Prim's MST Algorithm

In computer science, Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is also sometimes called the DJP algorithm, the Jarník algorithm, or the Prim–Jarník algorithm.

The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all vertices.

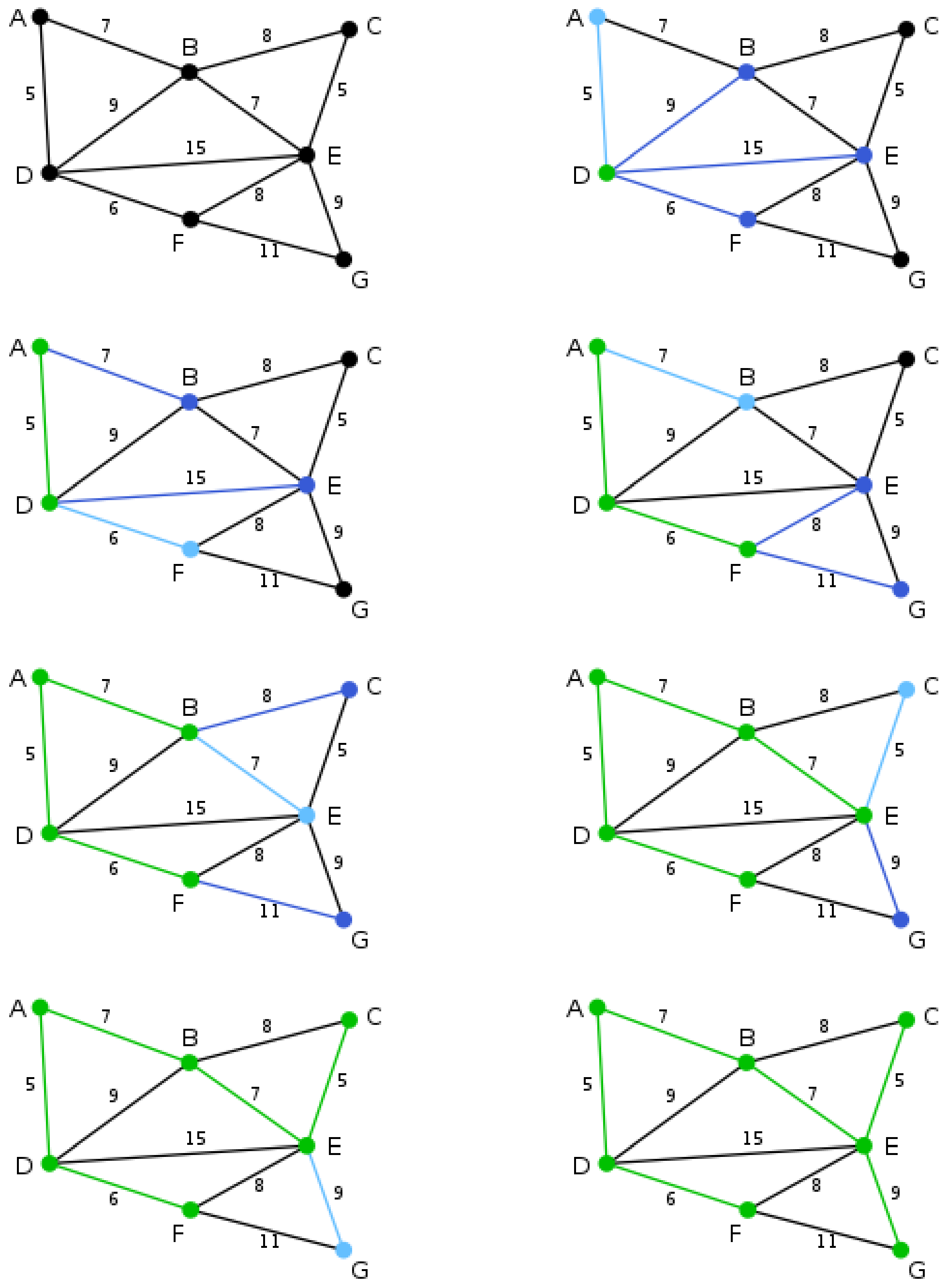
- **Input**: A non-empty connected weighted graph with vertices V and edges E.
- **Initialise**: $V_{new} = \{x\}$, where x is an arbitrary node (starting point) from V, $E_{new} = \{\}$
- **Repeat** until $V_{new} = V$:
    - Choose an edge (u, v) with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, any of them may be picked)
    - Add v to V_{new}, and (u, v) to E_{new}
- **Output**: V_{new} and E_{new} describe a minimal spanning tree

