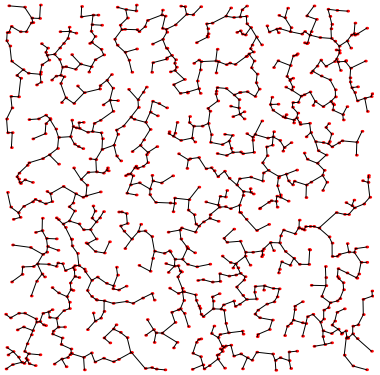
# Algorithms and Analysis

## Lesson 21: *Know Your Graph Algorithms*

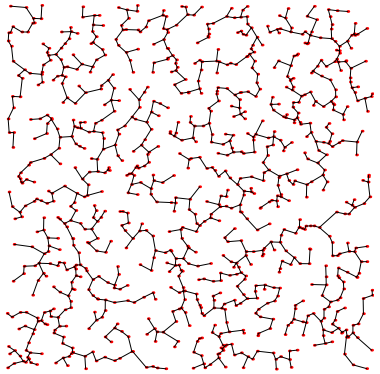


*Weighted graph algorithms, Minimum spanning tree, Prim, Kruskal, shortest path, Dijkstra*



# Outline

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





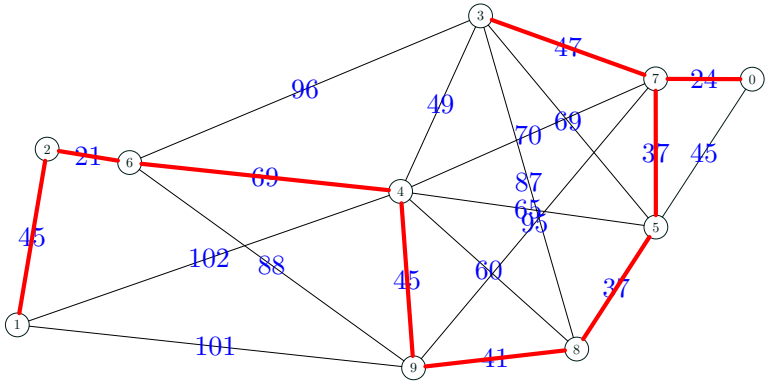
# Graph Algorithms

- 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



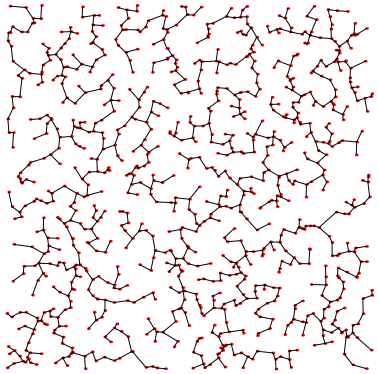
## Greedy Strategy

- 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

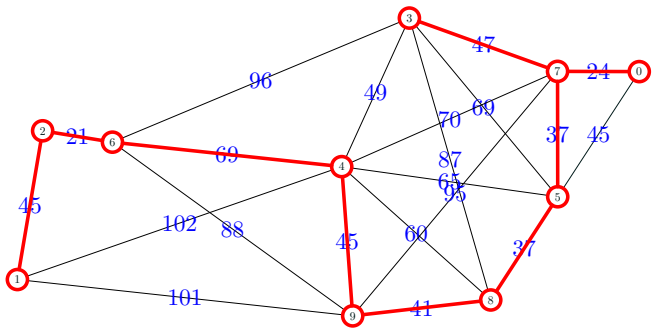
1. Minimum Spanning Tree
2. **Prim's Algorithm**
3. Kruskal's Algorithm
4. Shortest Path





