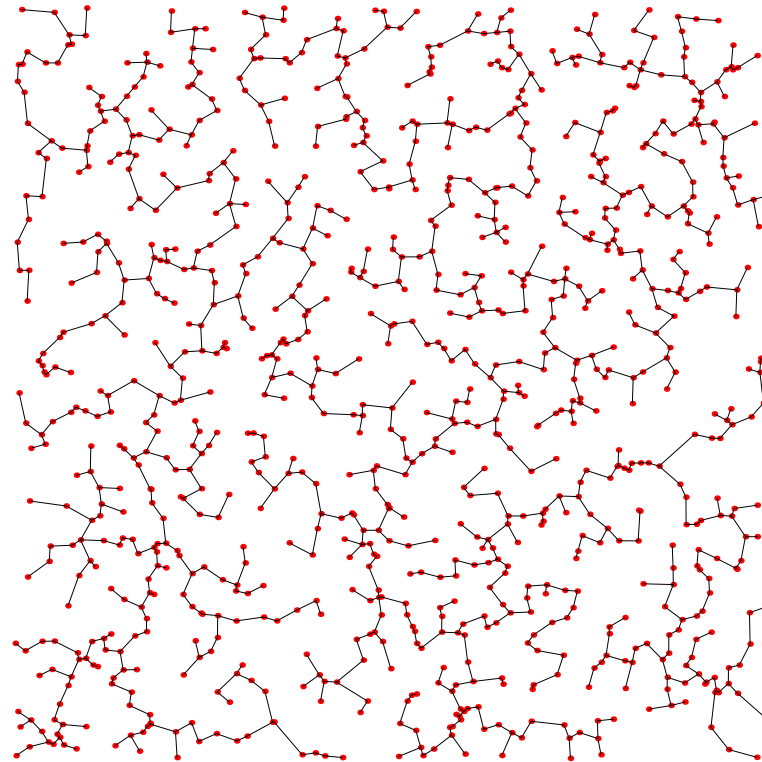


# Further Mathematics and Algorithms

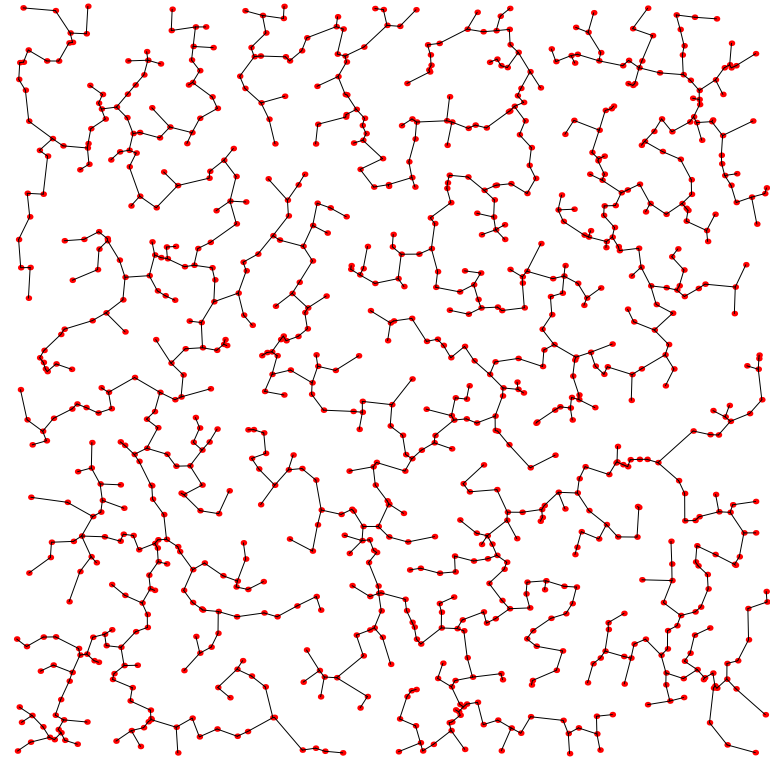
## Lesson 18: *Know Your Graph Algorithms*



*Weighted graph algorithms, Minimum spanning tree, Prim, Kruskal, shortest path, Dijkstra*

# Outline

1. **Minimum Spanning Tree**
2. Prim's Algorithm
3. Kruskal's Algorithm
4. Union Find
5. Shortest Path

