Algorithms & Data Structures

Prim & Kruskal

**Name:** Alexandru Cardas

**ID:** C17504869

**Class:** DT228

**Year:** 2nd

Table of Contents

[Introduction 4](#__RefHeading___Toc1767_626663100)

[Note 4](#__RefHeading___Toc2598_704612103)

[Minimum Spanning Tree 4](#__RefHeading___Toc2600_704612103)

[Growing a minimum spanning Tree 5](#__RefHeading___Toc2602_704612103)

[Starting Graph for Kruskal and Prim 5](#__RefHeading___Toc430_626663100)

[Prim 6](#__RefHeading___Toc422_626663100)

[Definition 6](#__RefHeading___Toc2604_704612103)

[Efficiency 6](#__RefHeading___Toc7818_857513756)

[Adjacency Lists 7](#__RefHeading___Toc263_1039298202)

[Diagram 7](#__RefHeading___Toc424_626663100)

[Traversal 8](#__RefHeading___Toc265_1039298202)

[Depth-First Search 8](#__RefHeading___Toc1769_626663100)

[Steps 9](#__RefHeading___Toc1732_2635631669)

[Terminal Output 9](#__RefHeading___Toc7880_857513756)

[Breadth-First Search 13](#__RefHeading___Toc1771_626663100)

[Steps 14](#__RefHeading___Toc1734_2635631669)

[Terminal Output 15](#__RefHeading___Toc2434_704612103)

[PrimGraph 18](#__RefHeading___Toc2436_704612103)

[Parent & Distance Arrays 23](#__RefHeading___Toc2438_704612103)

[Graph Traversal 24](#__RefHeading___Toc2606_704612103)

[First Step 24](#__RefHeading___Toc2442_704612103)

[Second Step 25](#__RefHeading___Toc2444_704612103)

[Third Step 26](#__RefHeading___Toc2446_704612103)

[Fourth Step 27](#__RefHeading___Toc2448_704612103)

[Fifth Step 28](#__RefHeading___Toc2450_704612103)

[Sixth Step 29](#__RefHeading___Toc2452_704612103)

[Seventh Step 30](#__RefHeading___Toc2454_704612103)

[Eight Step 31](#__RefHeading___Toc2456_704612103)

[Ninth Step 32](#__RefHeading___Toc2458_704612103)

[Tenth Step 33](#__RefHeading___Toc2460_704612103)

[Eleventh Step 34](#__RefHeading___Toc2462_704612103)

[Twelfth Step 35](#__RefHeading___Toc2464_704612103)

[Complete Graph 36](#__RefHeading___Toc2466_704612103)

[Terminal Output 37](#__RefHeading___Toc2468_704612103)

[Kruskal 41](#__RefHeading___Toc275_1039298202)

[Definition 41](#__RefHeading___Toc1798_626663100)

[Efficiency 41](#__RefHeading___Toc432_626663100)

[Path Compression 42](#__RefHeading___Toc2470_704612103)

[UnionFindSets 43](#__RefHeading___Toc7820_857513756)

[KruskalGraph 45](#__RefHeading___Toc7822_857513756)

[Set Formation 47](#__RefHeading___Toc7824_857513756)

[Terminal Output 47](#__RefHeading___Toc7826_857513756)

[Union Formation 49](#__RefHeading___Toc7828_857513756)

[Initial Union 49](#__RefHeading___Toc7830_857513756)

[First Union 50](#__RefHeading___Toc7832_857513756)

[Second Union 51](#__RefHeading___Toc7834_857513756)

[Third Union 52](#__RefHeading___Toc7836_857513756)

[Fourth Union 53](#__RefHeading___Toc7838_857513756)

[Fifth Union 54](#__RefHeading___Toc7840_857513756)

[Sixth Union 55](#__RefHeading___Toc7842_857513756)

[Seventh Union 56](#__RefHeading___Toc7844_857513756)

[Eighth Union 57](#__RefHeading___Toc7846_857513756)

[Graph Traversal 58](#__RefHeading___Toc7848_857513756)

[First Move 58](#__RefHeading___Toc1800_626663100)

[Second Move 59](#__RefHeading___Toc2068_2635631669)

[Third Move 60](#__RefHeading___Toc1804_626663100)

[Fourth Move 61](#__RefHeading___Toc2070_2635631669)

[Fifth Move 62](#__RefHeading___Toc2072_2635631669)

[Sixth Move 63](#__RefHeading___Toc2074_2635631669)

[Seventh Move 64](#__RefHeading___Toc2076_2635631669)

[Eight Move 65](#__RefHeading___Toc2078_2635631669)

[Ninth Move 66](#__RefHeading___Toc2080_2635631669)

[Tenth Move 67](#__RefHeading___Toc2082_2635631669)

[Eleventh Move 68](#__RefHeading___Toc2084_2635631669)

[Twelfth Move 69](#__RefHeading___Toc2086_2635631669)

[Thirteenth Move 70](#__RefHeading___Toc2088_2635631669)

[Fourteenth Move 71](#__RefHeading___Toc2090_2635631669)

[Complete Graph 72](#__RefHeading___Toc2092_2635631669)

[Reflection 73](#__RefHeading___Toc7922_857513756)

# Introduction

|  |  |  |
| --- | --- | --- |
| Green for traversing | Blue for visited | Red for ignored |

### Note

* For this assignment, vertex **C** is picked for demonstrating traversal and as starting point for Prim’s algorithm.
* The traversal Breadth and Depth first searches use an adjacency list that does not have a sentinel, while the Prim’s algorithm has been implemented using an adjacency list with a sentinel.
* The diagrams are very big in order for them to be readable and so I recommend scrolling via the arrow keys with the size of the page set to optimum so one key input will traverse straight into the next page rather than scrolling down.
* Some of the code that is indented too much due to too many loops will display poorly. Check the code inside the files if it becomes too difficult to track each line of code.
* Prim’s heap is implemented via an adjacency list and Kruskal’s heap is implemented using a simple array.
* Screen captures for the output have been replaced with terminal output. A script with an output to a file has been created and once the class has been executed, the output of that class gets recorded onto the .txt file and then copy pasted here.
* The code has comments explaining how each section works.

