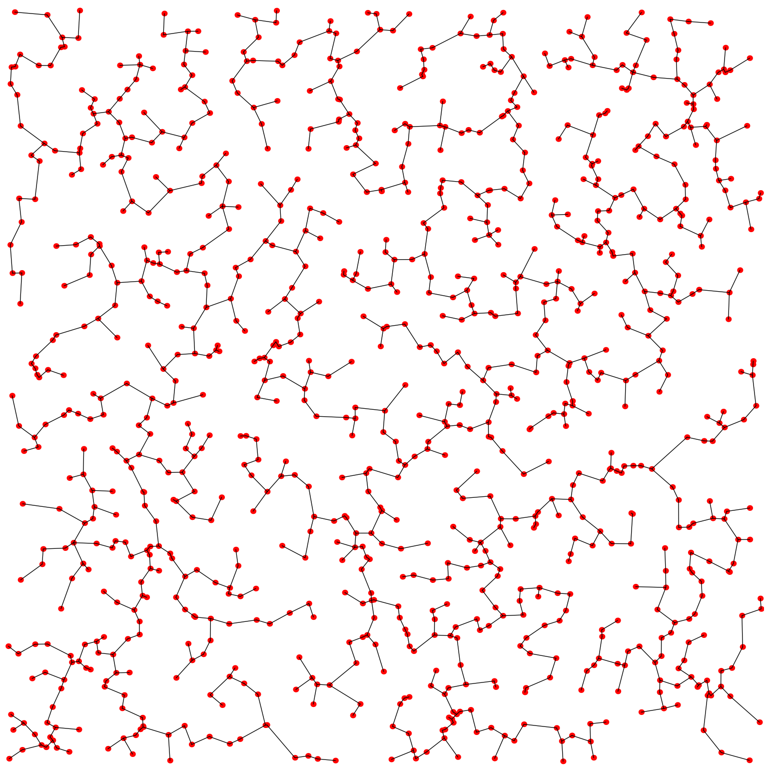
Algorithms and Analysis

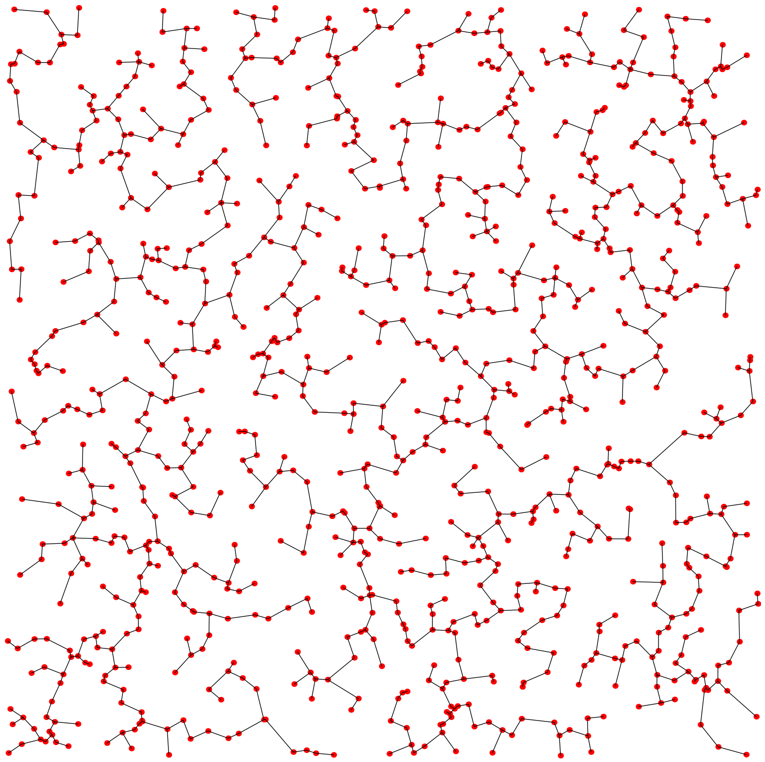
Lesson 21: *Know Your Graph Algorithms*



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. 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

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

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

Graph Algorithms

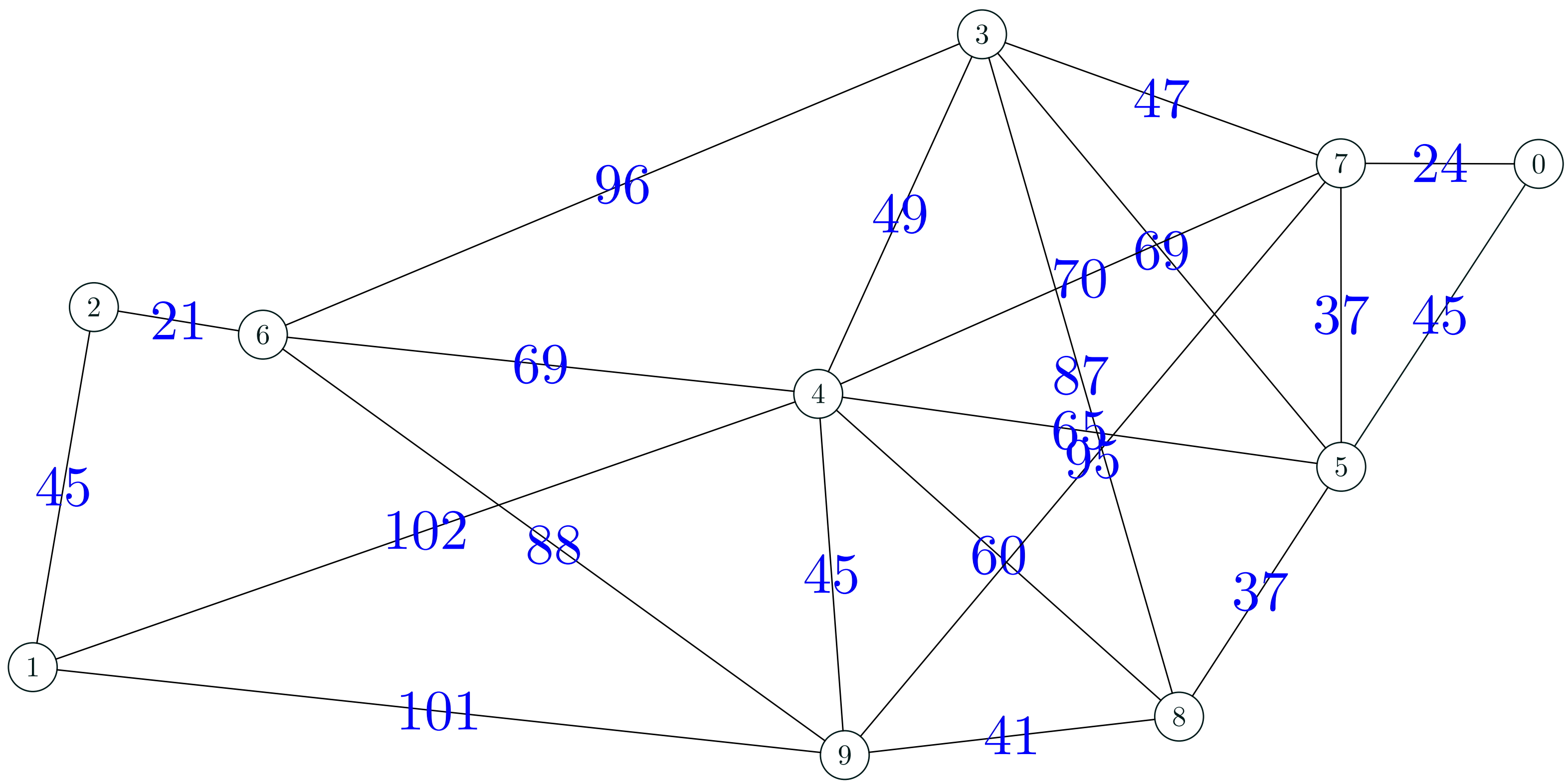
- We consider a graph algorithm to be **efficient** if it can solve a graph problem in $O(n^a)$ time for some fixed a
- That is, an efficient algorithm runs in polynomial time
- A problem is **hard** if there is no known efficient algorithm
- This does **not** mean the best we can do is to look through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

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

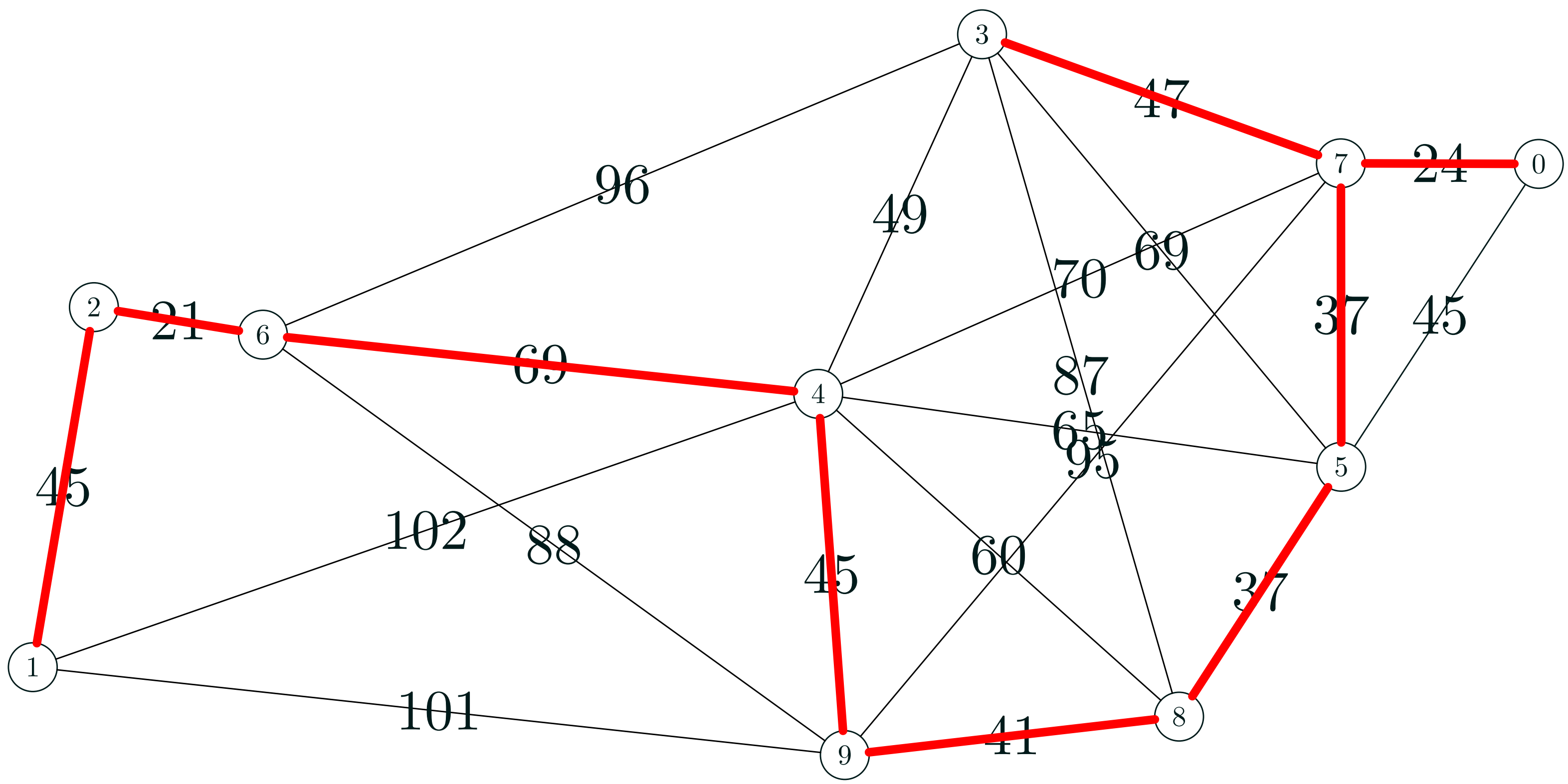
Minimum spanning tree

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



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

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

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

Greedy Strategy

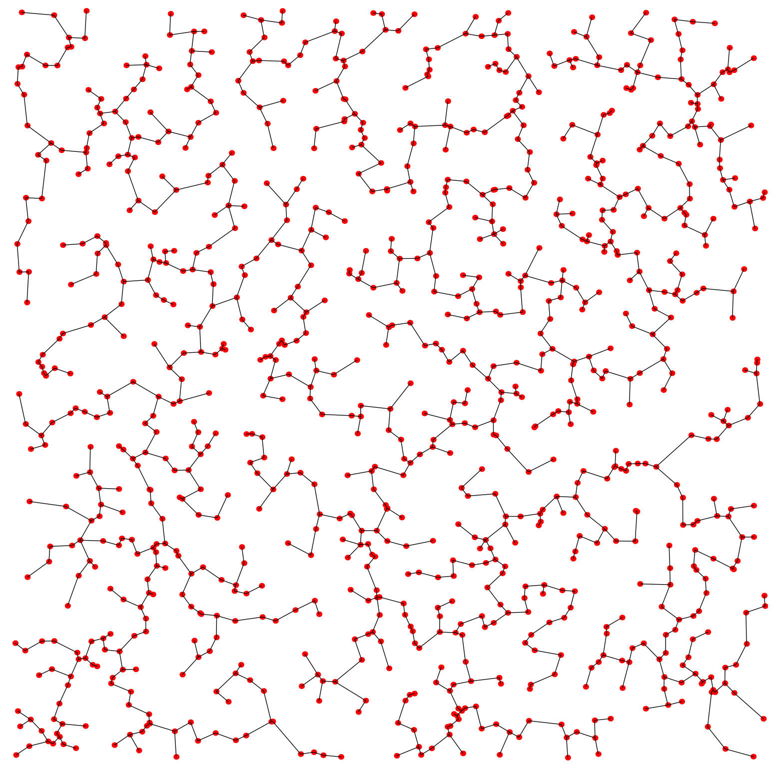
- We consider two algorithms for solving the problem
  - ⋆ Prim's algorithm (discovered 1957)
  - ⋆ Kruskal's algorithm (discovered 1956)
- Both algorithms use a **greedy strategy**
- Generally greedy strategies are not guaranteed to give globally optimal solutions
- There exists a class of problems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