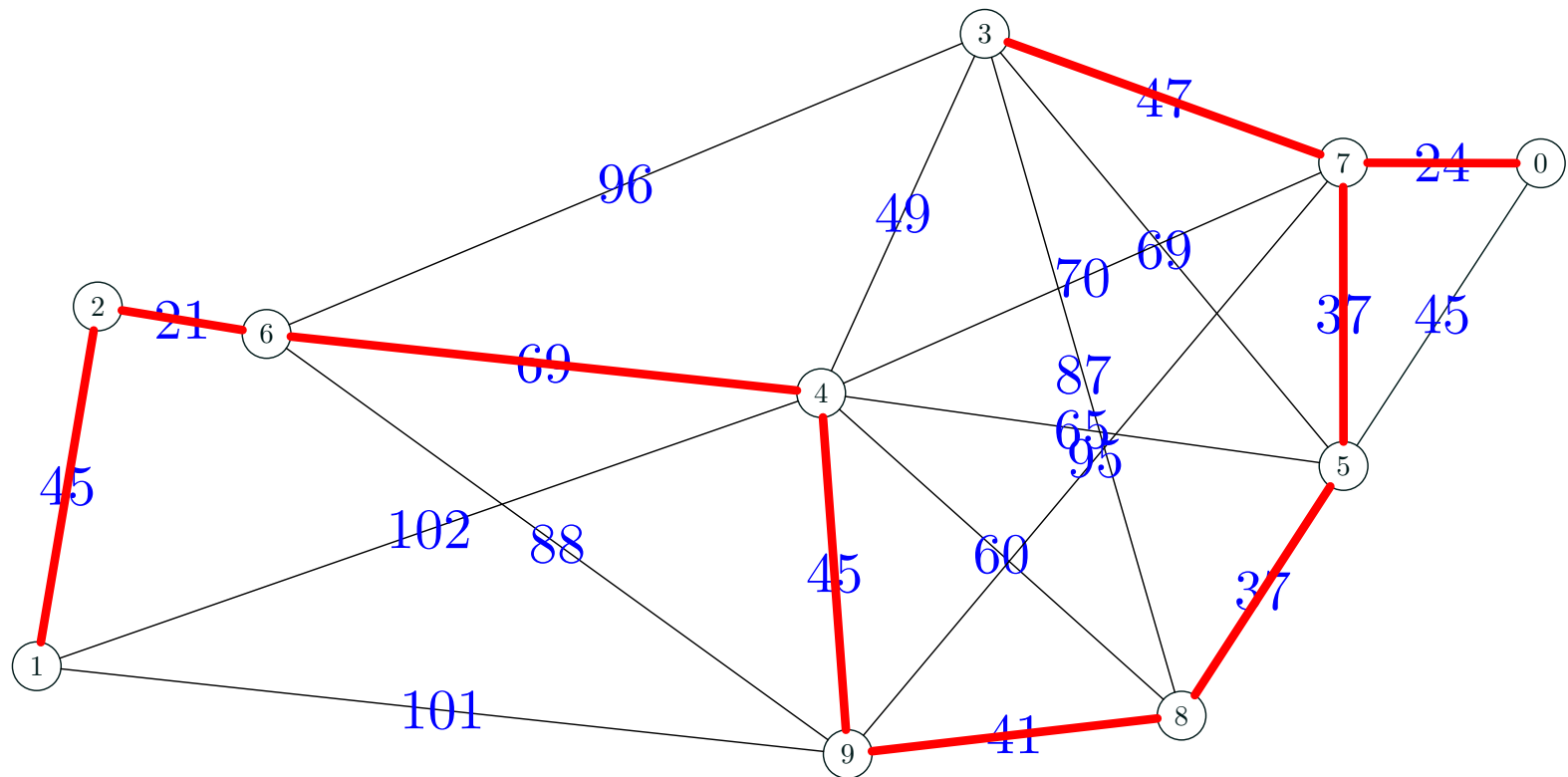


# Graph Algorithms

- We consider a graph algorithm to be **efficient** if it can solve a graph problem in  $O(n^a)$  time for some fixed  $a$ ■
- That is, an efficient algorithm runs in polynomial time■
- A problem is **hard** if there is no known efficient algorithm■
- This does **not** mean the best we can do is to look through all possible solutions—see later lectures■
- In this lecture we are going to look at some efficient graph algorithms for weighted graphs■

# Minimum spanning tree

- A minimal spanning tree is the shortest tree which spans the entire graph■

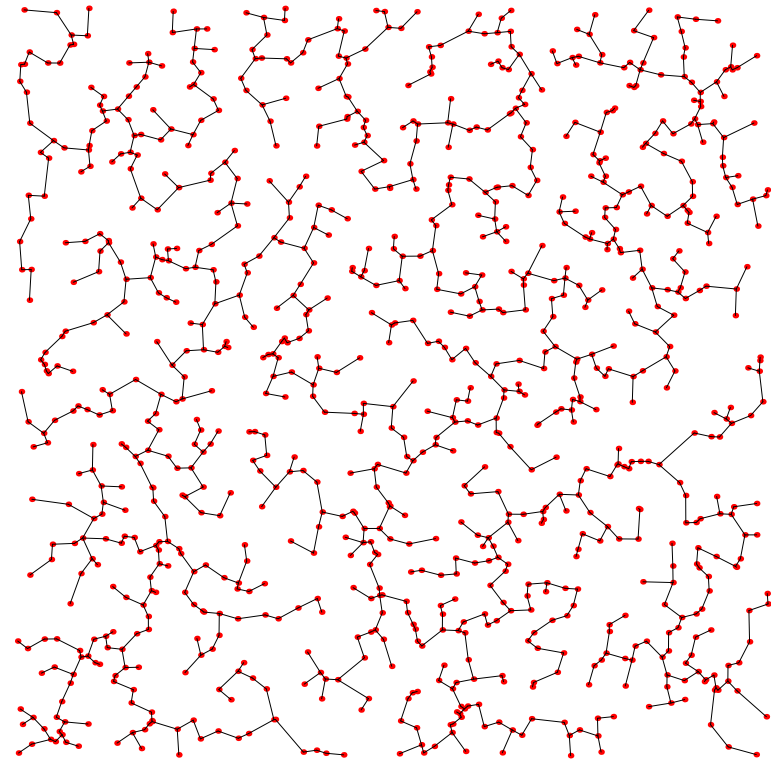


# Greedy Strategy

- We consider two algorithms for solving the problem
  - ★ Prim's algorithm (discovered 1957)
  - ★ Kruskal's algorithm (discovered 1956)■
- Both algorithms use a **greedy strategy**■
- Generally greedy strategies are not guaranteed to give globally optimal solutions■
- There exists a class of problems with a **matroid** structure where greedy algorithms lead to globally optimal solutions■
- Minimum spanning trees, Huffman codes and shortest path problems are matroids■