Validity of this algorithm is based on the property (which can be proved):

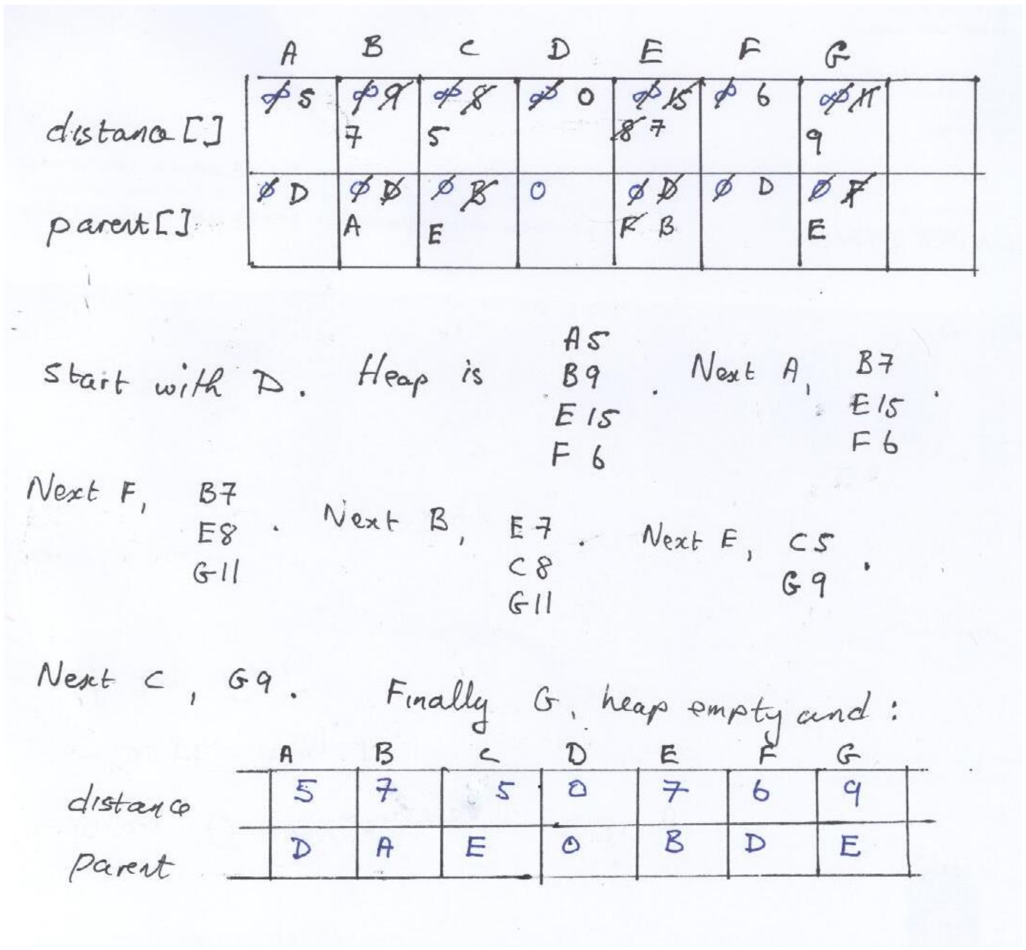
MST Property

Given any division of the vertices of a graph into two sets, the shortest edge which connects a vertex in one of the sets to a vertex in the other set is part of a MST.