For coding purposes and readability I decided to make a few changes to the naming convention in my code so that anyone that reads it will understand where I am getting these values from. Most notable ones are:

* V which stands for vertex amount has been changed to **vertexAmount**.
* E which stands for edge amount has been changed to **edgeAmount**.
* u which is the starting point of an edge has been changed to **origin**.
* v which is the ending point of an edge has been changed to **destination**.

## Minimum Spanning Tree

The two minimum-spanning-tree algorithms used, elaborate on the generic method. They each use a specific rule to determine a safe edge of Generic-MST. In Kruskal’s algorithm, the set A is a forest whose vertices are all those of the given graph. The safe edge added to A is always a least-weight edge in the graph that connects two distinct components. In Prim’s algorithm, the set A forms a single tree. The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

### Growing a minimum spanning Tree

Assume that we have a connected, undirected graph G = (V, E) with a weight function w : E -> R, and we wish to find a minimum spanning tree for G. The two algorithms considered use a greedy approach to the problem, although they differ in how they apply this approach. This greedy strategy is captured by the following generic method, which grows the minimum spanning tree one edge at a time. The generic method manages a set of edges A, maintaining the following loop invariant: **Prior to each iteration, A is a subset of some minimum spanning tree.** At each step, we determine an edge (u, v) that we can add to A without violating this invariant, in the sense that A union {(u, v)} is also a subset of a minimum spanning tree. We call such an edge a safe edge for A, since we can add it safely to A while maintaining the invariant.

## Starting Graph for Kruskal and Prim

# Prim

## Definition

Like Kruskal’s algorithm, Prim’s algorithm is a special case of the generic min-minimum-spanning-tree method. Prim’s algorithm operates much like Dijkstra’s algorithm for finding shortest paths in a graph. Prim’s algorithm has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V . Each step adds to the tree A a light edge that connects A to an isolated vertex - one on which no edge of A is incident. By Corollary, this rule adds only edges that are safe for A; therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree’s weight. In order to implement Prim’s algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A. The connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm. During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute. For each vertex v, the attribute v.key is the minimum weight of any edge connecting v to a vertex in the tree; by convention, v:key = infinite if there is no such edge. The attribute v.pi names the parent of v in the tree.

## Efficiency

The running time of Prim’s algorithm depends on how we implement the min-priority queue Q. If we implement Q as a binary min-heap we can use the BUILD-MIN-HEAP procedure to perform in O(V) time. The body of the while loop executes |V| times, and since each EXTRACT -MIN operation takes O(lg V) time, the total time for all calls to EXTRACT-MIN is O(V lg V). The for loop at later stage executes O(E) times altogether, since the sum of the lengths of all adjacency lists is 2 |E|. Within the for loop, we can implement the test for membership in Q in constant time by keeping a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q. The assignment involves an implicit DECREASE-KEY operation on the min-heap, which a binary min-heap supports in O(lg V) time. Thus, the total time for Prim’s algorithm is O(V lg V + E lg V) = O(E lg V), which is asymptotically the same as for our implementation of Kruskal’s algorithm.

## Adjacency Lists

Using the document headers we can create lists of edges, the first parameter takes in account the amount of vertices and the second parameter accounts for the number of edges.

### Diagram

Once we have established the amount of vertices we can create the array that holds the head of each linked lists, hence creating the adjacency list. Using a for loop that iterates through all the edges, we assign each parameter accordingly. First parameter is the first vertex, the second is the second vertex which both combined gives us the edge and the third parameter is the weight of the edge. Creating the edge object code is the same for a linked list, we add an element to the head of the original vertex and re-establish the link between the previously added vertex and the new one.

## Traversal

### Depth-First Search

*/\*\**  
 *\* This class doesn't use a stack, but it does use recursion which resembles a stack for operations.*  
 *\* Everything surrounded by square brackets constitutes the final traversal. ex: [A] marked as visited*  
 *\**  
 *\** ***@param*** *s used as the starting node for the graph traversal*  
 *\*/*  
void dfsRecursive(int s)  
{  
 // we create a clone of adjacency list because we do not want to modify anything from the original adj  
 adj = adjacencyList.clone();  
 boolean[] visited = new boolean[vertexAmount + 1];  
  
 System.*out*.println("Add root " + *toChar*(s));  
  
 dfsVisit(s, visited);  
}  
  
private void dfsVisit(int v, boolean[] visited)  
{  
 // every vertex that gets passed into the method and is also inserted into visited array  
 visited[v] = true;  
  
 System.*out*.println("\n///// [" + *toChar*(v) + "] marked as visited /////\n");  
  
 // check the children of that vertex  
 while (!*isEmpty*(adj[v]))  
 {  
 // get the immediate child  
 int destination = adj[v].vertex;  
  
 System.*out*.println("Point to the next vertex connected to " + *toChar*(v));  
  