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

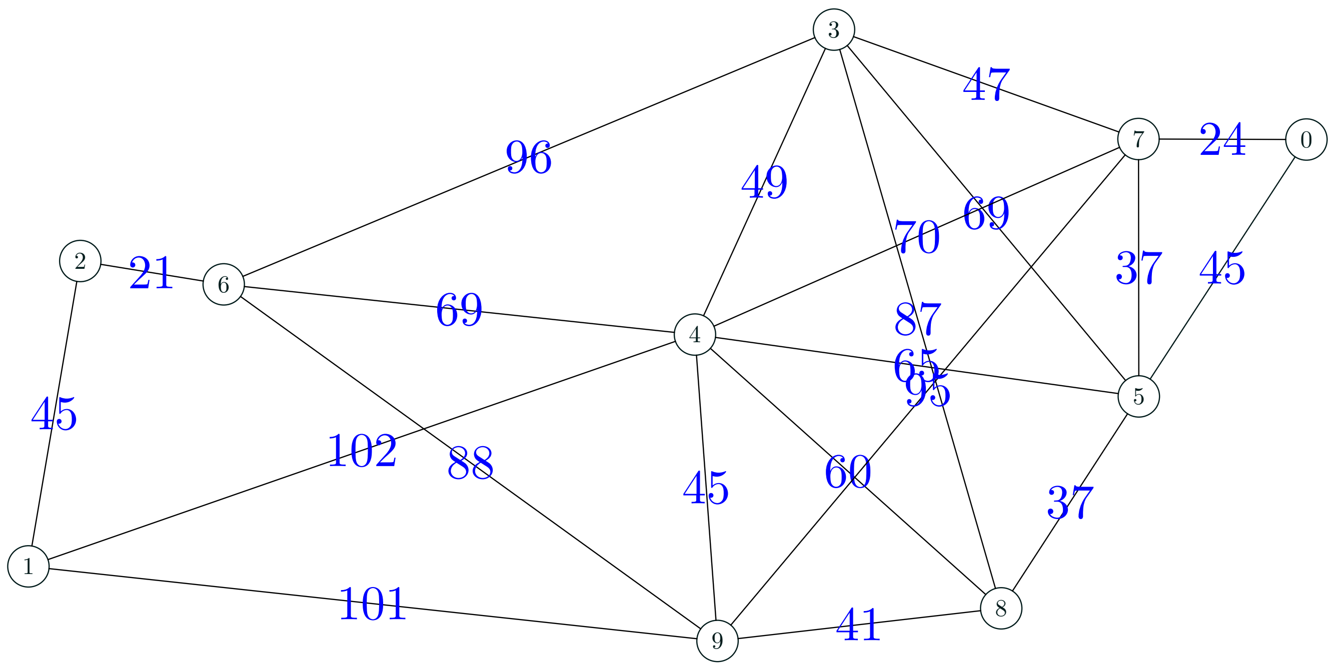
Outline

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



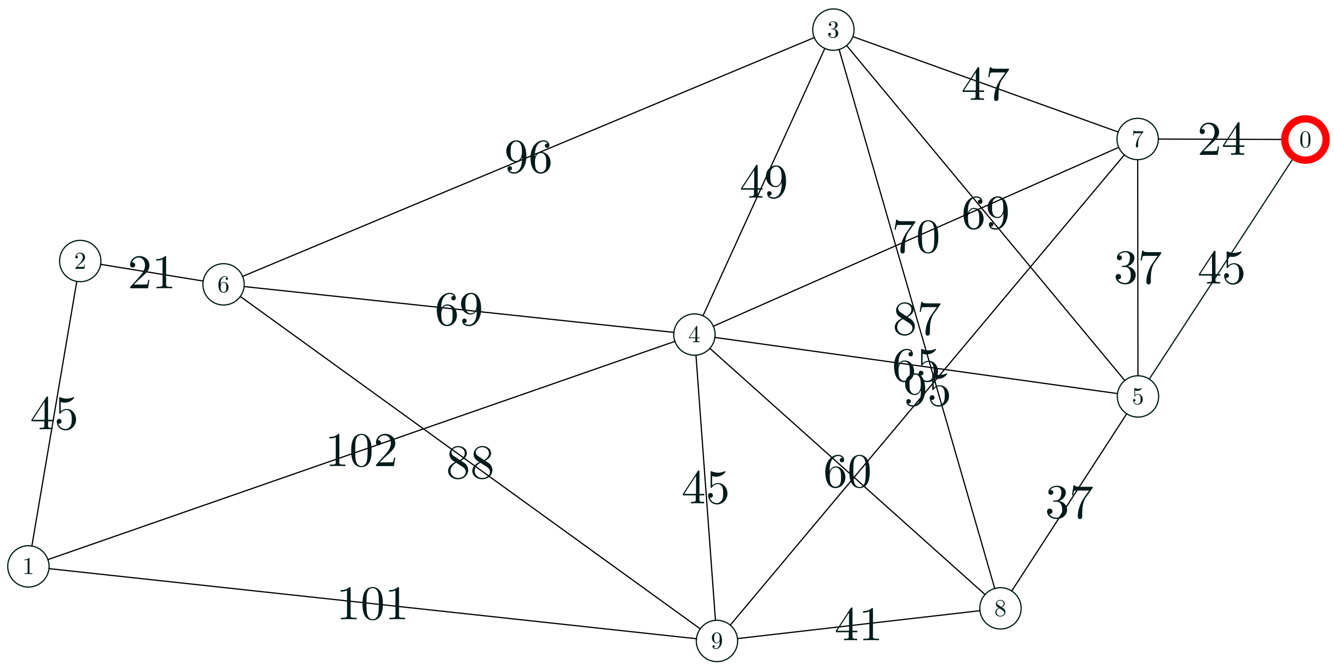
Prim's Algorithm

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



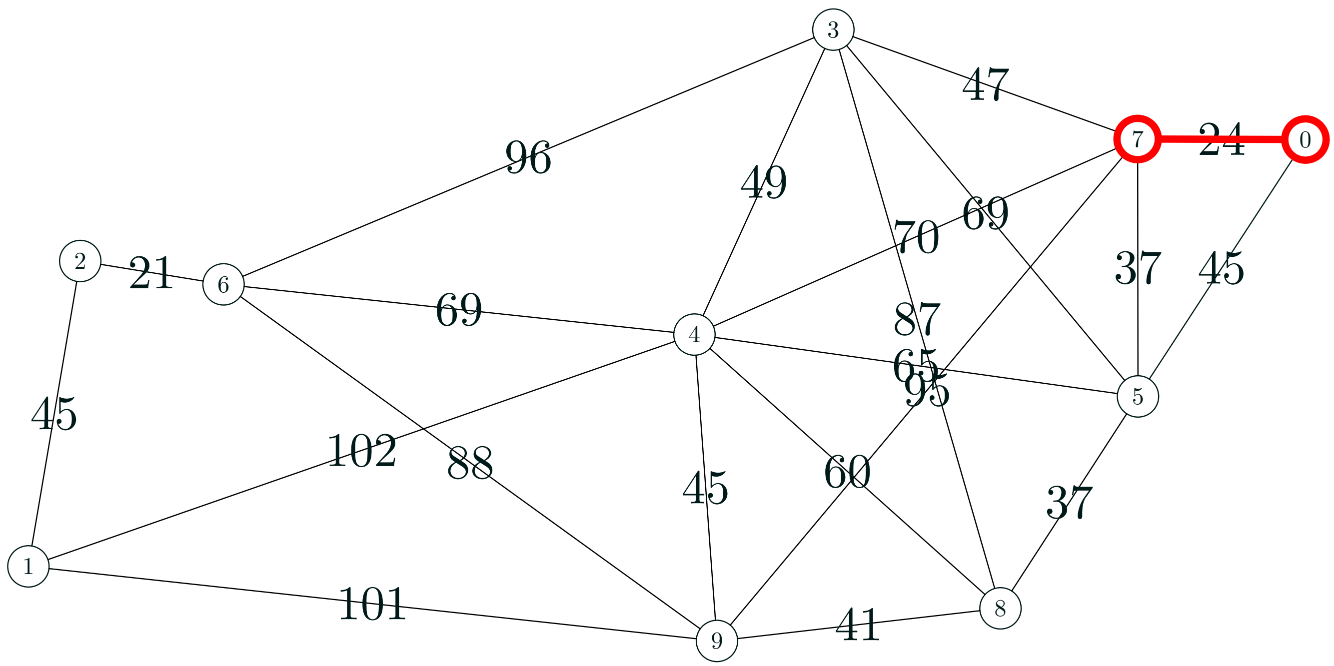
Prim's Algorithm

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



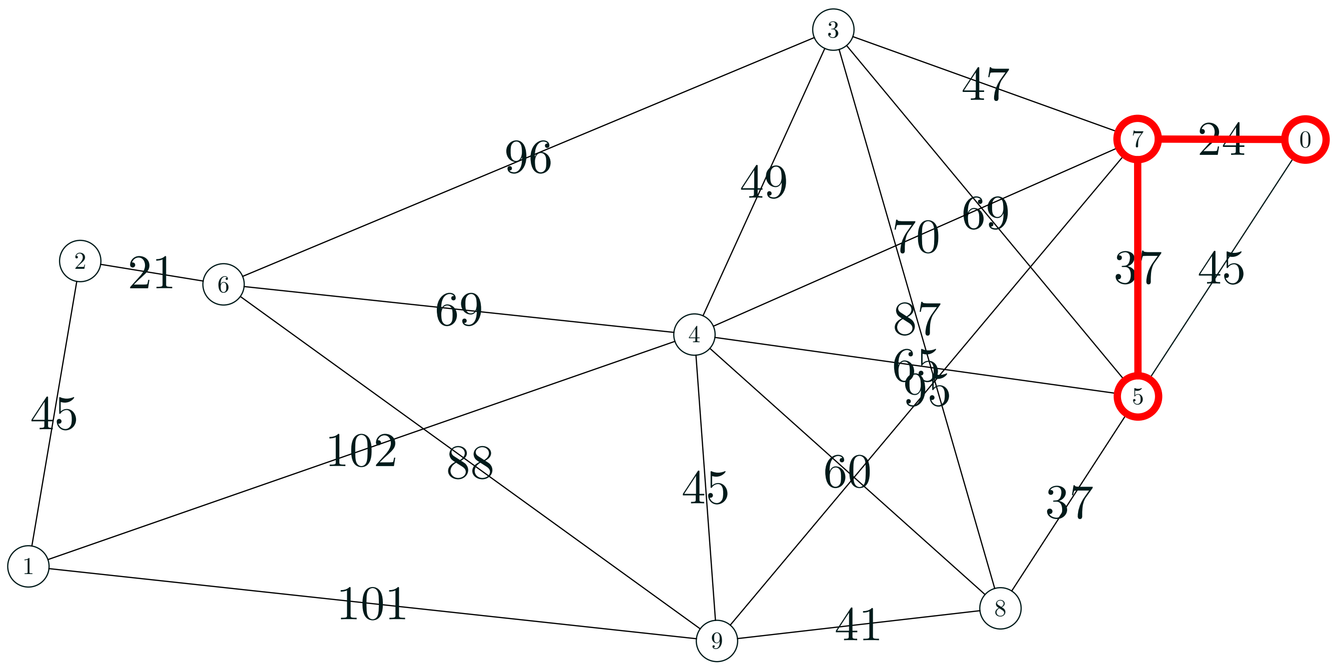
Prim's Algorithm

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



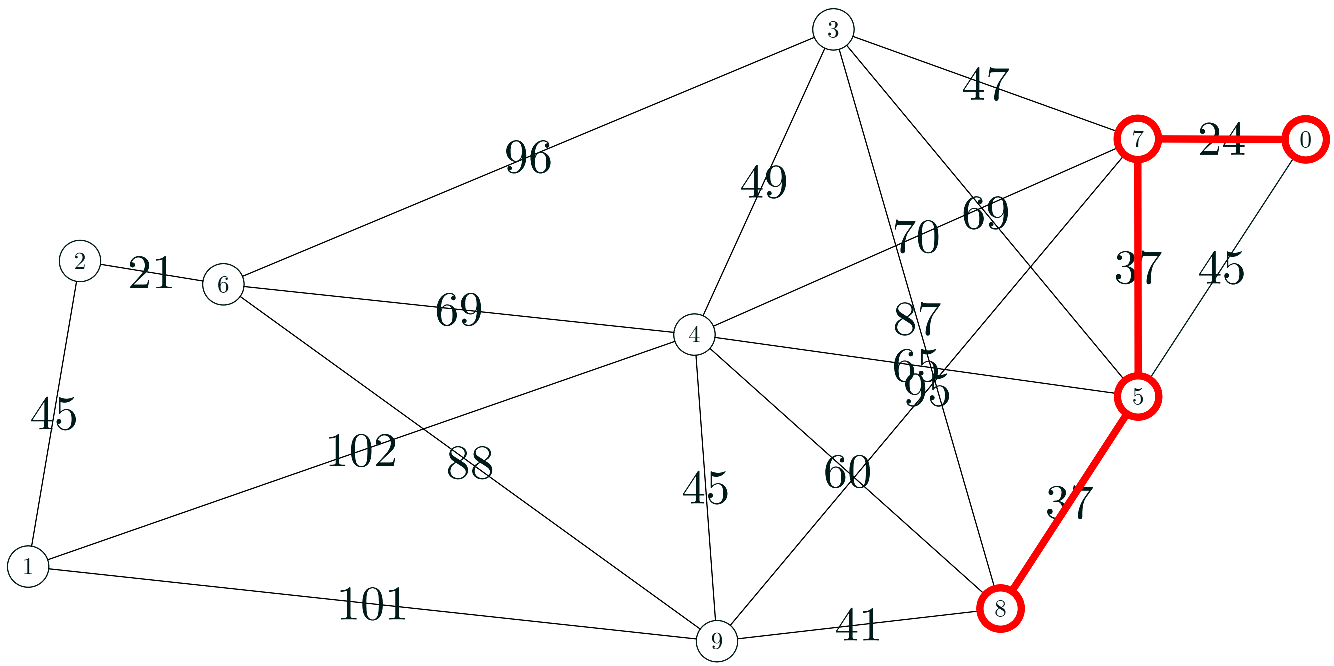
Prim's Algorithm

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



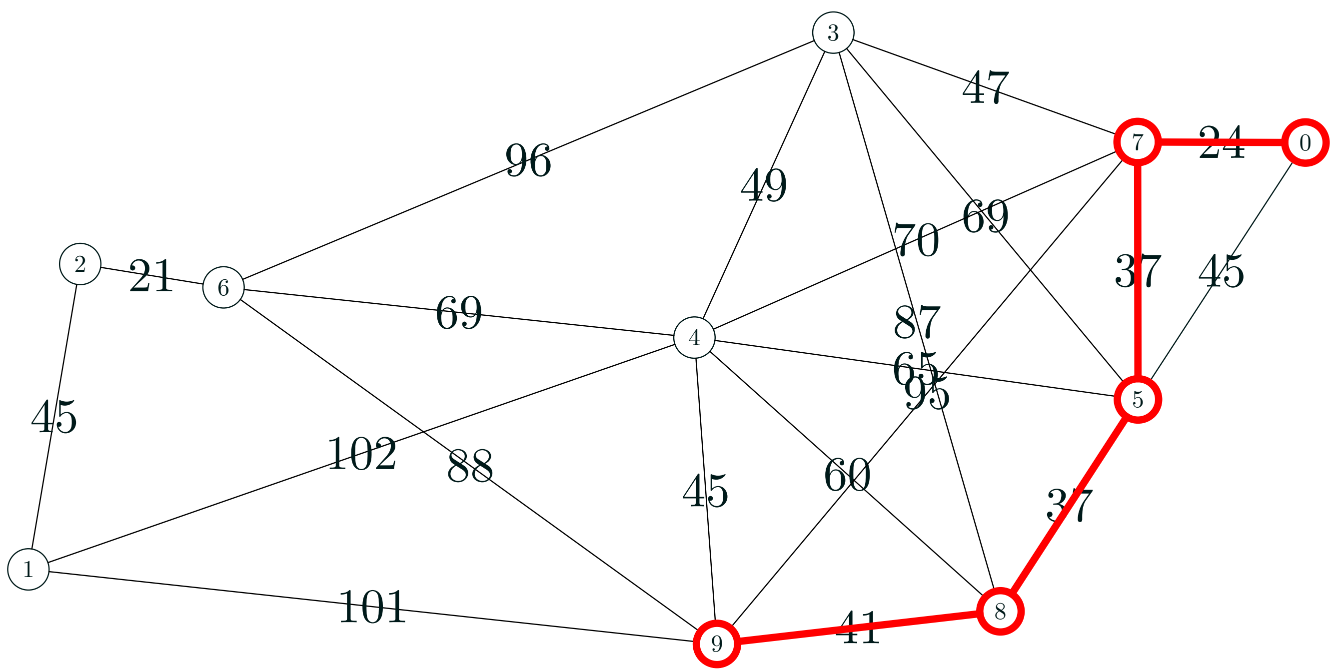
Prim's Algorithm

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



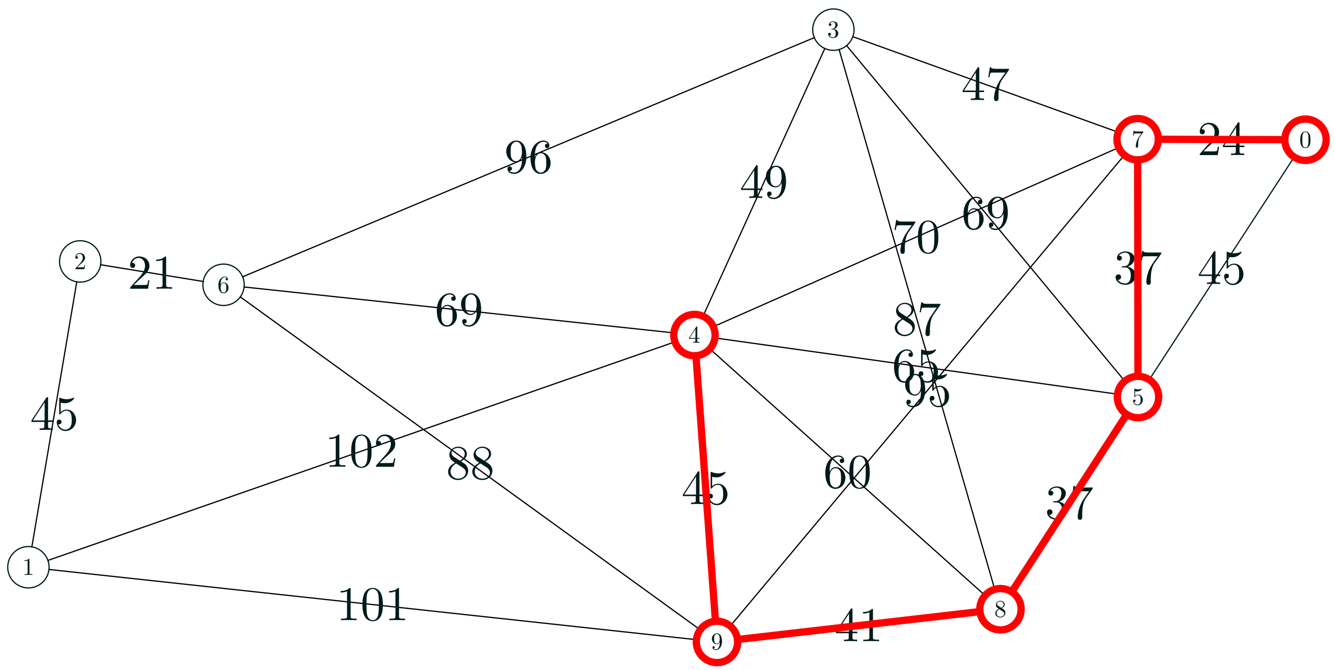
Prim's Algorithm

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



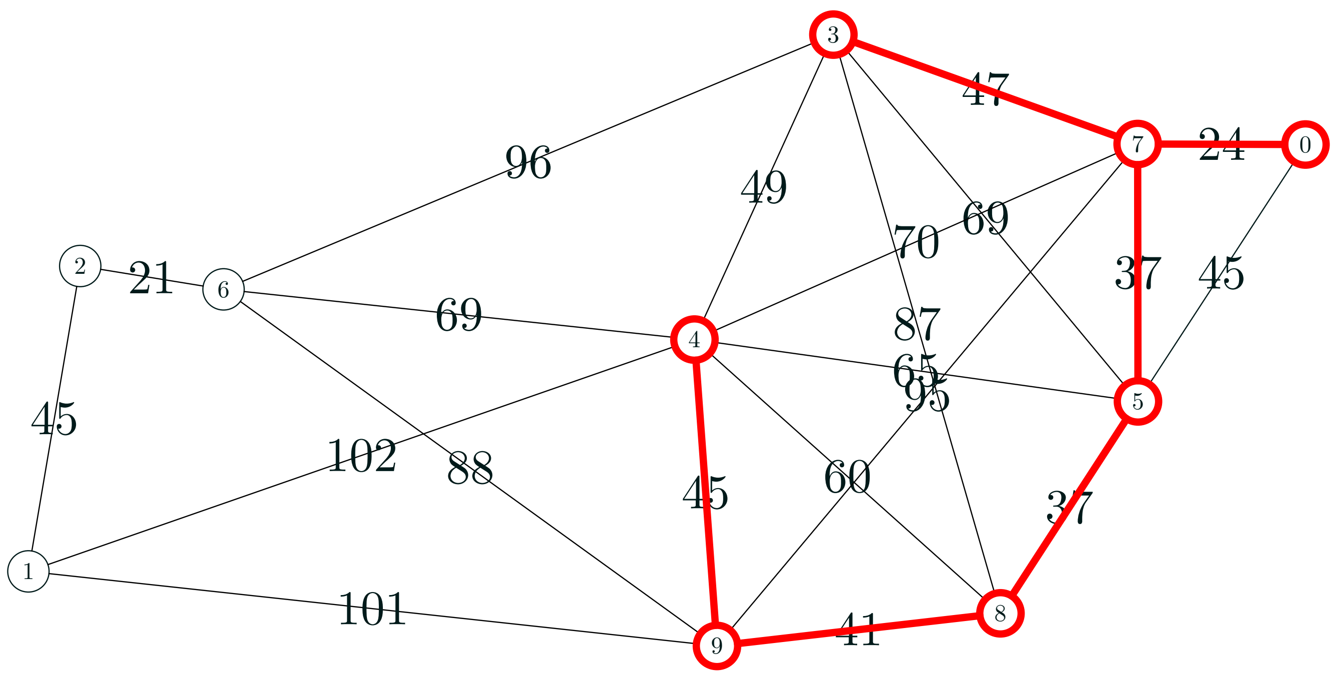
Prim's Algorithm

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



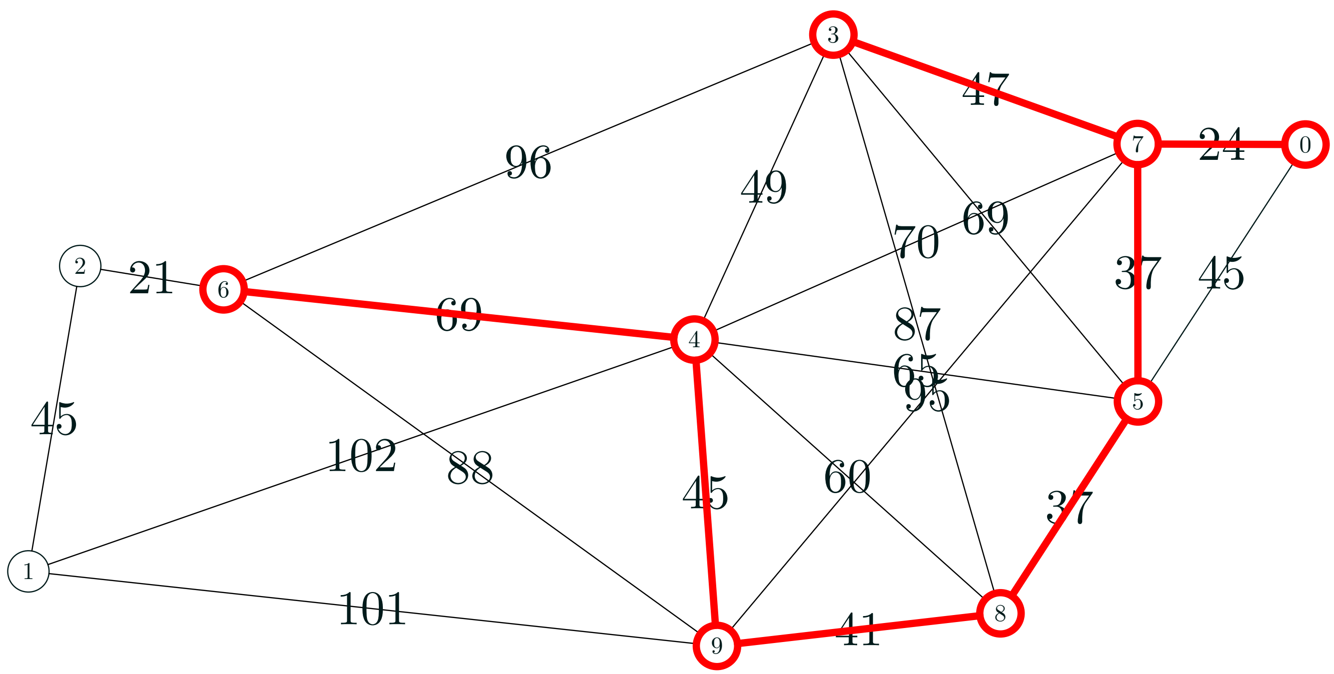
Prim's Algorithm

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



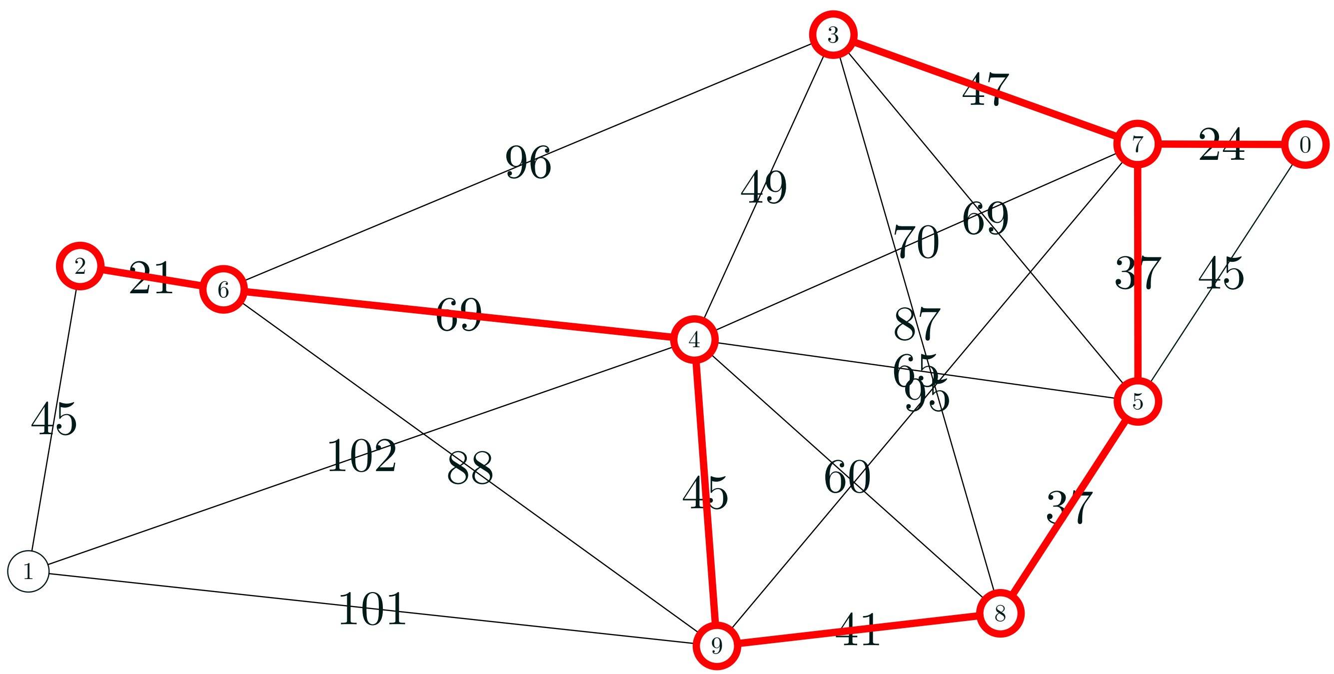
Prim's Algorithm

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



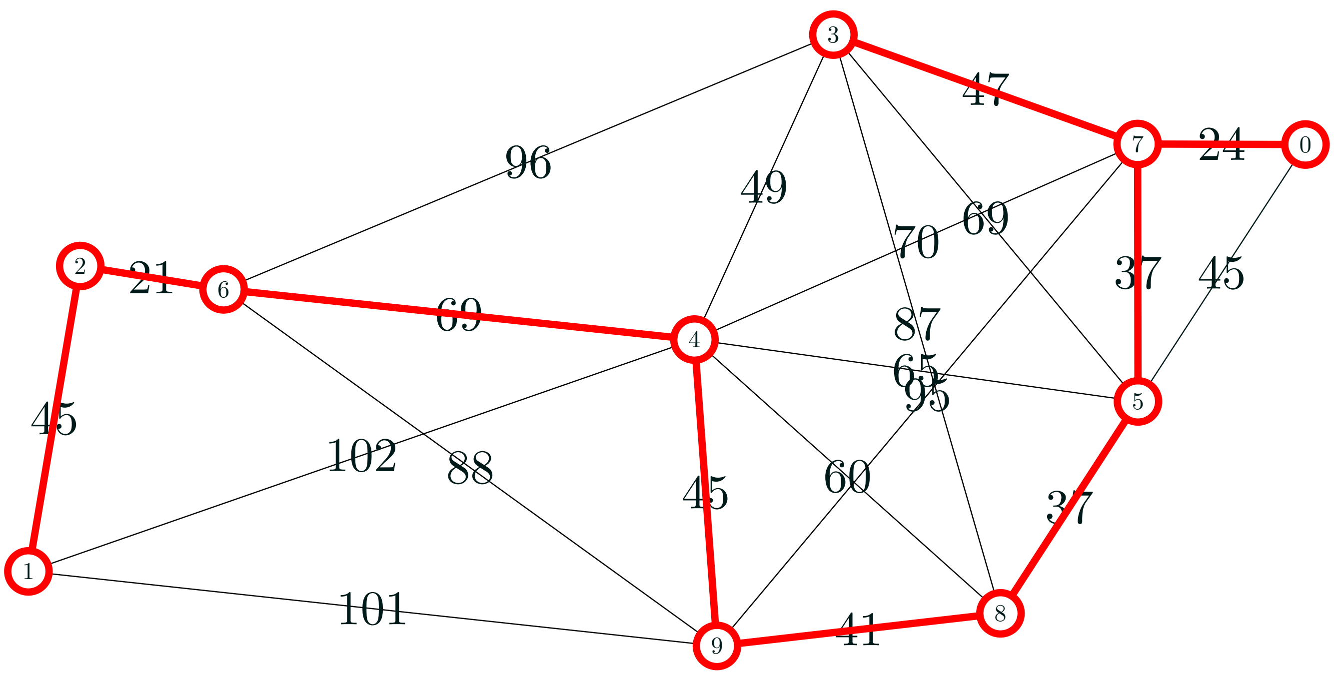
Prim's Algorithm

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



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 k ∈ {v ∈ V|(node, v) ∈ E} \\ k  is a neighbours of node
         if ( w_node,k < d_k  )
            d_k ←w_node,k
            PQ.add( (d_k, (node,k)) )
         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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E} \\ k is a neighbours of node
      if ( w_node,k < d_k )
        d_k ←w_node,k
        PQ.add( (d_k, (node,k)) )
      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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E} \\ k is a neighbours of node
      if ( w_node,k < d_k )
        d_k ←w_node,k
        PQ.add( (d_k, (node,k)) )
      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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E} \\ k  is a neighbours of node
      if ( w_node,k < d_k  )
        d_k ←w_node,k
        PQ.add( (d_k, (node,k)) )
      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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E} \\ k  is a neighbours of node
      if ( w_node,k < d_k  )
        d_k ←w_node,k
        PQ.add( (d_k, (node,k)) )
      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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E} \\ k  is a neighbours of node
         if ( w_node,k < d_k  )
            d_k ←w_node,k
            PQ.add( (d_k, (node,k)) )
         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
}
```

Pseudo Code

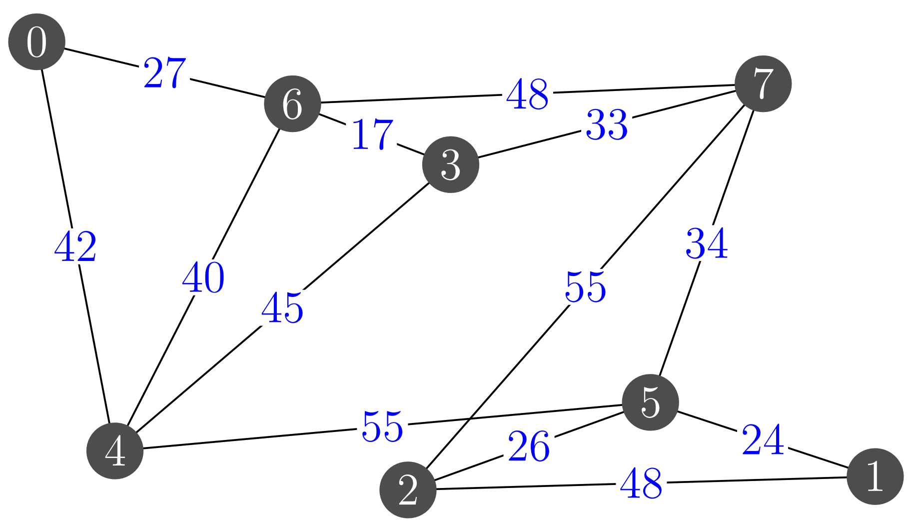
```
PRIM(G = (𝒱, ℰ, w)) {
  for i←1 to |𝒱|
    d_i ←∞              \\ Minimum 'distance' to subtree
  endfor
  ℰ_T ←∅                \\ Set of edges in subtree
  PQ.initialise()       \\ initialise an empty priority queue
  node ← v_1            \\ where v_1 ∈ 𝒱 is arbitrary
  for i←1 to |𝒱| − 1
    d_node ←0
    for k ∈ {v ∈ 𝒱|(node, v) ∈ ℰ} \\ k is a neighbours of node
      if ( w_node,k < d_k )
        d_k ←w_node,k
        PQ.add( (d_k, (node,k)) )
      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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E} \\ k  is a neighbours of node
      if ( w_node,k < d_k  )
        d_k ←w_node,k
        PQ.add( (d_k, (node,k)) )
      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

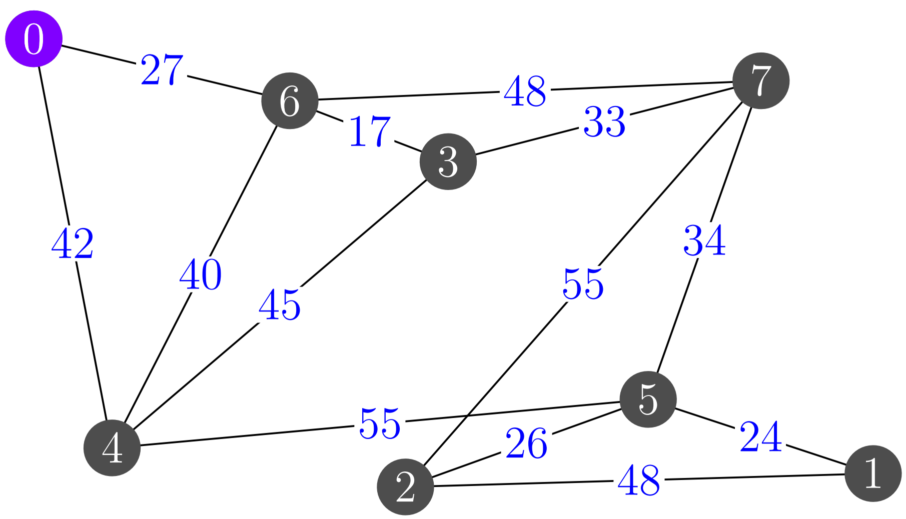
	0	1	2	3	4	5	6	7
`d[]`	∞	∞	∞	∞	∞	∞	∞	∞



Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
`d[]`	0	∞	∞	∞	∞	∞	∞	∞

node=0



Prim's Algorithm in Detail

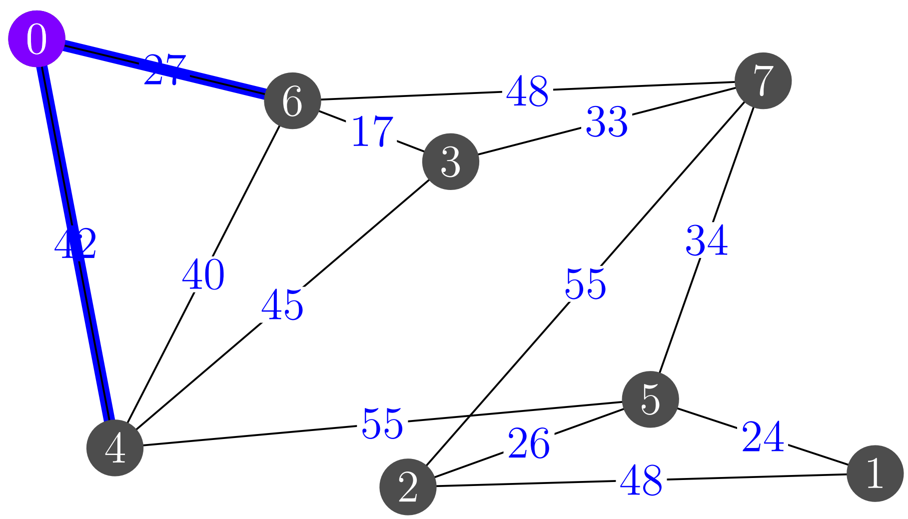
	0	1	2	3	4	5	6	7
d[]	0	∞	∞	∞	42	∞	27	∞

neighbours of node 0 added to PQ

node=0

PQ (27, (0,6))

(42, (0,4))



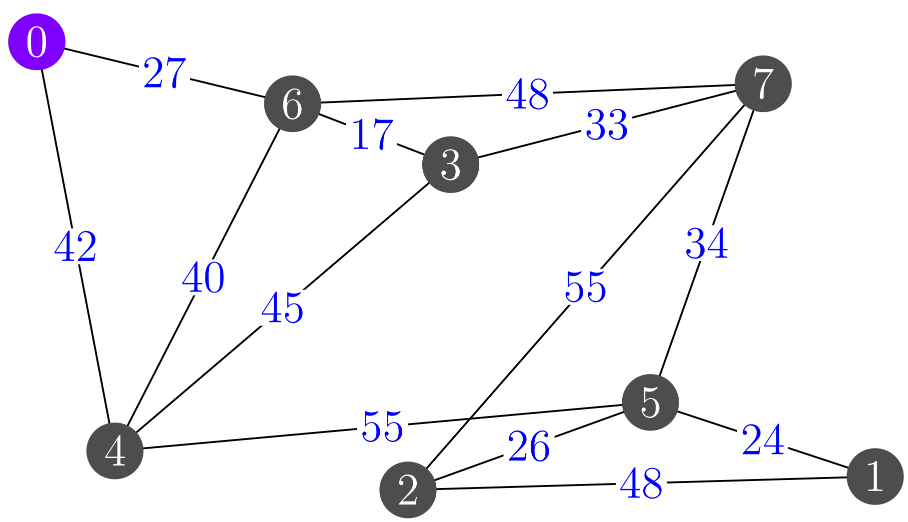
Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
d[]	0	∞	∞	∞	42	∞	27	∞

node=0

PQ (27, (0,6)) → nearest node=6

(42, (0,4))