## 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

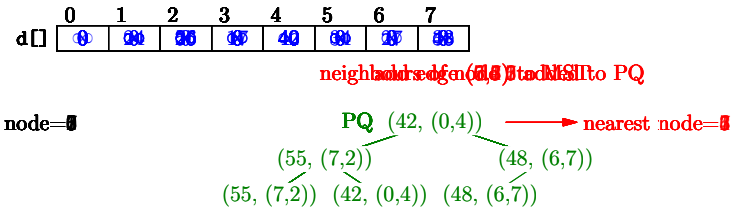


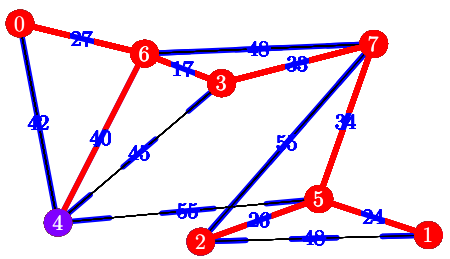



## Pseudo Code

```
PRIM(G = (V, E, w)) {
  for i←1 to |V|
    d_i ←∞            \\ Minimum 'distance' to subtree
  endfor
  E_T ←∅              \\ Set of edges in subtree
  PQ.initialise()     \\ initialise an empty priority queue
  node ← v_1          \\ where v_1 ∈ V is arbitrary
  for i←1 to |V| − 1
    d_node ←0
    for neigh ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,neigh < d_neigh )
        d_neigh ←w_node,neigh
        PQ.add( (d_neigh, (node,neigh)) )
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    until (d_next_node > 0)
    E_T ←E_T ∪ {(a_node, next_node)}
    node ←next_node
  endfor
  return E_T
}
```

# Prim's Algorithm in Detail






# 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

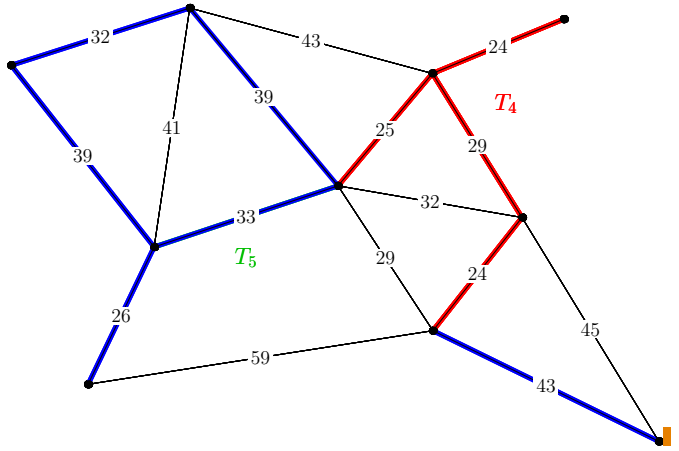


# Proof by induction

- We want to show that each subtree, $T_i$, for $i = 1, 2, \cdots, n$ 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 = (V, E, w)) {
  for i←0 to |V|
    d_i ←∞
  endfor
  E_T ←∅
  PQ.initialise()
  node ← v_1
  for i←1 to |V| − 1                    // loop 1  O(|V|)
    d_node ←0
    for k ∈ {v ∈ V|(node, v) ∈ E}  // inner loop O(|E|/|V|)
      if ( w_node,k < d_k )
        d_k ←w_node,k
        PQ.add( (d_k, (node,k)) )      // O(log(|E|))
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    until (d_next_node > 0)
    E_T ←E_T ∪ {(node, next_node)}
    node ←next_node
  endfor
  return E_T
}
```



## Run Time

- The worst time is

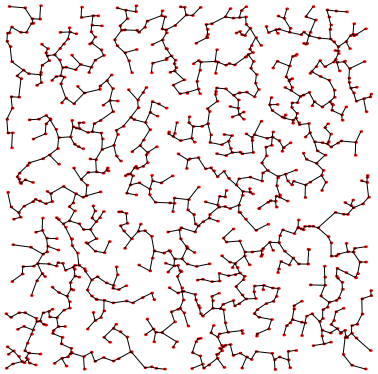
$$O(|\mathcal{V}|) \times O\left(\frac{|\mathcal{E}|}{|\mathcal{V}|}\right) \times O\big(\log(|\mathcal{E}|)\big) = O\left(|\mathcal{E}|\log(|\mathcal{E}|)\right)$$

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



## Outline

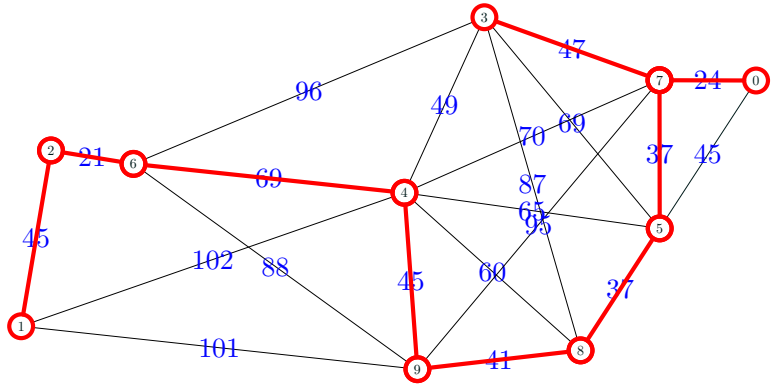
1. Minimum Spanning Tree
2. Prim's Algorithm
3. **Kruskal's Algorithm**
4. Shortest Path





## Kruskal's Algorithm

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



## Pseudo Code