 // go to the next vertex  
 adj[v] = adj[v].next;  
  
 if (visited[destination])  
 {  
 System.*out*.println("\t" + *toChar*(destination) + " has already been visited so ignore");  
 }  
  
 // check if the current vertex has been visited  
 if (!visited[destination])  
 {  
 // make a recursive call  
 System.*out*.println("--- Make recursive call with " + *toChar*(destination) + " ---");  
 dfsVisit(destination, visited);  
 }  
 }  
}

#### Steps

1. Add root C and mark C as visited.
2. Make a recursive call with the first child of C which is I.
3. Mark I as visited.
4. Make a recursive call with the first child of I which is H.
5. Mark H as visited.
6. I has been visited so ignore, make a recursive call with the second child of H which is G.
7. Mark G as visited.
8. I has been visited so ignore, H has been visited so ignore, make a recursive call with the third child of G which is F.
9. ...Now the steps keep repeating until all the vertices are visited and the rest can be ignored.

#### Terminal Output

///// Depth-First Search /////

Add root C

///// [C] marked as visited /////

Point to the next vertex connected to C

--- Make recursive call with I ---

///// [I] marked as visited /////

Point to the next vertex connected to I

--- Make recursive call with H ---

///// [H] marked as visited /////

Point to the next vertex connected to H

I has already been visited so ignore

Point to the next vertex connected to H

--- Make recursive call with G ---

///// [G] marked as visited /////

Point to the next vertex connected to G

I has already been visited so ignore

Point to the next vertex connected to G

H has already been visited so ignore

Point to the next vertex connected to G

--- Make recursive call with F ---

///// [F] marked as visited /////

Point to the next vertex connected to F

G has already been visited so ignore

Point to the next vertex connected to F

--- Make recursive call with E ---

///// [E] marked as visited /////

Point to the next vertex connected to E

F has already been visited so ignore

Point to the next vertex connected to E

--- Make recursive call with D ---

///// [D] marked as visited /////

Point to the next vertex connected to D

F has already been visited so ignore

Point to the next vertex connected to D

E has already been visited so ignore

Point to the next vertex connected to D

C has already been visited so ignore

Point to the next vertex connected to F

D has already been visited so ignore

Point to the next vertex connected to F

C has already been visited so ignore

Point to the next vertex connected to H

--- Make recursive call with B ---

///// [B] marked as visited /////

Point to the next vertex connected to B

H has already been visited so ignore

Point to the next vertex connected to B

C has already been visited so ignore

Point to the next vertex connected to B

--- Make recursive call with A ---

///// [A] marked as visited /////

Point to the next vertex connected to A

H has already been visited so ignore

Point to the next vertex connected to A

B has already been visited so ignore

Point to the next vertex connected to H

A has already been visited so ignore

Point to the next vertex connected to I

G has already been visited so ignore

Point to the next vertex connected to I

C has already been visited so ignore

Point to the next vertex connected to C

F has already been visited so ignore

Point to the next vertex connected to C

D has already been visited so ignore

Point to the next vertex connected to C

B has already been visited so ignore

### Breadth-First Search

*/\*\**  
 *\* This method will print out the traversal and the correct order is drawn from the vertices that have been visited.*  
 *\* Everything surrounded by square brackets constitutes the final traversal. ex: [A] marked as visited*  
 *\**  
 *\** ***@param*** *s starting vertex from where the traversal begins*  
 *\** ***@throws*** *MyExceptions for demo purposes*  
 *\*/*  
void breadthFirstSearch(int s) throws MyExceptions  
{  
 // create a clone of the adjacency list because we do not want to modify anything from the original adj  
 Node[] adj = adjacencyList.clone();  
 int v;  
 boolean[] visited = new boolean[vertexAmount + 1];  
  
 // put the starting node on the queue  
 System.*out*.println("Enqueue root " + *toChar*(s));  
 queue.enQueue(s);  
  
 // run until queue is empty  
 while (!queue.isEmpty())  
 {  
 // remove the vertex from the queue and traverse to his children  
 v = queue.deQueue();  
  
 System.*out*.println("--- Dequeue vertex " + *toChar*(v) + " ---");  
  
 if (visited[v])  
 {  
 System.*out*.println("\t" + *toChar*(v) + " has already been visited so ignore");  
 }  
  
 // check if the current vertex has been visited  
 if (!visited[v])  
 {  
 // mark the current vertex as visited  
 visited[v] = true;  
  
 System.*out*.println("\n///// [" + *toChar*(v) + "] marked as visited /////\n");  
  
 // check the vertices attached to the current one  
 while (!*isEmpty*(adj[v]))  
 {  
 if (!visited[adj[v].vertex])  
 {  
 // put the child onto the queue  
 System.*out*.println("Enqueue the connected vertex " + *toChar*(adj[v].vertex));  
 queue.enQueue(adj[v].vertex);  
 }  
 // go to the next node linked to it  
 adj[v] = adj[v].next;  
 }  
 }  
 }  
}

#### Steps

These steps do not include the duplicated values inside the queue unlike the output of the actual algorithm. Also in the code implementation, we queue the vertices and when we remove them from the queue, that’s when we also mark them as visited and start exploring the connected vertices.