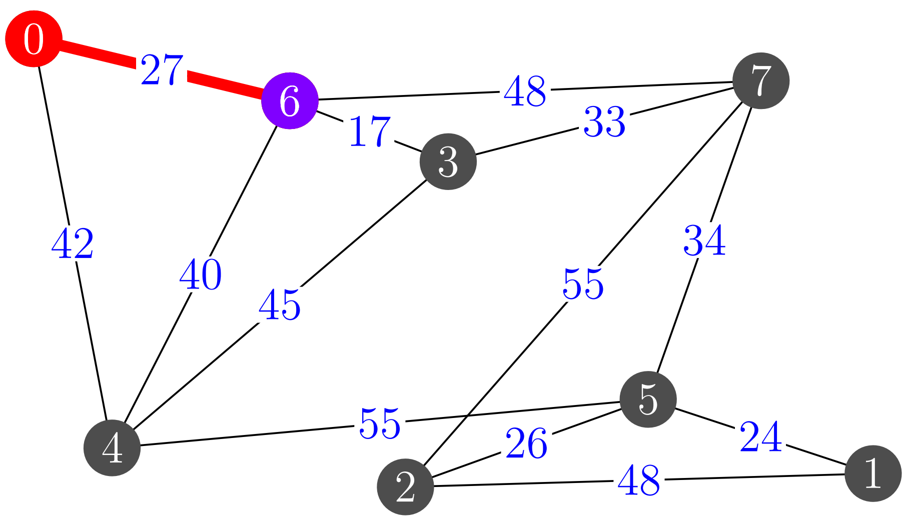
Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
d[]	0	∞	∞	∞	42	∞	0	∞

add edge (0,6) to MST

node=6

PQ (42, (0,4))

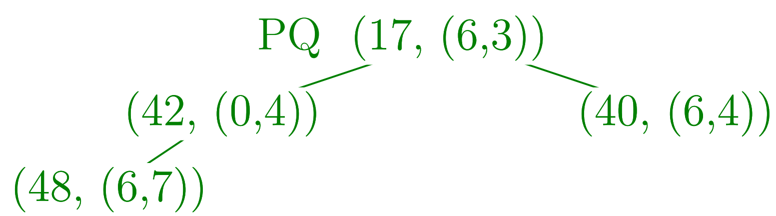


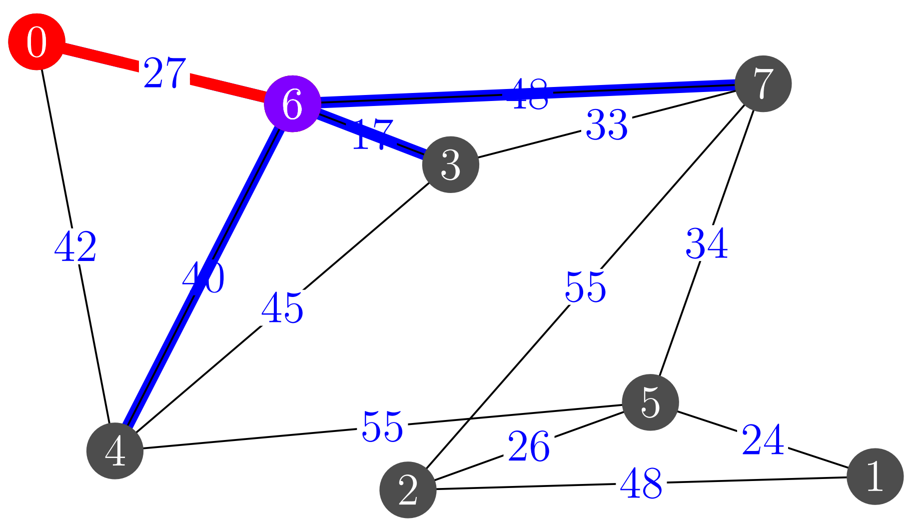
Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
d[]	0	∞	∞	17	40	∞	0	48

neighbours of node 6 added to PQ

node=6





Prim's Algorithm in Detail

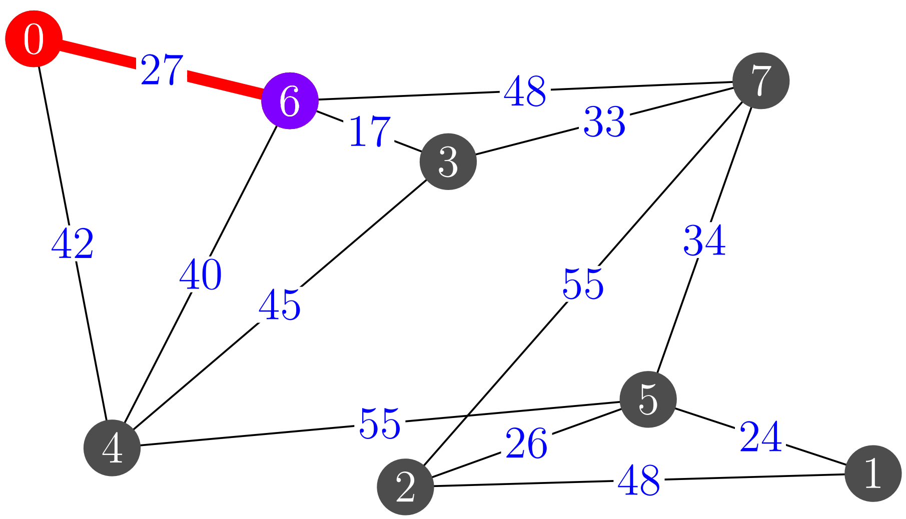
	0	1	2	3	4	5	6	7
`d[]`	0	∞	∞	17	40	∞	0	48

node=6

PQ (17, (6,3)) → nearest node=3

(42, (0,4)) (40, (6,4))

(48, (6,7))



Prim's Algorithm in Detail

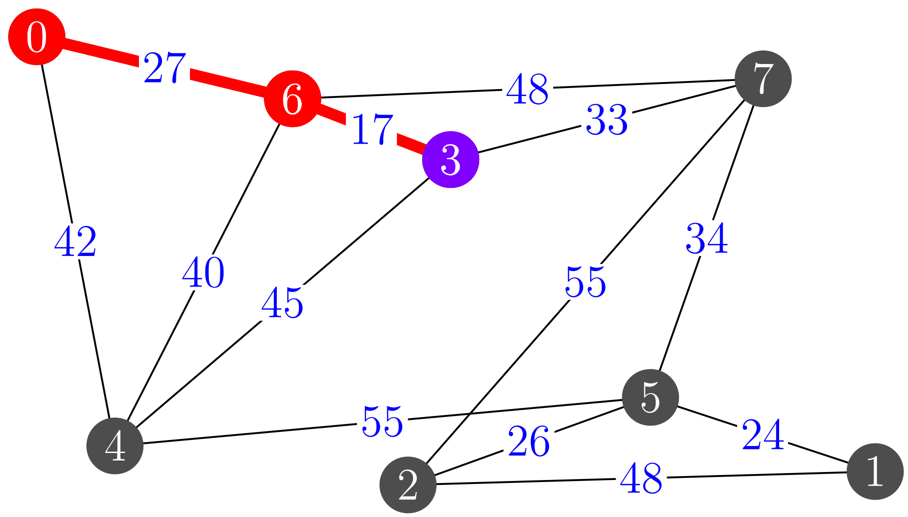
	0	1	2	3	4	5	6	7
`d[]`	0	∞	∞	0	40	∞	0	48

add edge (6,3) to MST

node=3

PQ (40, (6,4))

(42, (0,4)) (48, (6,7))



Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
`d[]`	0	∞	∞	0	40	∞	0	33

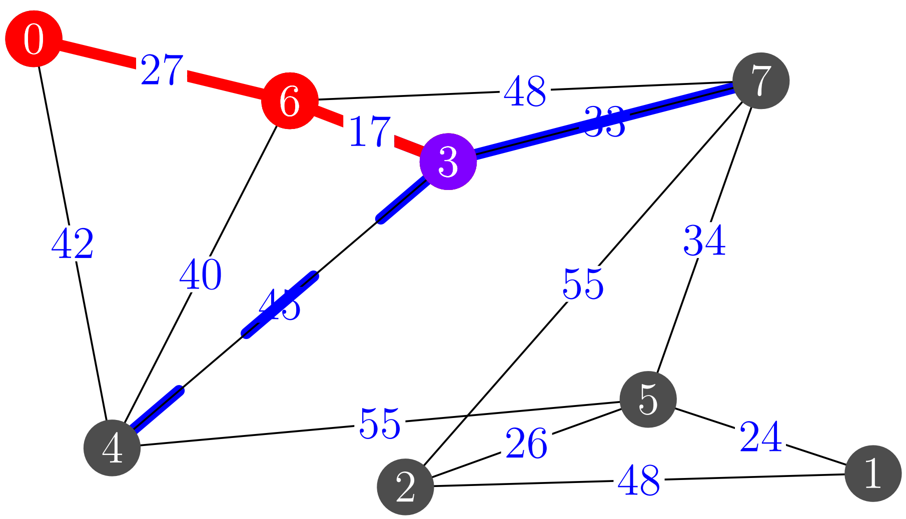
neighbours of node 3 added to PQ

node=3

PQ (33, (3,7))

(40, (6,4)) (48, (6,7))

(42, (0,4))



Prim's Algorithm in Detail

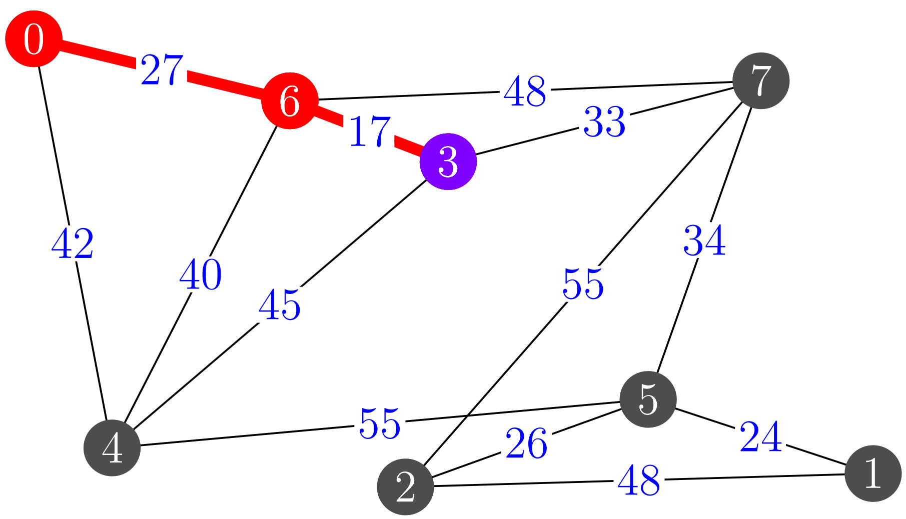
	0	1	2	3	4	5	6	7
d[]	0	∞	∞	0	40	∞	0	33

node=3

PQ (33, (3,7)) $\longrightarrow$ nearest node=7

(40, (6,4)) (48, (6,7))

(42, (0,4))



Prim's Algorithm in Detail

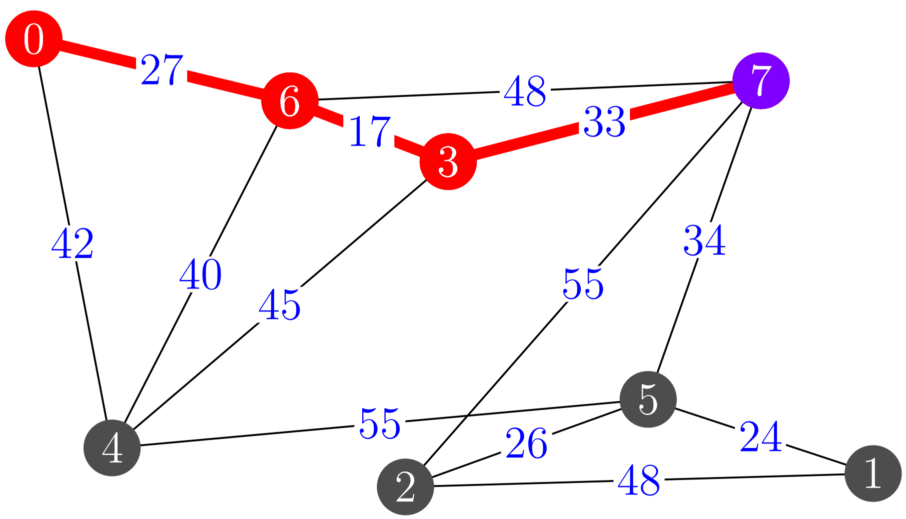
	0	1	2	3	4	5	6	7
`d[]`	0	∞	∞	0	40	∞	0	0

add edge (3,7) to MST

node=7

PQ (40, (6,4))

(42, (0,4)) (48, (6,7))



Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
`d[]`	0	∞	55	0	40	34	0	0

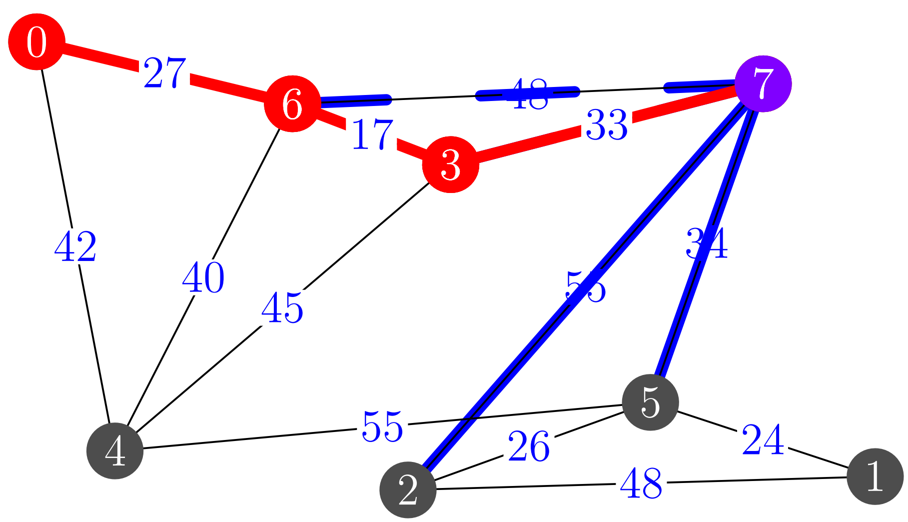
neighbours of node 7 added to PQ

node=7

PQ (34, (7,5))

(40, (6,4)) (48, (6,7))

(55, (7,2)) (42, (0,4))



Prim's Algorithm in Detail

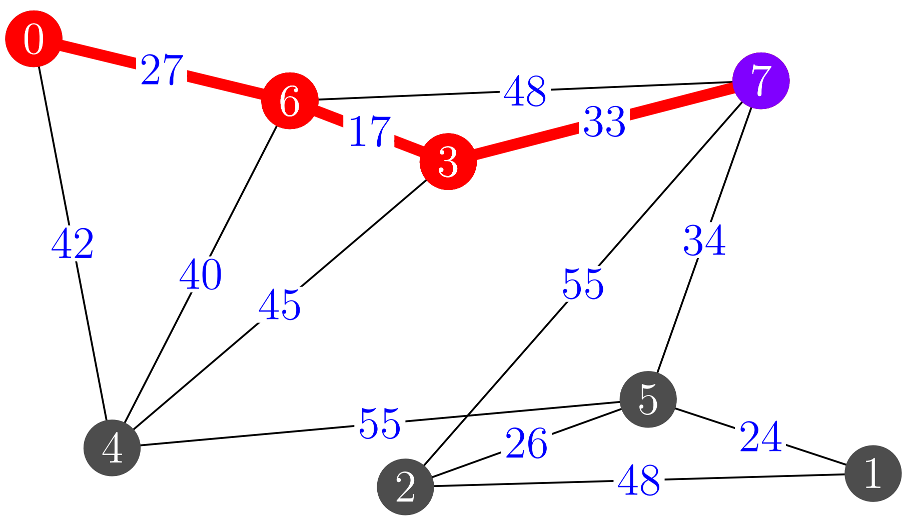
	0	1	2	3	4	5	6	7
`d[]`	0	∞	55	0	40	34	0	0

node=7

PQ (34, (7,5)) $\longrightarrow$ nearest node=5

(40, (6,4)) (48, (6,7))

(55, (7,2)) (42, (0,4))



Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
d[]	0	∞	55	0	40	0	0	0

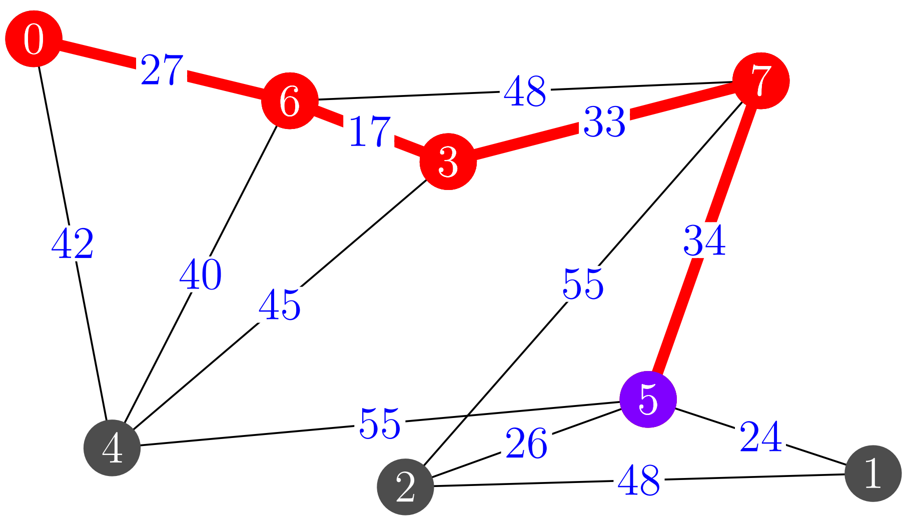
add edge (7,5) to MST

node=5

PQ (40, (6,4))

(42, (0,4)) (48, (6,7))

(55, (7,2))



Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
d[]	0	24	26	0	40	0	0	0

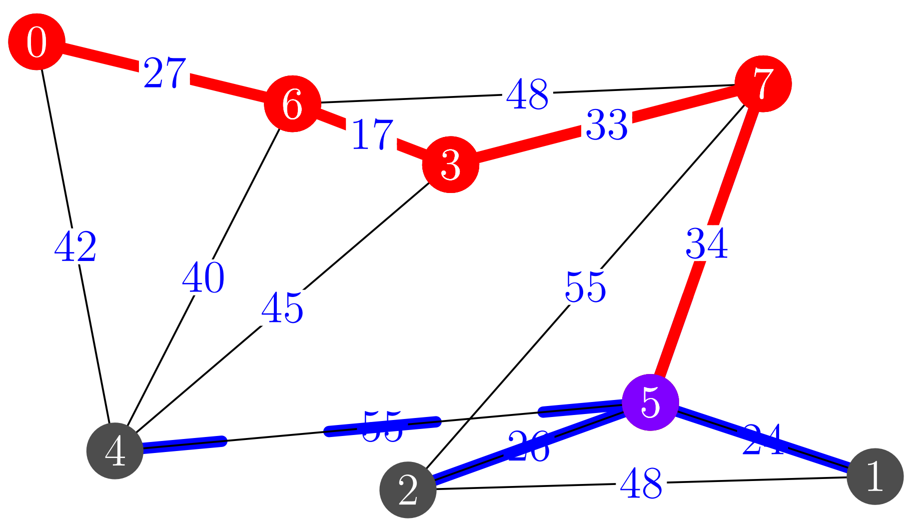
neighbours of node 5 added to PQ

node=5

PQ (24, (5,1))

(40, (6,4)) (26, (5,2))

(55, (7,2)) (42, (0,4)) (48, (6,7))



Prim's Algorithm in Detail

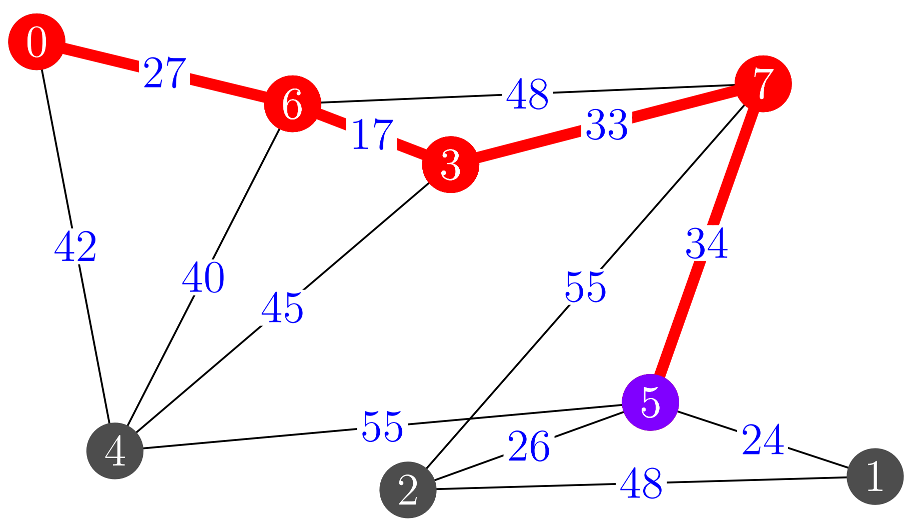
	0	1	2	3	4	5	6	7
`d[]`	0	24	26	0	40	0	0	0

node=5

PQ (24, (5,1)) → nearest node=1

(40, (6,4)) (26, (5,2))

(55, (7,2)) (42, (0,4)) (48, (6,7))

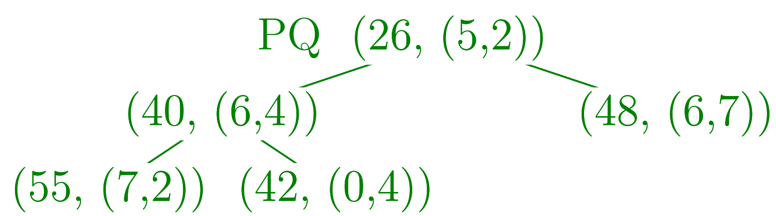


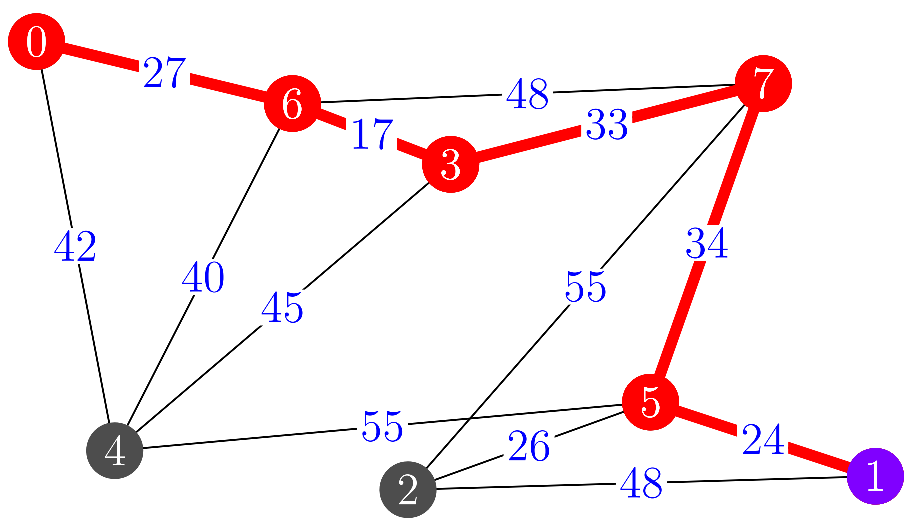
Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
`d[]`	0	0	26	0	40	0	0	0

add edge (5,1) to MST

node=1





Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
d[]	0	0	26	0	40	0	0	0

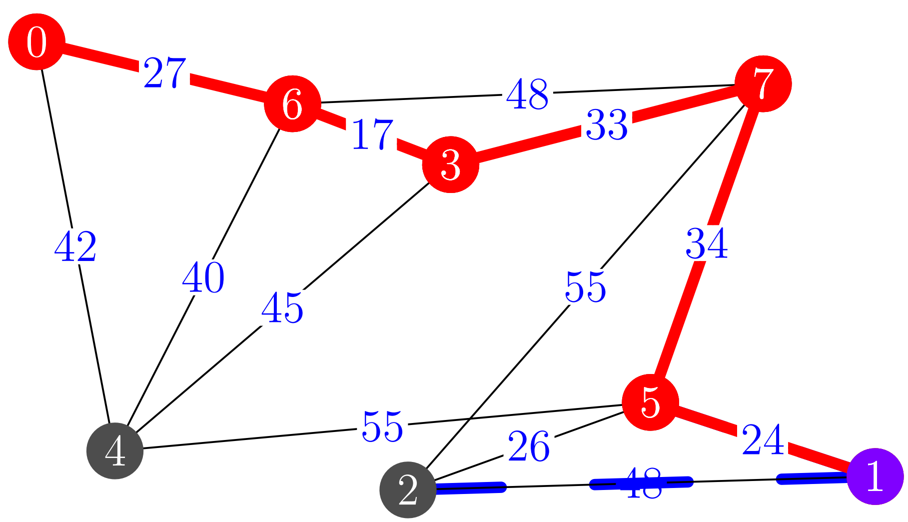
neighbours of node 1 added to PQ

node=1

PQ (26, (5,2))

(40, (6,4)) (48, (6,7))

(55, (7,2)) (42, (0,4))



Prim's Algorithm in Detail

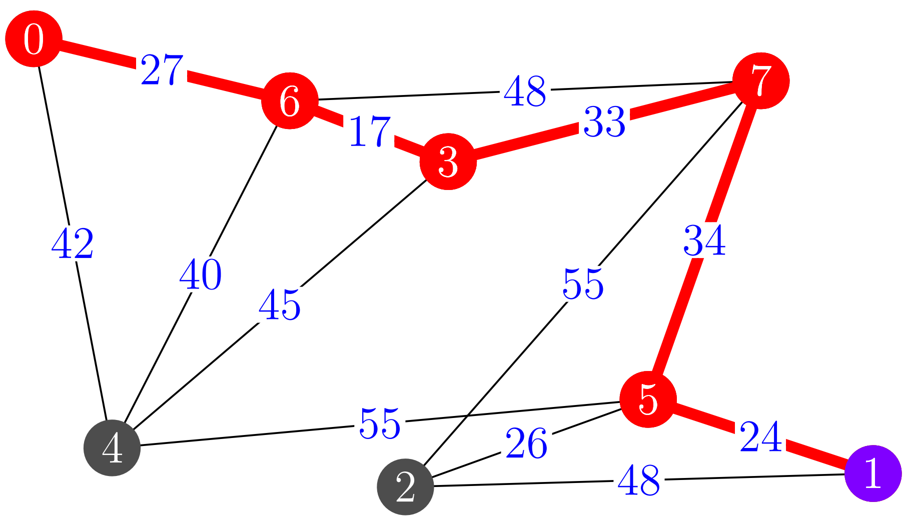
	0	1	2	3	4	5	6	7
d[]	0	0	26	0	40	0	0	0

node=1

PQ (26, (5,2)) → nearest node=2

(40, (6,4)) (48, (6,7))