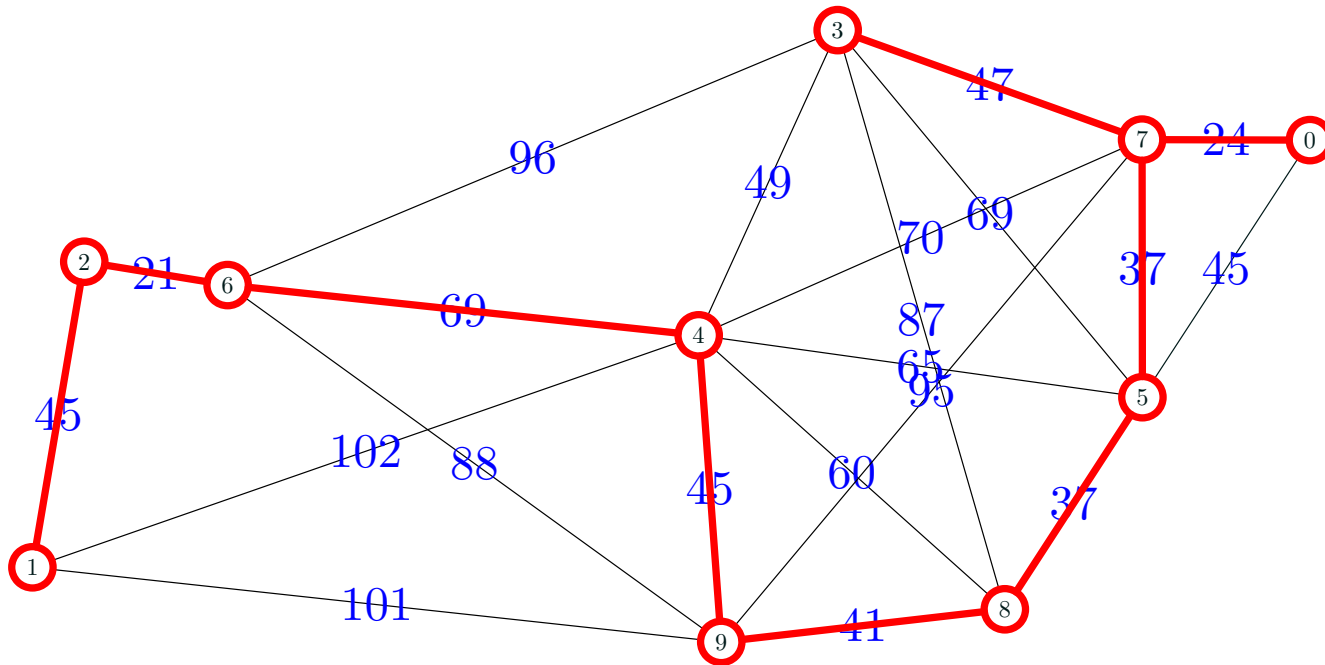
# Outline

1. Minimum Spanning Tree
2. **Prim's Algorithm**
3. Kruskal's Algorithm
4. Union Find
5. Shortest Path



# Prim's Algorithm

- Prim's algorithm grows a subtree greedily
- Start at an arbitrary node
- Add the shortest edge to a node not in the tree



# Pseudo Code

```
PRIM( $G = (\mathcal{V}, \mathcal{E}, w)$ ) {  
  for  $i \leftarrow 1$  to  $|\mathcal{V}|$   
     $d_i \leftarrow \infty$           \ \ Minimum 'distance' to subtree  
  endfor  
   $\mathcal{E}_T \leftarrow \emptyset$           \ \ Set of edges in subtree  
  PQ.initialise() \ \ initialise an empty priority queue  
  node  $\leftarrow v_1$           \ \ where  $v_1 \in \mathcal{V}$  is arbitrary  
  for  $i \leftarrow 1$  to  $|\mathcal{V}| - 1$   
     $d_{\text{node}} \leftarrow 0$   
    for  $k \in \{v \in \mathcal{V} | (\text{node}, v) \in \mathcal{E}\}$  \ \  $k$  is a neighbours of node  
      if (  $w_{\text{node},k} < d_k$  )  
         $d_k \leftarrow w_{\text{node},k}$   
        PQ.add( (  $d_k$ , (node, k) ) )  
      endif  
    endfor  
    do  
      (a_node, next_node)  $\leftarrow$  PQ.getMin()  
    until (  $d_{\text{next\_node}} > 0$  )  
     $\mathcal{E}_T \leftarrow \mathcal{E}_T \cup \{(\text{a\_node}, \text{next\_node})\}$   
    node  $\leftarrow$  next_node  
  endfor  
  return  $\mathcal{E}_T$   
}
```

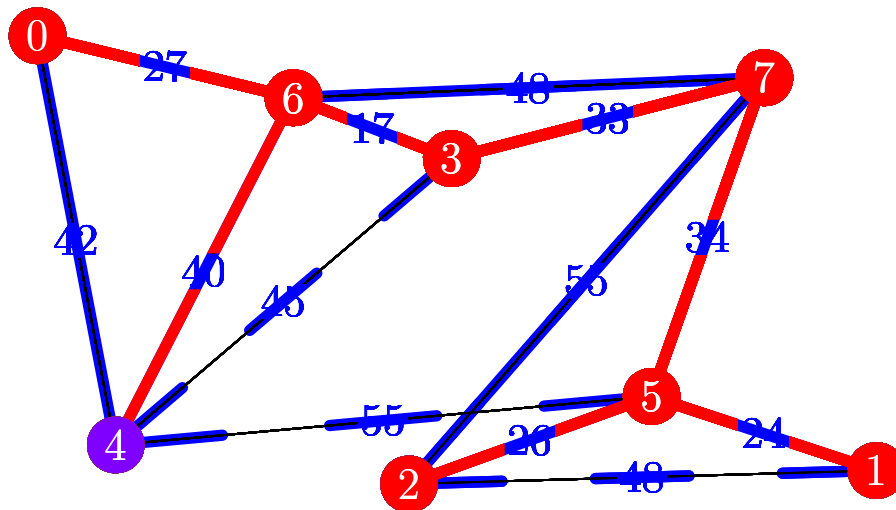
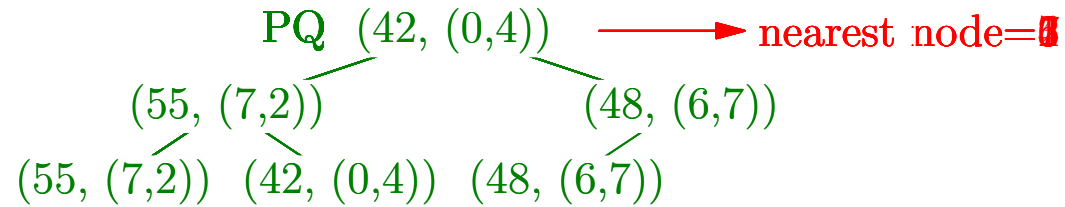


# Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
d[]	<del>0</del>	<del>21</del>	<del>55</del>	<del>17</del>	<del>40</del>	<del>31</del>	<del>27</del>	<del>38</del>

