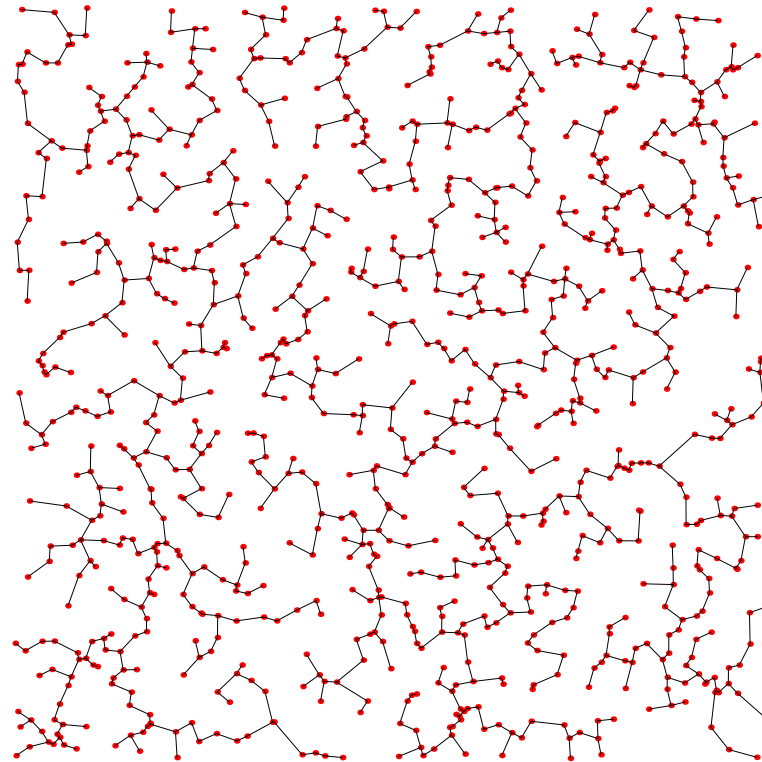


Algorithms and Analysis

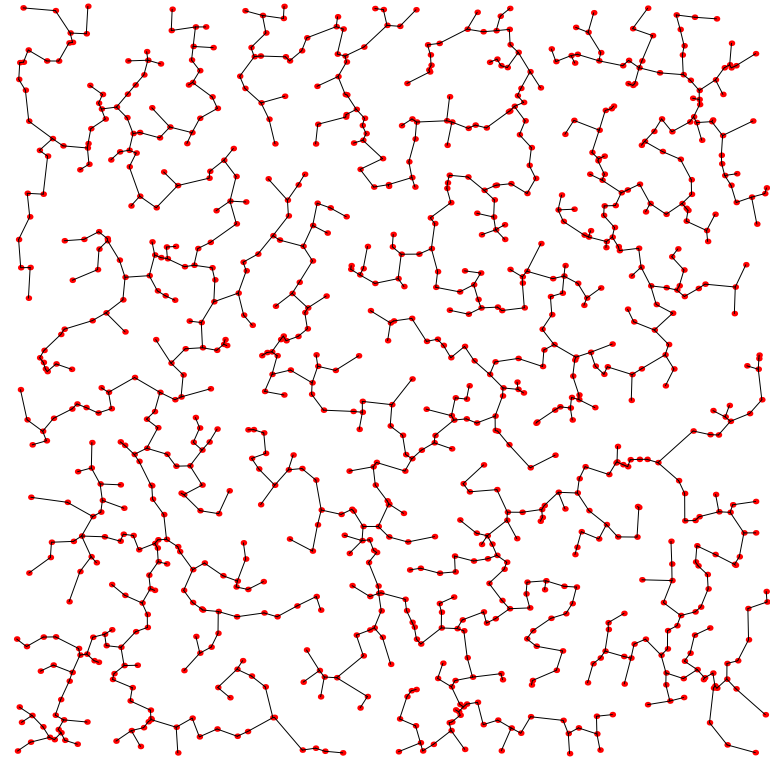
Lesson 21: *Know Your Graph Algorithms*



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. Shortest Path

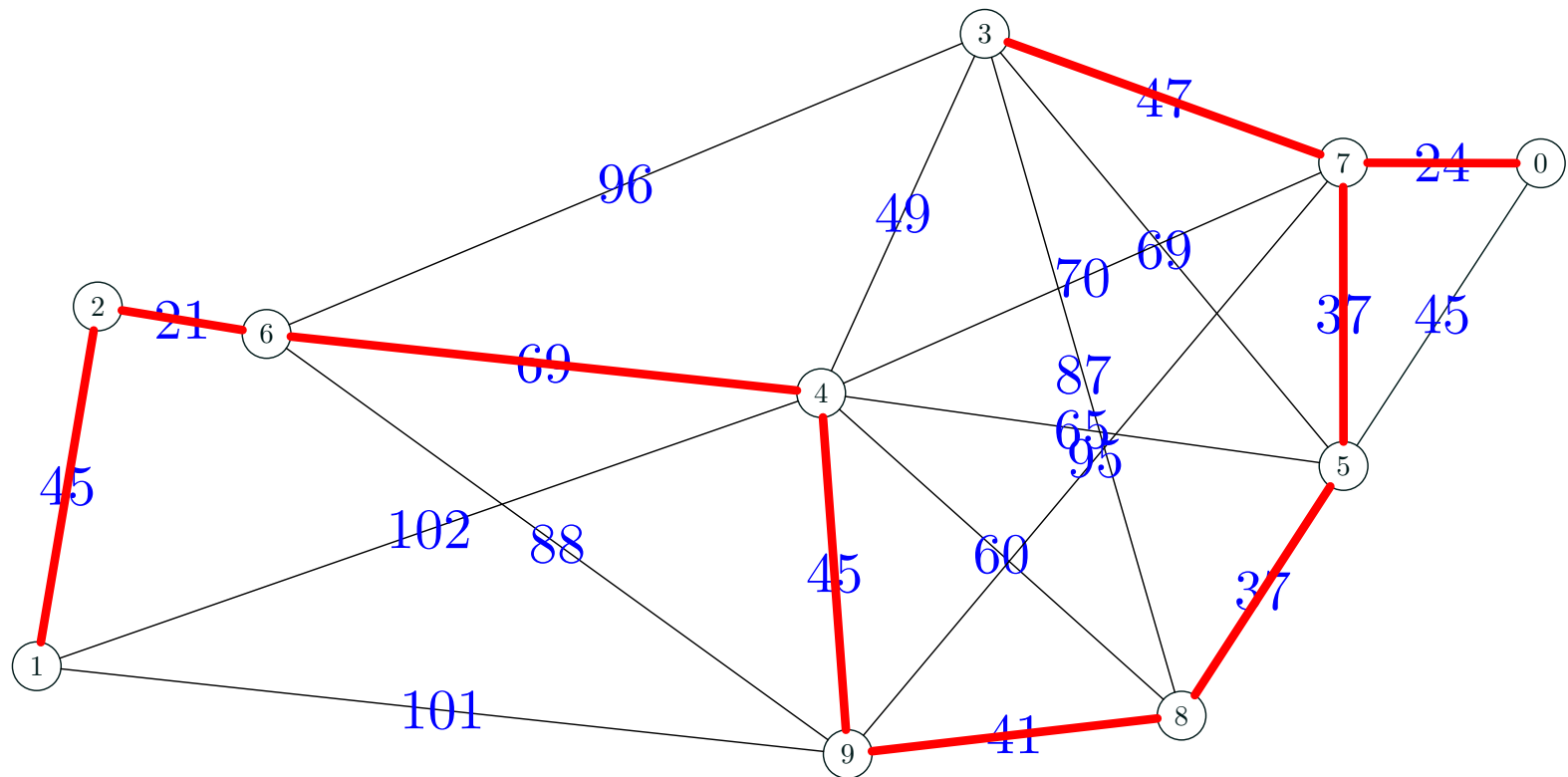