(55, (7,2)) (42, (0,4))



Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
d[]	0	0	0	0	40	0	0	0

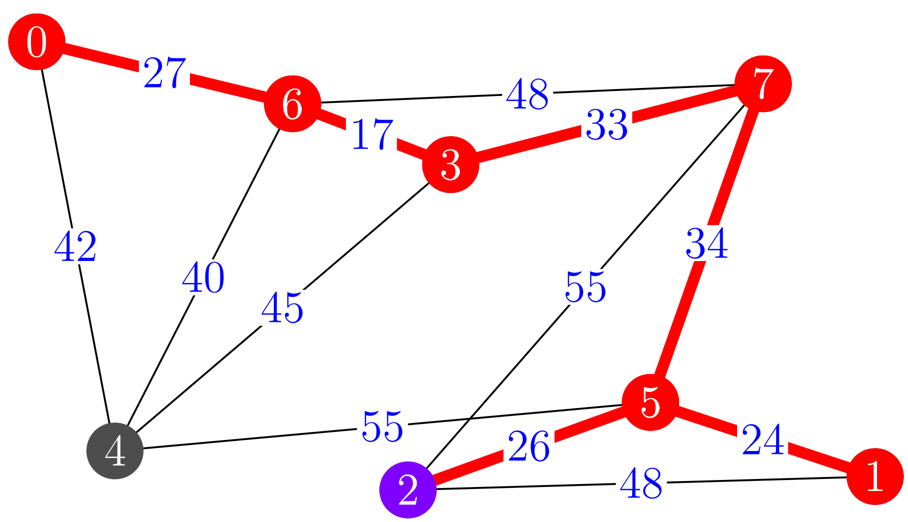
add edge (5,2) to MST

node=2

PQ (40, (6,4))

(42, (0,4)) (48, (6,7))

(55, (7,2))



Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
`d[]`	0	0	0	0	40	0	0	0

neighbours of node 2 added to PQ

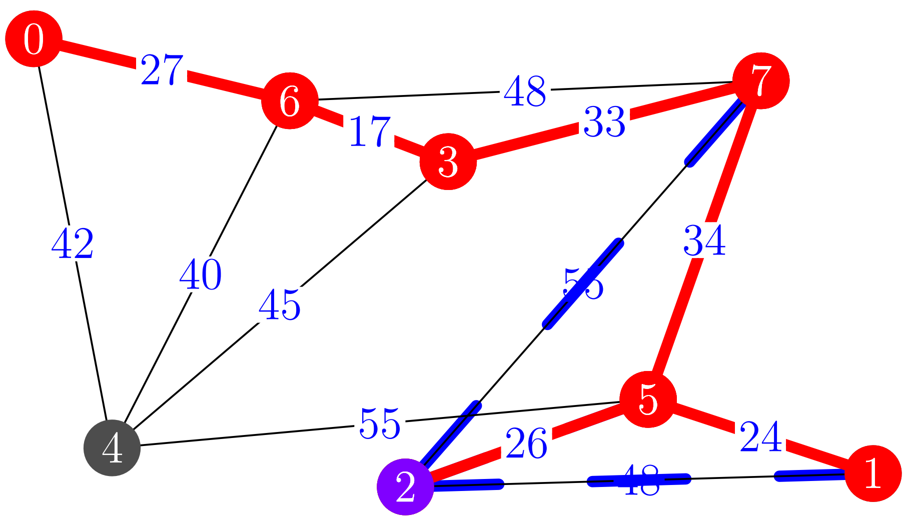
node=2

PQ (40, (6,4))

(42, (0,4))

(48, (6,7))

(55, (7,2))



Prim's Algorithm in Detail

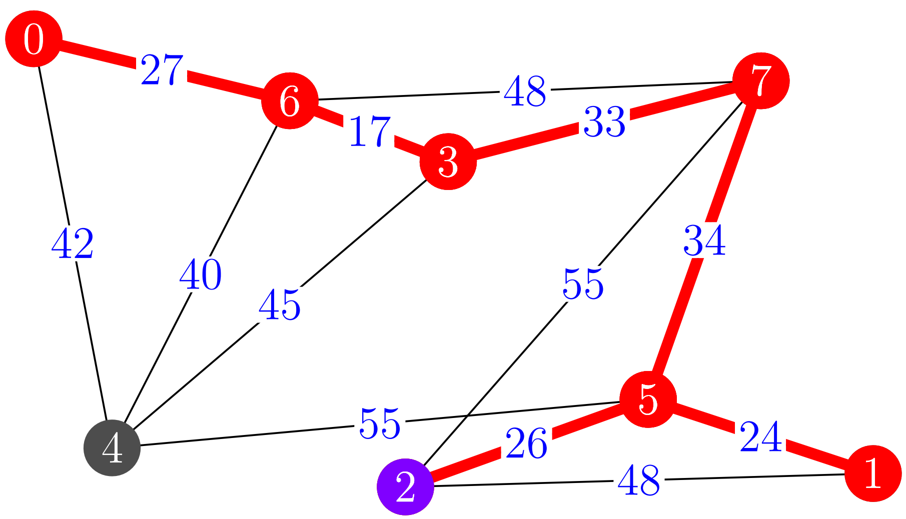
	0	1	2	3	4	5	6	7
`d[]`	0	0	0	0	40	0	0	0

node=2

PQ (40, (6,4)) → nearest node=4

(42, (0,4)) (48, (6,7))

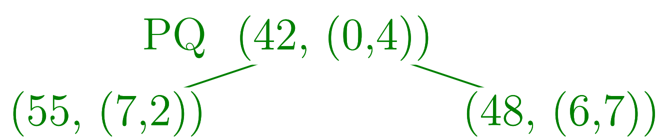
(55, (7,2))

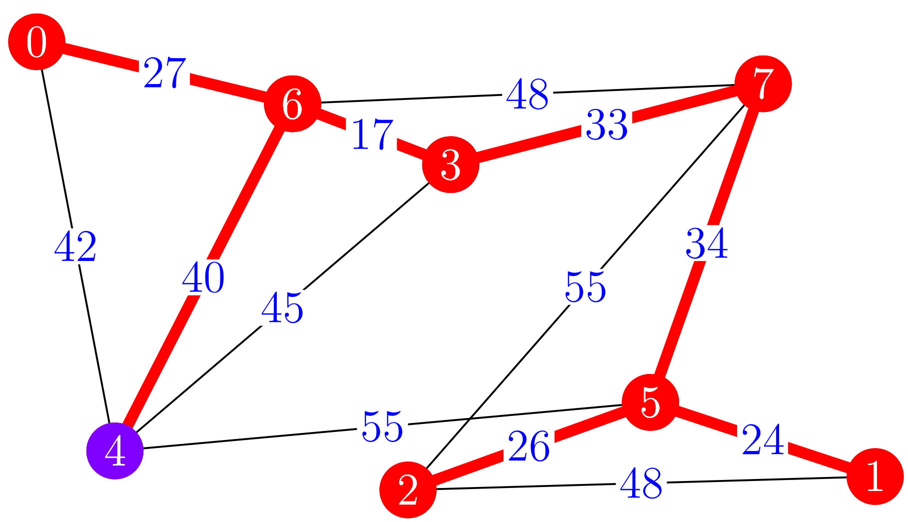


Prim's Algorithm in Detail

	0	1	2	3	4	5	6	7
d[]	0	0	0	0	40	0	0	0

add edge (6,4) to MST





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

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

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

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

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

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

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

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

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

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

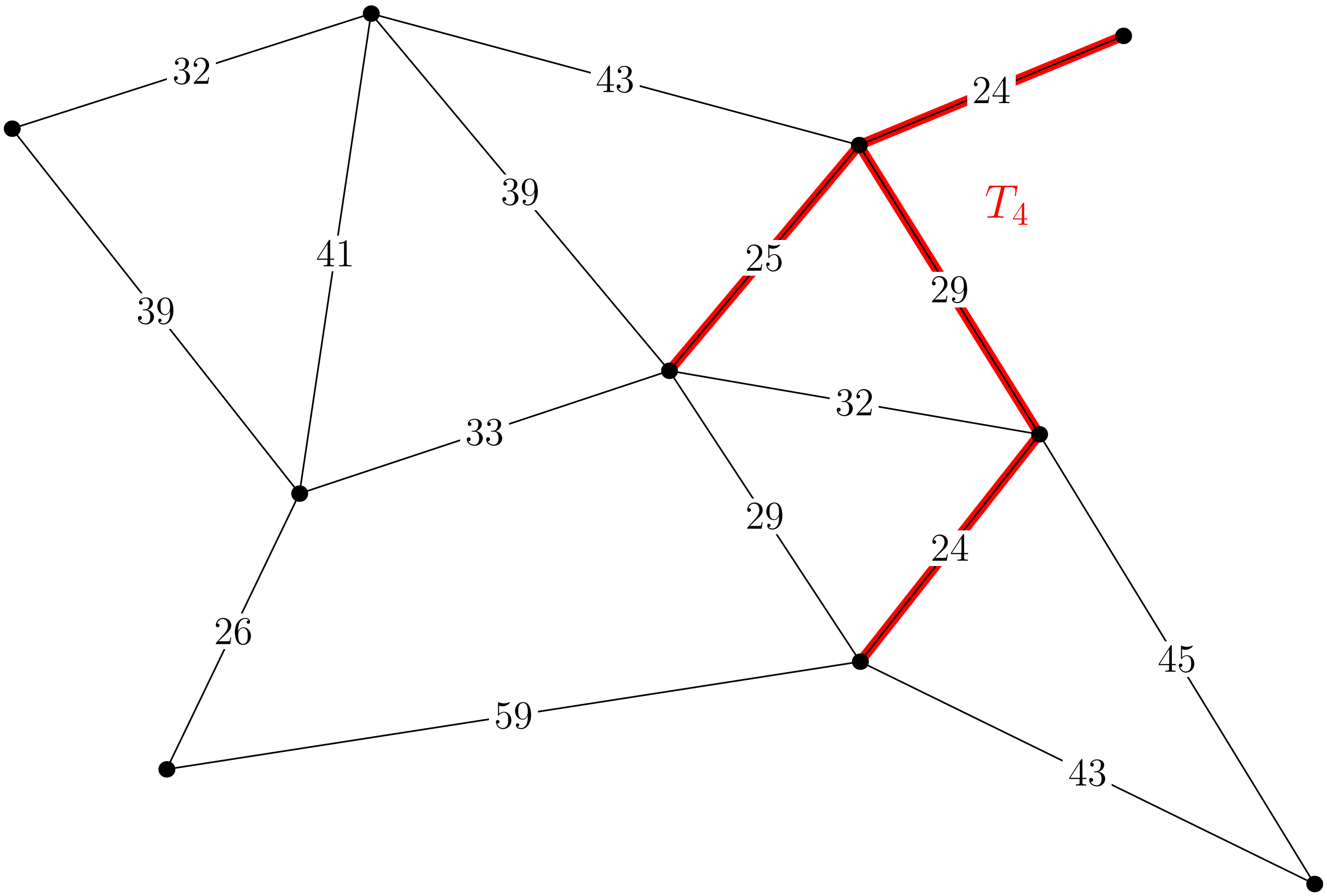
- In the base case, T_1 consists of a tree with no edges, but this has to be part of the minimum spanning tree

- To prove the inductive case we assume that T_i is part of the minimum spanning tree

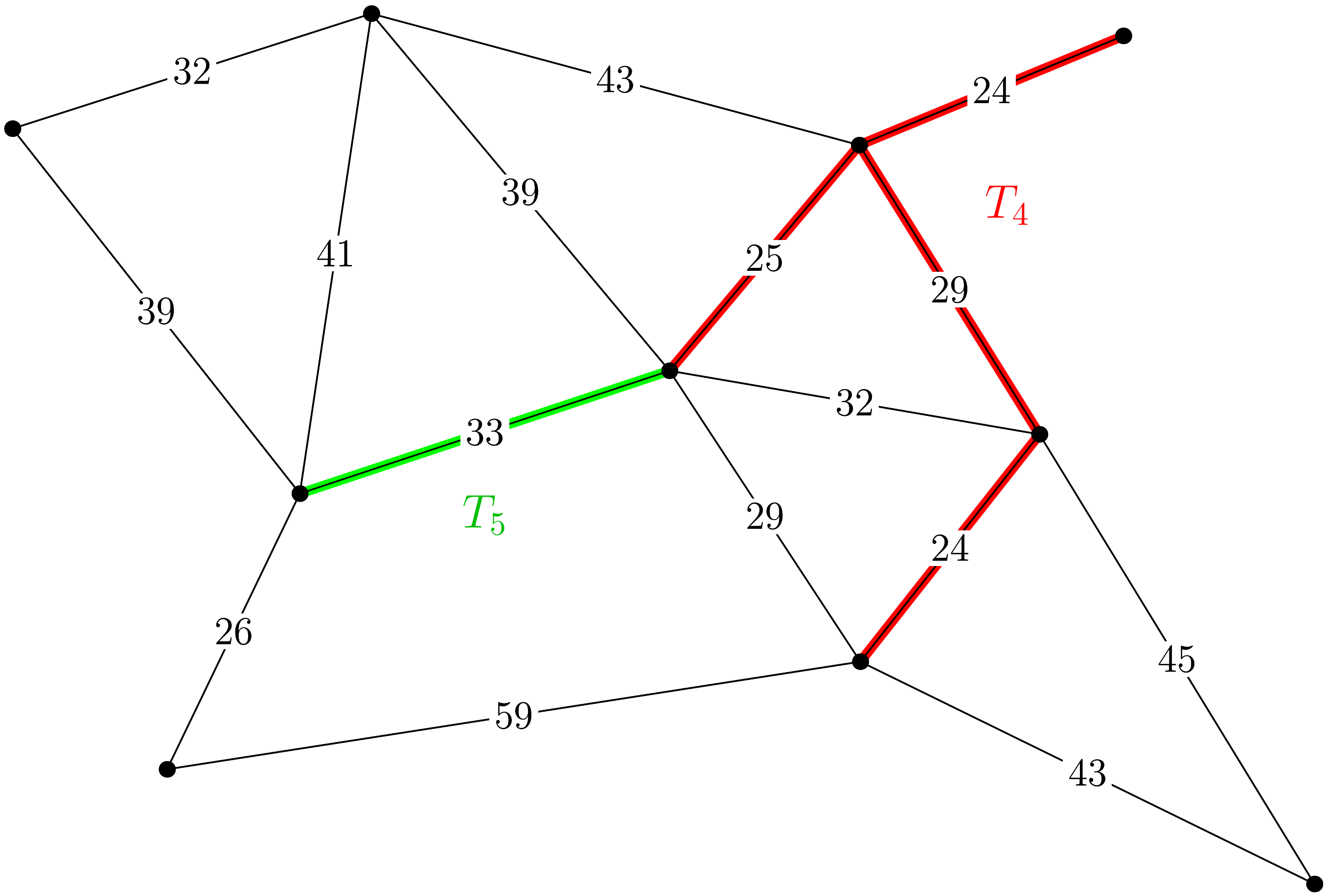
- We want to prove that T_{i+1} formed by adding the shortest edge is also part of the minimum spanning tree

- We perform the proof by contradiction—we assume that this added edge isn't part of the minimum spanning tree

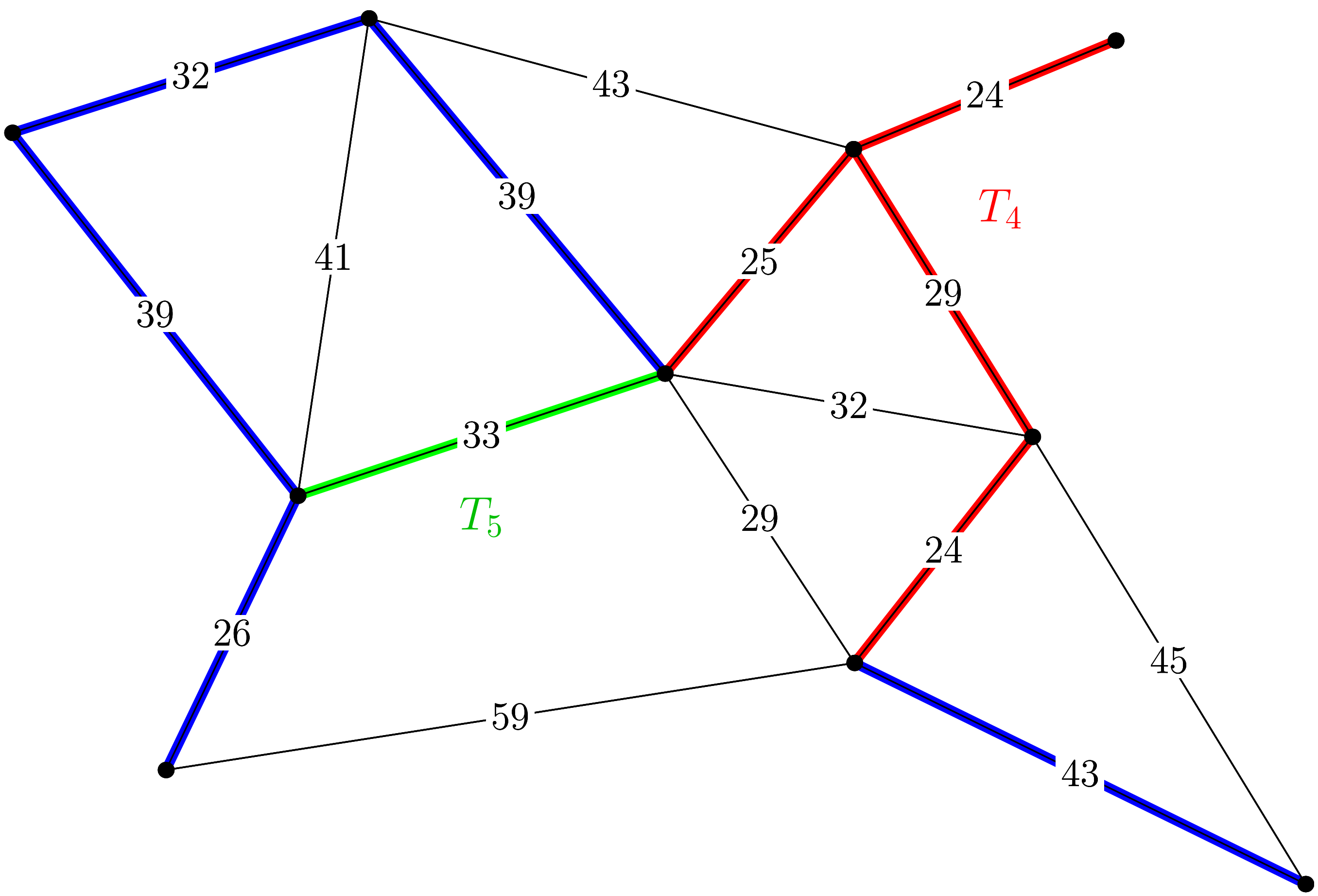
Contrariwise



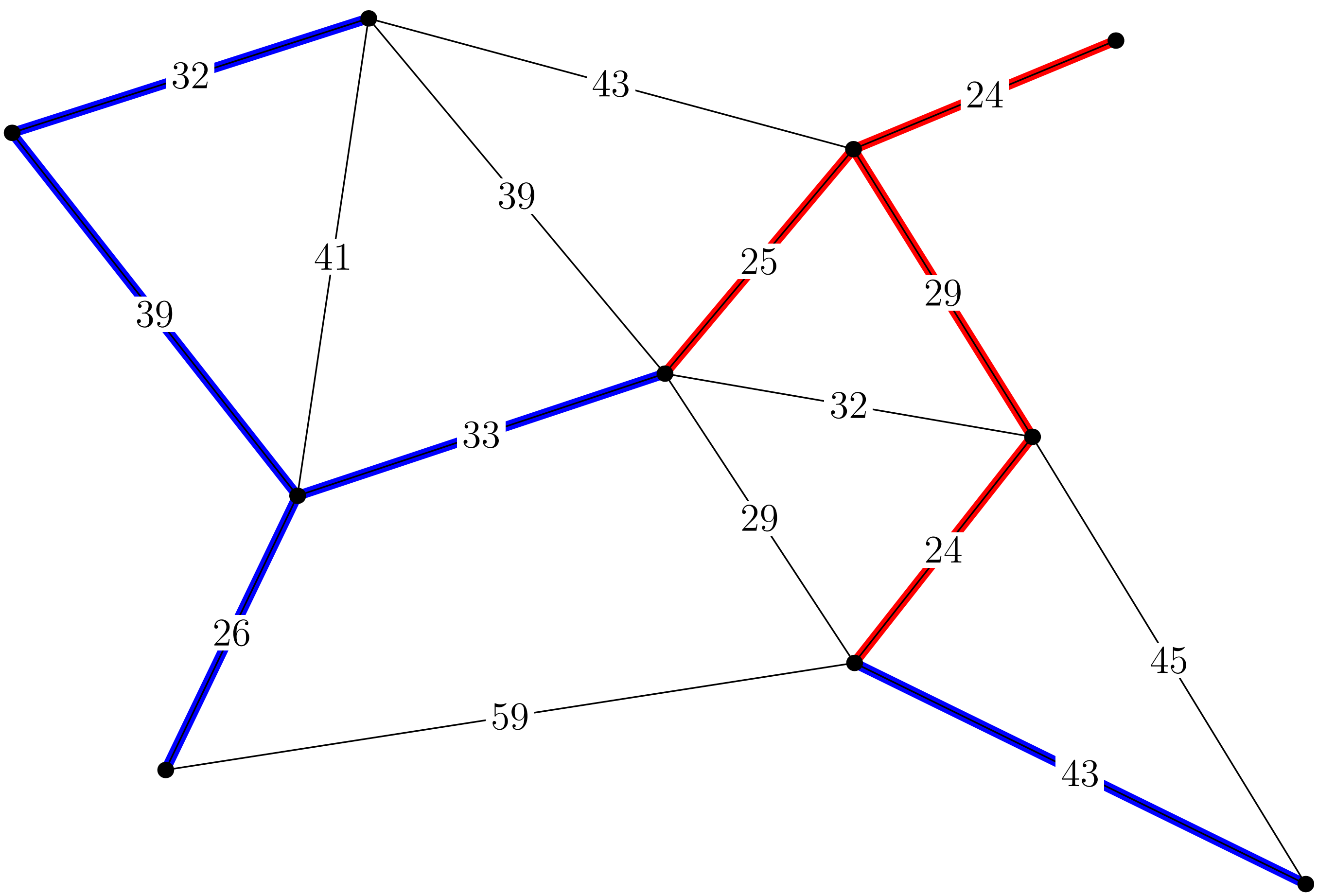
Contrariwise



Contrariwise



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

Run Time

- The worst time is

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

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

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

Run Time

- The worst time is

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

- Note that $|\mathcal{E}| < |\mathcal{V}|^2$

- Thus, $\log(|\mathcal{E}|) < 2\log(|\mathcal{V}|) = O\,\big(\log(|\mathcal{V}|)\big)$