1. Start exploring at C, put C on the queue.
2. Visit the first connected vertex of C which is I, put I on the queue.
3. Visit the second connected vertex of C which is F, put F on the queue.
4. Visit the third connected vertex of C which is D, put D on the queue.
5. Visit the fourth connected vertex of C which is B, put B on the queue.
6. C has been fully explored and so we remove it from the queue.
7. Start exploring at I, I is already in the queue.
8. Visit the first connected vertex of I which is H, put H on the queue.
9. Visit the second connected vertex of I which is G, put G on the queue.
10. C has been visited so ignore.
11. I has been fully explored and so we remove it from the queue.
12. Start exploring the first vertex in the queue which is F.
13. G has been visited so ignore.
14. Visit the second connected vertex of F which is E, put E on the queue.
15. D has been visited so ignore.
16. C has been visited so ignore.
17. F has been fully explored and so we remove it from the queue.
18. Start exploring the first vertex in the queue which is D.
19. D has been fully explored so we remove it from the queue.
20. Start exploring the first vertex in the queue which is B.
21. H has been visited so ignore, C has been visited so ignore, visit the third connected vertex of B which is A, put A on the queue.
22. All vertices have been visited, but queue is not empty.
23. B has been fully explored and so we remove it from the queue.
24. Start exploring the first vertex in the queue which is H.
25. H has been fully explored so we remove it from the queue.
26. Start exploring the first vertex in the queue which is G.
27. G has been fully explored so we remove it from the queue.
28. Start exploring the first vertex in the queue which is E.
29. E has been fully explored so we remove it from the queue.
30. Start exploring the first vertex in the queue which is A.
31. A has been fully explored so we remove it from the queue.
32. Queue is now empty, terminate the breadth-first search.

#### Terminal Output

///// Breadth-First Search /////

Enqueue root C

--- Dequeue vertex C ---

///// [C] marked as visited /////

Enqueue the connected vertex I

Enqueue the connected vertex F

Enqueue the connected vertex D

Enqueue the connected vertex B

--- Dequeue vertex I ---

///// [I] marked as visited /////

Enqueue the connected vertex H

Enqueue the connected vertex G

--- Dequeue vertex F ---

///// [F] marked as visited /////

Enqueue the connected vertex G

Enqueue the connected vertex E

Enqueue the connected vertex D

--- Dequeue vertex D ---

///// [D] marked as visited /////

Enqueue the connected vertex E

--- Dequeue vertex B ---

///// [B] marked as visited /////

Enqueue the connected vertex H

Enqueue the connected vertex A

--- Dequeue vertex H ---

///// [H] marked as visited /////

Enqueue the connected vertex G

Enqueue the connected vertex A

--- Dequeue vertex G ---

///// [G] marked as visited /////

--- Dequeue vertex G ---

G has already been visited so ignore

--- Dequeue vertex E ---

///// [E] marked as visited /////

--- Dequeue vertex D ---

D has already been visited so ignore

--- Dequeue vertex E ---

E has already been visited so ignore

--- Dequeue vertex H ---

H has already been visited so ignore

--- Dequeue vertex A ---

///// [A] marked as visited /////

--- Dequeue vertex G ---

G has already been visited so ignore

--- Dequeue vertex A ---

A has already been visited so ignore

||| ================================================================ |||

## PrimGraph

public class PrimGraph extends CommonFunctionsP  
{  
 private int vertexAmount;  
 private Node[] adj;  
 private Node sentinel;  
 private int[] mst;  
  
 private PrimGraph(String graphFile) throws IOException  
 {  
 int origin, destination;  
 int weight;  
 Node temp;  
  
 FileReader fr = new FileReader(graphFile);  
 BufferedReader reader = new BufferedReader(fr);  
  
 String splits = "\\s+";  
 String line = reader.readLine();  
 String[] parts = line.split(splits);  
  
 vertexAmount = Integer.*parseInt*(parts[0]);  
 int edgeAmount = Integer.*parseInt*(parts[1]);  
  
 sentinel = new Node();  
 sentinel.next = sentinel;  
  
  
 // here the sentinel gets placed at the end of the other vertices  
 adj = new Node[vertexAmount + 1];  
 for (destination = 1; destination <= vertexAmount; destination++)  
 {  
 adj[destination] = sentinel;  
 }  
  
 System.*out*.println("Vertices = " + parts[0] + " Edges = " + parts[1]);  
 System.*out*.println("Vertex[1] -- Vertex[2] (weight)");  
 System.*out*.println("Reading edges from text file");  
  
 for (int edge = 0; edge < edgeAmount; edge++)  
 {  
 line = reader.readLine();  
 parts = line.split(splits);  
 origin = Integer.*parseInt*(parts[0]);  
 destination = Integer.*parseInt*(parts[1]);  
 weight = Integer.*parseInt*(parts[2]);  
  
 System.*out*.println(*toChar*(origin) + "--" + *toChar*(destination) + "(" + weight + ")");  
  
 temp = new Node();  
 temp.vertex = destination;  
 temp.wgt = weight;  
 temp.next = adj[origin];  
 adj[origin] = temp;  
  
 temp = new Node();  
 temp.vertex = origin;  
 temp.wgt = weight;  
 temp.next = adj[destination];  
 adj[destination] = temp;  
 }  
 }  
  
 private void display()  
 {  
 Node node;  
  
 for (int v = 1; v <= vertexAmount; ++v)  
 {  
 System.*out*.print("\nadj[" + *toChar*(v) + "] ->");  
  
 for (node = adj[v]; node != sentinel; node = node.next)  
 {  
 if (node.next == sentinel)  
 {  
 System.*out*.print(" |" + *toChar*(node.vertex) + " | " + node.wgt + "|");  
 }  
 else  
 {  
 System.*out*.print(" |" + *toChar*(node.vertex) + " | " + node.wgt + "| ->");  
 }  
 }  
 }  
 System.*out*.println();  
 }  
  
 private void MSTPrim(int s)  
 {  
 int weightSum = 0;  
  
 int[] distance = new int[vertexAmount + 1];  
 int[] parent = new int[vertexAmount + 1];  
 int[] heapPosition = new int[vertexAmount + 1];  
  