Example 1



Heap, distance[] and parent[] array updates can be seen from what follows.



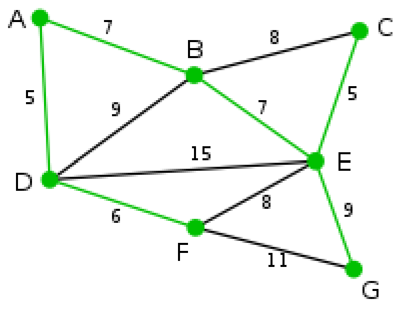
	A	B	C	D	E	F	G	
distance[]	~~∅~~ 5	~~∅~~ ~~9~~ 7	~~∅~~ ~~8~~ 5	~~∅~~ 0	~~∅~~ ~~15~~ ~~8~~ 7	~~∅~~ 6	~~∅~~ ~~11~~ 9	
parent[]	~~∅~~ D	~~∅~~ ~~D~~ A	~~∅~~ ~~B~~ E	0	~~∅~~ ~~D~~ ~~F~~ B	~~∅~~ D	~~∅~~ ~~F~~ E	

Start with D. Heap is A5 B9 E15 F6. Next A, B7 E15 F6.

Next F, B7 E8 G11. Next B, E7 C8 G11. Next E, C5 G9.

Next C, G9. Finally G, heap empty and:

	A	B	C	D	E	F	G	
distance	5	7	5	0	7	6	9	
parent	D	A	E	0	B	D	E	



Example 2 - shows more algorithm detail

From Sedgewick

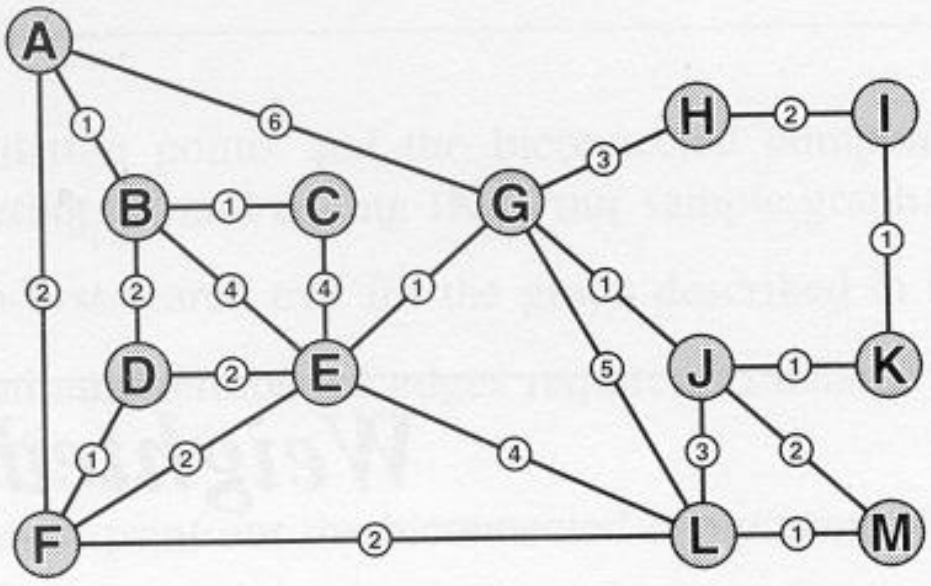


Figure 31.1 A weighted undirected graph.