- Thus the worst case time complexity is $|\mathcal{E}|\log(|\mathcal{V}|)$

Run Time

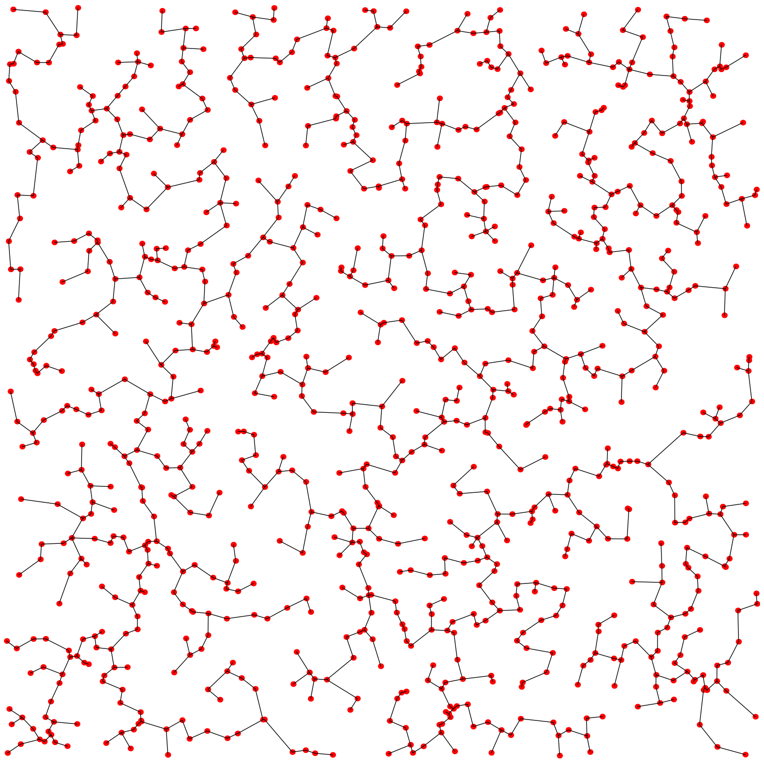
- The worst time is

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

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

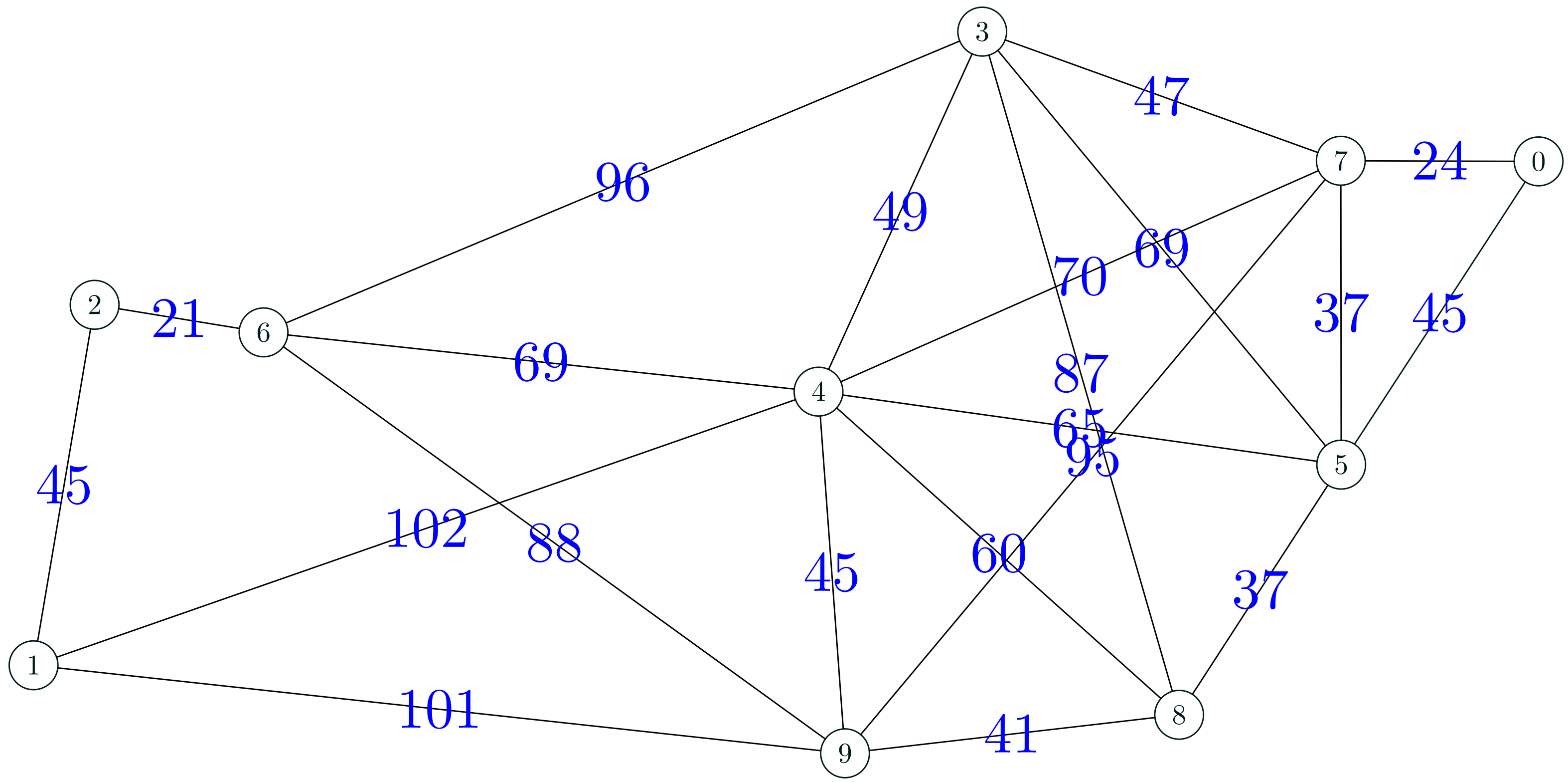
Outline

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



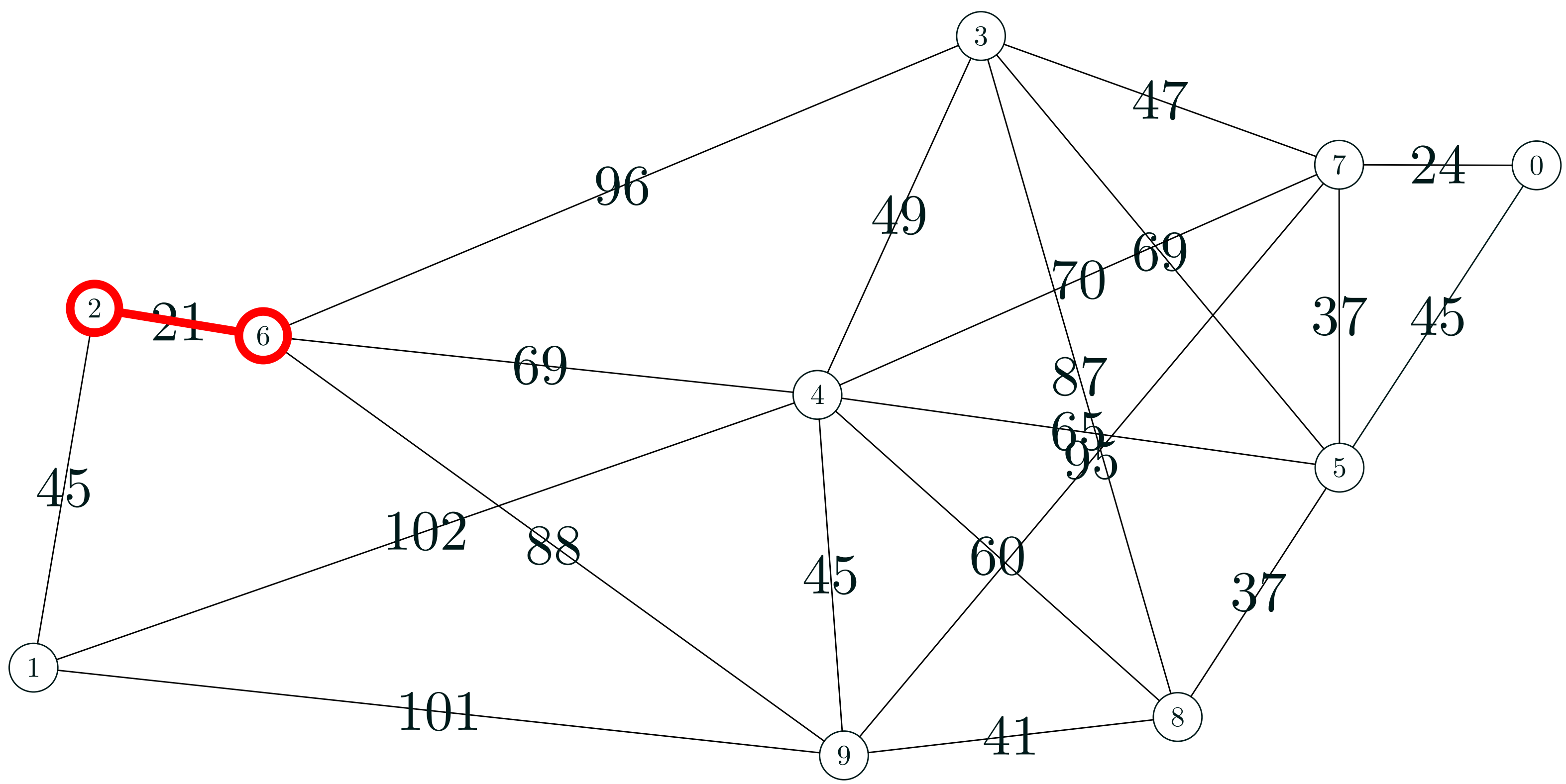
Kruskal's Algorithm

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



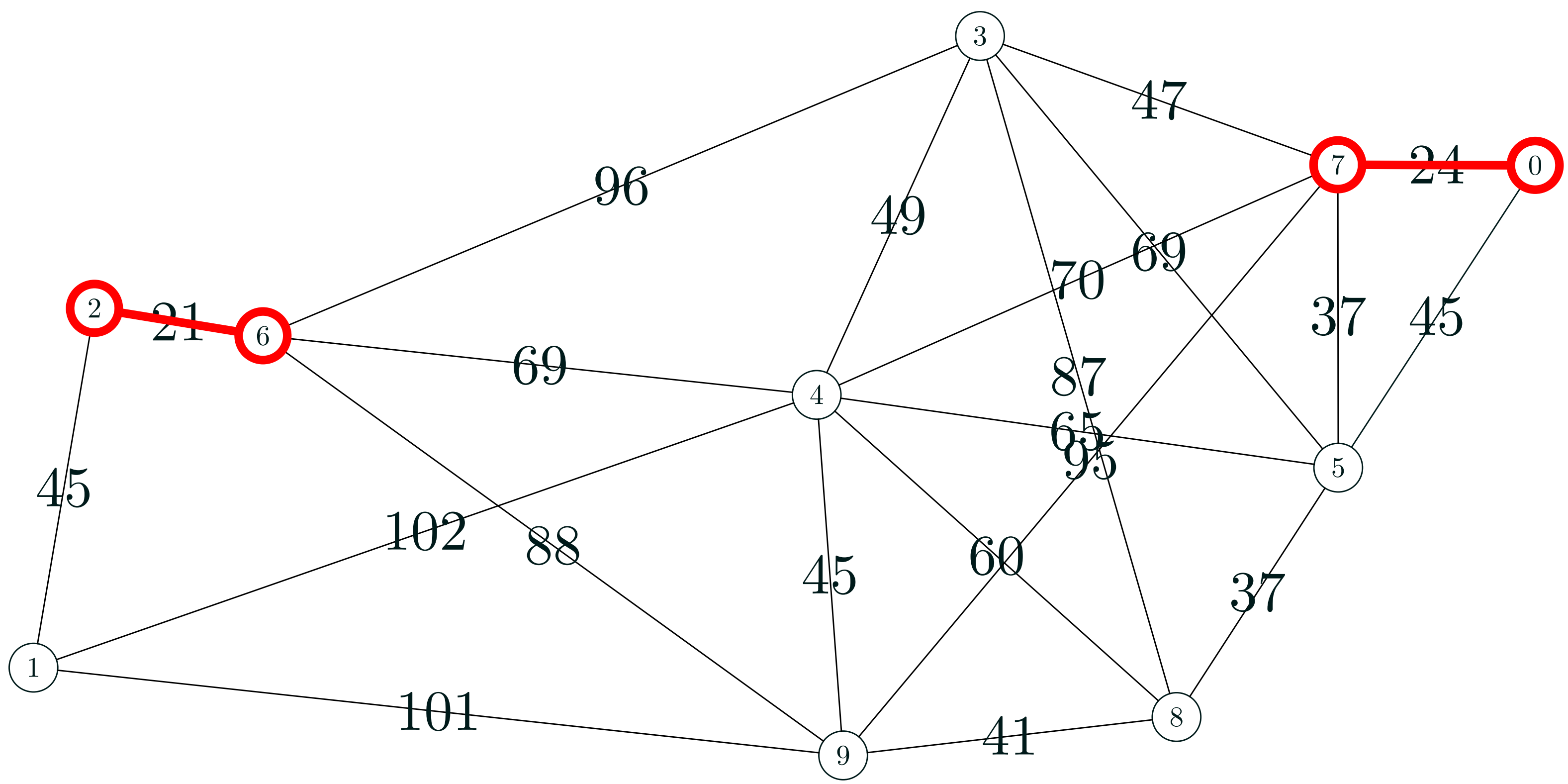
Kruskal's Algorithm

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



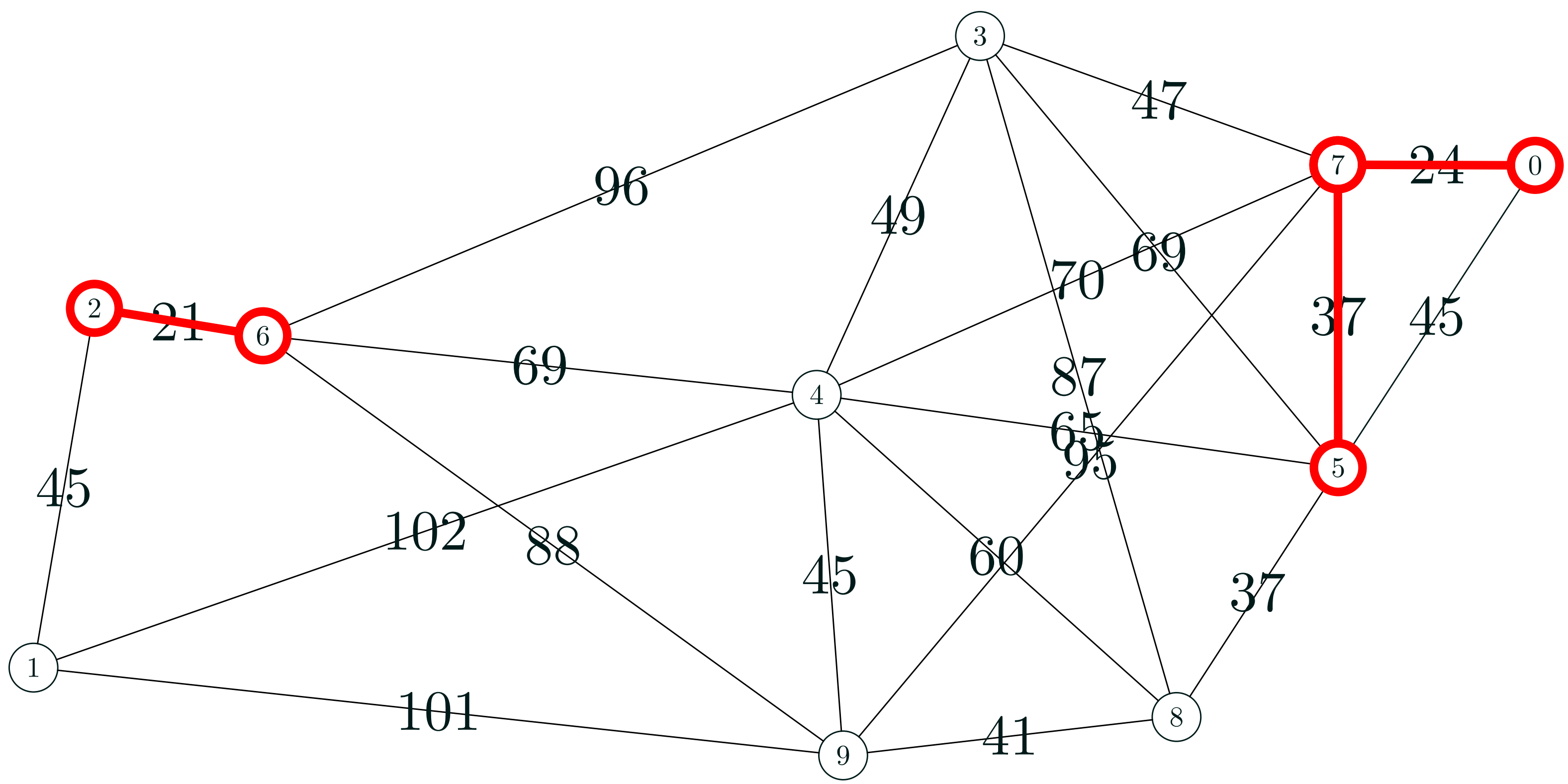
Kruskal's Algorithm

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



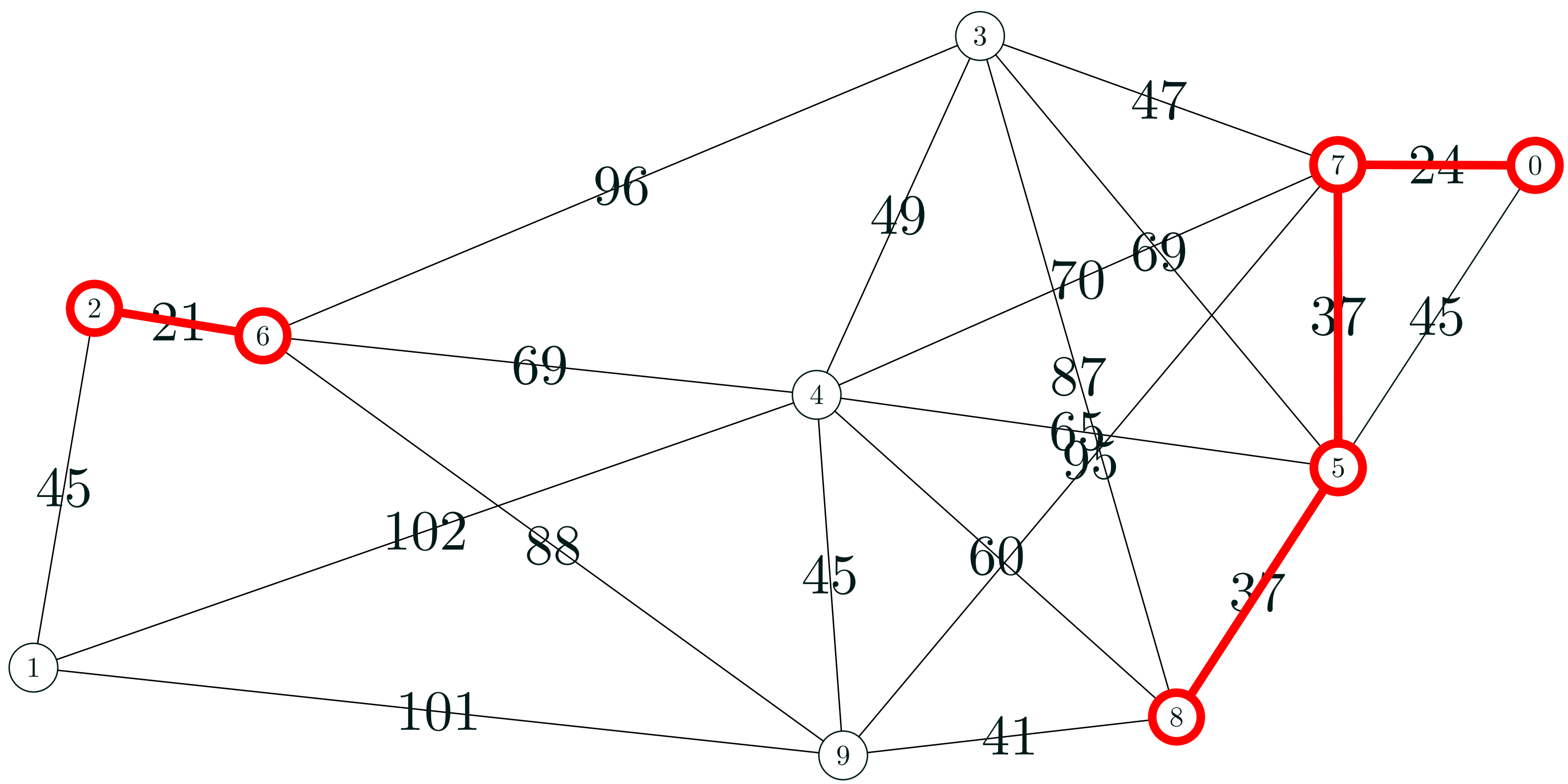
Kruskal's Algorithm

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



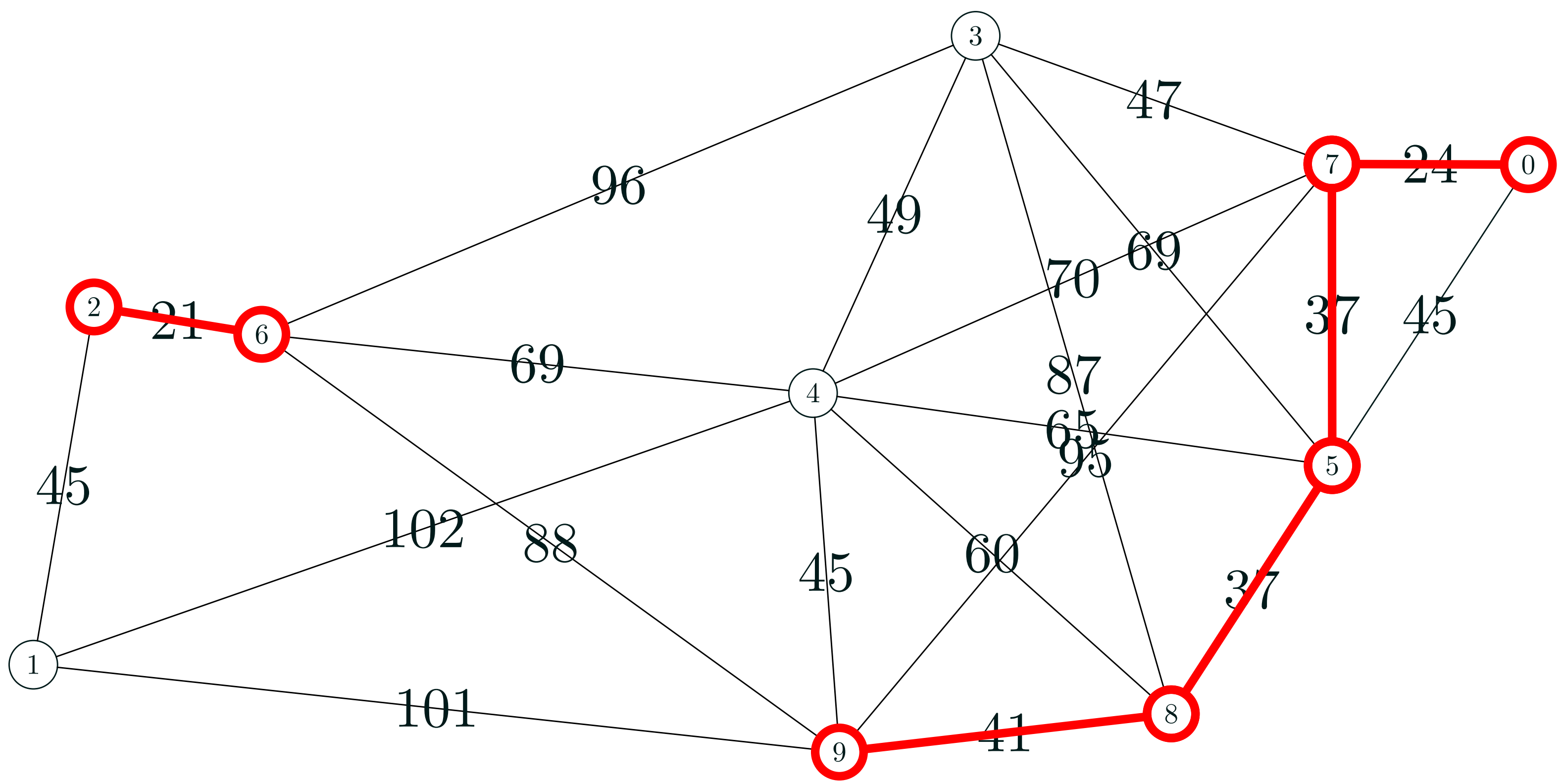
Kruskal's Algorithm

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



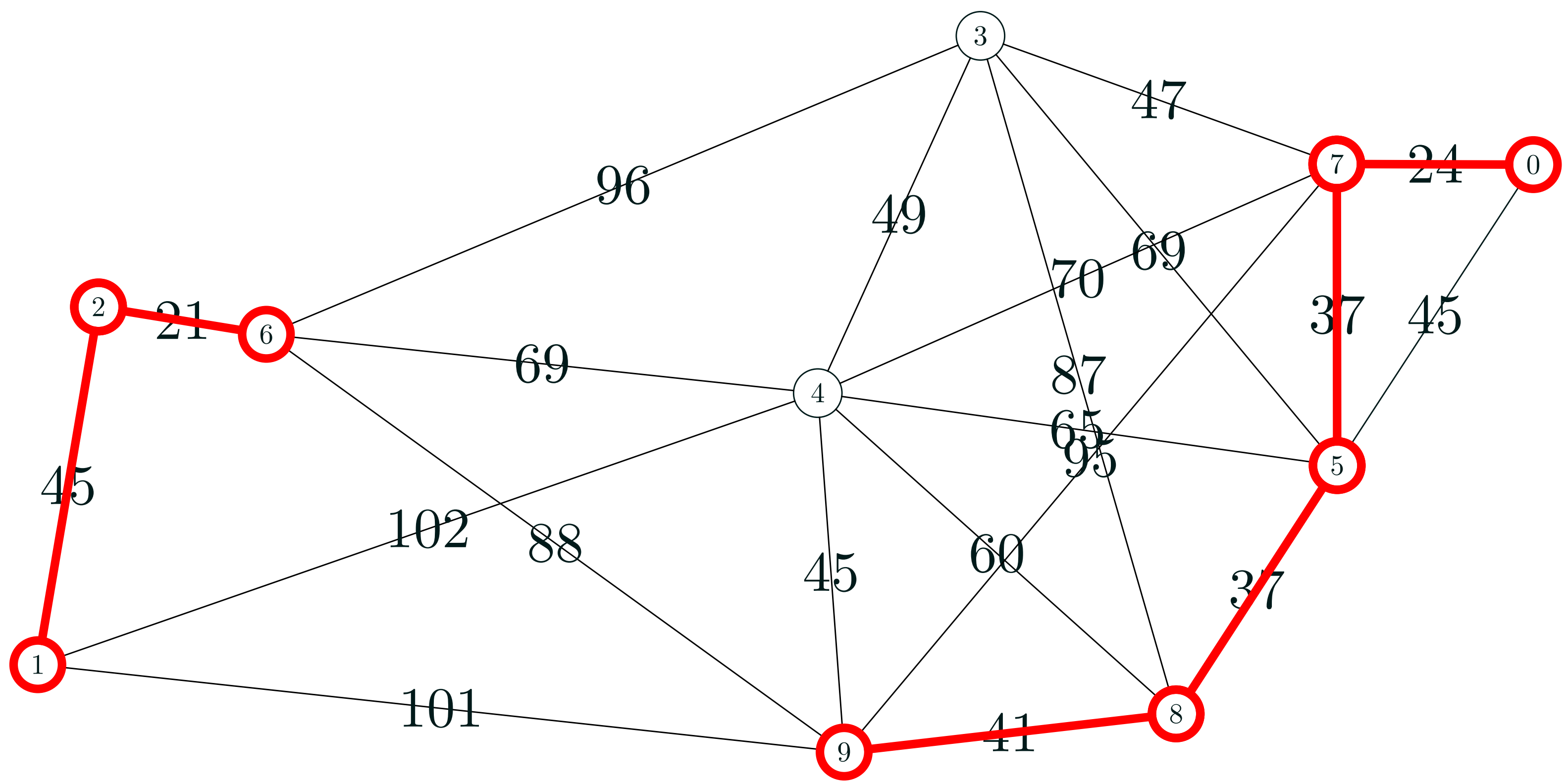
Kruskal's Algorithm

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



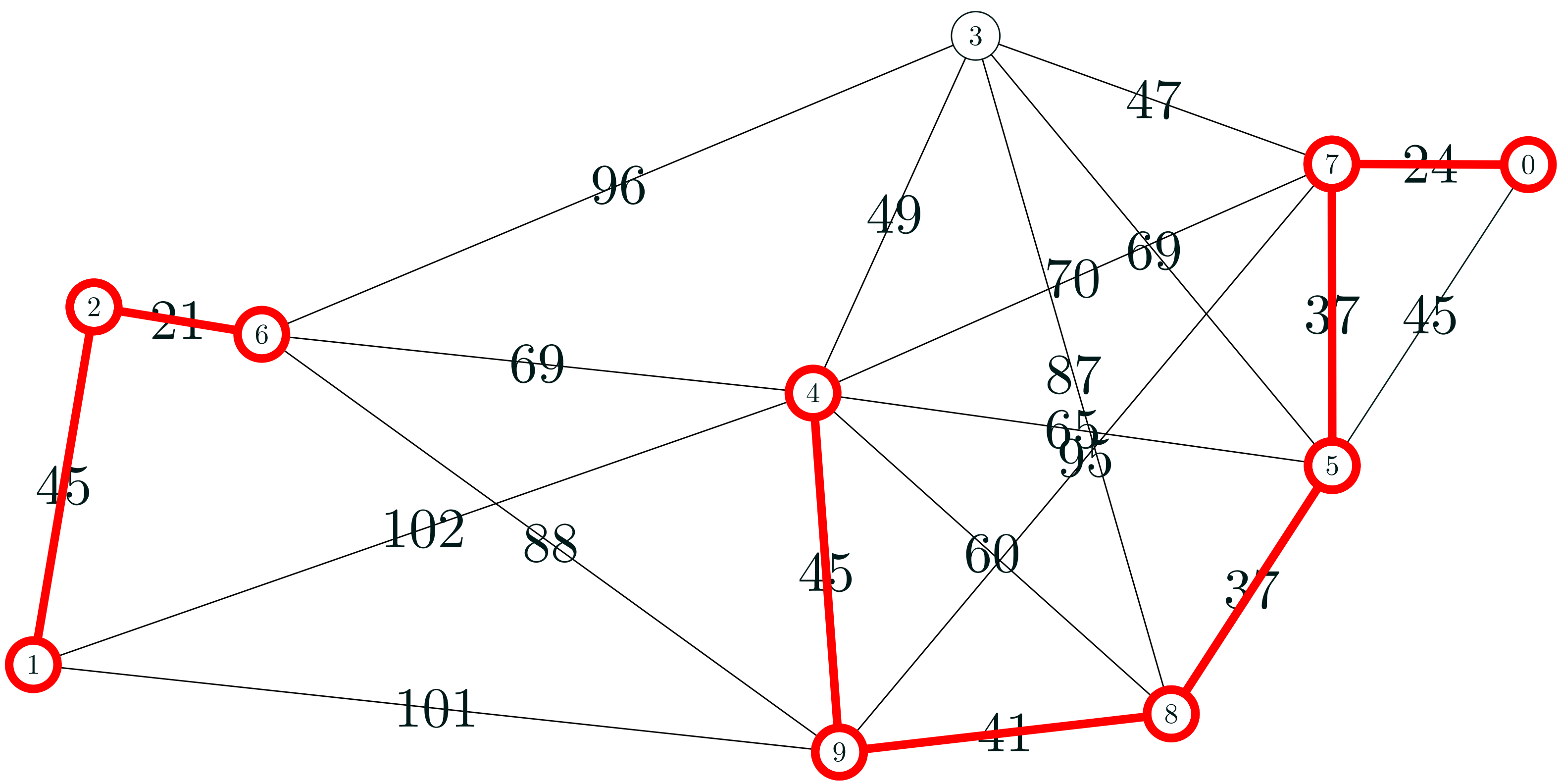
Kruskal's Algorithm

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



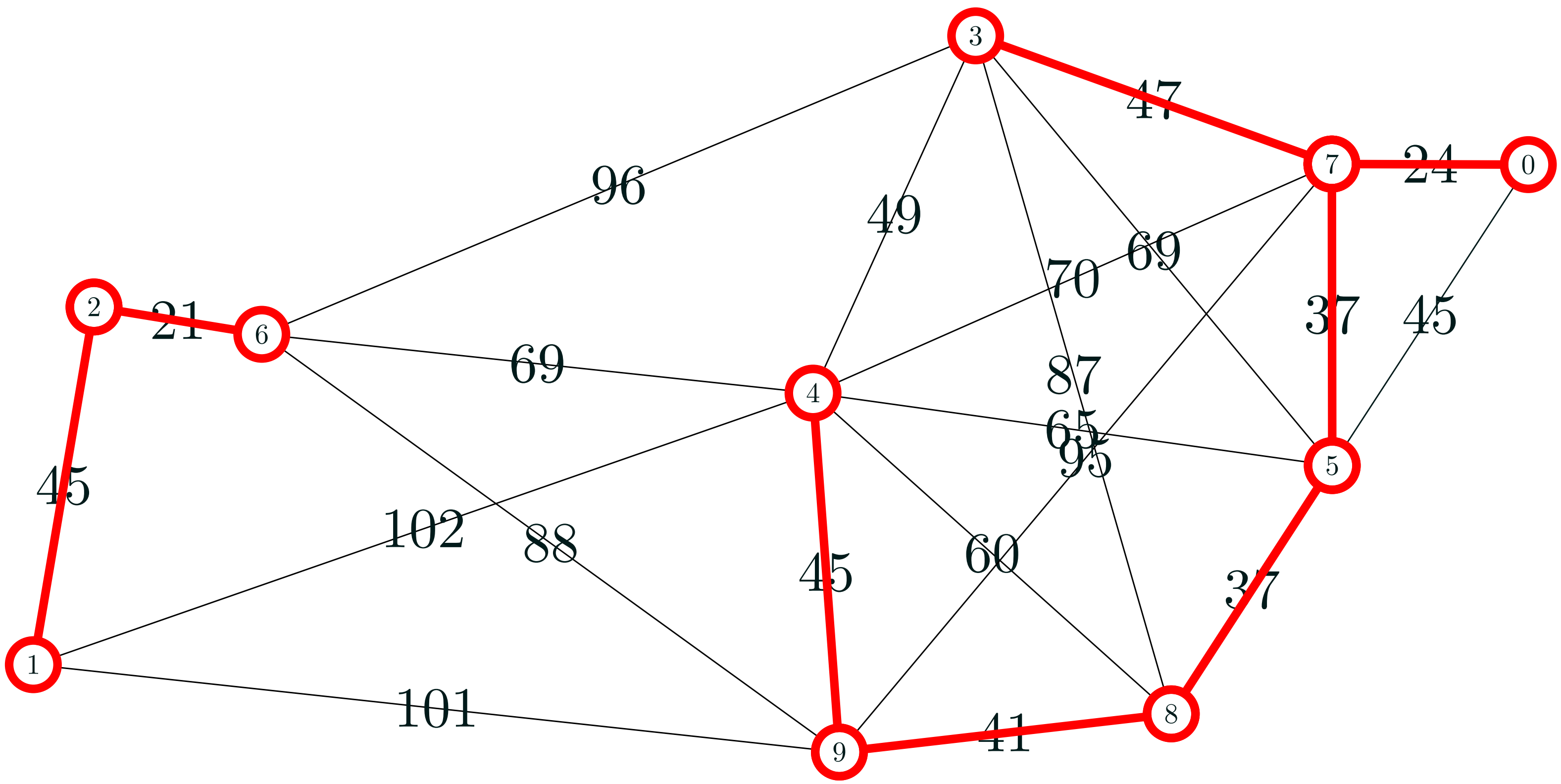
Kruskal's Algorithm

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



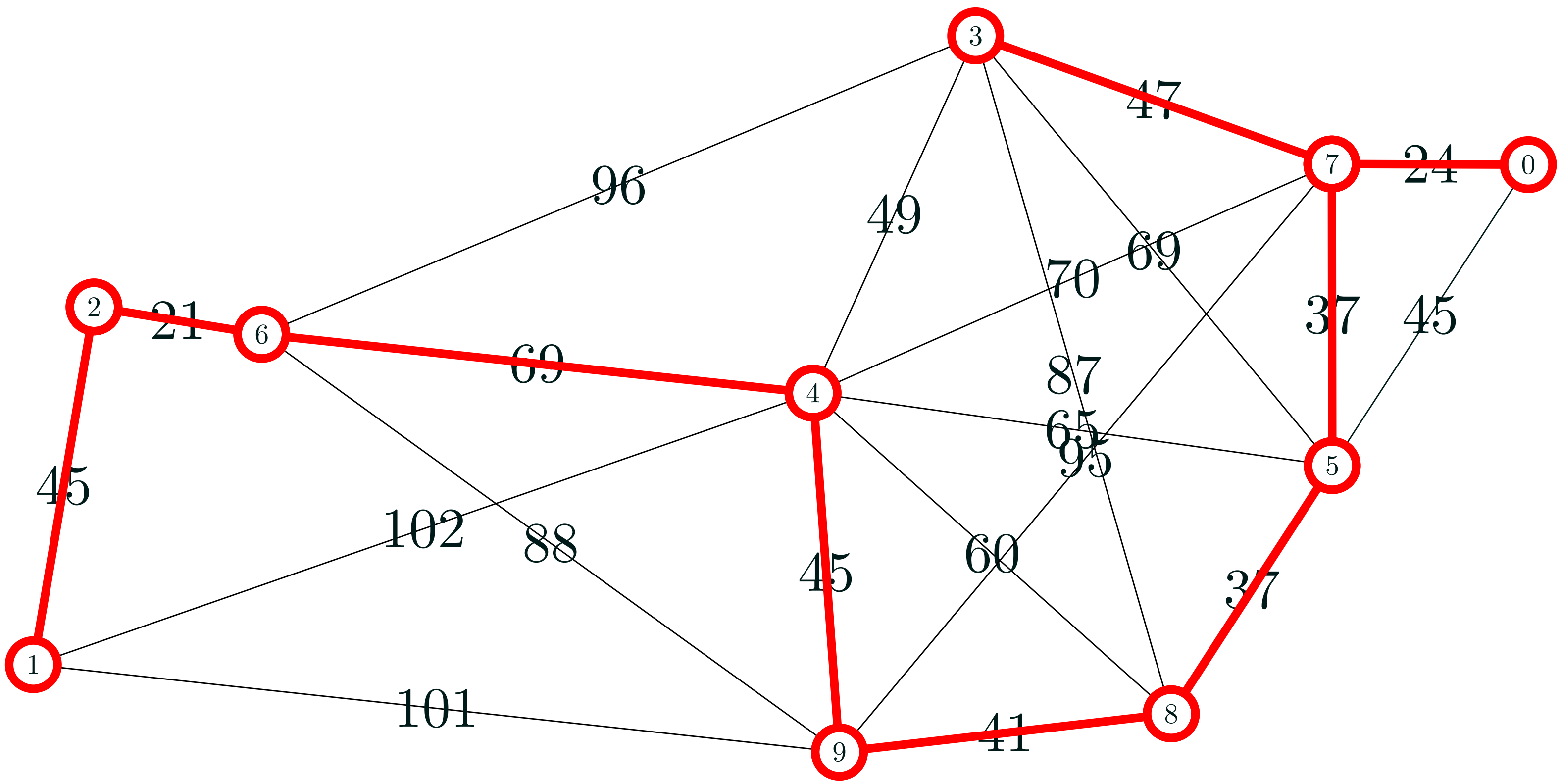
Kruskal's Algorithm

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



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
}
```

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
}
```

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
}
```

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
}
```

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

Analysis

- Kruskal's algorithm looks much simpler than Prim's
- The sorting takes most of the time, thus Prim's algorithms is $O\big(|\mathcal{E}|\log(|\mathcal{E}|)\big) = O\big(|\mathcal{E}|\log(|\mathcal{V}|)\big)$
- 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

Analysis

- Kruskal's algorithm looks much simpler than Prim's
- The sorting takes most of the time, thus Prim's algorithms is $O\big(|\mathcal{E}|\log(|\mathcal{E}|)\big) = O\big(|\mathcal{E}|\log(|\mathcal{V}|)\big)$
- 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

Analysis

- Kruskal's algorithm looks much simpler than Prim's