 // fill the distance array with max values  
 Arrays.*fill*(distance, Integer.*MAX\_VALUE*);  
  
 // initialise the distance of the starting point to 0  
 distance[s] = 0;  
  
 // create the heap and pass the vertex amount and the distance array along with the lookup table  
 HeapPrim primHeap = new HeapPrim(vertexAmount, distance, heapPosition);  
  
 // insert the starting point at the root of the heap  
 primHeap.insert(s);  
  
 System.*out*.println("\nInsert root node of the heap " + *toChar*(s));  
  
 while (!primHeap.isEmpty())  
 {  
 // get the smallest element of the heap which is the root  
 int vertexRemoved = primHeap.remove();  
  
 distance[vertexRemoved] = 0;  
  
 Node node = adj[vertexRemoved];  
  
 while (node != sentinel)  
 {  
 int vertex = node.vertex;  
 int weight = node.wgt;  
  
 // check if the current node's weight value is smaller than the weight stored in the distance array  
 if (weight < distance[vertex])  
 {  
 parent[vertex] = vertexRemoved;  
  
 System.*out*.println("\n///// ========== PARENT [" + *toChar*(parent[vertex]) + "] to adjacent vertex " + *toChar*(vertex) );  
  
 System.*out*.println("\n///// Traverse from origin [" + *toChar*(parent[vertex]) + "] to adjacent vertex " + *toChar*(vertex) + " with a distance of " + weight);  
  
 System.*out*.print("Add the smallest weighted edge \t\t sum + " + weight);  
  
 // increment the weight of mst with the smaller weight  
 weightSum += weight;  
  
 System.*out*.println(" and results in a weight of " + weightSum);  
  
 // remove the previous larger weight from the sum  
 if (distance[vertex] != Integer.*MAX\_VALUE*)  
 {  
 System.*out*.print("Remove the bigger weighted edge \t sum - " + distance[vertex]);  
 weightSum -= distance[vertex];  
 System.*out*.println(" and results in a weight of " + weightSum);  
 }  
 else  
 {  
 // swap if smaller  
 distance[vertex] = weight;  
 }  
  
 // check if the current vertex is inside the heap or else insert it  
 if (heapPosition[vertex] == 0)  
 {  
 primHeap.insert(vertex);  
 }  
 else  
 {  
 System.*out*.println("\n///// Sift up " + *toChar*(heapPosition[vertex]));  
  
 primHeap.siftUp(heapPosition[vertex]);  
 }  
 }  
  
 // move pointer to the next node  
 node = node.next;  
 }  
 }  
 System.*out*.println("\nWeight of MST = " + weightSum);  
  
 // make the mst array point to the parent array so it can be used with printing  
 mst = parent;  
 }  
  
  
 private void showMST()  
 {  
 System.*out*.println("\nMinimum Spanning tree parent array is:");  
 for (int vertex = 1; vertex <= vertexAmount; vertex++)  
 {  
 if (*toChar*(mst[vertex]) == '@')  
 {  
 System.*out*.println(*toChar*(vertex));  
 }  
 else  
 {  
 System.*out*.println(*toChar*(vertex) + " -> " + *toChar*(mst[vertex]));  
 }  
 }  
 System.*out*.println();  
 }  
  
 public static void main(String[] args) throws IOException  
 {  
 int starVert;  
  
 String fileName1 = "myGraph.txt";  
 AdjacencyList myList = new AdjacencyList(fileName1);  
  
 BFS bfs = new BFS(myList.getAdjacencyList(), myList.getVertexAmount());  
 DFS dfs = new DFS(myList.getAdjacencyList(), myList.getVertexAmount());  
  
 System.*out*.println("\n///// Adjacency List /////\n");  
 // print the adjacency list  
 myList.printAdj();  
  
 System.*out*.println();  
 System.*out*.println("||| ================================================================ |||");  
 System.*out*.println();  
  
 System.*out*.println("///// Breadth-First Search /////\n");  
  
 try  
 {  
 bfs.breadthFirstSearch(3);  
 } catch (MyExceptions myExceptions)  
 {  
 myExceptions.printStackTrace();  
 }  
  
 System.*out*.println();  
 System.*out*.println("||| ================================================================ |||");  
 System.*out*.println();  
 System.*out*.println("///// Depth-First Search /////\n");  
  
 dfs.dfsRecursive(3);  
  
 Scanner textScanner = new Scanner(System.*in*);  
 String fileName2 = "myGraph.txt";  
 System.*out*.print("\nEnter starting vertex: ");  
 starVert = textScanner.nextInt();  
  
 PrimGraph graph = new PrimGraph(fileName2);  
  
 graph.display();  
 graph.MSTPrim(starVert);  
 graph.showMST();  
 }  
}

## Parent & Distance Arrays

@ stands for element pointing to itself and infinity is set as an initial distance. **C** is the starting vertex and so it does not have any parents.

