**Dijkstra Vs. Kruskal Minimum spanning tree algorithm**

**Part 01.**

**1. Motivation***What is the importance of this problem in the world?*Minimum spanning tree is extremely important for networking, and routing since the weight associated with an edge in a given graph, can represent time to travel the edge, risk, distance or really anything (it can be applied to a very diverse set of problems). So finding the most efficient algorithm or the algorithm that has the lowest cost edges helps applications that need a to find the minimum spanning tree run faster which is needed since hardware like cpu’s are getting faster but a faster algorithm will always be better and noticeable for graphs with a large number of vertices. Hardware innovations can never really keep up with a better algorithm since they can always be a large enough data set that would allow weaker hardware with a better algorithm to be faster than better hardware and a worse algorithm.

*What do we want to do?*We want to know when Kruskal’s minimum spanning tree algorithm and Dijkstra’s algorithm for shortest path from a single source, should be used instead of the other, in order to accomplish this we are going to design and implement both of these algorithms mentioned into c++, using assert statements and invariants to ensure correctness, and test them using randomly generated undirected connected graphs with edge density P and randomly generated weight for each edge. And gathering data each time for the average routing distance from a single source, and cost of the edges chosen by the algorithm, then compare the data from each algorithm to decide when to use each of them for a given application.

*Why?*Finding the data for each algorithm and comparing it to determine when to use either Kruskals’s algorithm or Dijkstra’s algorithm, will allow use to optimize applications that need a Minimum spanning tree, which is a very important problem to solve for routing and networking for example.

**2. Background***What is the historical perspective of algorithms, when and why developed?*The first ever algorithm was developed around 1700 BC by the Egyptians to multiply two numbers together. However the first instance of a minimum spanning tree algorithm, Boruvka’s algorithm, was developed in 1926 by Otakar Boruvka a Czech scientist, for the purpose of finding an efficient way to route electricity in Moravia. The rational for developing new algorithms is to solve new problems or solve old problems better or at least better in some aspect.

*Which algorithm is easier to understand, why?*Kruskal’s minimum spanning tree algorithm is much easier to understand since in order to understand the algorithm you must understand sorting which is very simple and intuitive, and understand loops in a graph which is also simple and intuitive. Whereas Dijkstra’s algorithm requires you to understands heaps and min heaps, which is complicated when compared to sorting and loops, and also a deep understanding of data structures and multiple data structures at the same time.

*How do they work?*Kruskal’s minimum spanning tree algorithm works by sorting all of edge values, then taking the lowest weighted edges if the edges do not form a loop, do this till you have collected V – 1 edges / the tree visit all vertices. Dijkstra’s algorithm works by making a min heap with each point in the min heap tied to a vertex, and then setting all values in the min heap to infinity, then pick the source vertice any vertice will do, and set its value in the heap to 0, then look at all the edges connected to the vertice and if the weight of that edge is lower than the value of the vertice in the heap than change it to weight of the edge, do this for all edges connected to the vertice. Remove the root value of the min heap, and then adjust it, now taking the new root node and look its edges, do this for all vertices/ all edges, and the result will be a heap / tree that gives a minimum spanning tree from the source vertice, to find the edges you must, while doing the algorithm above, keep track of the parent vertice for each vertices value, and with this information you can generate the minimum spanning tree.

**3. Procedures**

*Overall Structure of the Program*

*Classes with two MS Tree methods*

class KruskalCls  
{  
 public:

void setUnion(int arr[], int a, int b);

int find(int arr[], int x);

void HeapSort(int arr[], int n);

void MaxHeapify(int arr[], int pos, int n);

void MakeMaxHeap(int arr[], int n);

void KruskalMST(int matrix[][], int V);

}

class DijkstraCls

{

public:

void DijkstraMST(int matrix[][], inv V);

}

*Driver that uses these methods to get the minimum spanning tree*

int main()