neighbors of node 4 added to PQ

node=4



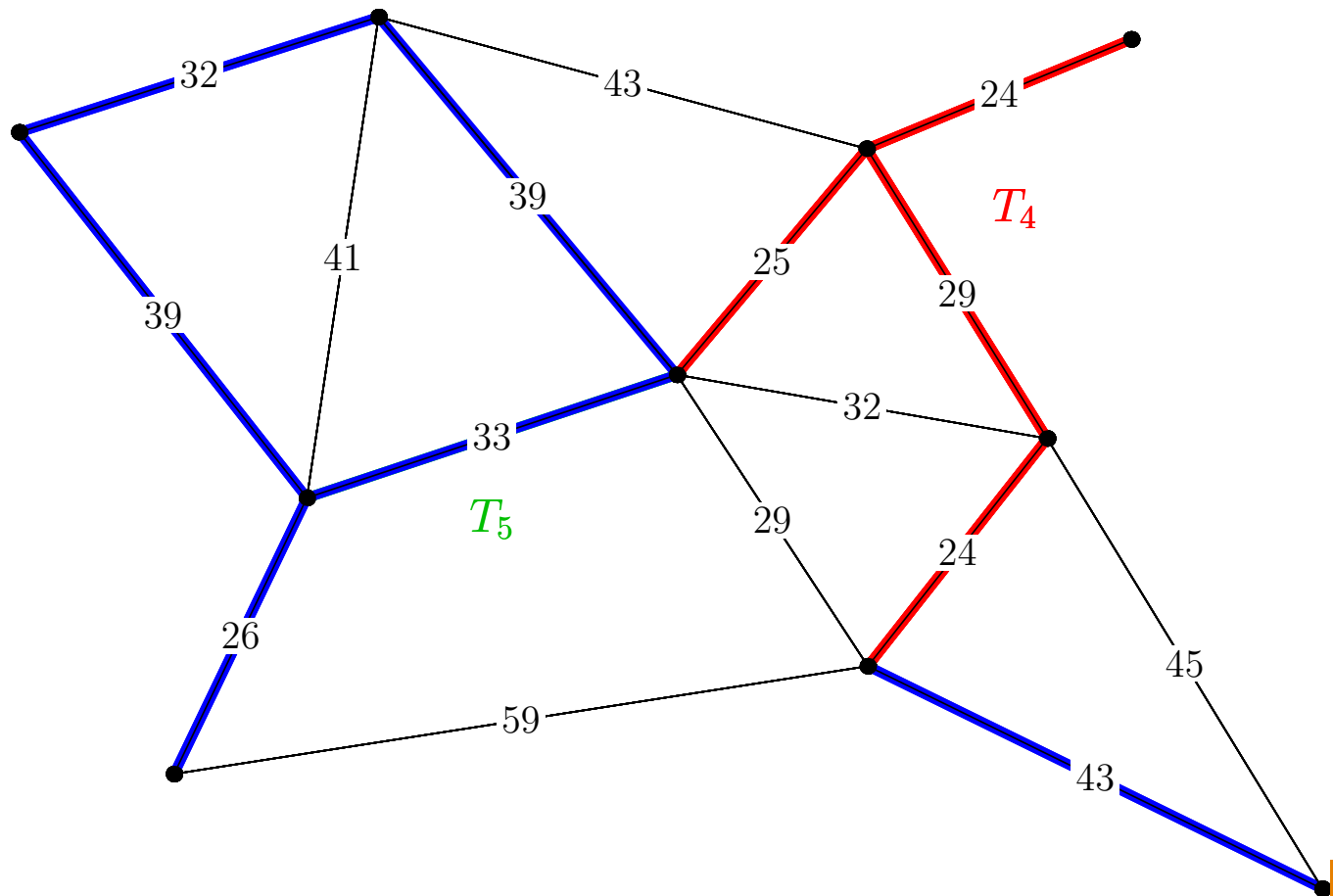
Finished MST

# Why Does This Work?

- Clearly Prim's algorithm produces a spanning tree■
  - ★ It is a tree because we always choose an edge to a node not in the tree■
  - ★ It is a spanning tree because it has  $|\mathcal{V}| - 1$  edges■
- Why is this a minimum spanning tree?■
- Once again we look for a proof by induction■

# Proof by induction

- We want to show that each subtree,  $T_i$ , for  $i = 1, 2, \dots, n$  is part of (a subgraph) of some minimum spanning tree■
- In the base case,  $T_1$  consists of a tree with no edges, but this has to be part of the minimum spanning tree■
- To prove the inductive case we assume that  $T_i$  is part of the minimum spanning tree■
- We want to prove that  $T_{i+1}$  formed by adding the shortest edge is also part of the minimum spanning tree■
- We perform the proof by contradiction■—we assume that this added edge isn't part of the minimum spanning tree■



# Loop Counting

```
PRIM( $G = (\mathcal{V}, \mathcal{E}, w)$ ) {  
  for  $i \leftarrow 0$  to  $|\mathcal{V}|$   
     $d_i \leftarrow \infty$   
  endfor  
   $\mathcal{E}_T \leftarrow \emptyset$   
  PQ.initialise()  
  node  $\leftarrow v_1$   
  for  $i \leftarrow 1$  to  $|\mathcal{V}| - 1$  // loop 1  $O(|\mathcal{V}|)$   
     $d_{node} \leftarrow 0$   
    for  $k \in \{v \in \mathcal{V} | (node, v) \in \mathcal{E}\}$  // inner loop  $O(|\mathcal{E}|/|\mathcal{V}|)$   
      if ( $w_{node,k} < d_k$ )  
         $d_k \leftarrow w_{node,k}$   
        PQ.add( $(d_k, (node, k))$ ) //  $O(\log(|\mathcal{E}|))$   
      endif  
    endfor  
    do  
      ( $a\_node, next\_node$ )  $\leftarrow$  PQ.getMin()  
    until ( $d_{next\_node} > 0$ )  
     $\mathcal{E}_T \leftarrow \mathcal{E}_T \cup \{(node, next\_node)\}$   
    node  $\leftarrow next\_node$   
  endfor  
  return  $\mathcal{E}_T$   
}
```