## Graph Traversal

### First Step

### Second Step

### Third Step

### Fourth Step

### Fifth Step

### Sixth Step

### Seventh Step

### Eight Step

### Ninth Step

### Tenth Step

### Eleventh Step

### Twelfth Step

### Complete Graph

#### Terminal Output

Enter starting vertex: 3

Vertices = 9 Edges = 14

Vertex[1] -- Vertex[2] (weight)

Reading edges from text file

A--B(4)

A--H(8)

B--C(8)

B--H(11)

C--D(7)

C--F(4)

C--I(2)

D--E(9)

D--F(14)

E--F(10)

F--G(2)

G--H(1)

G--I(6)

H--I(7)

adj[A] -> |H | 8| -> |B | 4|

adj[B] -> |H | 11| -> |C | 8| -> |A | 4|

adj[C] -> |I | 2| -> |F | 4| -> |D | 7| -> |B | 8|

adj[D] -> |F | 14| -> |E | 9| -> |C | 7|

adj[E] -> |F | 10| -> |D | 9|

adj[F] -> |G | 2| -> |E | 10| -> |D | 14| -> |C | 4|

adj[G] -> |I | 6| -> |H | 1| -> |F | 2|

adj[H] -> |I | 7| -> |G | 1| -> |B | 11| -> |A | 8|

adj[I] -> |H | 7| -> |G | 6| -> |C | 2|

Insert root node of the heap C

///// ========== PARENT [C] to adjacent vertex I

///// Traverse from origin [C] to adjacent vertex I with a distance of 2

Add the smallest weighted edge sum + 2 and results in a weight of 2

///// ========== PARENT [C] to adjacent vertex F

///// Traverse from origin [C] to adjacent vertex F with a distance of 4

Add the smallest weighted edge sum + 4 and results in a weight of 6

///// ========== PARENT [C] to adjacent vertex D

///// Traverse from origin [C] to adjacent vertex D with a distance of 7

Add the smallest weighted edge sum + 7 and results in a weight of 13

///// ========== PARENT [C] to adjacent vertex B

///// Traverse from origin [C] to adjacent vertex B with a distance of 8

Add the smallest weighted edge sum + 8 and results in a weight of 21

///// ========== PARENT [I] to adjacent vertex H

///// Traverse from origin [I] to adjacent vertex H with a distance of 7

Add the smallest weighted edge sum + 7 and results in a weight of 28

///// ========== PARENT [I] to adjacent vertex G

///// Traverse from origin [I] to adjacent vertex G with a distance of 6

Add the smallest weighted edge sum + 6 and results in a weight of 34

///// ========== PARENT [F] to adjacent vertex G

///// Traverse from origin [F] to adjacent vertex G with a distance of 2

Add the smallest weighted edge sum + 2 and results in a weight of 36

Remove the bigger weighted edge sum - 6 and results in a weight of 30

///// Sift up B

///// ========== PARENT [F] to adjacent vertex E

///// Traverse from origin [F] to adjacent vertex E with a distance of 10

Add the smallest weighted edge sum + 10 and results in a weight of 40

///// ========== PARENT [G] to adjacent vertex H

///// Traverse from origin [G] to adjacent vertex H with a distance of 1

Add the smallest weighted edge sum + 1 and results in a weight of 41

Remove the bigger weighted edge sum - 7 and results in a weight of 34

///// Sift up B

///// ========== PARENT [H] to adjacent vertex A

///// Traverse from origin [H] to adjacent vertex A with a distance of 8

Add the smallest weighted edge sum + 8 and results in a weight of 42

///// ========== PARENT [D] to adjacent vertex E

///// Traverse from origin [D] to adjacent vertex E with a distance of 9

Add the smallest weighted edge sum + 9 and results in a weight of 51

Remove the bigger weighted edge sum - 10 and results in a weight of 41

///// Sift up C

///// ========== PARENT [A] to adjacent vertex B

///// Traverse from origin [A] to adjacent vertex B with a distance of 4

Add the smallest weighted edge sum + 4 and results in a weight of 45

Remove the bigger weighted edge sum - 8 and results in a weight of 37

///// Sift up B

Weight of MST = 37

Minimum Spanning tree parent array is:

A -> H

B -> A

C

D -> C

E -> D

F -> C

G -> F

H -> G

I -> C

# Kruskal

## Definition

Kruskal’s algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge of least weight. Kruskal’s algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge of least possible weight. Our implementation of Kruskal’s algorithm is like the algorithm to compute connected components from the following diagrams. It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest. The operation FIND-SET returns a representative element from the set that contains a vertex. Thus, we can determine whether two vertices origin and destination belong to the same tree by testing whether FIND-SET A equals FIND-SET B. To combine trees, Kruskal’s algorithm calls the UNION procedure.

## Efficiency

The running time of Kruskal’s algorithm for a graph G = (V, E) depends on how we implement the disjoint-set data structure. We assume that we use the disjoint-set-forest implementation with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known. Initializing the set A in line 1 takes O(1) time, and the time to sort the edges is O(E lg E). (We will account for the cost of the |V| MAKE-SET operations in the for loop. The for loop later performs O(E) FIND-SET and UNION operations on the disjoint-set forest. Along with the |V| MAKE-SET operations, these take a total of O((V + E) alpha (V)) time, where alpha is the very slowly growing function. Because we assume that G is connected, we have |E| >= |V| - 1, and so the disjoint-set operations take O(E set V)time. Moreover, since alpha(|V|) = O(lg V) = O(lg E) the total running time of Kruskal’s algorithm is O(E lg E). Observing that |E| < |V ^ 2| , we have lg |E| = O(lg V) and so we can restate the running time of Kruskal’s algorithm as O(E lg V).