- The sorting takes most of the time, thus Prim's algorithms is $O\big(|\mathcal{E}|\log(|\mathcal{E}|)\big) = O\big(|\mathcal{E}|\log(|\mathcal{V}|)\big)$
- 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**

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**

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**

Cycling

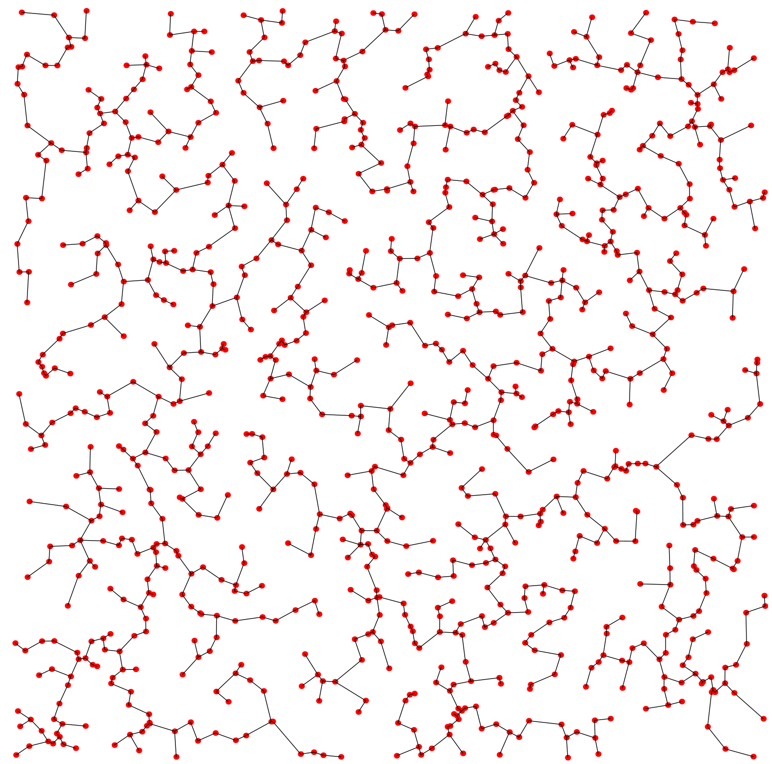
- For a path to be a cycle the edge has to join two nodes representing the same subtree
- To compute this we need to quickly **find** which subtree a node has been assigned to
- Initially all nodes are assigned to a separate subtree
- When two subtrees are combined by an edge we have to perform the **union**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

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 $\mathcal{S}_1$, $\mathcal{S}_2$, . . .
- 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 `union`s and `find`s

Union-Find

- In the union-find algorithm we have a set of objects $x \in \mathcal{S}$ which are to be grouped into subsets $\mathcal{S}_1$, $\mathcal{S}_2$, ...
- 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 `union`s and `find`s

Union-Find

- In the union-find algorithm we have a set of objects $x \in \mathcal{S}$ which are to be grouped into subsets $\mathcal{S}_1$, $\mathcal{S}_2$, . . .

- 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 `union`s and `find`s

Union-Find

- In the union-find algorithm we have a set of objects $x \in \mathcal{S}$ which are to be grouped into subsets $\mathcal{S}_1$, $\mathcal{S}_2$, . . .
- 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 `union`s and `find`s

Union-Find

- In the union-find algorithm we have a set of objects $x \in \mathcal{S}$ which are to be grouped into subsets $\mathcal{S}_1$, $\mathcal{S}_2$, . . .
- 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 `union`s and `find`s

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)`

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)`

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?

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?

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?

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?

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?

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?

