Minimum Spanning Trees

Ahmed Waseem Raslan
Ahmed Elbarbary

March 14, 2023

Methodology

The general approach was to implement both Kruskal's and Prim's algorithms for finding the Minimum Spanning Tree of a given Graph/ grid of Houses. We then profile both algorithms to find the faster approach. Both algorithms are available to choose from, along with random graph generation, to allow for the ease of profiling and testing purposes. Any output is saved to files specified by the user.

Approach:

We construct a simple graph class as follows:

graph
$TABLE[INT] adjacency\ Matrix$ $TABLE[INT] Minimum\ spanning\ tree$ $HEAP[EDGE, ARRAY[EDGE]]$ $POINTER\ INT\ parent$ $INT\ number\ of\ verticies$
methods
$unionSet(INT\ u, v)$ $INT\ findSet(INT\ u)$ $kruskal()$ $prim()$

with EDGE being a tuple of
$INT\ source\ node, INT\ destination\ node, INT\ edge\ weight$

Methods description

$unionSet(INT\ u, v)$:

Union function used for the union of the sets of vertices to be able to detect cycles when choosing a minimum spanning edge.[**?**]

$INT\ findSet(INT\ u)$:
Find function used to find a vertix in a set and return its parent.[**?**]

kruskal() :
Kruskal's Algorithim for finding the minimum spanning tree.[**?**]

prim() :
Prim's Algorithim for finding the minimum spanning tree.[**?**]

We also have custom utility functions as follows:

utilities
PAIR[INT, ARRAY[EDGE]] *parseFile*(File Path) PAIR[ADJACENCY MATRIX, ARRAY[EDGE]] *generateRandomEdges*(INT N, INT Wmin, INT Wmax) PAIR[FLOAT, FLOAT]*getExecutionTimes*(GRAPH g) *profileAlgorithims*(INT Nstart, INT Nend, INT Wmin, INT Wmax)

Method Description
PAIR[INT, ARRAY[EDGE]] *parseFile*(File Path):
Parses single text files containing a graph given its file path. returns both vertix number of graph and its array of edges

PAIR[ADJACENCY MATRIX, ARRAY[EDGE]] *generateRandomEdges*(INT N, INT Wmin, INT Wmax):
Generates a random graph given its number of vertices N, minimum edge Weight $Wmin$ and maximum edge weight $Wmax$.

PAIR[FLOAT, FLOAT]*getExecutionTimes*(GRAPH g):
Given a graph, it calls its *kruskal*() and *prim*() algorithms to compute their running times and returns them in a pair.

profileAlgorithims(INT Nstart, INT Nend, INT Wmin, INT Wmax):
Iteratively calls *getExecutionTimes* to retrieve profiling times of Kruskal and Prim's algorithms for $Nstart$ vertices to $Nend$ vertices. $Wmin$ and $Wmax$ parameters are given as the function calls *generateRandomEdges*

Driver code is responsible for navigation, writing generated graphs to files specified by the user, and using appropriate implementation for user choices.

Algorithim Specifications

Kruskal

At first, all Edges are represented as disjointed sets (by using the member attribute of graph *parent*).

Non-Zero weighted edges are then put into a min-heap; in our case, every graph is initialized with a minimum heap of non-zero weighted edges using priority queues in C++.

Using the minimum heap, the first minimum edge is removed, and the heap is rebuilt. Using *FindSet*() and *union*() functions, we detect if that edge makes a cycle in our minimum spanning tree. We discard edges that make a cycle and only include the minimum edges that do not complete a cycle.

Time Complexity

Building Heap of edges: $\mathcal{O}(ELog_2E)$ where E is the number of Edges in the graph.
Removing front element of heap: $\mathcal{O}(Log_2E)$
Find function: $\mathcal{O}(Log_2V)$ where V is the number of vertices in a graph.

Pseudo code:

```
\\heap built with initialization of graph O(ELogE)
kruskal()
{
    arr[Edge];
    while(i < V - 1) AND heap is not empty)
    {
        Edge = heap.pop() \\O(LogE) * (V - 1)
        i = findSet(Edge.source); \\O(LogV) * (V - 1)
        j = findSet(Edge.destination); \\O(LogV) * (V - 1)
        if(i != j)
        {
            arr[i] = Edge;
            unionSet(i, j);
        }
    }
```

From the figure above, we can deduce that the complexity is of $\mathcal{O}(ELog_2V)$

Prim

Prim's algorithm incorporates the classical greedy method approach. It begins at a vertex and traverses the graph's minimum edges. It skips a vertex if it has been visited before. Each minimum edge is then put into an edge array.