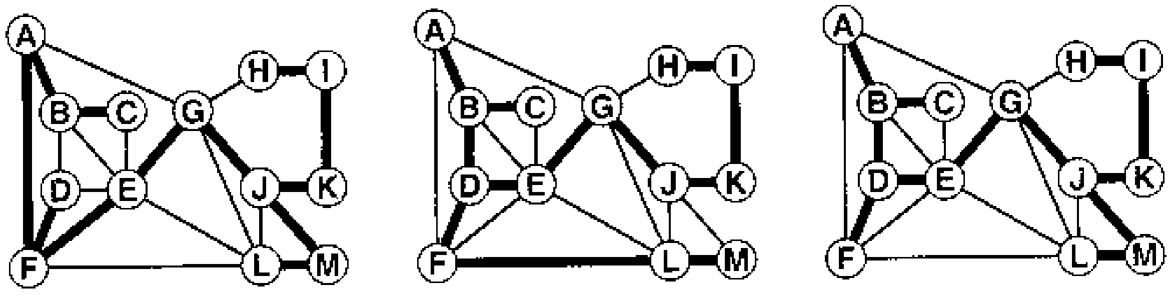


Figure 31.2 Minimum spanning trees.

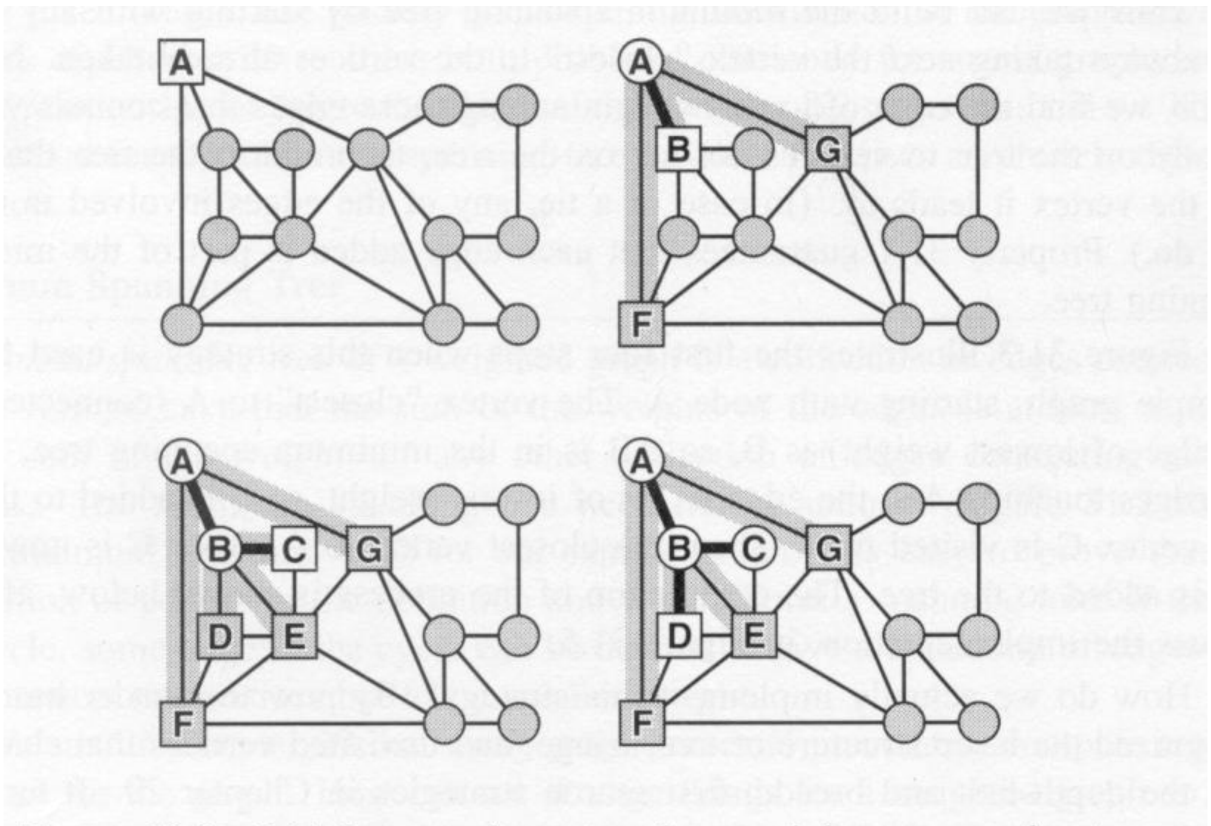


Figure 31.3 Initial steps of constructing a minimum spanning tree.