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

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

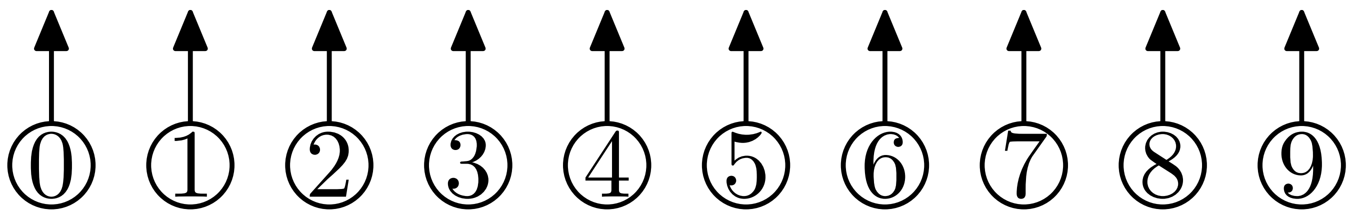
Fast Union

- 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

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

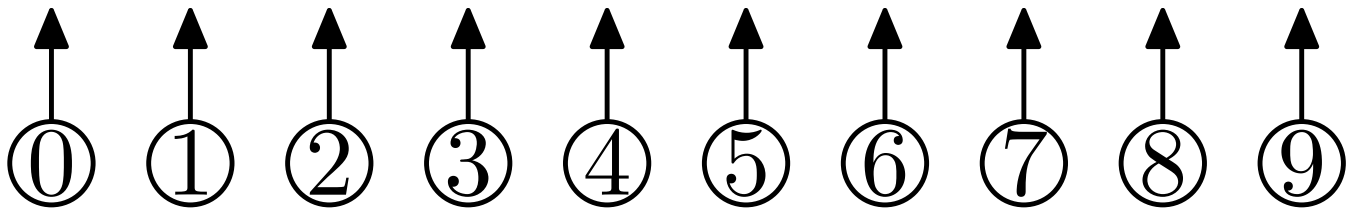
Putting it Together



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

Putting it Together

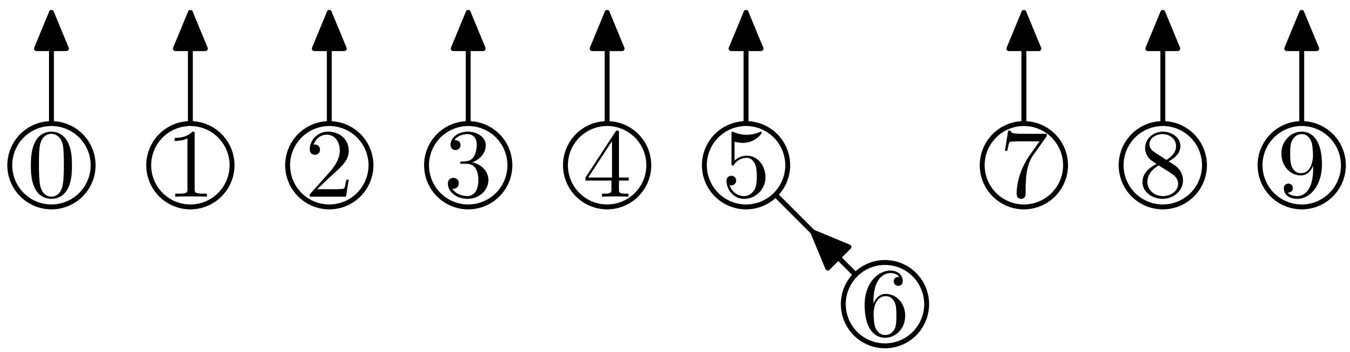
```
union(find(5),find(6))
```



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

Putting it Together

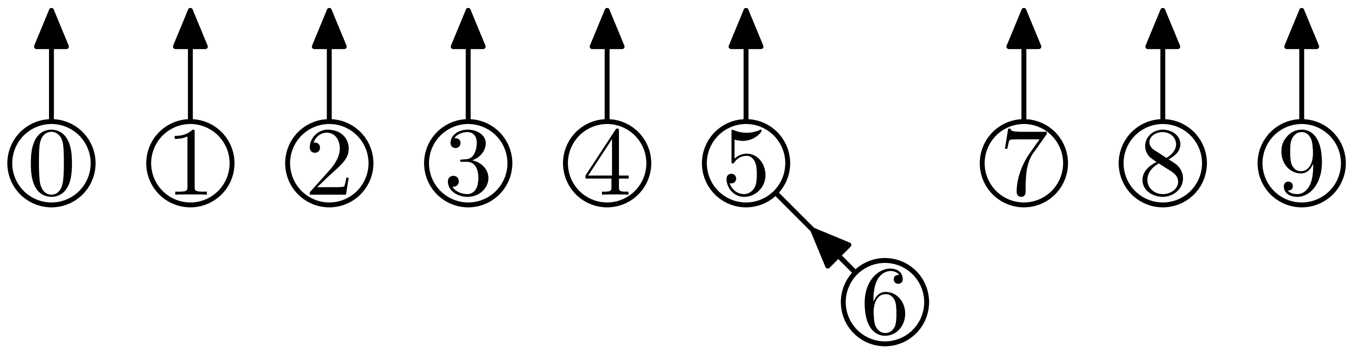
`union(find(5),find(6))`



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

Putting it Together

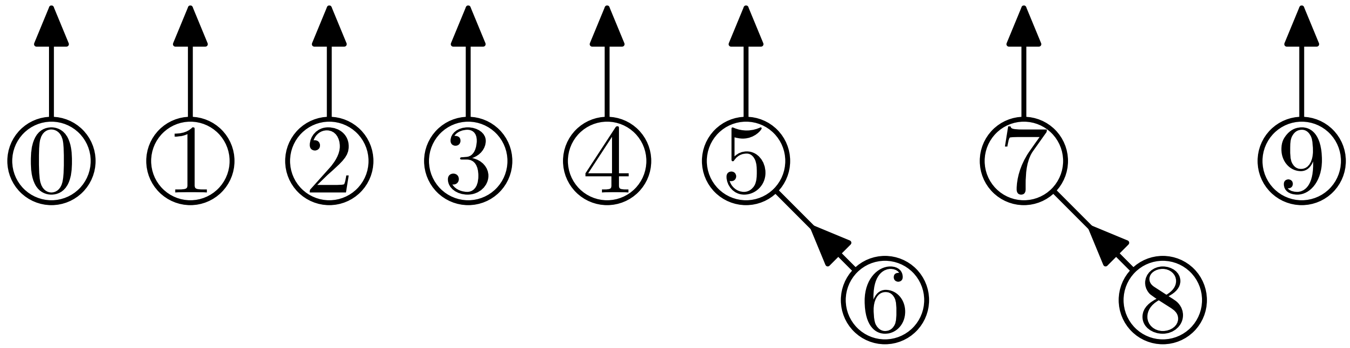
`union(find(7),find(8))`



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

Putting it Together

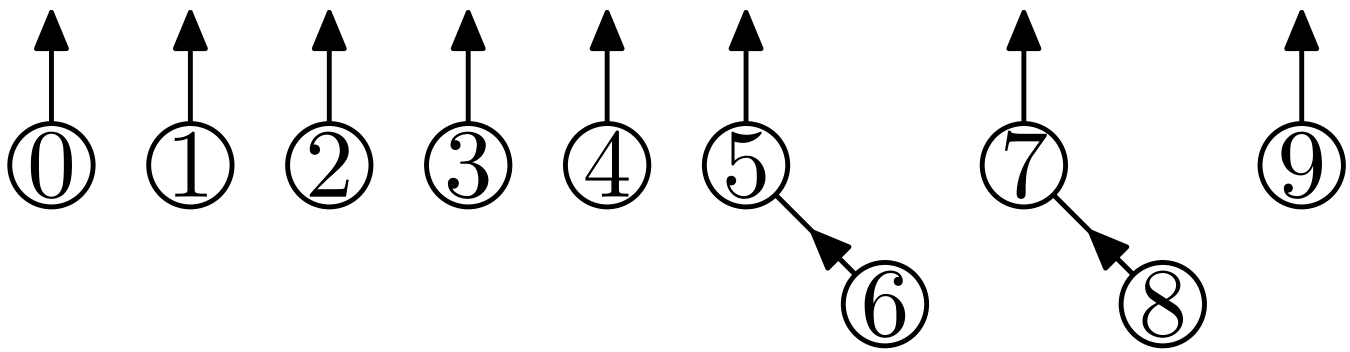
```
union(find(7),find(8))
```



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

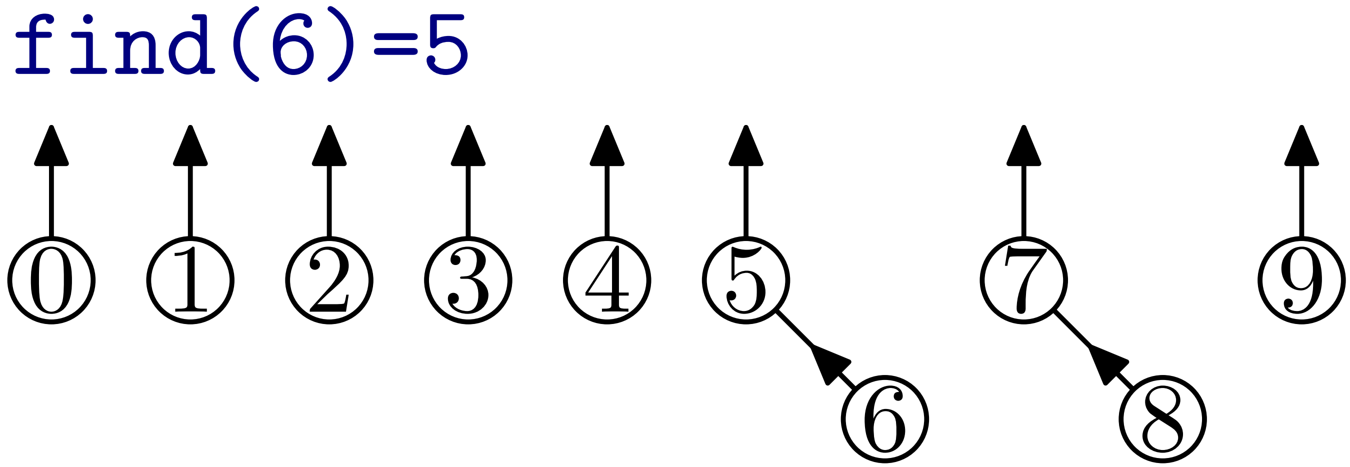
Putting it Together

`find(6)`



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

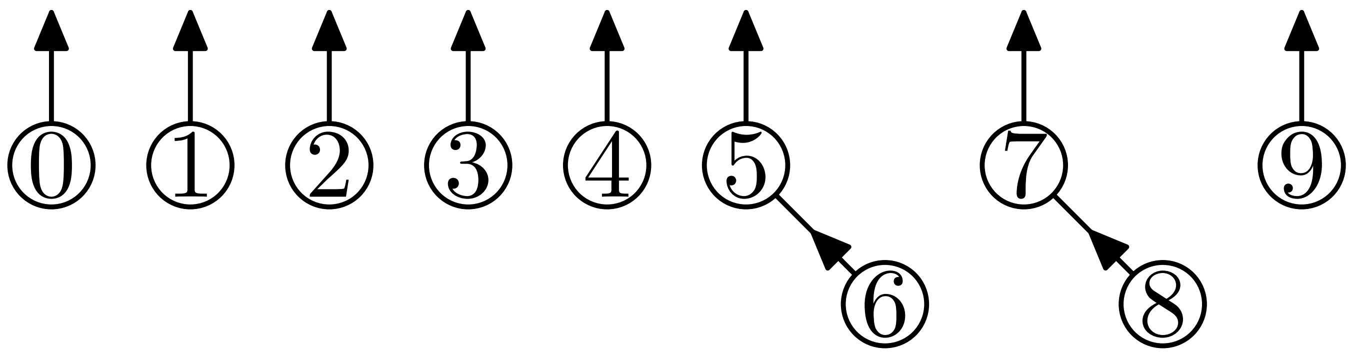
Putting it Together



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

Putting it Together

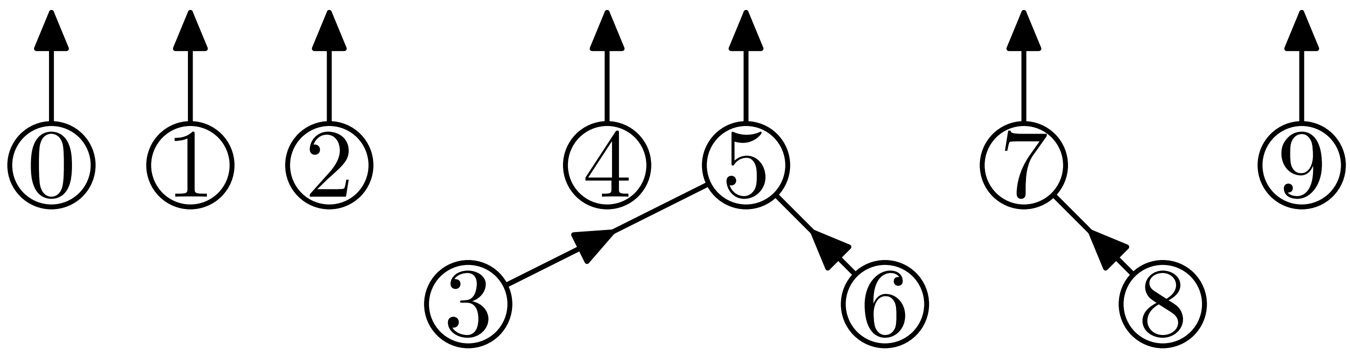
`union(find(3),find(6))`



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

Putting it Together

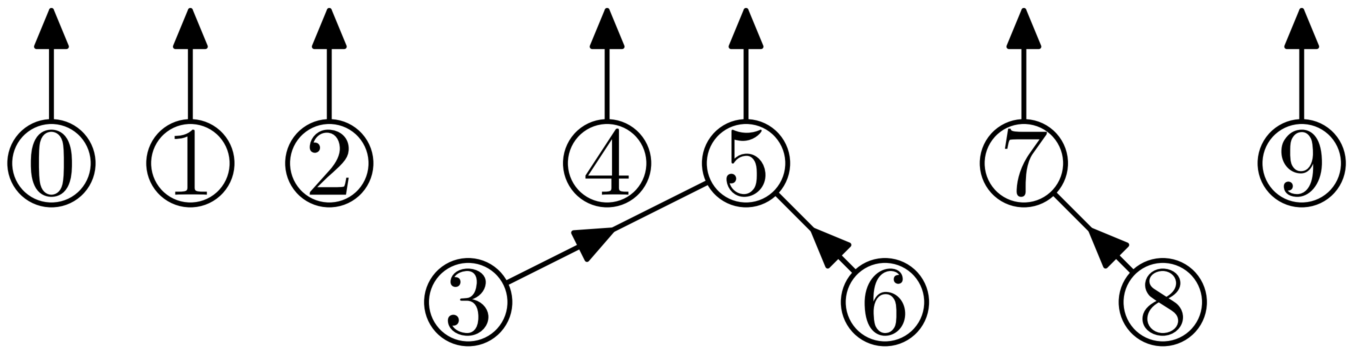
`union(find(3),find(6))`



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

Putting it Together

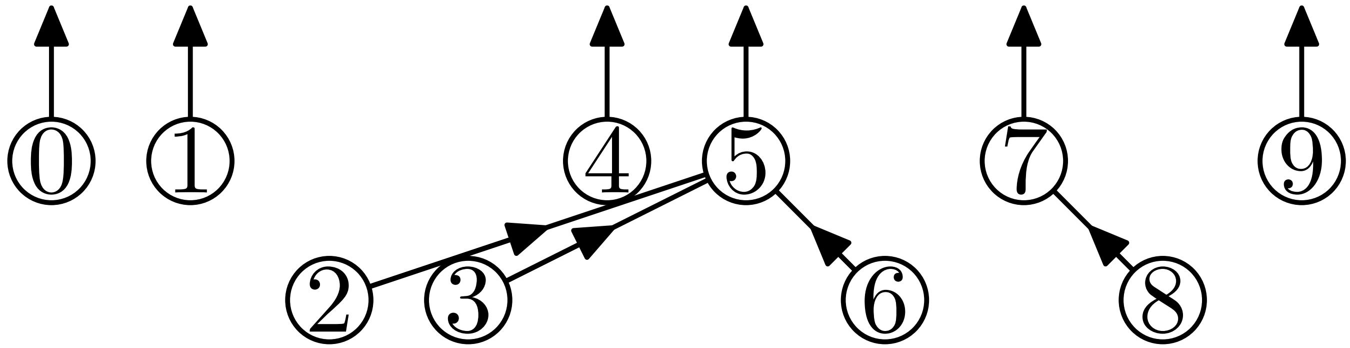
`union(find(2),find(6))`



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

Putting it Together

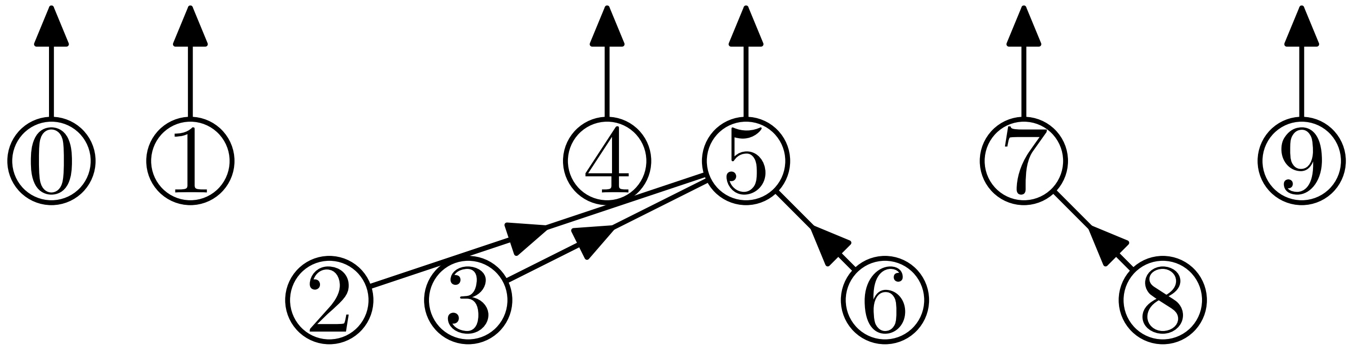
`union(find(2),find(6))`



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

Putting it Together

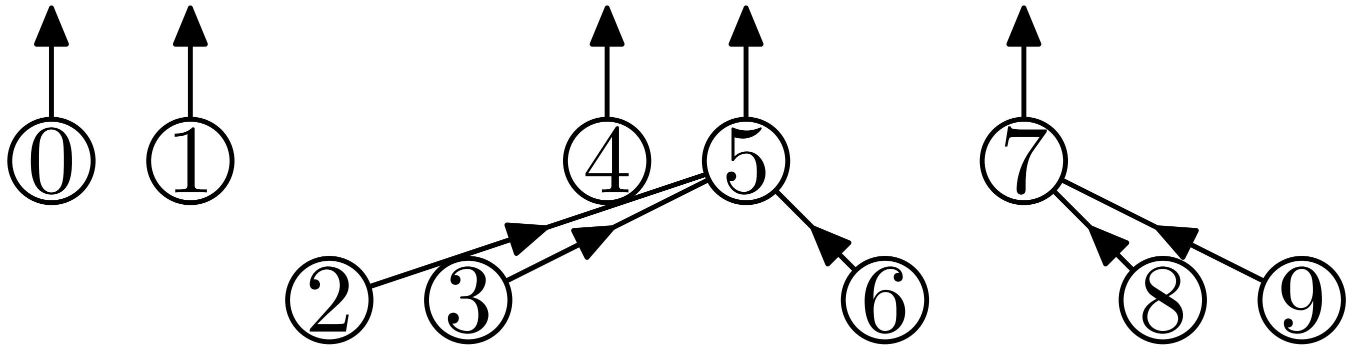
`union(find(9),find(8))`



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

Putting it Together

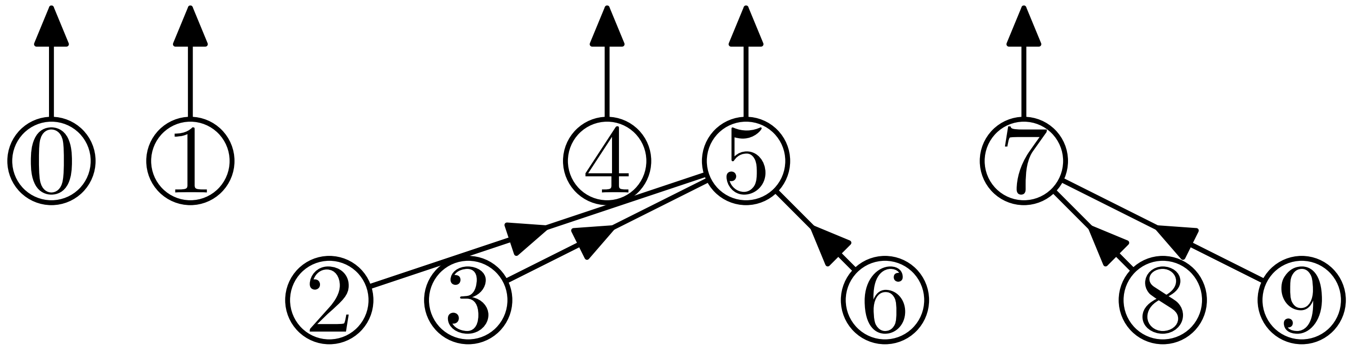
`union(find(9),find(8))`



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

Putting it Together

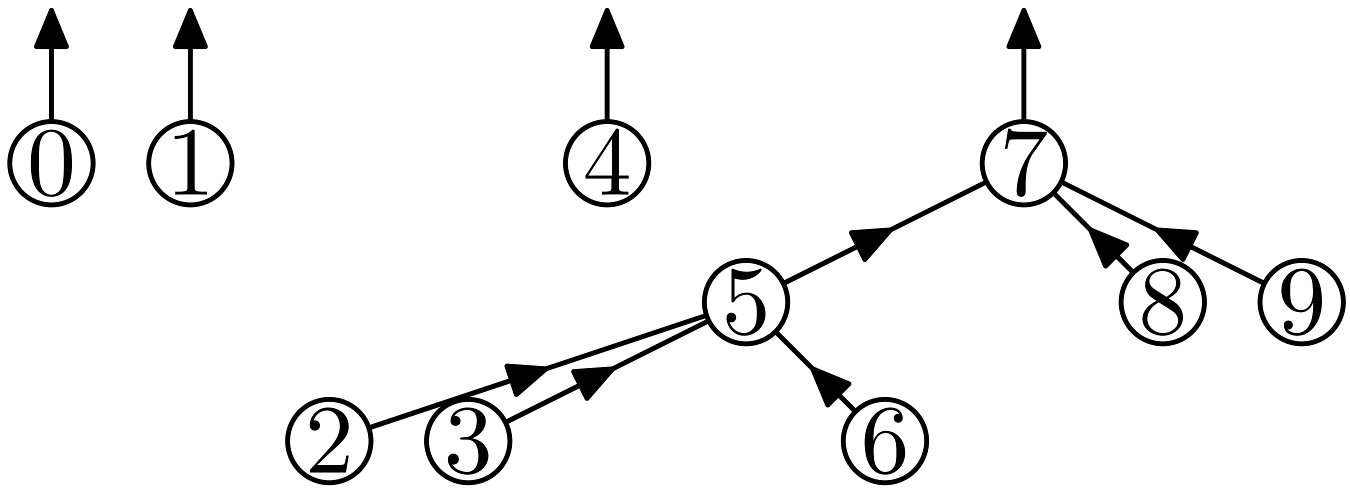
`union(find(9),find(3))`



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

Putting it Together

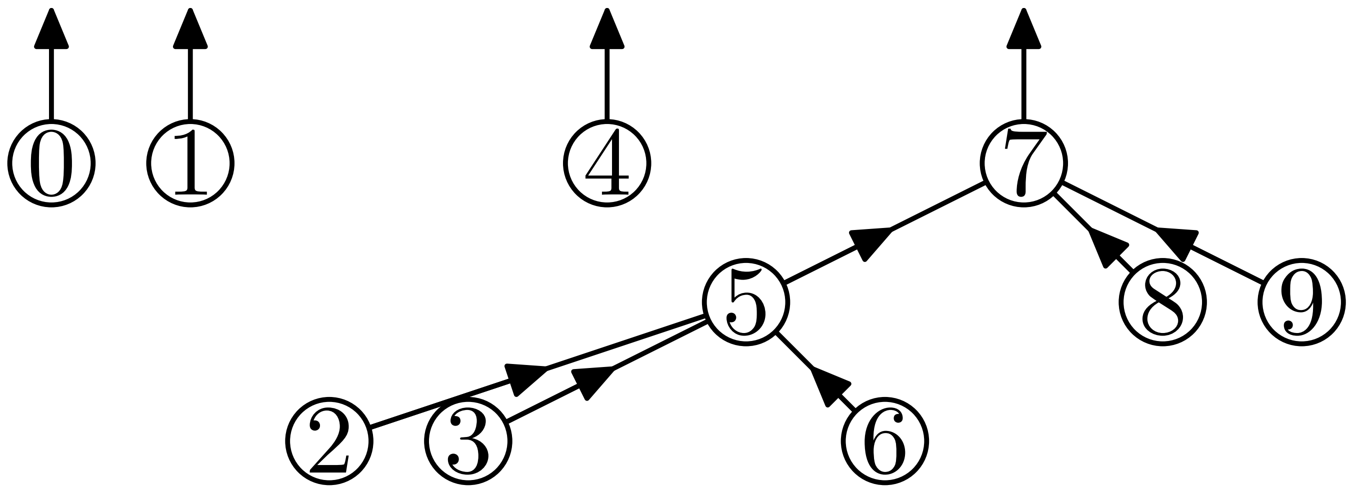
`union(find(9),find(3))`



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

Putting it Together

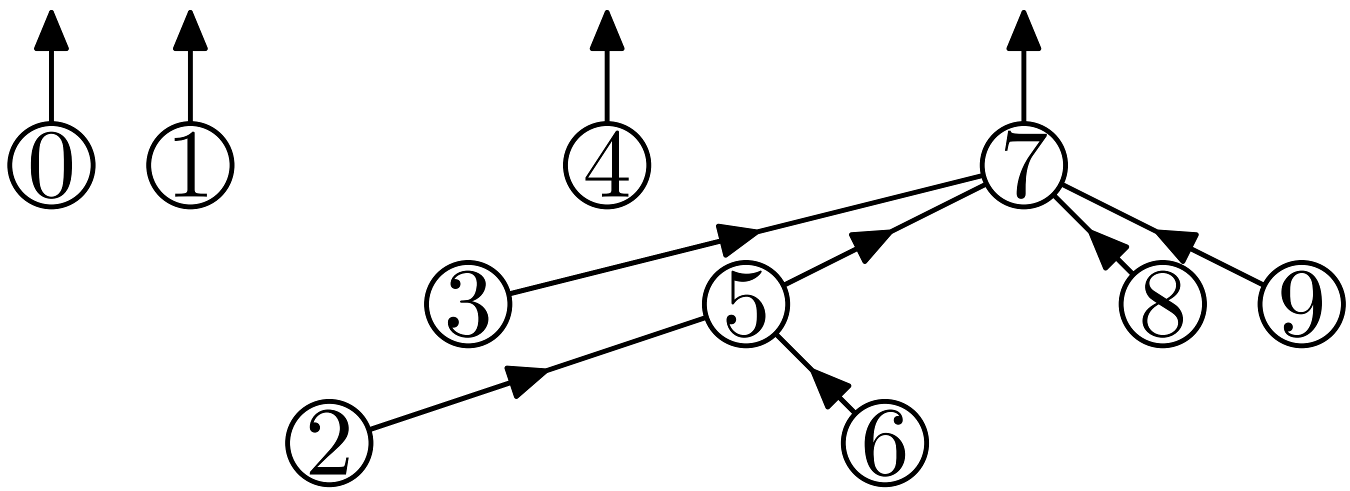
`find(3)`



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

Putting it Together

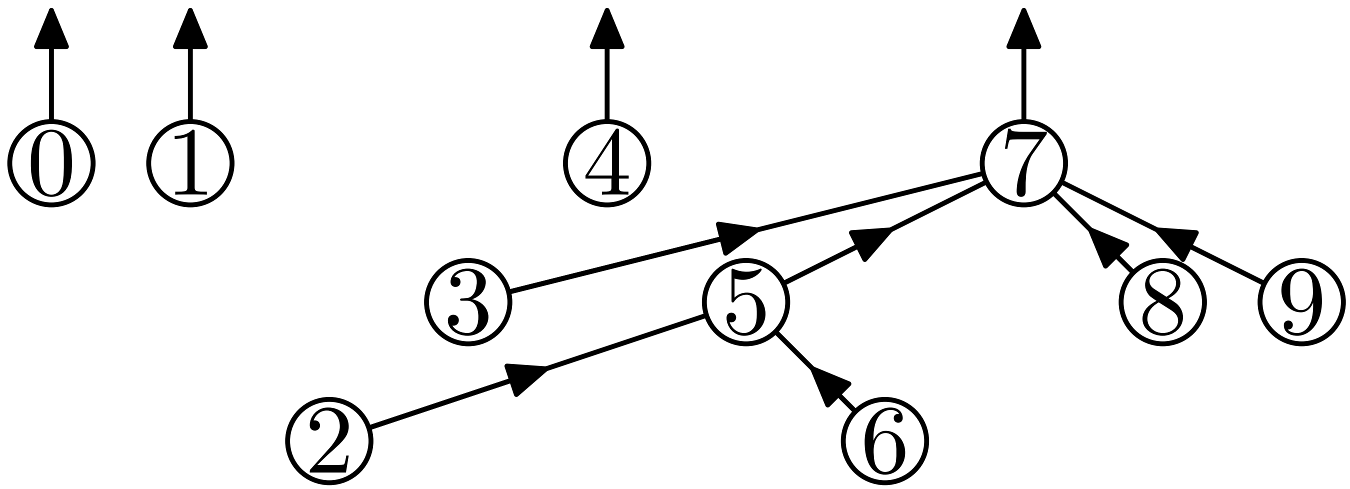
`find(3)=7`



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

Putting it Together

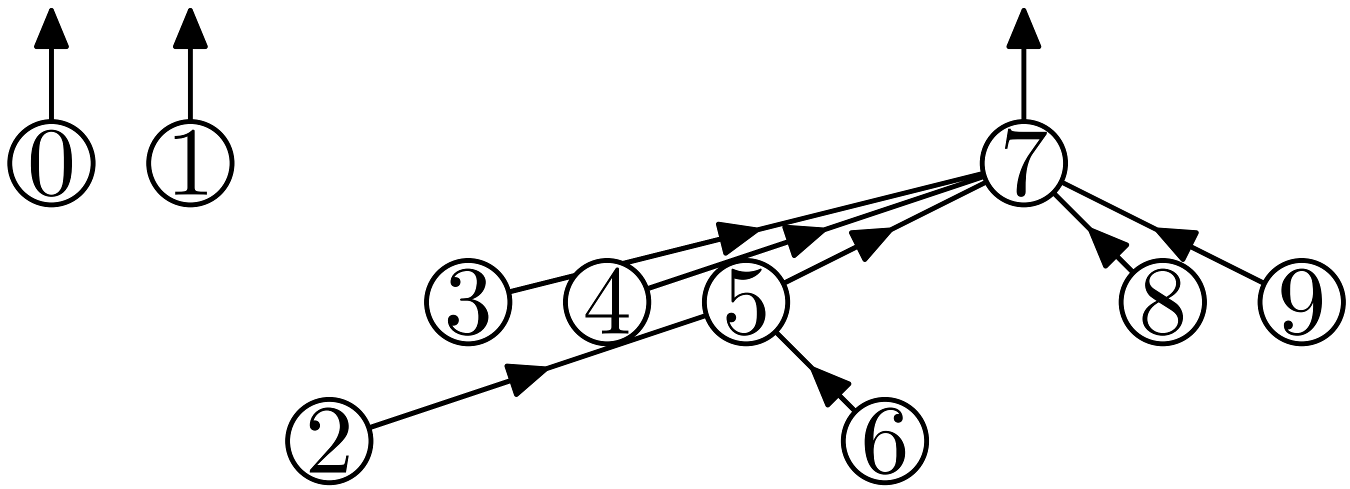
`union(find(3),find(4))`



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

Putting it Together

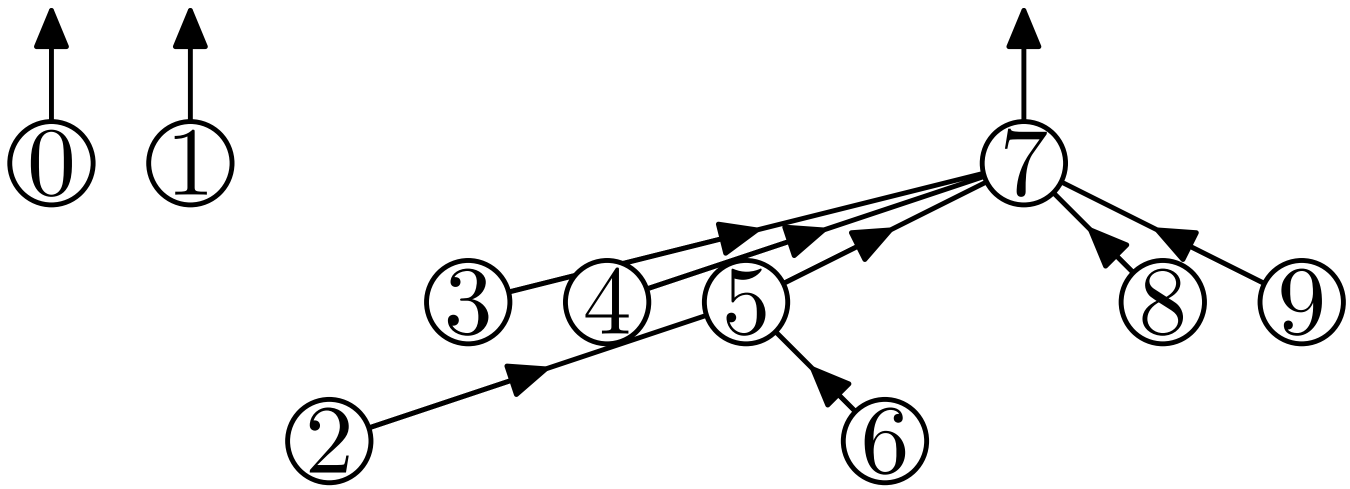
`union(find(3),find(4))`



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

Putting it Together

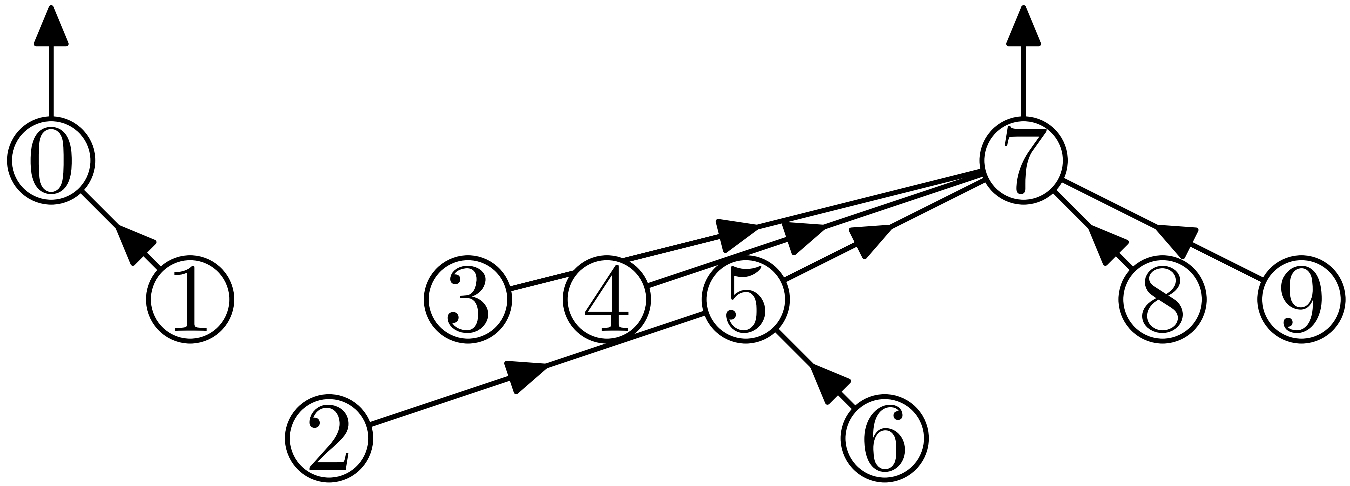
```
union(find(0),find(1))
```



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

Putting it Together

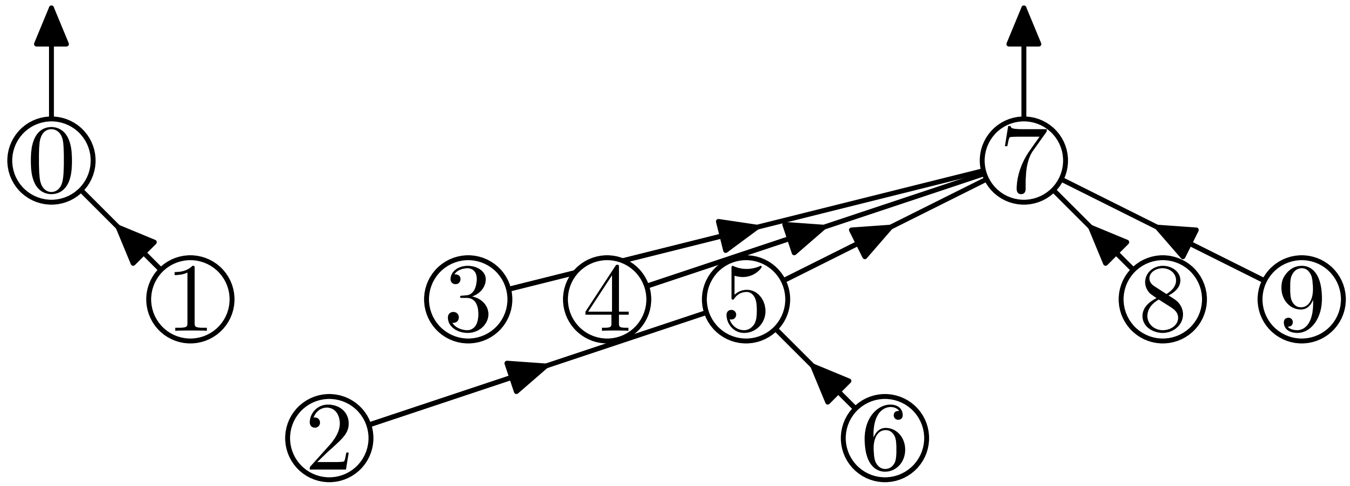
`union(find(0),find(1))`



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

Putting it Together

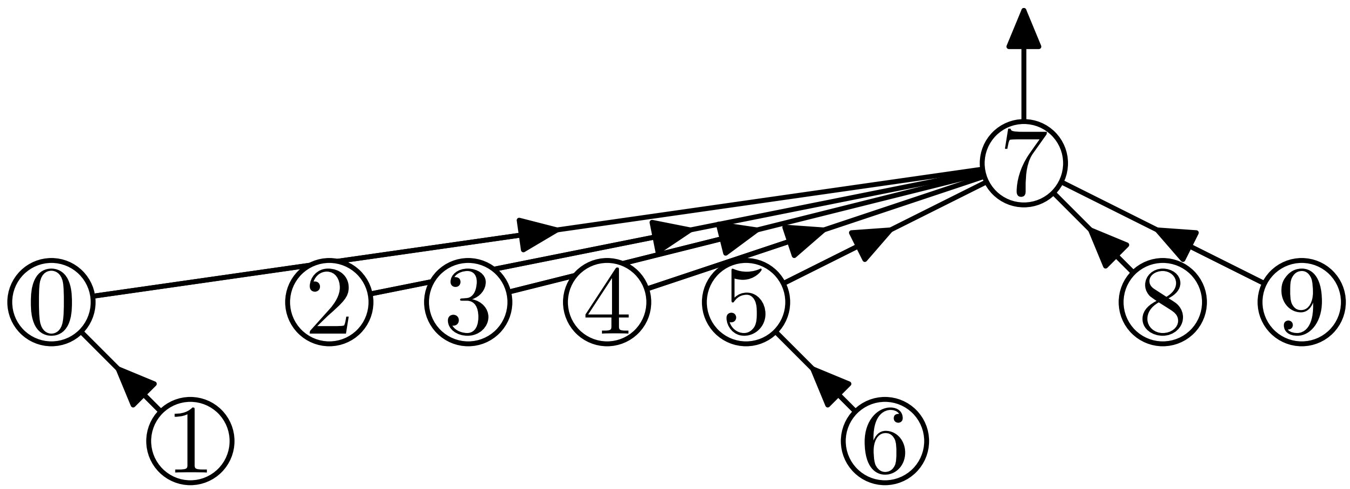
`union(find(1),find(2))`



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

Putting it Together

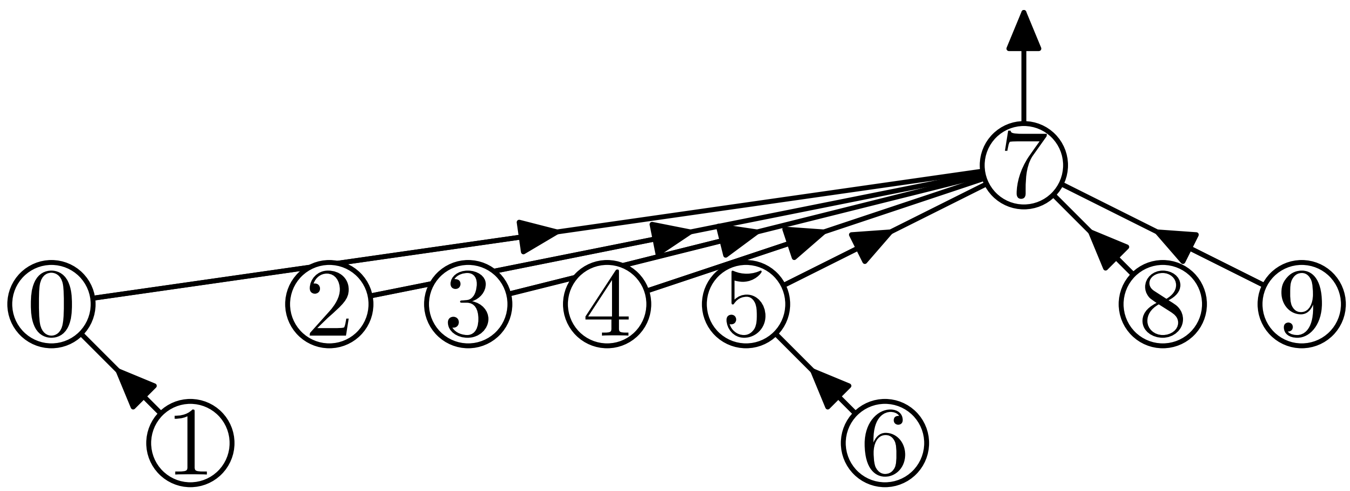
`union(find(1),find(2))`



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