# Run Time

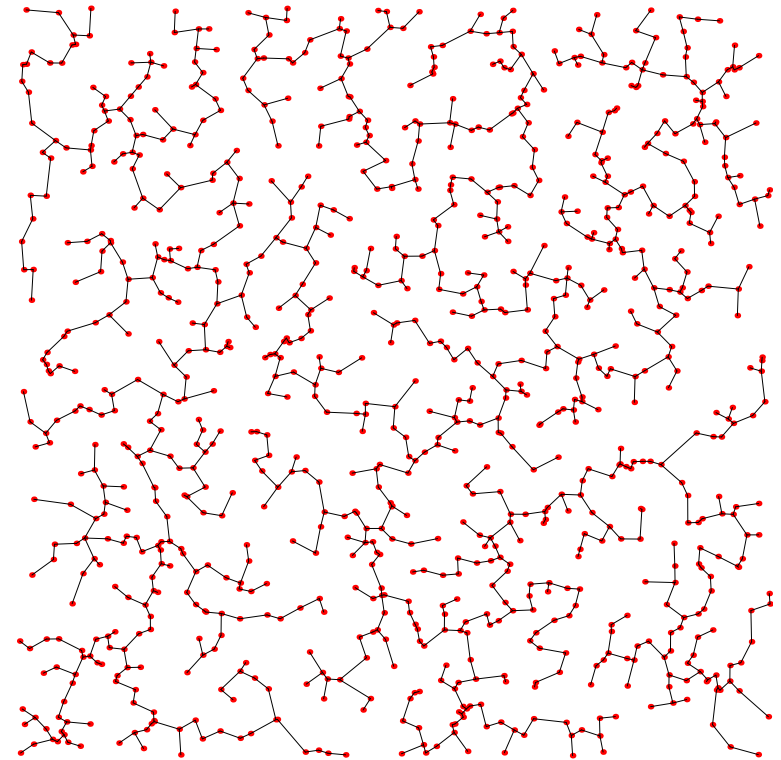
- The worst time is

$$O(|\mathcal{V}|) \times O\left(\frac{|\mathcal{E}|}{|\mathcal{V}|}\right) \times O(\log(|\mathcal{E}|)) = O(|\mathcal{E}| \log(|\mathcal{E}|))$$

- Note that  $|\mathcal{E}| < |\mathcal{V}|^2$
- Thus,  $\log(|\mathcal{E}|) < 2 \log(|\mathcal{V}|) = O(\log(|\mathcal{V}|))$
- Thus the worst case time complexity is  $|\mathcal{E}| \log(|\mathcal{V}|)$

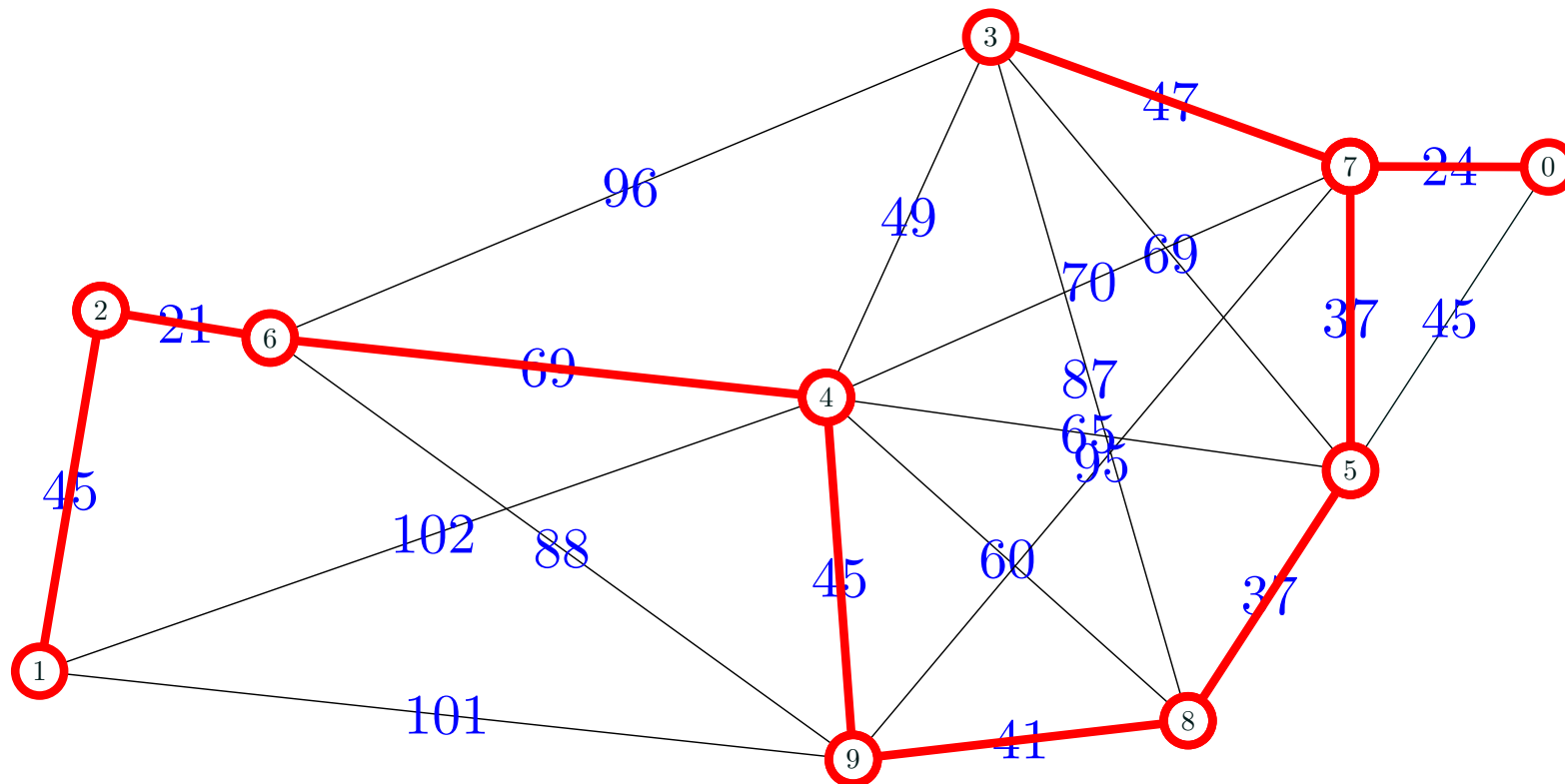
# Outline

1. Minimum Spanning Tree
2. Prim's Algorithm
3. **Kruskal's Algorithm**
4. Union Find
5. Shortest Path



# Kruskal's Algorithm

- Kruskal's algorithm works by choosing the shortest edges which don't form a loop





# Pseudo Code

```
KRUSKAL( $G = (\mathcal{V}, \mathcal{E}, w)$ )  
{  
    PQ.initialise()  
    for edge  $\in |\mathcal{E}|$   
        PQ.add( ( $w_{edge}$ , edge) )  
    endfor  
  
     $\mathcal{E}_T \leftarrow \emptyset$   
    noEdgesAccepted  $\leftarrow 0$   
  
    while (noEdgesAccepted  $< |\mathcal{V}| - 1$ )  
        edge  $\leftarrow$  PQ.getMin()  
        if  $\mathcal{E}_T \cup \{\text{edge}\}$  is acyclic  
             $\mathcal{E}_T \leftarrow \mathcal{E}_T \cup \{\text{edge}\}$   
            noEdgesAccepted  $\leftarrow$  noEdgesAccepted + 1  
        endif  
    endwhile  
  
    return  $\mathcal{E}_T$   
}
```