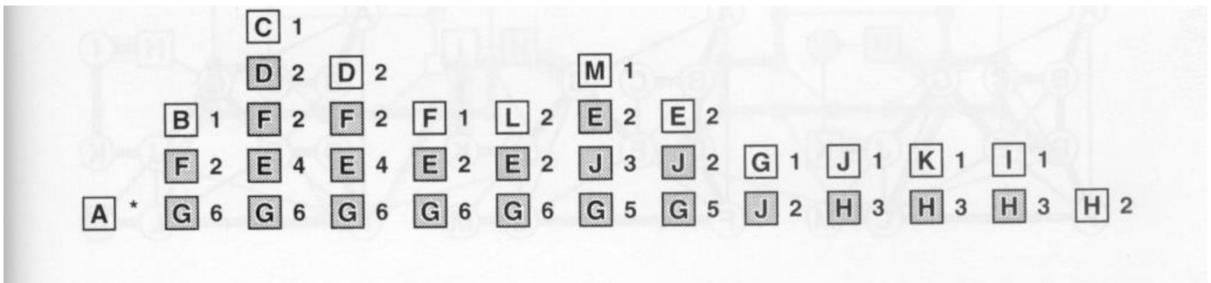


Figure 31.4 Contents of priority queue during minimum spanning tree construction.

Initial state of parent and distance arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
parent		0	0	0	0	0	0	0	0	0	0	0	0	0
distance		0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Changes to distance and parent arrays as algorithm progresses:

	0	A	B	C	D	E	F	G	H	I	J	K	L	M			
parent		0	A	B	B	~~B~~ D	~~A~~ D	~~A~~ E	~~G~~ I	K	~~L~~ ~~M~~ G	J	F	L			
distance		0	1	1	2	~~4~~ 2	~~2~~ 1	~~6~~ 1	~~3~~ 2	1	~~3~~ ~~2~~ 1	1	2	1			

A	B1 F2 G6
B	C1 D2 E4 F2 G6
C	D2 E4 F2 G6
D	F1 E2 G6
F	E2 L2 G6
L	M1 E2 G6 J3
M	E2 J2 G6
E	G1 J2
G	J1 H3
J	K1 H3
K	I1 H3
I	H2

Data Structures Required for Adjacency Lists representation

- a Java/C#/C++ class called **GraphLists** to encapsulate most of the data structures and methods
- a Node structure for lined lists. Use these to store graph edges in the linked lists
- array of linked lists to store the graph in memory. Declared with

```
Node[] adj;    (Node ** adj; //in C++)
```

- an int array *dist[ ]* to record the current distance of a vertex from the MST.
- an int array *parent[ ]* to store the MST
- a heap h to find the next vertex nearest the MST so that it can be added to the MST. The heap minimises the number of steps in finding the next nearest vertex. log V steps instead of V steps in sequestial search.
- an int array *hPos[ ]* which records the position of any vertex with the heap array a[ ]. So vertex *v* is in position *hPos[v]* in a*[]*.