## Path Compression

This a way of flattening the structure of the tree whenever findSet() is used on it. The idea is that each node visited on the way to a root node may as well be attached directly to the root node; they all share the same representative. To effect this, as findSet() recursively traverses up the tree, it changes each node's parent reference to point to the root that it found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them.

int findSet(int vertex)  
{  
 int root = vertex;  
 while (root != treeParent[root])  
 {  
 root = treeParent[root];  
 }  
  
 /\*  
 this operation is called path compression, compress the path leading back to  
 the root which gives amortised time complexity  
 \*/  
 while (vertex != root)  
 {  
 int newRoot = treeParent[vertex];  
 treeParent[vertex] = root;  
 vertex = newRoot;  
 }  
  
 return root;  
}

## UnionFindSets

class UnionFindSets extends CommonFunctionsK  
{  
 private int[] treeParent;  
 private int heapSize;  
  
 // constructor which makes all the vertices point to themselves first  
 UnionFindSets(int vertexAmount)  
 {  
 heapSize = vertexAmount;  
 treeParent = new int[vertexAmount + 1];  
  
 for (int i = 1; i <= heapSize; i++)  
 {  
 treeParent[i] = i;  
 }  
 }  
  
 */\*\**  
 *\* Determine which subset a particular element is in. It returns the root element of it's cluster.*  
 *\* It can be determined whether two elements are in the same subset by comparing*  
 *\* the result of two findSet operations.*  
 *\**  
 *\** ***@param*** *vertex self explanatory*  
 *\** ***@return*** *root node*  
 *\*/*  
int findSet(int vertex)  
 {  
 int root = vertex;  
 while (root != treeParent[root])  
 {  
 root = treeParent[root];  
 }  
  
 /\*  
 this operation is called path compression, compress the path leading back to  
 the root which gives amortised time complexity  
 \*/  
 while (vertex != root)  
 {  
 int newRoot = treeParent[vertex];  
 treeParent[vertex] = root;  
 vertex = newRoot;  
 }  
  
 return root;  
 }  
  
 /\*  
 when we get two sets, we take the parent tree, we get the root vertex of the first set and we attach it to  
 the other set at the specified root node  
 \*/  
 void union(int set1, int set2)  
 {  
 int firstRoot = findSet(set1);  
 int secondRoot = findSet(set2);  
  
 // merge the two sets together  
 treeParent[secondRoot] = firstRoot;  
 }  
  
 // display function to show what each vertex is attached to  
 void showTrees()  
 {  
 for (int i = 1; i <= heapSize; ++i)  
 {  
 System.*out*.print(*toChar*(treeParent[i]) + "<-" + *toChar*(i) + "\t");  
 }  
 System.*out*.println();  
 }  
  
 */\*\**  
 *\* Similar to Prim's visited edge function.*  
 *\*/*  
void showSets()  
 {  
 int u, root;  
 boolean[] visited = new boolean[heapSize + 1];  
 for (u = 1; u <= heapSize; ++u)  
 {  
 root = findSet(u);  
 if (!visited[root])  
 {  
 showSet(root);  
 visited[root] = true;  
 }  
 }  
 System.*out*.println();  
 }  
  
 // display the elements of each set  
 private void showSet(int root)  
 {  
 System.*out*.print("Set{");  
 for (int vertex = 1; vertex <= heapSize; vertex++)  
 {  
 if (findSet(vertex) == root)  
 {  
 System.*out*.print(*toChar*(vertex) + " ");  
 }  
 }  
 System.*out*.print("} ");  
 }  
}

## KruskalGraph

class KruskalGraph extends CommonFunctionsK  
{  
 private int vertexAmount, edgeAmount;  
 private Edge[] edgeArray;  
 private Edge[] mst;  
 private int totalWeight;  
  
 private KruskalGraph(String graphFile) throws IOException  
 {  
 FileReader fr = new FileReader(graphFile);  
 BufferedReader reader = new BufferedReader(fr);  
  
 String splits = "\\s+";  
 String line = reader.readLine();  
 String[] parts = line.split(splits);  
  
 vertexAmount = Integer.*parseInt*(parts[0]);  
 edgeAmount = Integer.*parseInt*(parts[1]);  
 edgeArray = new Edge[edgeAmount + 1];  
  
 System.*out*.println("Vertices = " + parts[0] + " Edges = " + parts[1]);  
 System.*out*.println("Vertex[1] -- Vertex[2] (weight)");  
 System.*out*.println("Reading edges from text file");  
  
 for (int edge = 1; edge <= edgeAmount; ++edge)  
 {  
 line = reader.readLine();  
 parts = line.split(splits);  
 int origin = Integer.*parseInt*(parts[0]);  
 int destination = Integer.*parseInt*(parts[1]);  
 int weight = Integer.*parseInt*(parts[2]);  
  
 System.*out*.println(*toChar*(origin) + "--" + *toChar*(destination) + "(" + weight + ")");  
  
 this.edgeArray[edge] = new Edge(origin, destination, weight);  
 }  
  
 System.*out*.println();  
 }  
  
 private void MSTKruskal()  
 {  
 int indexOfSmallestEdge, vertex = 0;  
 int firstSet, secondSet;  
 Edge edge;  
 UnionFindSets partition;  
  
 totalWeight = 0;  
  
 mst = new Edge[vertexAmount - 1];  
 HeapKruskal heapKruskal = new HeapKruskal(edgeAmount, edgeArray);  
 partition = new UnionFindSets(vertexAmount);  
  
 partition.showSets();  
  
 while (vertex < vertexAmount - 1)  
 {  
 indexOfSmallestEdge = heapKruskal.remove();  
 edge = edgeArray[indexOfSmallestEdge];  
  
 firstSet = partition.findSet(edge.origin);  
 secondSet = partition.findSet(edge.destination);  
  