Graph Algorithms

- 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

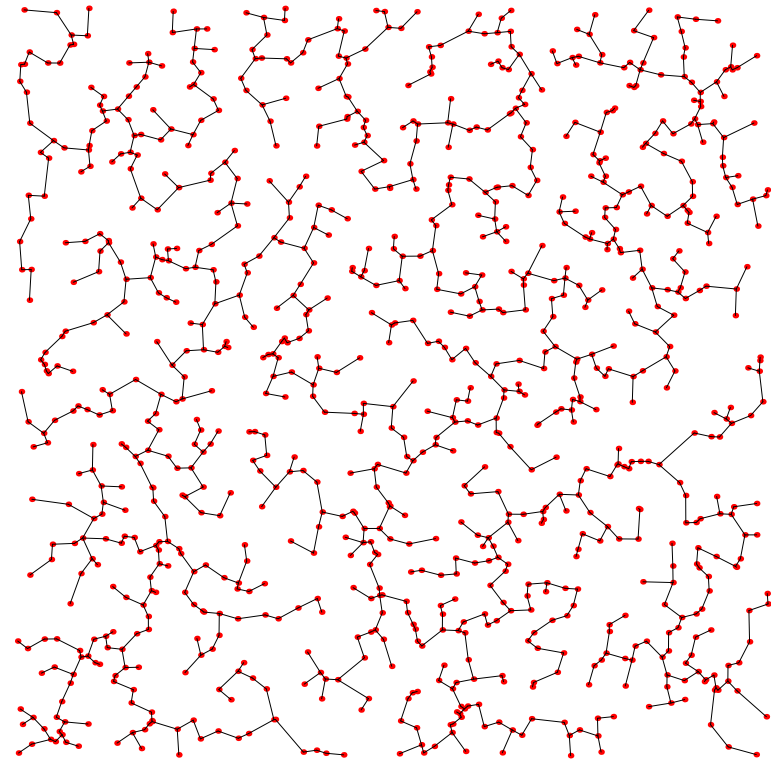


Greedy Strategy

- 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■

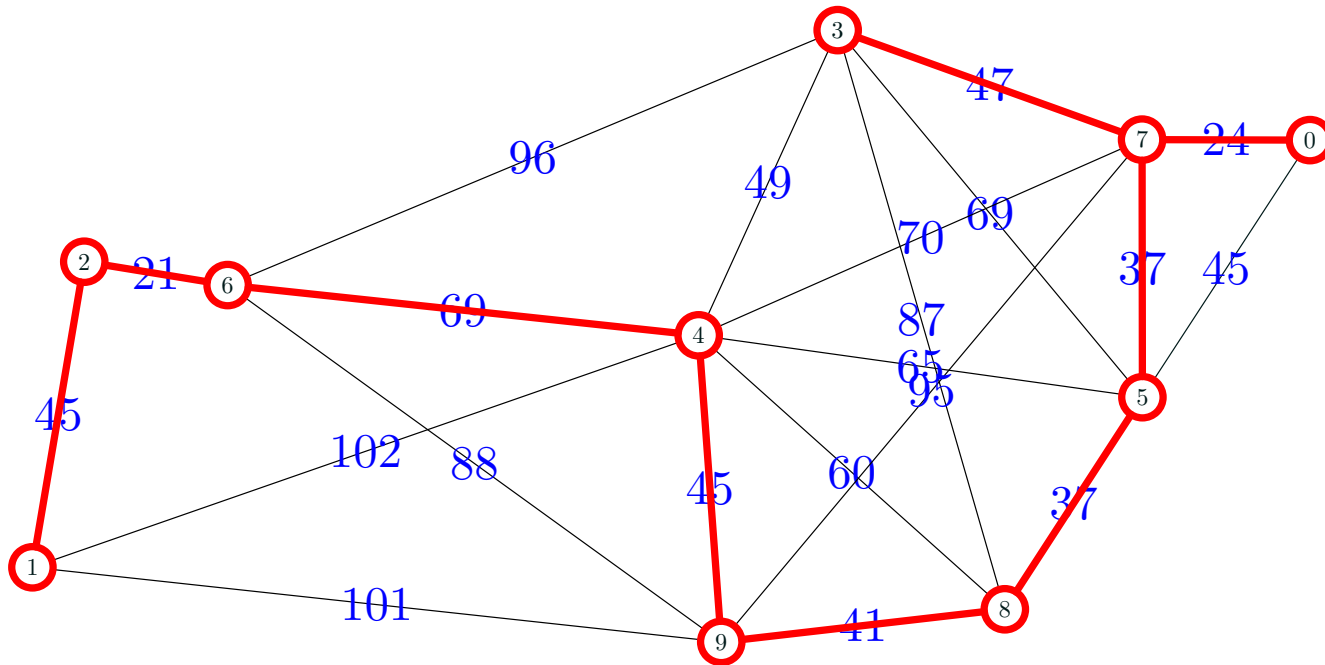
Outline