# Analysis

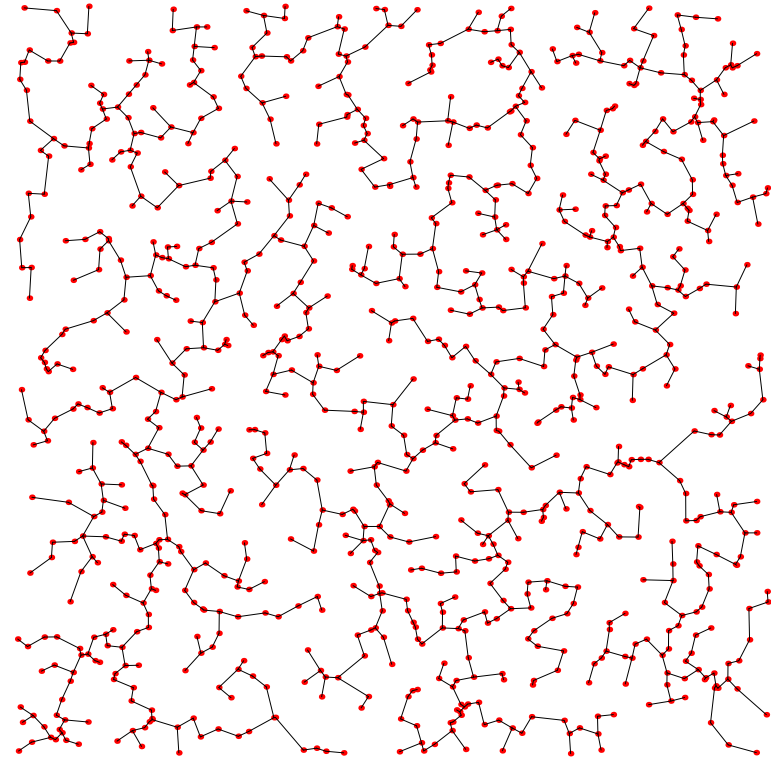
- Kruskal's algorithm looks much simpler than Prim's■
- The sorting takes most of the time, thus Prim's algorithm is  $O(|\mathcal{E}| \log(|\mathcal{E}|)) = O(|\mathcal{E}| \log(|\mathcal{V}|))$ ■
- We can sort the edges however we want—we could use quick sort rather than heap sort using a priority queue■
- But we haven't specified how we determine if the added edge would produce a cycle■

# Cycling

- For a path to be a cycle the edge has to join two nodes representing the same subtree■
- To compute this we need to quickly **find** which subtree a node has been assigned to■
- Initially all nodes are assigned to a separate subtree■
- When two subtrees are combined by an edge we have to perform the **union** of the two subtrees■
- This is a tricky but standard operation known as **union-find**■

# Outline

1. Minimum Spanning Tree
2. Prim's Algorithm
3. Kruskal's Algorithm
4. **Union Find**
5. Shortest Path



# Union-Find

- In the union-find algorithm we have a set of objects  $x \in \mathcal{S}$  which are to be grouped into subsets  $\mathcal{S}_1, \mathcal{S}_2, \dots$  ■
- Initially each object is in its individual subset (no relationships) ■
- We want to make the **union** of two subsets (add relationship between elements) ■
- We also want to **find** the subset given an element ■
- This is a common problem for which we will write a class `DisjointSets` to perform fast unions and finds ■

# DisjointSets

- We want to create a class

```
class DisjointSets
{
    DisjointSets(int numElements) {/* Constructor */}
    int find(int x) {/* Find root */}
    void union(int root1, int root2) {/* Union */}

    private:
        int[] s;
}
```

- Where `find(x)` returns a unique identifier for the subset which element `x` belongs to
- The array `s` contains labelling information to implement `find(x)`

# The Union-Find Dilemma

- A natural algorithm to perform finds is to maintain an array returning a subset label for each element—this makes `find` fast
- However, every time we combine two subset we have to change all the labels in this array (taking  $O(n)$  operations)
- If we are unlucky the cost of performing  $n$  unions is  $\Theta(n^2)$
- If we ensure that we relabel the smaller subset then the time complexity is  $\Theta(n \log(n))$
- Fast *finds* seems to give slow(ish) *unions*
- What about the other way around?

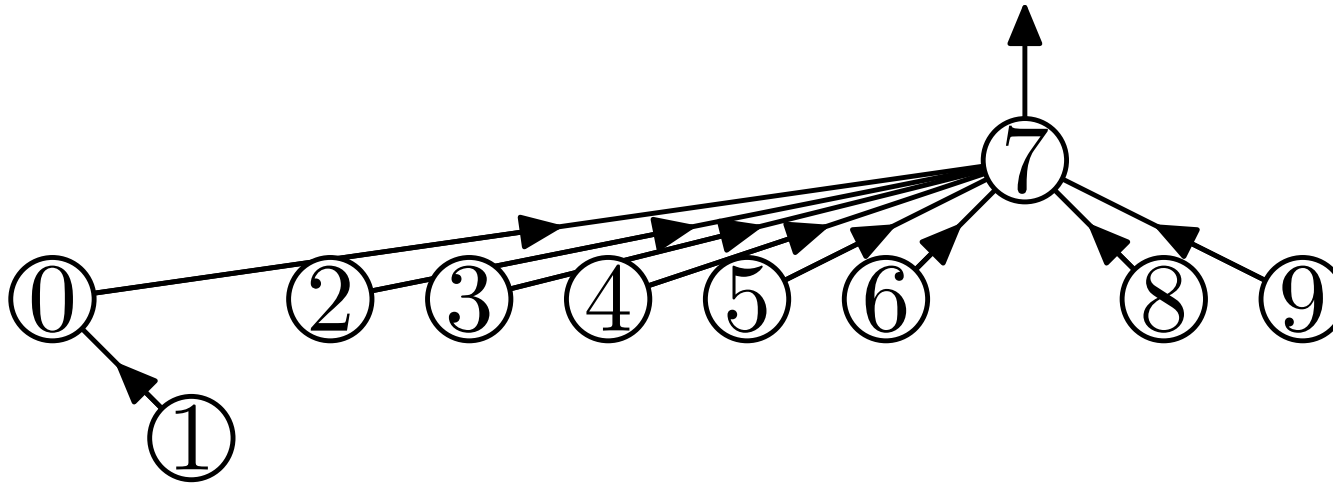
# Fast Union

- To achieve fast unions we can represent our disjoint sets as a forest (many disjoint trees)■
- Every time we perform a union we make one of the trees point to the head of the other tree■
- The cost of `find` depends on the depth of the tree■
- To make unions efficient we make the shallow tree a subtree of the deeper tree■