 // check if the sets will create a circle  
 if (firstSet != secondSet)  
 {  
 mst[vertex++] = edge;  
  
 System.*out*.print("\n///// Edge " + vertex + ": ");  
 edge.show();  
 System.*out*.println();  
  
 // create the union between the two sets using these partitions  
 partition.union(firstSet, secondSet);  
 partition.showSets();  
 partition.showTrees();  
  
 totalWeight += edge.weight;  
 }  
 else  
 {  
 System.*out*.print("\nIgnoring the edge\t");  
 edge.show();  
 System.*out*.println();  
 }  
 }  
 }  
  
 // display function  
 private void showMST()  
 {  
 System.*out*.print("\nMinimum spanning tree build from following edges:\n");  
 for (int edge = 0; edge < vertexAmount - 1; edge++)  
 {  
 mst[edge].show();  
 System.*out*.println();  
 }  
 System.*out*.println("\nWeight of the MST = " + totalWeight);  
 }  
  
 public static void main(String[] args) throws IOException  
 {  
 String fname2 = "myGraph.txt";  
  
 KruskalGraph graph = new KruskalGraph(fname2);  
  
 graph.MSTKruskal();  
 graph.showMST();  
 }  
}

## Set Formation

I used the output of the terminal to represent the set formation.

#### Terminal Output

Set{A } Set{B } Set{C } Set{D } Set{E } Set{F } Set{G } Set{H } Set{I }

///// Edge 1: G--H(1)

Set{A } Set{B } Set{C } Set{D } Set{E } Set{F } Set{G H } Set{I }

A<-A B<-B C<-C D<-D E<-E F<-F G<-G G<-H I<-I

///// Edge 2: F--G(2)

Set{A } Set{B } Set{C } Set{D } Set{E } Set{F G H } Set{I }

A<-A B<-B C<-C D<-D E<-E F<-F F<-G F<-H I<-I

///// Edge 3: C--I(2)

Set{A } Set{B } Set{C I } Set{D } Set{E } Set{F G H }

A<-A B<-B C<-C D<-D E<-E F<-F F<-G F<-H C<-I

///// Edge 4: C--F(4)

Set{A } Set{B } Set{C F G H I } Set{D } Set{E }

A<-A B<-B C<-C D<-D E<-E C<-F C<-G C<-H C<-I

///// Edge 5: A--B(4)

Set{A B } Set{C F G H I } Set{D } Set{E }

A<-A A<-B C<-C D<-D E<-E C<-F C<-G C<-H C<-I

Ignoring the edge G--I(6)

Ignoring the edge H--I(7)

///// Edge 6: C--D(7)

Set{A B } Set{C D F G H I } Set{E }

A<-A A<-B C<-C C<-D E<-E C<-F C<-G C<-H C<-I

///// Edge 7: B--C(8)

Set{A B C D F G H I } Set{E }

A<-A A<-B A<-C A<-D E<-E A<-F A<-G A<-H A<-I

Ignoring the edge A--H(8)

///// Edge 8: D--E(9)

Set{A B C D E F G H I }

A<-A A<-B A<-C A<-D A<-E A<-F A<-G A<-H A<-I

Minimum spanning tree build from following edges:

G--H(1)

F--G(2)

C--I(2)

C--F(4)

A--B(4)

C--D(7)

B--C(8)

D--E(9)

## Union Formation

Kruskal’s algorithm creates a set for each vertex and then we merge these sets together until every vertex hangs from one root.

### Initial Union

### First Union

### Second Union

### Third Union

### Fourth Union

### Fifth Union

### Sixth Union

### Seventh Union

### Eighth Union

## Graph Traversal

### First Move

### Second Move

### Third Move

### Fourth Move

### Fifth Move

### Sixth Move

### Seventh Move

### Eight Move

### Ninth Move

### Tenth Move

### Eleventh Move

### Twelfth Move

### Thirteenth Move

### Fourteenth Move

### Complete Graph

# Reflection

Throughout this assignment I meticulously dived deeper into the graph theory in order to get a better grasp its principles. I used the lecture notes, along with YouTube videos and the Algorithms book by Cormen et al. in order to boost my knowledge on the graphs and get an explanation from different perspectives.

Everything in regards to this assignment was useful especially because graphs are one of the most common questions asked in the interview. I can say with confidence that in the case I get a question on Prim or Kruskal, I will be able to answer it and excel while doing so. Moreover it has enhanced my perception of Java and improved my coding speed. I can visualize algorithms better now from writing them down on paper and making these diagrams. It helps me to see every step of the way.

At the beginning I had difficulties with integrating the heaps into the algorithms for both Prim’s and Kruskal’s but eventually I got the hang of it and I understood what the distance and parent array are doing and how they improve efficiency. The pseudo-code proved very useful and it was to transform it into Java code. I’ve also spent a great deal of time in making the print statements very explicit and it made it very easy to construct the diagrams based on those statements.

My only regret is that I did not get a chance to build the heap with the same implementation found in the Cormen et al. book and I could have made the code even better and simpler if I had the possibility of giving each class its own file. A lot of the methods and classes repeat themselves because they are common to both files.

This assignment is moreover a great addition to my portfolio on GitHub and it will contribute towards my final year project as I intend to create a software for teaching algorithms.