1. Minimum Spanning Tree
2. **Prim's Algorithm**
3. Kruskal's Algorithm
4. 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

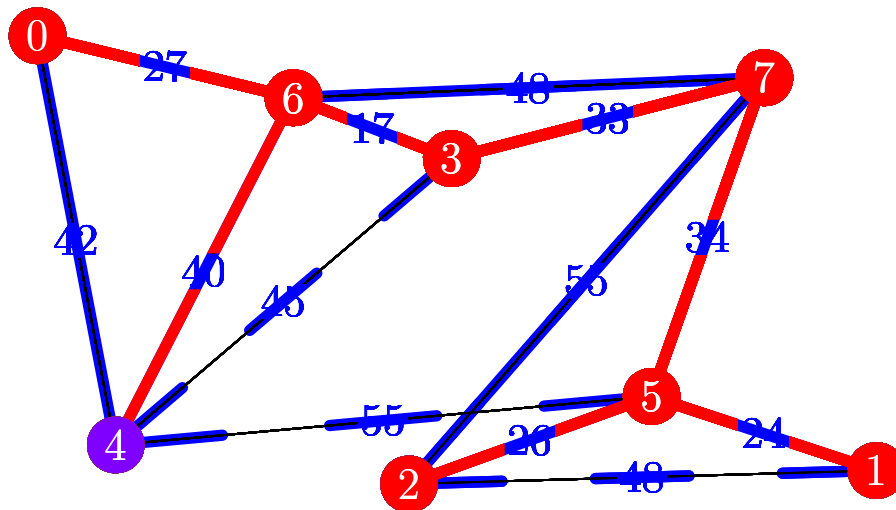
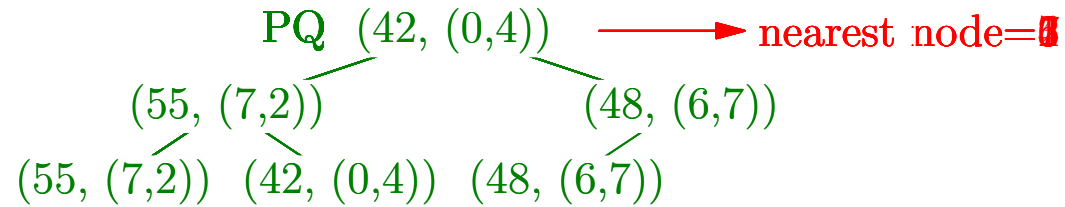
```
PRIM( $G = (\mathcal{V}, \mathcal{E}, w)$ ) {  