{

const int NUM\_OF\_VERTICES = 20;

const int MAX\_EDGES = 20 \* 19 / 2;

const int MAX\_WEIGHT = 25;

KruskalCls Krsk;

DijkstraCls Djk;

int p;

int edges;

int GraphKruskal[NUM\_OF\_VERTICES][NUM\_OF\_VERTICES];

int GraphDijkstra[NUM\_OF\_VERTICES][NUM\_OF\_VERTICES];

int numOfEdgesLeft;

int tempRandom;

srand(time(NULL));

for(edges = 0; edges <= MAX\_EDGES – NUM\_OF\_VERTICES – 1; edges ++)

{

clearMatrix(GraphKruskal);

clearMatrix(GraphDijkstra);

numOfEdgesLeft = edges;

p = 2 \* edges / ( 20 \* 19 );

while(numOfEdgesLeft > 0)

{

for(int i = 0; i < NUM\_OF\_VERTICES; i++)

{

for(int j = i + 1; j < NUM\_OF\_VERTICES; j++)

{

if(j == i + 1)

{

tempRandom = rand() % MAX\_WEIGHT + 1;

GraphKruskal[i][j] = tempRandom;

GraphKruskal[j][i] = tempRandom;

GraphDijkstra[i][j] = tempRandom;

GraphDijkstra[j][i] = tempRandom;

}

else if(numOfEdgesLeft > 0)

{

tempRandom = rand() % 20;

if ( tempRandom == 5 )

{

numOfEdgesLeft --;

GraphKruskal[i][j] = tempRandom;

GraphKruskal[j][i] = tempRandom;

GraphDijkstra[i][j] = tempRandom;

GraphDijkstra[j][i] = tempRandom;

}

}

}

}

}

Krsk.KruskalMST(GraphKruskal, NUM\_OF\_VERTICES);

Djk.DijkstraMST(GraphKijkstra, NUM\_OF\_VERTICES);

}

return 0;

}

*Graphical display for visualizing the layout of classes/methods*

*Pseudocode with Correct Program Headers for usability*

void KruskalMST( int matrix[][], int V)

{

int edges = 0

int edgesInGraph = 0

int edgeRank[V\*(V-1)/2] = INT\_MAX

int parent[V]

int MST[V][V] = 0

for i = 0 up to V

parent[i] = i

for I and j up to V

if ( matrix[i][j] > 0)

edgeRank[edgesInGraph] = matrix[i][j]

edgesInGraph ++

HeapSort(edgeRank, (V\*(V-1)/2))

edgesInGraph = 0

while edges < V – 1

for I and j up to V

if(matrix[i][j] == edgeRank[edgesInGraph])

edgesInGraph++

if(find(i) != find(j))

edges ++

setUnion(parent,i,j)

MST[i][j] = matrix[i][j]

}

int find( int arr[], int x)

{

if(arr[x] != x)

x = arr[x]

return x

}

void setUnion(int arr[], int a, int b)

{

arr[find(a)] = find(b)

}

void HeapSort(int arr[], int n)

{

MakeMaxHeap(arr(1..n), n)

for i = n down to 2

swap arr(1) and arr(i)

MaxHeapify(arr(1..n), 1, i-1)

}

void MaxHeapify(int arr[], int pos, int n)

{

index = 2\* pos

if index > n

return

if index < n

if arr(index + 1) > arr(index)

index += 1

if arr(index) > arr(pos)

swap arr(index) and arr(pos)

MaxHeapify(arr(1..n), index, n)

}

void MakeMaxHeap(int arr[], int n)

{

for i = floor (n / 2) down to 1

MaxHeapify(arr(1..n), i, n)

}

void DijkstraMST( int matrix[][], int V)