# Putting it Together

$\text{find}(6)=7$



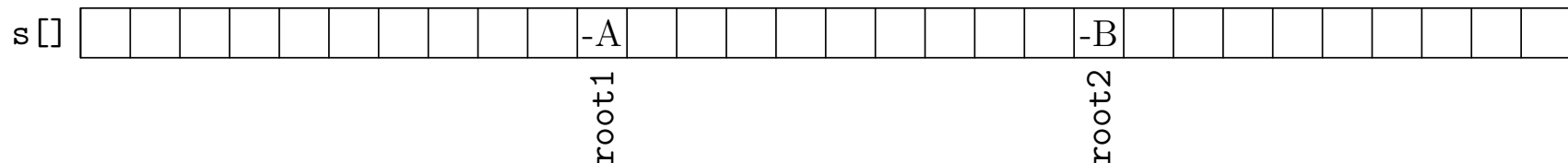
7	0	7	7	7	7	7	-3	7	7
0	1	2	3	4	5	6	7	8	9



# Smart Union

```
DisjointSets::DisjointSets(int numElements)
{
    s = new int[numElements];
    for(int i=0; i<s.length; i++)
        s[i] = -1; // roots are negative number
}
```

```
void DisjointSets::union(int root1, int root2)
{
    if (s[root2]<s[root1]) {           // root2 is deeper
        s[root1] = root2;             // make root2 the root
    } else {
        if (s[root1]==s[root2])
            s[root1]--;               // update height if same
        s[root2] = root1;             // make root1 new root
    }
}
```



# Path Compression

- To speed up `find` we relabel all nodes we visit during `find` by the root label

```
public int DisjointSets::find(int index)
{
    if (s[index]<0)
        return index;
    else
        return s[index] = find(s[index]);
}
```

s[]					10					20									-3									
					5					10									20									

# Mazes

- Union-Find is a data structure which can occur in very different applications■
- One application is building a maze■
- Start from a complete lattice■
- Remove a randomly chosen edge if it connects two unconnected regions■
- Stop when the start and end cell are connected■
- Or better after all cells are connected■

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24
25	26	27	28	29
30	31	32	33	34
35	36	37	38	39
40	41	42	43	44
45	46	47	48	49

# Time Complexity of Union-Find

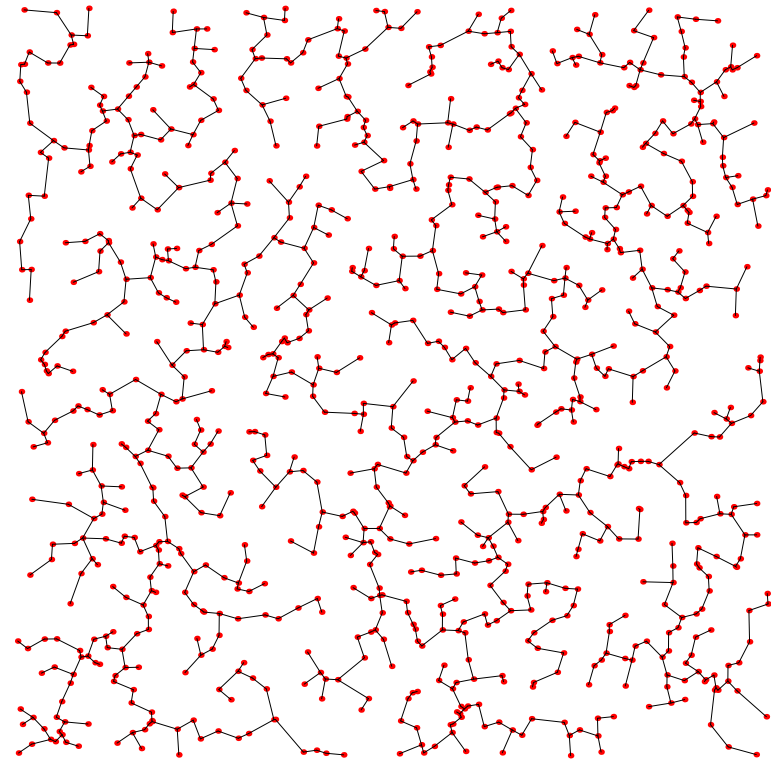
- If we perform  $M$  finds and  $N$  unions then the time complexity is  $O(M \log_2^*(N))$  ■
- Where  $\log_2^*(N)$  is the number of times you need to apply the logarithm function before you get a number less than 1 ■
- In practice  $\log_2^*(N) \leq 5$  for all conceivable  $N$  ■