  for  $i \leftarrow 1$  to  $|\mathcal{V}|$   
     $d_i \leftarrow \infty$           \ \ Minimum 'distance' to subtree  
  endfor  
   $\mathcal{E}_T \leftarrow \emptyset$           \ \ Set of edges in subtree  
  PQ.initialise() \ \ initialise an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


Prim's Algorithm in Detail

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

neighbors of node 4 added to PQ

node=4



Finished MST

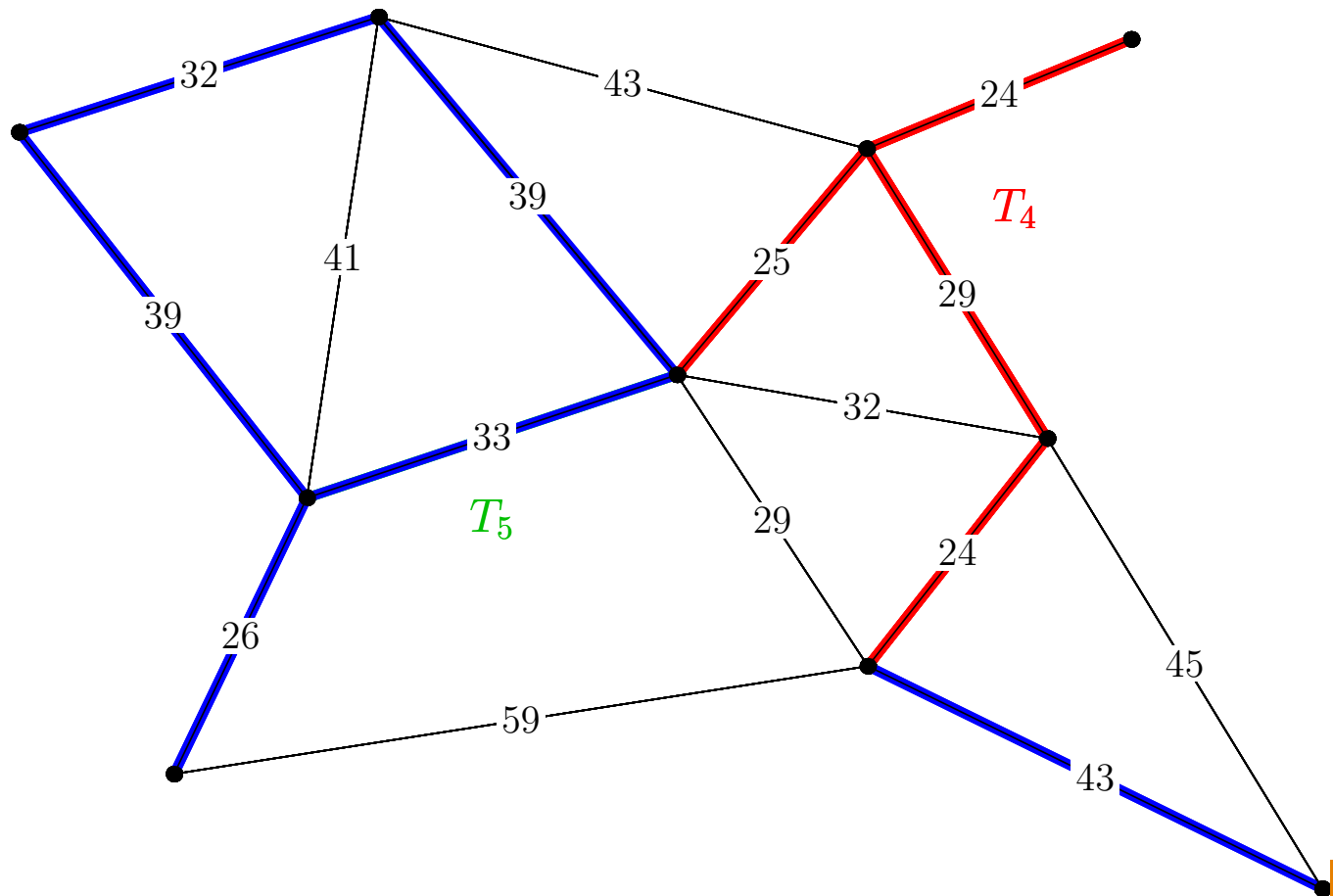
Why Does This Work?