Prim’s MST Algorithm on Adjacency Lists

```
Prim_Lists( Vertex s )
Begin
    // G = (V, E)
    foreach v ∈ V
        dist[v] := ∞
        parent[v] := 0                    // treat 0 as a special null vertex
        hPos[v] := 0                      // indicates that v ∉ heap

    h = new Heap(|V|, hPos, dist)         // priority queue (heap) initially empty
    h.insert(s)                           // s will be the root of the MST

    while (not h.isEmpty() )         // should repeat  |V|-1  times
        v := h.remove()              // add v to the MST
        dist[v] := -dist[v]          // marks v as now in the MST
        foreach u ∈ adj(v)           // examine each neighbour u of v
            if wgt(v, u) < dist[u]
                dist[u] := wgt(v, u)
                parent[u] := v
                if u ∉ h
                    h.insert( u)
                else
                    h.siftUp( hPos[u])

            end if
        end for
    end while
    return parent
End
```

Implementation Comments

- First of all, a simple way to represent the vertices is to number them as 1, 2, 3 … |V|.

- The most efficient way to represent the graph is to use and adjacency list, especially for sparse graphs. This is an array of linked lists. For dense ones an adjacency matrix would be fine.

- Also, a number of other data structures are used here

- The distance from a vertex u, outside of the current spanning tree, to some nearest vertex in the tree can be implemented as an array dist[u]. Initially all nodes are assumed to be an infinite distance away, ∞ (or INT_MAX in C++ or int.MaxValue in C#).

- AN array parent[u] keeps track of the parent vertex of u in the MST as it grows. In fact parent[ ] stores the MST.

- The heap vertex with the highest priority is the one closest to the MST being built, i.e. the vertex u where dist[u] is a minimum. This suggests a priority queue or heap to store vertices which are outside the current MST and are being considered for inclusion.

- When vertex u is stored on the heap, only the number that represents it is stored on the heap. Its priority is stored separately This cannot be used for positioning it within the heap. Instead use dist[u] to store its priority.

- Nodes will be positioned within the heap according to their distance from the MSP being built. So the heap data structure will need a pointer to dist[ ]. Then instead of have something like v > a[k/2], one would have dist[v] < dist[a[k/2]].

- If a is the heap array, then a[i] given the vertex at position i in the heap, i.e. a[] maps from a heap position or index to a vertex.

- When siftUp() is performed on a node u in the heap, its position in the heap will be required. We need to map from a vertex to a heap position, which is the opposite of what a[] does. Use a^{-1} for this. Both a and a^{-1} are be implemented as arrays. Call a^{-1}() hPos[] if you want.

  a : Index $\rightarrow$ Vertex
  hPos : Vertex $\rightarrow$ Index

siftUp() and siftDown() should take are of adjusting hPos[];

- On finding a shorter edge connecting a vertex u to the MST being built, its priority within the heap increases and so siftUp() would be called as in:

  siftUp(hPos [u])

- The heap will need to update both a[] and hPos[] and will need access to dist[].

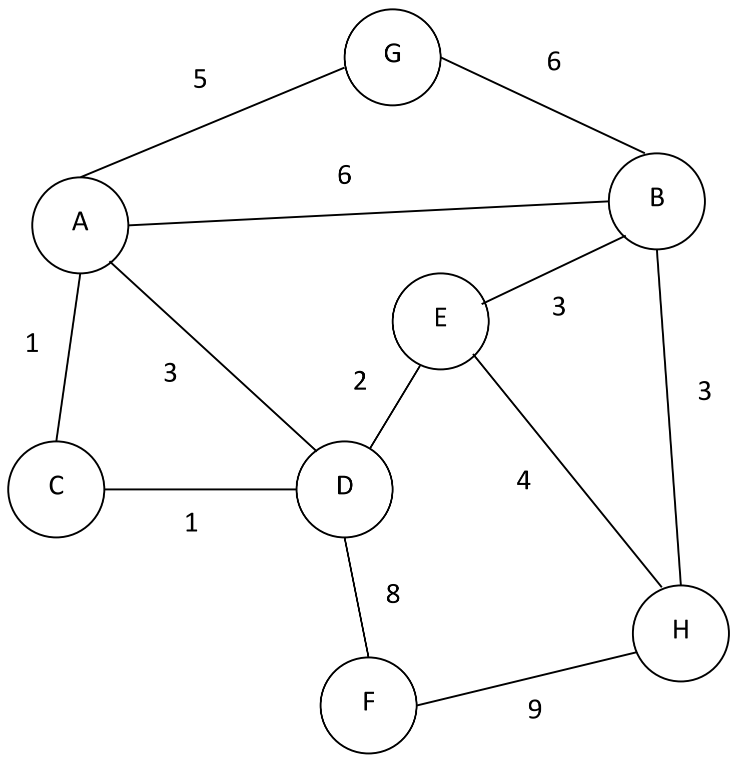
- A good way to give the heap access to the dist[] and hPos[] arrays is to pass them as parameter arguments to the heap constructor.