Pseudo Code:

```
prim()
{
    arr[Edge];
    while(number of Edges < V -1)
    {
        min = infinity;
        x = 0;
        y = 0;
        for(i = 0 to V)
        {
            if(selected[i])
            {
                for(j = 0 to V)
                {
                    if(!selected[j] AND adjacencyMatrix[i][j])
                    {
                        if(min > adjacencyMatrix[i][j])
                        {
                            min = adjacencyMatrix[i][j];
                            x = i;
                            y = j;
                        }
                    }
                }
            }
        }
        selected[y] = true;
        Number of Edges++;
        Edge e = {x, y, adjacencyMatrix[i][j]};
        arr[Edge].push_back(e);
    }
```

We can see that the algorithim above is $\mathcal{O}(V^2)$

Data specifications

Input

Single-input graph files are stored as .txt as the number of vertices of the graph followed by adjacency matrix values separated by spaces, i.e.,

7 0 15 30 0 20 0 0 15 0 0 40 0 0 0 30 0 0 35 10 0 0 0 40 35 0 0 11 0 20 0 10 0 0 50
75 0 0 0 11 50 0 16 0 0 0 0 75 16 0

The graph above is of 7 vertices, and is represented as follows:

0	15	30	0	20	0	0
15	0	0	40	0	0	0
30	0	0	35	10	0	0
0	40	35	0	0	11	0
20	0	10	0	0	50	75
0	0	0	11	50	0	16
0	0	0	0	75	16	0

Output

Program output can be divided into the following:

- MST analysis of a given graph file.

  The program outputs the analysis onto the terminal.

- Generation of a random graph. The program outputs the resulting graph into a file specified by user.

- Profiling of both algorithms. The program outputs the resulting profile onto a text file that is specified by the user.

Experimental results

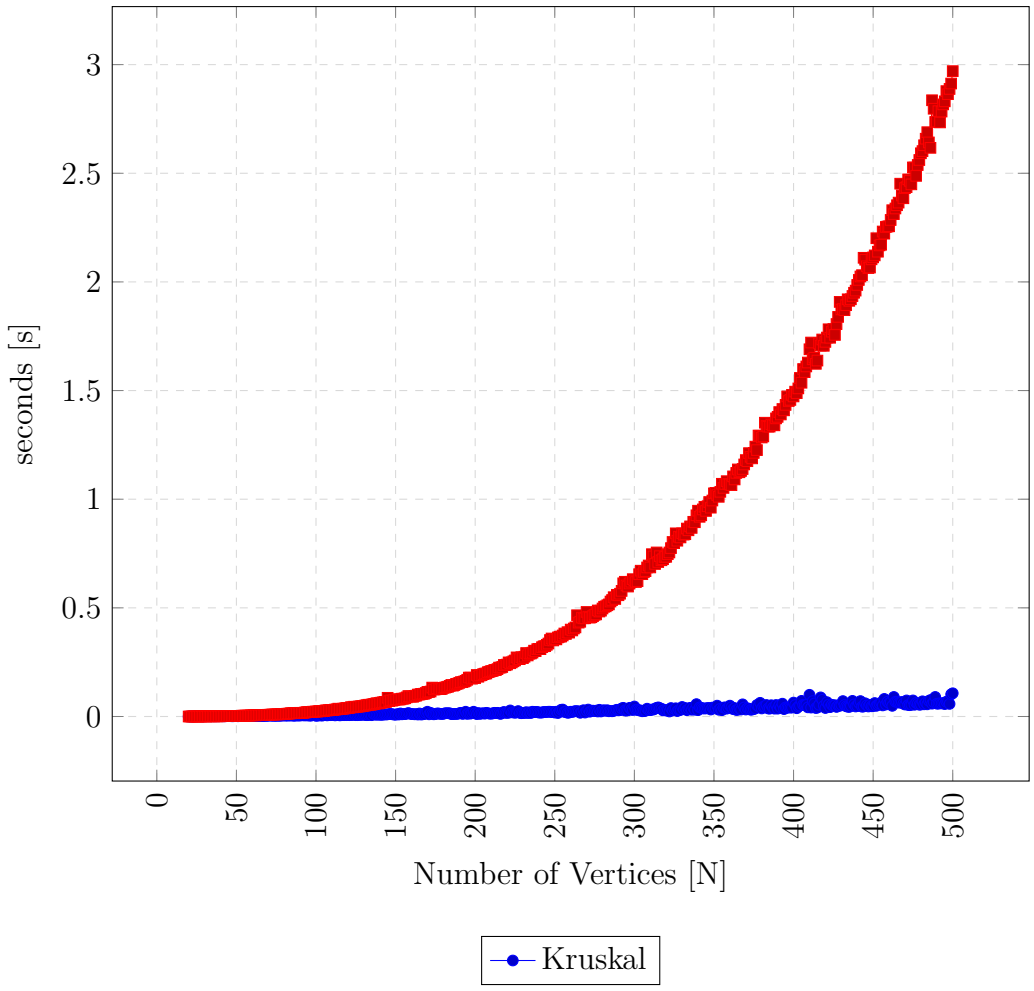

Figure 1: Kruskal's time profiling against Prim's time profiling