- 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■

Contrariwise



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  
       $(a\_node, next\_node) \leftarrow \text{PQ.getMin}()$   
    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

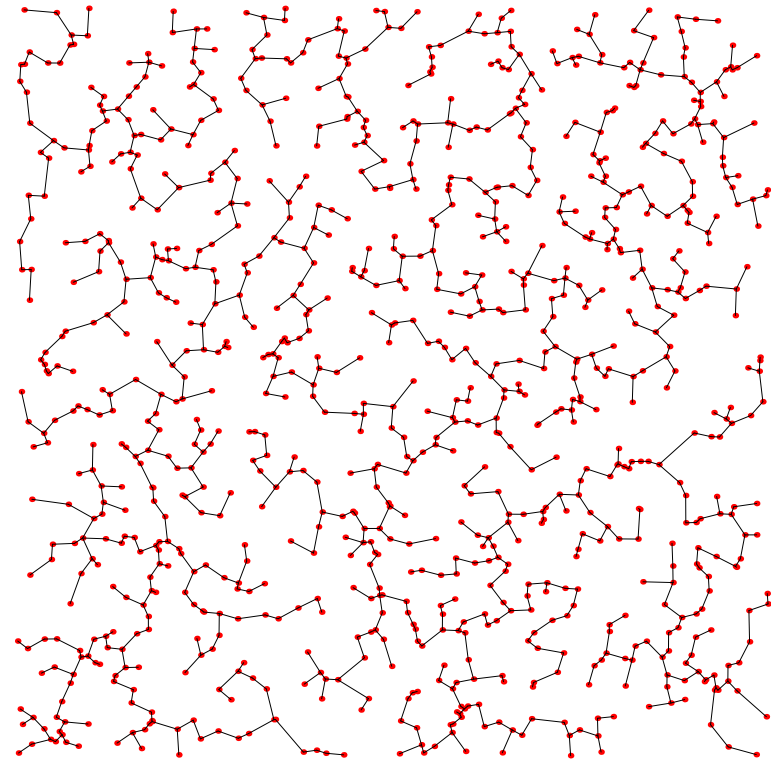
- The worst time is

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

- 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 $|\mathcal{E}| \log(|\mathcal{V}|)$

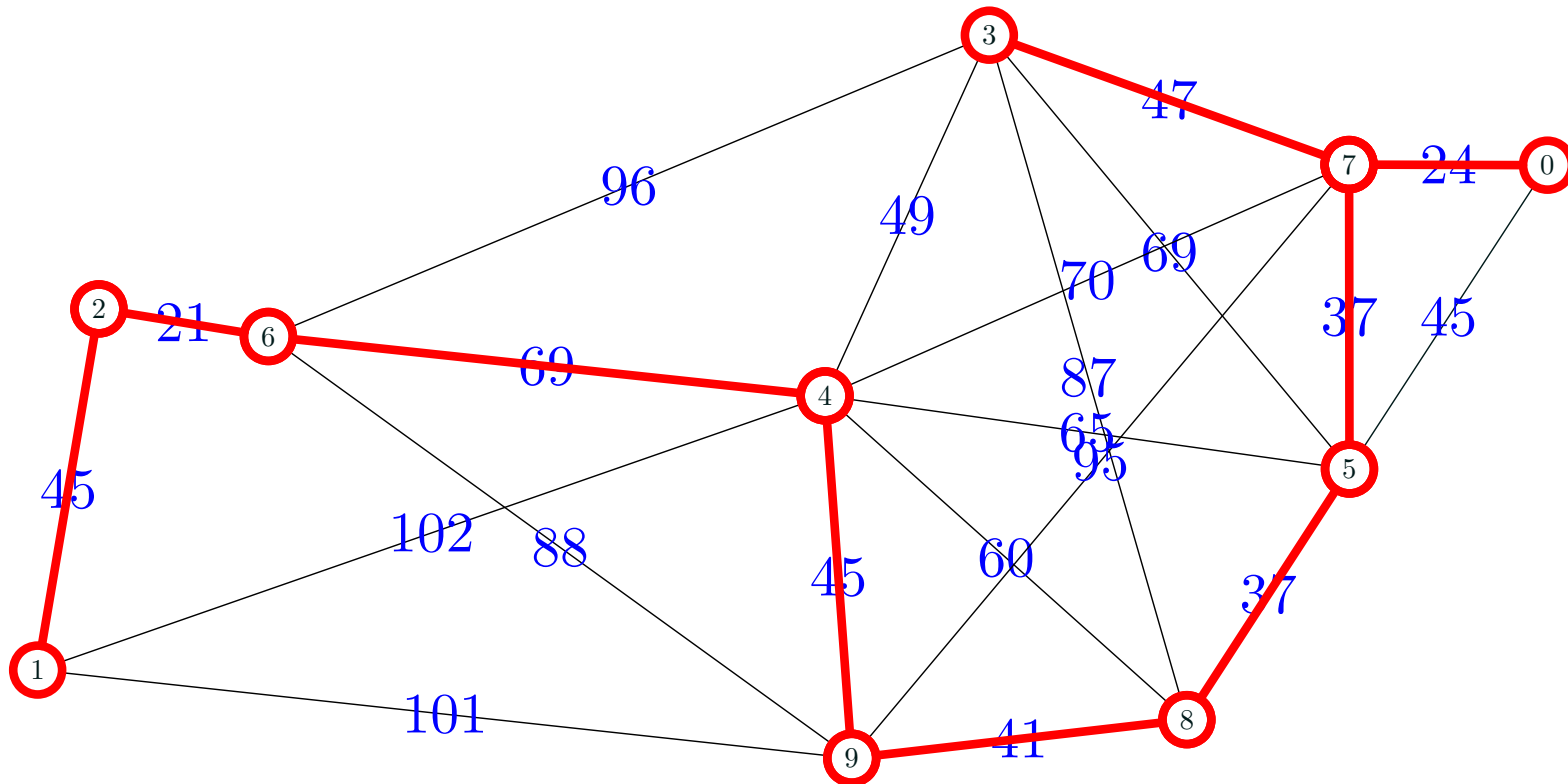
Outline

1. Minimum Spanning Tree
2. Prim's Algorithm
3. **Kruskal's Algorithm**
4. 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

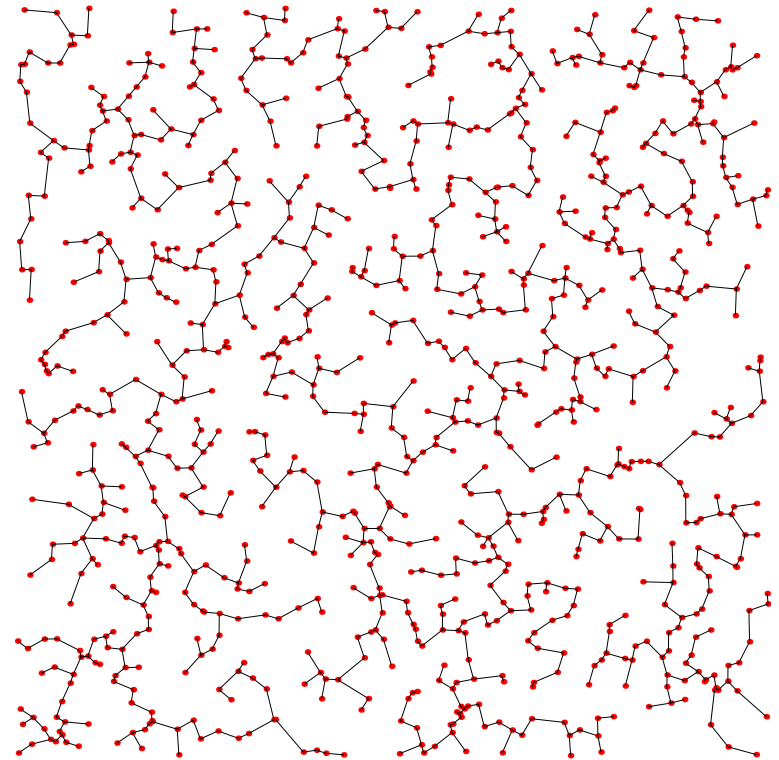
- Kruskal's algorithm looks much simpler than Prim's■