```
KRUSKAL(G = (V, E, w))
{
  PQ.initialise()
  for edge ∈ |E|
    PQ.add( (w_edge, edge) )
  endfor

  E_T ← ∅
  noEdgesAccepted ←0

  while (noEdgesAccepted < |V| − 1)
    edge ←PQ.getMin()
    if E_T ∪ {edge} is acyclic
      E_T ←E_T ∪ {edge}
      noEdgesAccepted ←noEdgesAccepted +1
    endif
  endwhile

  return 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



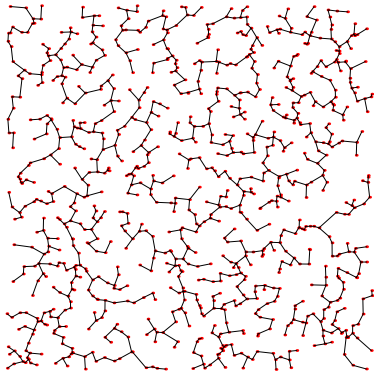
## Cycling

- 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**

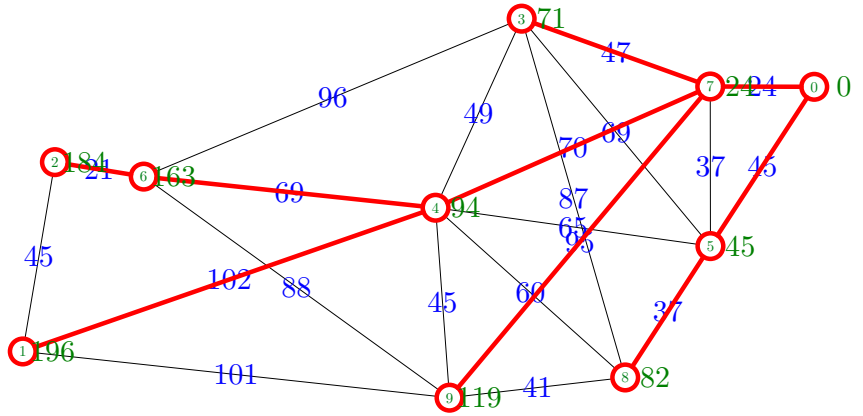


# Shortest path

- 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 = (V, E, w), source) {
  for i←0 to |V|
    d_i ←∞          \\ Minimum 'distance' to source
  endfor
  E_T ←∅            \\ Set of edges in subtree
  PQ.initialise()   \\ initialise an empty priority queue
  node ←source
  d_node ←0
  for i←1 to |V| − 1
    for neigh ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,neigh + d_node < d_neigh )
        d_neigh ←w_node,neigh + d_node
        PQ.add( (d_neigh, (node, neigh)) )
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    while next_node not in subtree
    E_T ←E_T ∪ {(a_node, next_node)}
    node ←next_node
  endfor
  return E_T
}
```



# Compare to Prim's Algorithm

```
PRIM(G = (V, E, w)) {
  for i←1 to |V|
    d_i ←∞          \\ Minimum 'distance' to subtree
  endfor
  E_T ←∅            \\ Set of edges in subtree
  PQ.initialise()   \\ initialise an empty priority queue
  node ← v_1        \\ where v_1 ∈ V is arbitrary
  for i←1 to |V| − 1
    d_node ←0
    for neigh ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,neigh < d_enigh )
        d_neigh ←w_node,neigh
        PQ.add( (d_neigh, (node,neigh)) )
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    until (d_next_node > 0)
    E_T ←E_T ∪ {(a_node, next_node)}
    node ←next_node
  endfor
  return 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