{

int edges = 1

int W[V]

int parent[V]

int visited[V]

int MST[V][V] = 0

int sortHelp

int target

for i up to V

W[i] = INT\_MAX

visited[i] = 0

parent[i] = i

W[0] = 0

while(edges != V-1)

sortHelp = INT\_MAX

for i up to V

if( W[i] < sortHelp and !visited[i])

sortHelp = W[i]

target = i

visited[target] = 1

for i up to V

if(matrix[target][i] < W[i] and matrix[target][i] != 0)

W[i] = matrix[target][i]

parent[i] = target

edges ++;

for i up to V

if(i != parent[i])

MST[parent[i]][i] = W[i]

MST[i][parent[i]] = W[i]

}

*Pre/Post Conditions, Invariants in Pseudocode major loop invariants*

Precondition:

matrix(1..V)(1..V) is a graph

void KruskalMST( int matrix[][], int V)

{

int edges = 0

int edgesInGraph = 0

int edgeRank[V\*(V-1)/2] = INT\_MAX

int parent[V]

int MST[V][V] = 0

for i = 0 up to V

parent[i] = i

for I and j up to V

if ( matrix[i][j] > 0)

edgeRank[edgesInGraph]

edgesInGraph ++

HeapSort(edgeRank, (V\*(V-1)/2))

edgesInGraph = 0

while edges < V – 1

Invariant: edges < V- 1

for I and j up to V

if(matrix[i][j] == edgeRank[edgesInGraph])

edgesInGraph++

if(find(i) != find(j))

edges ++

setUnion(parent,i,j)

MST[i][j] = matrix[i][j]

Invariant: edges <= V - 1

}

Postcondition:

MST contains V – 1 edges

PreCondition:

arr(1..n) is an array of numbers

void HeapSort(int arr[], int n)

{

MakeMaxHeap(arr(1..n), n)

for i = n down to 2

Invariant: arr(1..i) is max heap and arr(i+1..n) is sorted

swap arr(1) and arr(i)

MaxHeapify(arr(1..n), 1, i-1)

Invariant: arr(1..i-1) is max heap, arr(i..n) is sorted

}

PostCondition:

arr(1..n) is sorted

PreCondition:

arr(pos+1…n) has the max heap property

void MaxHeapify(int arr[], int pos, int n)

{

Invariant: arr(pos + 1…n) has the max heap property

index = 2\* pos

if index > n

return

if index < n

if arr(index + 1) > arr(index)

index += 1

if arr(index) > arr(pos)

swap arr(index) and arr(pos)

MaxHeapify(arr(1..n), index, n)

Invariant: arr(pos…n) has the max heap property

}

PostCondition:

arr(pos…n) has the max heap property

PreCondition:

arr(1..n) is an array of numbers

void MakeMaxHeap(int arr[], int n)

{

for i = floor (n / 2) down to 1

Invariant: arr(i+1…n) has max heap property

MaxHeapify(arr(1..n), i, n)

Invariant: arr(i…n) has max heap property

}

PostCondition:

arr(1...n) has the max heap property

Precondition:

matrix(1..V)(1..V) is a graph

void DijkstraMST( int matrix[][], int V)

{

int edges = 1

int W[V]

int parent[V]

int visited[V]

int MST[V][V] = 0

int sortHelp

int target

for i up to V

W[i] = INT\_MAX

visited[i] = 0

parent[i] = i

W[0] = 0

while(edges != V-1)

Invariant: edges < V - 1

sortHelp = INT\_MAX

for i up to V

if( W[i] < sortHelp and !visited[i])

sortHelp = W[i]

target = i

visited[target] = 1

for i up to V

if(matrix[target][i] < W[i] and matrix[target][i] != 0)

W[i] = matrix[target][i]

parent[i] = target

edges ++;

Invariant: edges <= V - 1

for i up to V

if(i != parent[i])

MST[parent[i]][i] = W[i]

MST[i][parent[i]] = W[i]

}

Postcondition:

MST contains V – 1 edges

*Invariants as they will be implemented*

There are 4 types of invariants in the pseudocode

1. That some part of two arrays is sorted by value / weight

bool isSorted(arr(1..n), start, stop)

this function will itterate from start to stop checking to make sure that the next element in the array is >= the previous, if this is ever false the function returns false

assert(isSorted(arr(1..n),start,stop))

2. That some part of an array has the max heap property

bool isMaxHeap(arr(1..n), start, stop)

this function will itterate from start to stop checking that if an index has child nodes that those child nodes are <= the node being indexed, if this is ever false the function returns false

assert(isMaxHeap(arr(1..n), start, stop))

3. That an array is made of numbers

This does not need an assert statement since a c++ program will not compile if the functions that are expecting an array of integers get anything that is not promotable to an array of integers

4. That a graph contains so many edges

bool edgesMatch(matrix[][], V, edges)

this function will itterate through through all relivant indices and if the index has a value > 0 then an edgesCounted variable will be incremented, and after itterating through the graph the function will return edgesCounted == edges

assert(edgesMatch(matrix[][], V, edges))

**4. Testing Plan**

*Describe what kind of data will be tested?*

The cost of the edges in the resulting spanning tree and the average routing distance from one vertice, a source, to all other nodes

*Is it real data or manufactored synthetic test?*

It is a manufactured synthetic test since the data is not real and has no meaning since it is randomly generated, but the data will be a good representation of how real data will work with the algorithms. Although the graphs are not 100% random since the graphs are made to be connected.

*Boundary cases p= 0 E > maximum size*

With how my driver generates graphs p = 0 is simply not possible, since it tests from V-1 edges up to V\*(V-1) / 2 edges, the max edges, since the graphs are always connected and V-1 edges are needed for a graph to be connected. Edges > maximum size are not possible with my driver.

*What size of data will be tested? What data will be tested?*

The data being tested is edges V-1 up to V\*(V-1) / 2 edges with V = 20 and the corresponding p values, and random weights up to max of 25.

**Part 02.**

**5. Correctness of Program**

For this part of the assignment see the 2 submitted files

Proj3.cpp – The entire expected program with comments and assert statements

proj3Data.txt – Sample output which contains the data used to create the graphs

**6. Implementation of Invariants**

For this part of the assigment see Proj3.cpp for the implementation of Invariants

**7. List of Problems Encountered**

The biggest problem I encountered was taking Kruskal’s and Dijkstra’s algorithms and implementing them into C++, the end result works but it was definetly a sturggle to find how to represent all the data in relevant data structures.

Another problem was working out how to randomly generate the graphs for the data, since the only way to guarantee that a graph is connected is to have a super dense graph, so I ended up making preset edges that appear in every graph and randomly generating the weight of the edges on the preset edges and the randomly generated edges.

Implementing a way to calculate the average routing distance was a difficult challenge but ultimately the end result works great.

**8. Performance Comparisons**

**9. Conclussion**

After implementing and working with both algorithms for minimum spanning tree, Kruskal and Dijkstra, it is not so clear which is the better algorithm. Both algorithms always make minimum spanning trees for a given graph, and therefore have the same total cost of edges, but Dijkstra more often than not outpreformed in the average routing distance, but that is not to say that Kruskal never outpreformed Dijkstra in this very same data point though this could be due to my implementation of Dijkstra’s or Kruskal’s algorithm not being correct. In the end Dijkstra’s algorithm was easier to implement for me and more often than not outpreformed Kruskal’s algorithm in average routing distance.

**10. References**

Karmakar Ankur, “Kruskal’s Algorithm (Simple Implementation for Adjacency Matrix)” *GeeksforGeeks*,

27 Mar. 2020, https://www.geeksforgeeks.org/kruskals-algorithm-simple-implementation-for-adjacency-matrix/

Sharma Sudhir, “C++ Program for Dijkstra’s shortest path algorithm?” *tutorialpoint*, 9 Aug. 2019,

https://www.tutorialspoint.com/cplusplus-program-for-dijkstra-s-shortest-path-algorithm