- The sorting takes most of the time, thus Prim's algorithms is $O(|\mathcal{E}| \log(|\mathcal{E}|)) = O(|\mathcal{E}| \log(|\mathcal{V}|))$ ■
- We can sort the edges however we want—we could use quick sort rather than heap sort using a priority 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■
- But that is precisely the **union-find** algorithm we covered in lecture 13■

Outline

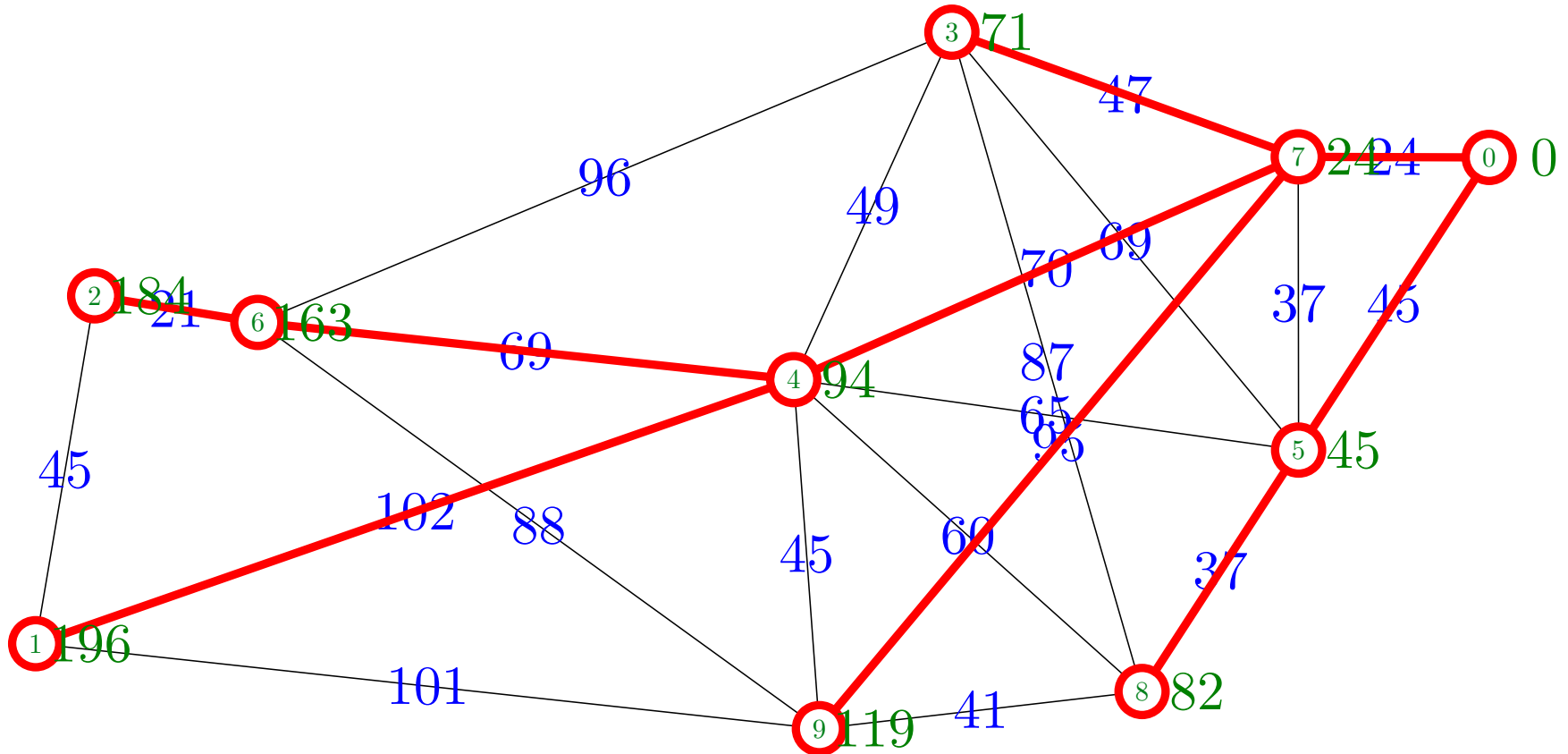
1. Minimum Spanning Tree
2. Prim's Algorithm
3. Kruskal's Algorithm
4. **Shortest Path**



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 neigh  $\in \{v \in \mathcal{V} | (node, v) \in \mathcal{E}\}$ 
      if (  $w_{node,neigh} + d_{node} < d_{neigh}$  )
         $d_{neigh} \leftarrow w_{node,neigh} + d_{node}$ 
        PQ.add( ( $d_{neigh}$ , (node, neigh)) )
      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 neigh  $\in \{v \in \mathcal{V} | (node, v) \in \mathcal{E}\}$   
      if (  $w_{\text{node,neigh}} < d_{\text{neigh}}$  )  
         $d_{\text{neigh}} \leftarrow w_{\text{node,neigh}}$   
        PQ.add( (  $d_{\text{neigh}}, (node, \text{neigh})$  ) )  
      endif  
    endfor  
    do  
      (a_node, next_node)  $\leftarrow$  PQ.getMin()  
    until (  $d_{\text{next\_node}} > 0$  )  
     $\mathcal{E}_T \leftarrow \mathcal{E}_T \cup \{(a\_node, \text{next\_node})\}$   
    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■