- In the main method could have

```
    Graph g = new Graph(fileName);
    mst = g.Prim( 3);
```

Example 3

(wgraph2c.txt)



A	C ->A	D ->C	E ->D	B ->E	H ->B	G ->A	F ->D
B 6	B 6	E 2	H 4	H 3	F 8	F 8	
C 1	D 1	F 8	F 8	F 8	G 5		
D 3	G 5	B 6	B 3	G 5			
G 5		G 5	G 5				

Prim’s Algorithm on an Adjacency Matrix

```
Prim_Matrix ( Vertex s )
    // G = (V, E)
    foreach v ∈ V
        dist[v] := ∞
        parent[v] := null
    v := s                                  // s will be the root of the MSP
    dist[null] := ∞

    while v ≠ null                          // should repeat  |V|-1  times
        dist[v] := -dist[v] // marks v as now in the MST
        min := null
        foreach u from 1 to |V|             // examine matrix row
            if u ∈ adj(v) ∧ wgt(v, u) < dist[u]
                dist[u] := wgt(v, u)
                parent[u] := v
            if dist[u] < dist[min] ∧ dist(u) > 0
                min := u
        v := min
    end while
    return parent
End
```

Time Complexity Analysis

Adjacency Lists Graph with Heap

Heap contains up to V vertices. So a heap operation requires a maximum of $\log_2 V$ steps.

Initialisation **for loop** – requires V steps

while loop

- each vertex is removed from heap once – so requires at most $V \log_2 V$ steps
- foreach loop within the while loop causes all the linked list nodes of the graph representation to be iterated thru once and since each list node representes an edge, the total number of iterations is E
- so each edge is examined once and may involve an *insert()* or a *siftUp()* . In total this means at most $E \log_2 V$ steps.

Therefore complexity = O($V + V \log_2 V + E \log_2 V$)

≈ O($E \log_2 V$)

For a **sparse** graph **E ≈ kV** (k some constant), so complexity ≈ O($E \log_2 V$) = $O(V \log_2 V)$

For a **dense** graph **E ≈ V²** , so complexity ≈ O($E \log_2 V$) = $O(V^2 \log_2 V)$

Adjacency Matrx Graph

In this representation no heap is used and the while loop + for loop combination iterates thru the whole matrix, i.e. **V^2** steps. So for both sparse and dense graphs in matrix represention:

Complexity = $O(V^2)$

Minimum edge weight data structure	**Time complexity (total)**
adjacency matrix for dense or sparse graph	O(**V^2**)
binary heap and adjacency lists for sparse graph	O(**V log V**)
binary heap and adjacency lists for dense graph (E ≈ V^2)	O(**V^2 log V**)
Fibonacci heap and adjacency list	O(**E + V log V**)

Since V^2 >> $V \log_2 V$ as V get larger, one would expect adjacency list performance to be much faster on sparse graphs.

Furthermore, adjacency lists use only the required memory for sparse graphs and are more efficient spacewisw for sparse graphs. Matrix representation is very inefficient both memory wise and in performance for sparse graphs.

Better to use an adjacency matrix for a dense graph, in which case which gives complexity $O(V^2)$ and adjacency lists for sparse graphs $O(V \log_2 V)$.