$\log_2(\log_2(\log_2(\log_2(\log_2(\log_2(\log_2(0000))))))) = 0$

- The proof of this time complexity is rather involved

# Outline

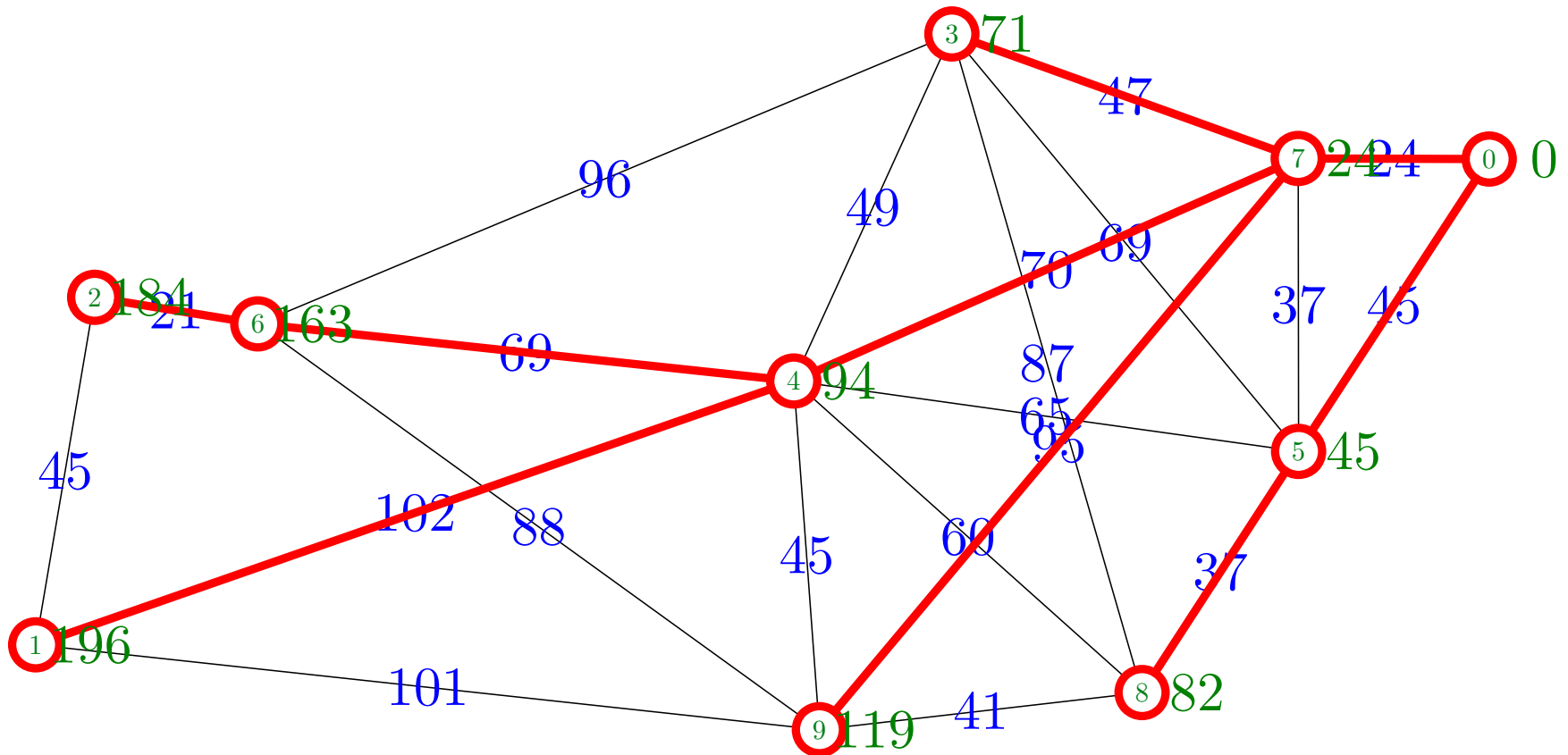
1. Minimum Spanning Tree
2. Prim's Algorithm
3. Kruskal's Algorithm
4. Union Find
5. **Shortest Path**



# Shortest path

- We can efficiently compute the shortest path from one vertex to any other vertex■
- This defines a spanning tree, but where the optimisation criteria is that we choose the vertex that are closest to the *source*■
- To find this spanning tree we use Dijkstra's algorithm where we successively add the nearest node to the source which is connected to the subtree built so far■
- This is very close to Prim's algorithm and has the same complexity■

# Dijkstra's Algorithm






# Pseudo Code

```
DIJKSTRA( $G = (\mathcal{V}, \mathcal{E}, w)$ , source) {  
  for  $i \leftarrow 0$  to  $|\mathcal{V}|$   
     $d_i \leftarrow \infty$           \ \ Minimum 'distance' to source  
  endfor  
   $\mathcal{E}_T \leftarrow \emptyset$       \ \ Set of edges in subtree  
  PQ.initialise() \ \ initialise an empty priority queue  
  node  $\leftarrow$  source  
   $d_{node} \leftarrow 0$   
  for  $i \leftarrow 1$  to  $|\mathcal{V}| - 1$   
    for  $k \in \{v \in \mathcal{V} | (node, v) \in \mathcal{E}\}$   
      if (  $w_{node,k} + d_{node} < d_k$  )  
         $d_k \leftarrow w_{node,k} + d_{node}$   
        PQ.add(  $(d_k, (node, k))$  )  
      endif  
    endfor  
    do  
      (a_node, next_node)  $\leftarrow$  PQ.getMin()  
      while next_node not in subtree  
         $\mathcal{E}_T \leftarrow \mathcal{E}_T \cup \{(a\_node, next\_node)\}$   
        node  $\leftarrow$  next_node  
      endwhile  
    endfor  
  return  $\mathcal{E}_T$   
}
```

# Compare to Prim's Algorithm

```
PRIM( $G = (\mathcal{V}, \mathcal{E}, w)$ ) {  
  for  $i \leftarrow 1$  to  $|\mathcal{V}|$   
     $d_i \leftarrow \infty$           \ \ Minimum 'distance' to subtree  
  endfor  
   $\mathcal{E}_T \leftarrow \emptyset$         \ \ Set of edges in subtree  
  PQ.initialise() \ \ initialise an empty priority queue  
  node  $\leftarrow v_1$           \ \ where  $v_1 \in \mathcal{V}$  is arbitrary  
  for  $i \leftarrow 1$  to  $|\mathcal{V}| - 1$   
     $d_{\text{node}} \leftarrow 0$    
    for  $k \in \{v \in \mathcal{V} | (\text{node}, v) \in \mathcal{E}\}$   
      if (  $w_{\text{node},k} < d_k$  )  
         $d_k \leftarrow w_{\text{node},k}$   
        PQ.add( (  $d_k, (\text{node}, k)$  ) )  
      endif  
    endfor  
    do  
      (a_node, next_node)  $\leftarrow$  PQ.getMin()  
    until (  $d_{\text{next\_node}} > 0$  )  
     $\mathcal{E}_T \leftarrow \mathcal{E}_T \cup \{(\text{a\_node}, \text{next\_node})\}$   
    node  $\leftarrow$  next_node  
  endfor  
  return  $\mathcal{E}_T$   
}
```

# Dijkstra Details

- Dijkstra is very similar to Prim's (it differs in the distances that are used)■
- It has the same time complexity■
- It can be viewed as using a greedy strategy■
- It can also be viewed as using the dynamic programming strategy (see lecture 22)■

# Lessons

- There are many efficient (i.e. polynomial  $O(n^a)$ ) graph algorithms■
- Some of the most efficient ones are based on the Greedy strategy■
- These are easily implemented using priority queues■
- Minimum spanning trees are useful because they are easy to compute■
- Dijkstra's algorithm is one of the classics■