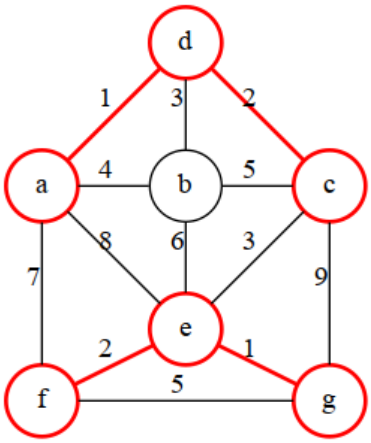
(1)



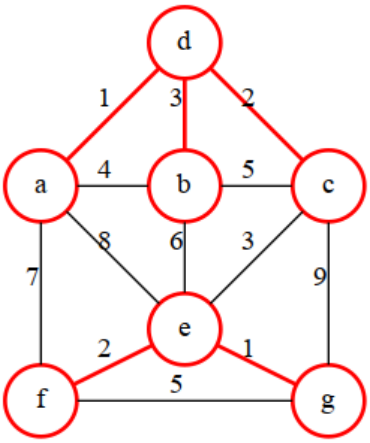
(2)



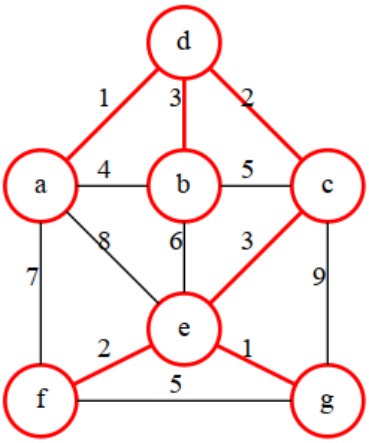
(3)



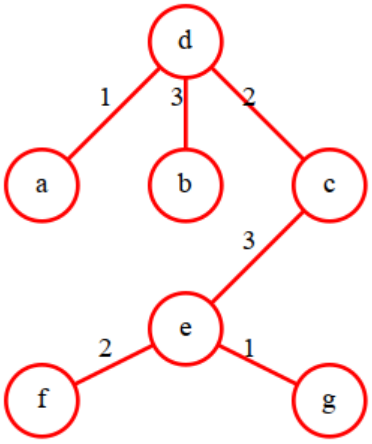
(4)



(5)



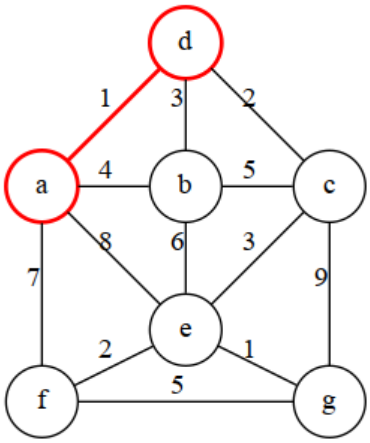
(6)



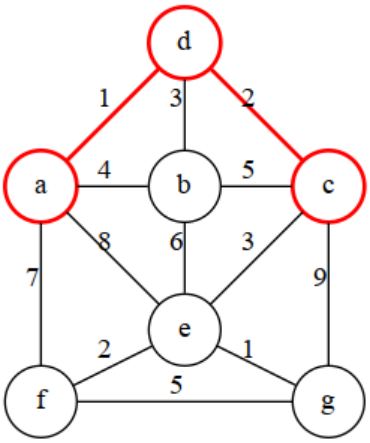
(7)

3.1.2 Prim's Algorithm

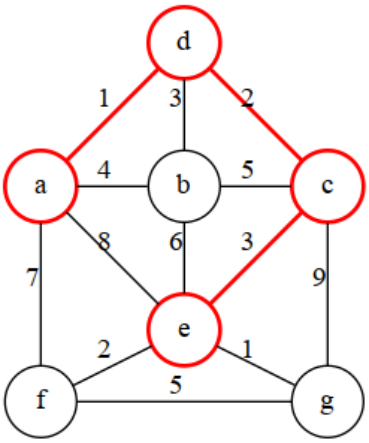
Begin with an empty tree. Start at any vertex of the graph. At each step, append the smallest edge that connects the vertices in the MST to the vertices not yet in the MST. Repeat until all vertices are in the tree.



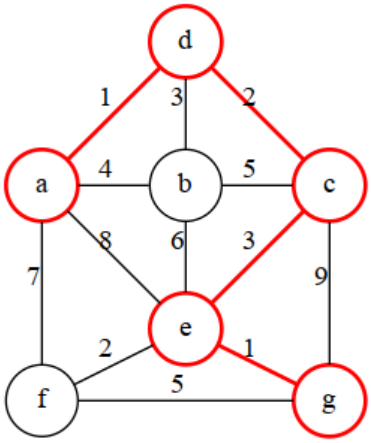
(1)



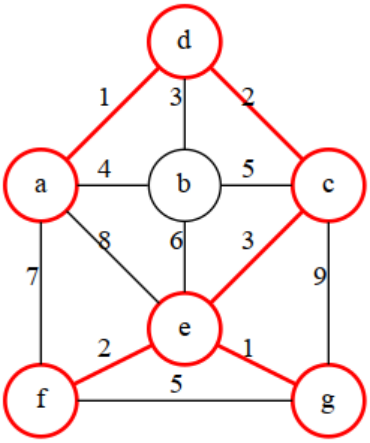
(2)



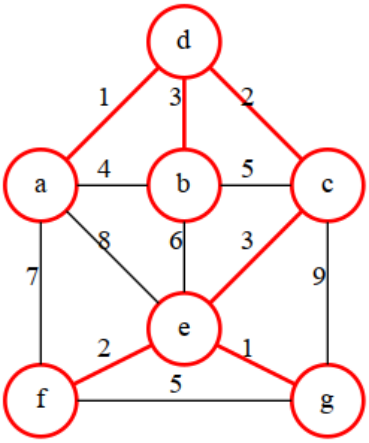
(3)



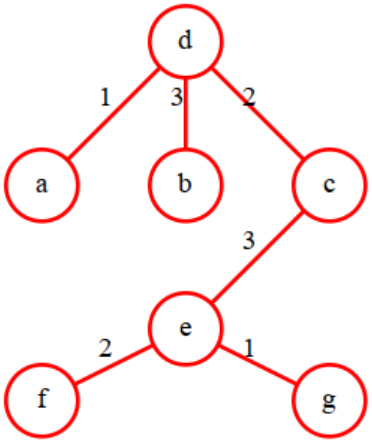
(4)



(5)



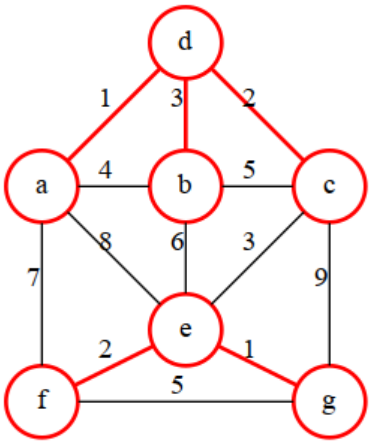
(6)



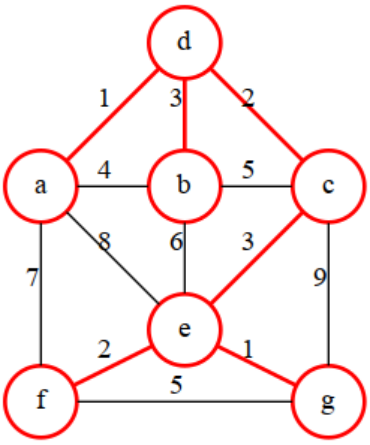
(7)

3.1.3 Boruvka's algorithm

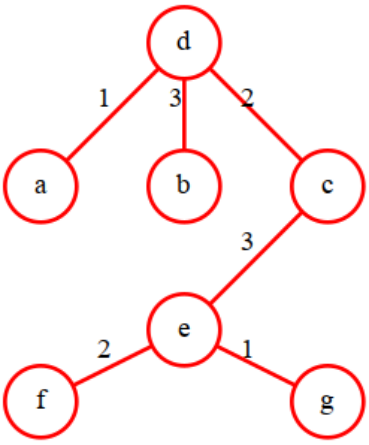
Begin with an empty tree and with all vertices of the graph as separate components. At each step, add the smallest edge connected to each component to the tree (these edges will not necessarily be unique). Repeat until all vertices are in the tree.



(1)



(2)



(3)

3.2 Shortest Paths

3.2.1 Dijkstra's Algorithm

3.2.2 A* Search

3.2.3 Bellman-Ford Algorithm

3.2.4 Floyd–Warshall Algorithm