Putting it Together

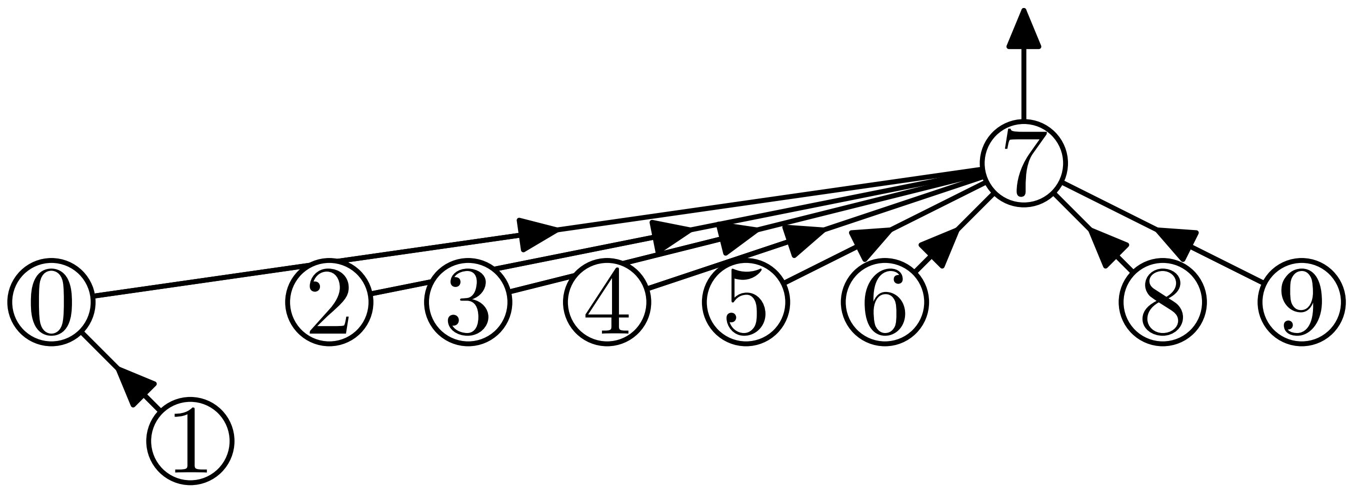
`find(6)`



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

Putting it Together

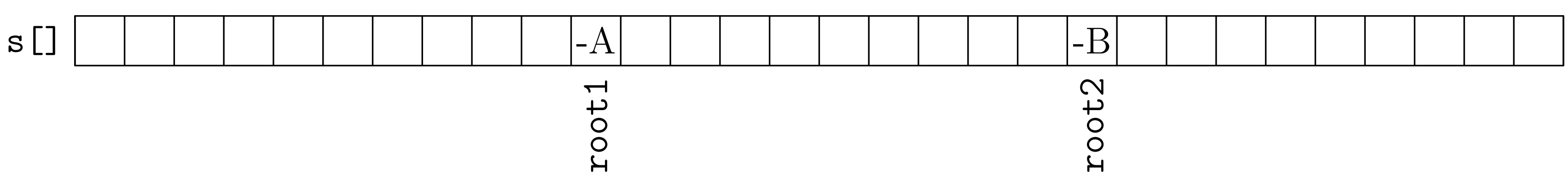
`find(6)=7`



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

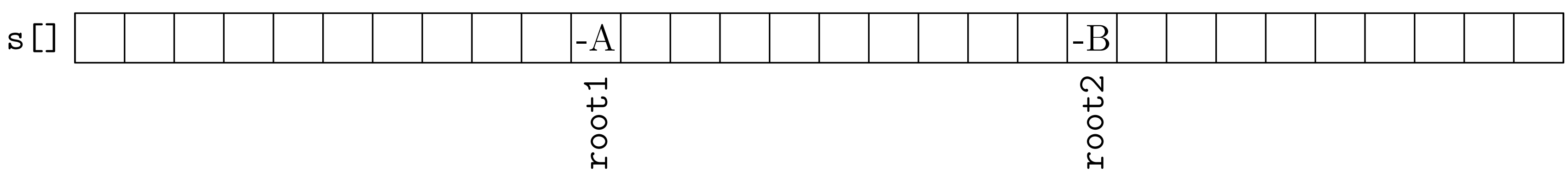
Smart Union

```
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
    }
}
```



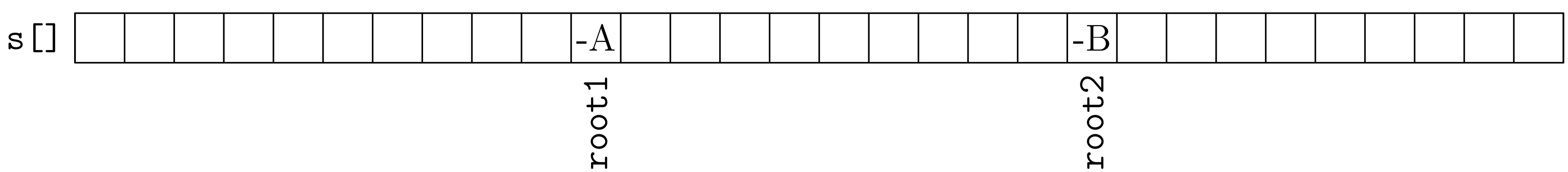
Smart Union

```
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
    }
}
```



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									

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

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

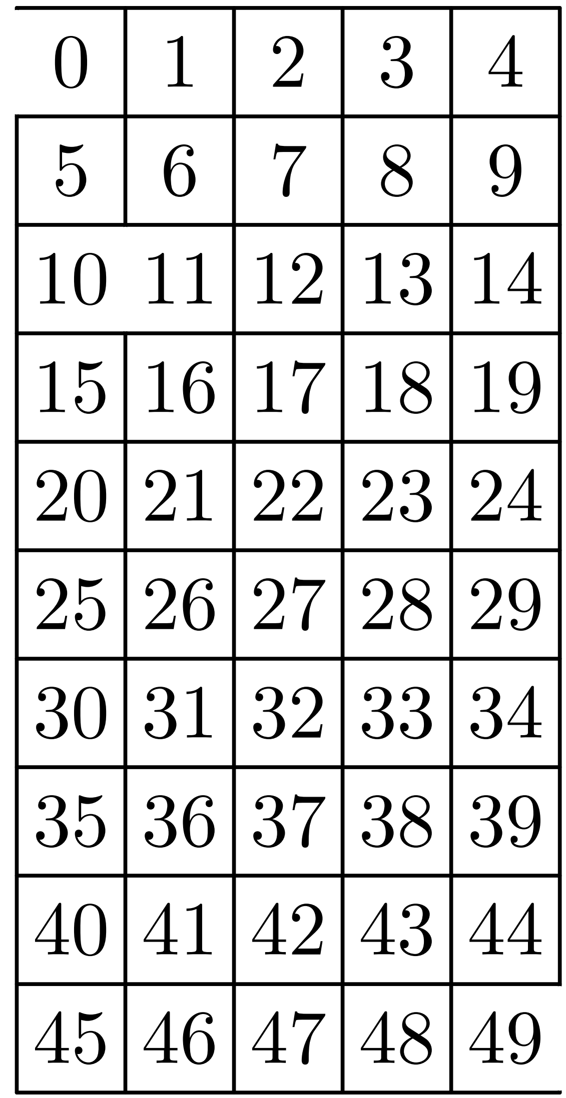
Mazes

- 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

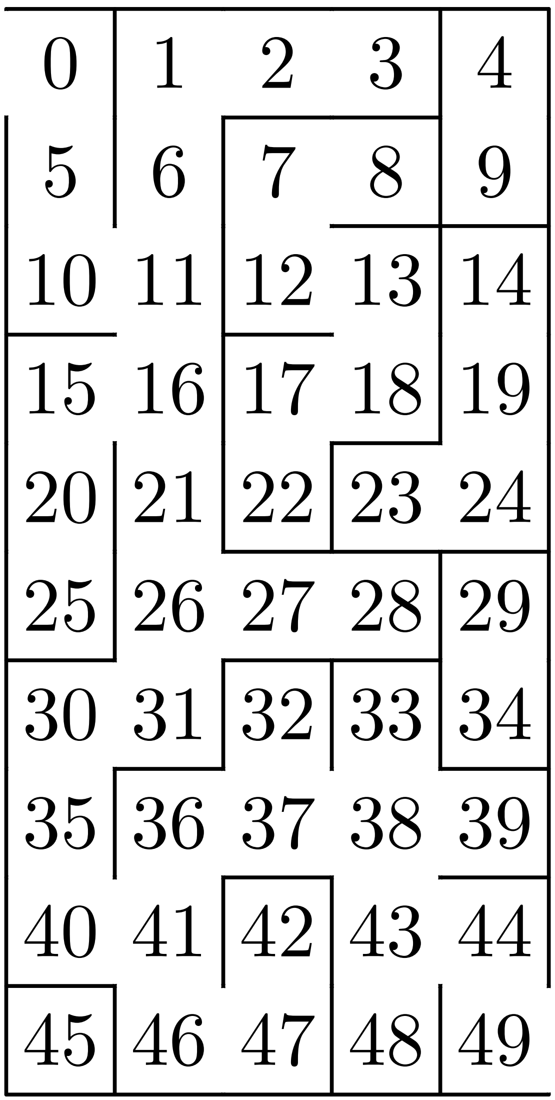
Mazes

- 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



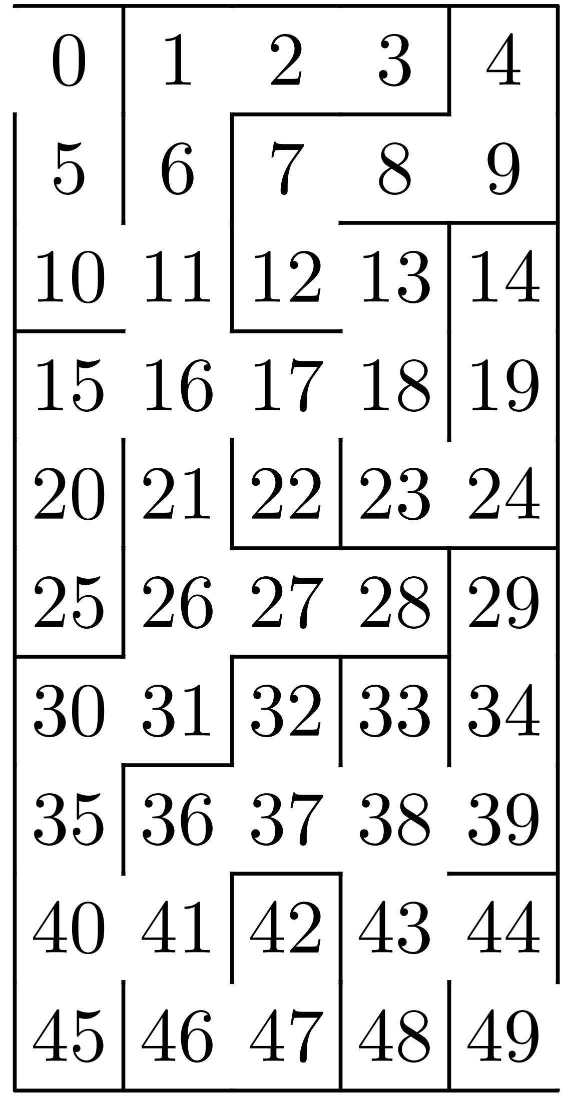
Mazes

- 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



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



Time Complexity of Union-Find

- If we perform M finds and N unions then the time complexity is $O\big(M\log_2^*(N)\big)$
- 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

Time Complexity of Union-Find

- If we perform M finds and N unions then the time complexity is $O\big(M \log_2^*(N)\big)$
- 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

Time Complexity of Union-Find

- If we perform M finds and N unions then the time complexity is $O\big(M \log_2^*(N)\big)$
- 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

Time Complexity of Union-Find

- If we perform M finds and N unions then the time complexity is $O\big(M \log_2^*(N)\big)$
- 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

$$\log_2(10^{80}) = 265.75$$

Time Complexity of Union-Find

- If we perform M finds and N unions then the time complexity is $O\big(M \log_2^*(N)\big)$
- 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

$$\log_2(\log_2(10^{80})) = 8.0539$$

Time Complexity of Union-Find

- If we perform M finds and N unions then the time complexity is $O\big(M\log_2^*(N)\big)$
- 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

$$\log_2(\log_2(\log_2(10^{80}))) = 3.0097$$

Time Complexity of Union-Find

- If we perform M finds and N unions then the time complexity is $O\big(M \log_2^*(N)\big)$
- 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

$$\log_2(\log_2(\log_2(\log_2(10^{80})))) = 1.5896$$

Time Complexity of Union-Find

- If we perform M finds and N unions then the time complexity is $O\big(M \log_2^*(N)\big)$

- 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

$$\log_2(\log_2(\log_2(\log_2(\log_2(10^{80}))))) = 0.66868$$

Time Complexity of Union-Find

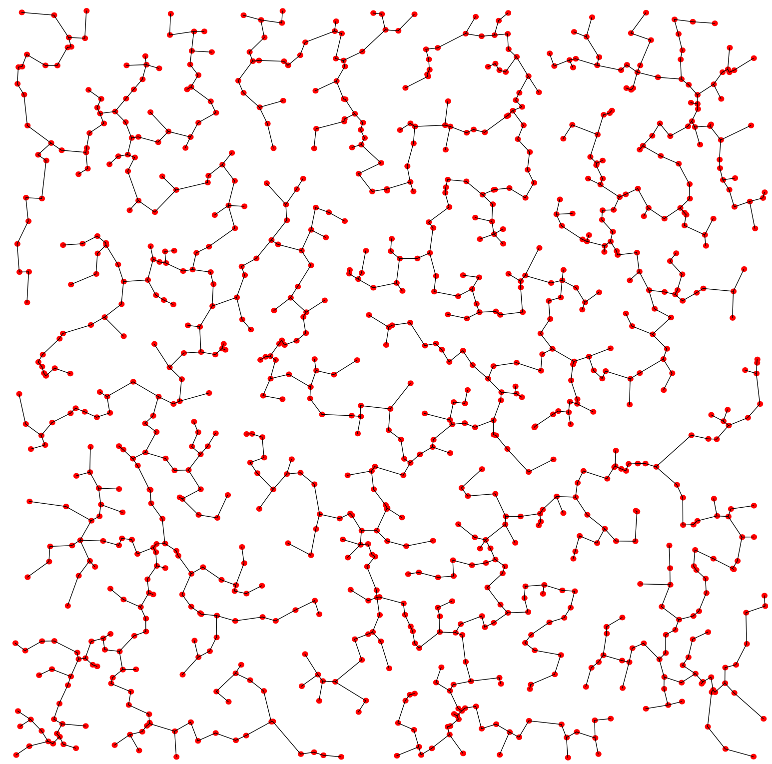
- If we perform M finds and N unions then the time complexity is $O\big(M \log_2^*(N)\big)$
- Where $\log_2^*(N)$ is the number of times you need to apply the logarithm function before you get a number less than 1
- In practice $\log_2^*(N) \leq 5$ for all conceivable N

$$\log_2(\log_2(\log_2(\log_2(\log_2(10^{80}))))) = 0.66868$$

- 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

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

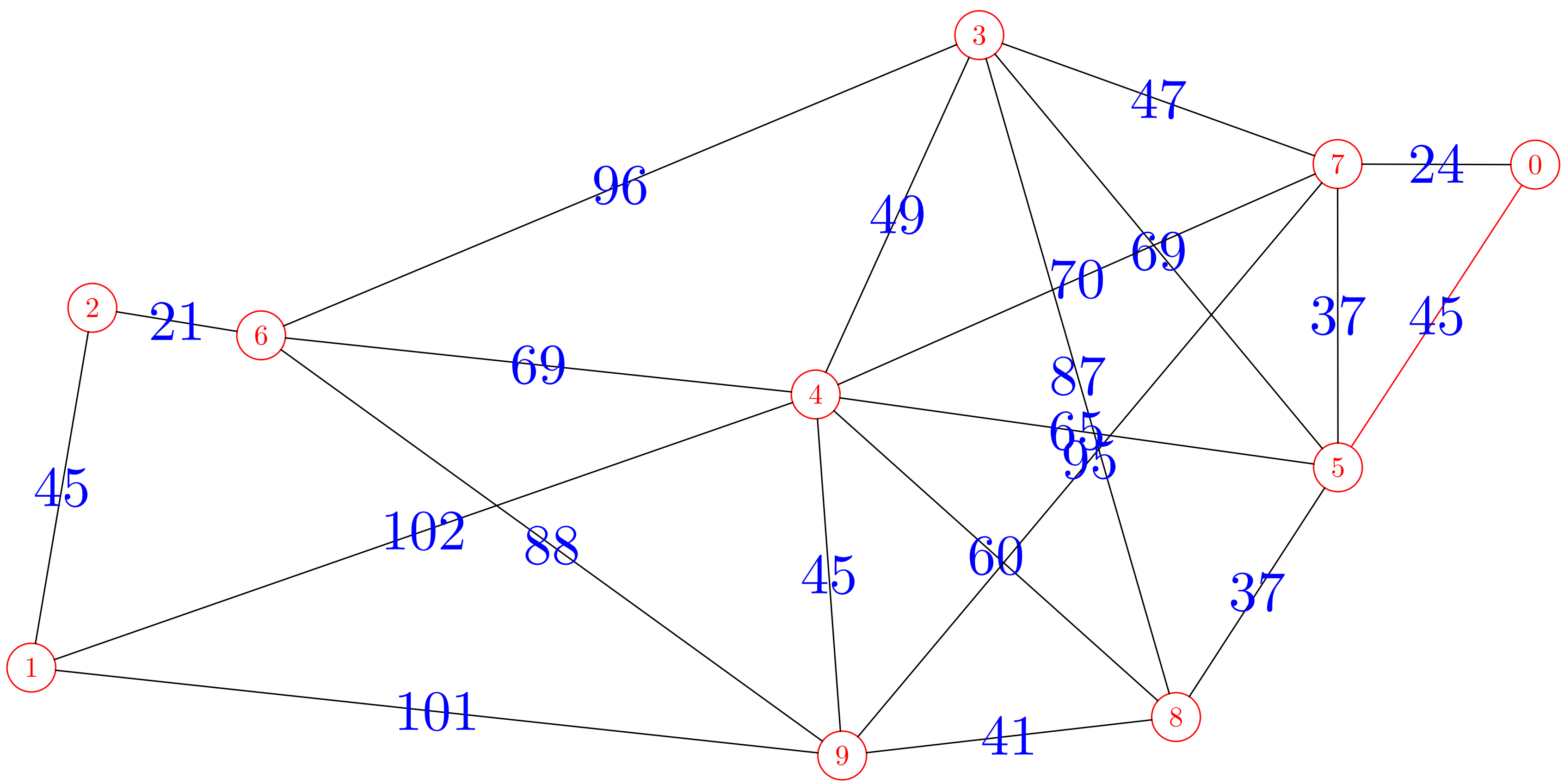
Shortest path

- 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

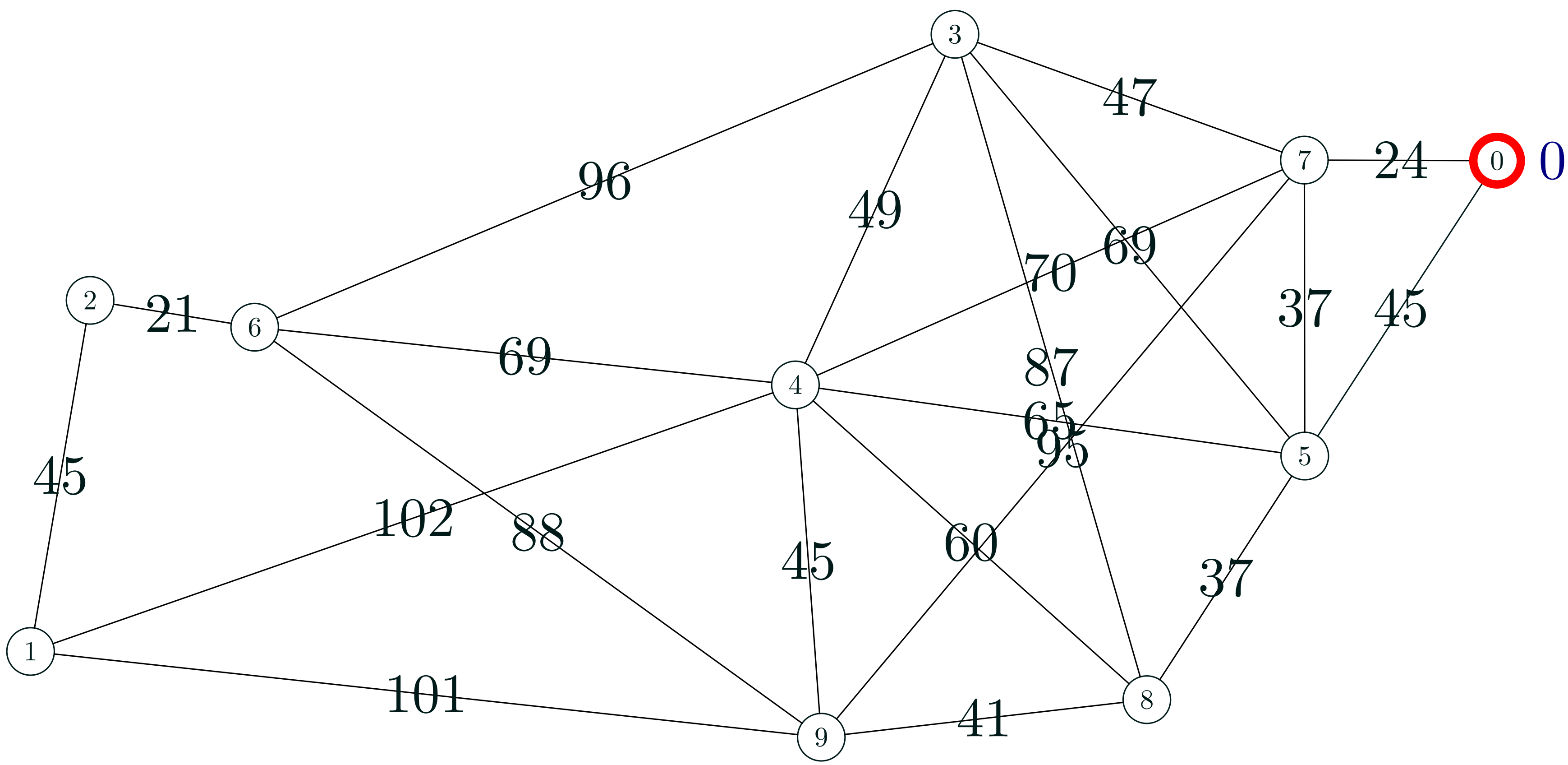
Shortest path

- 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

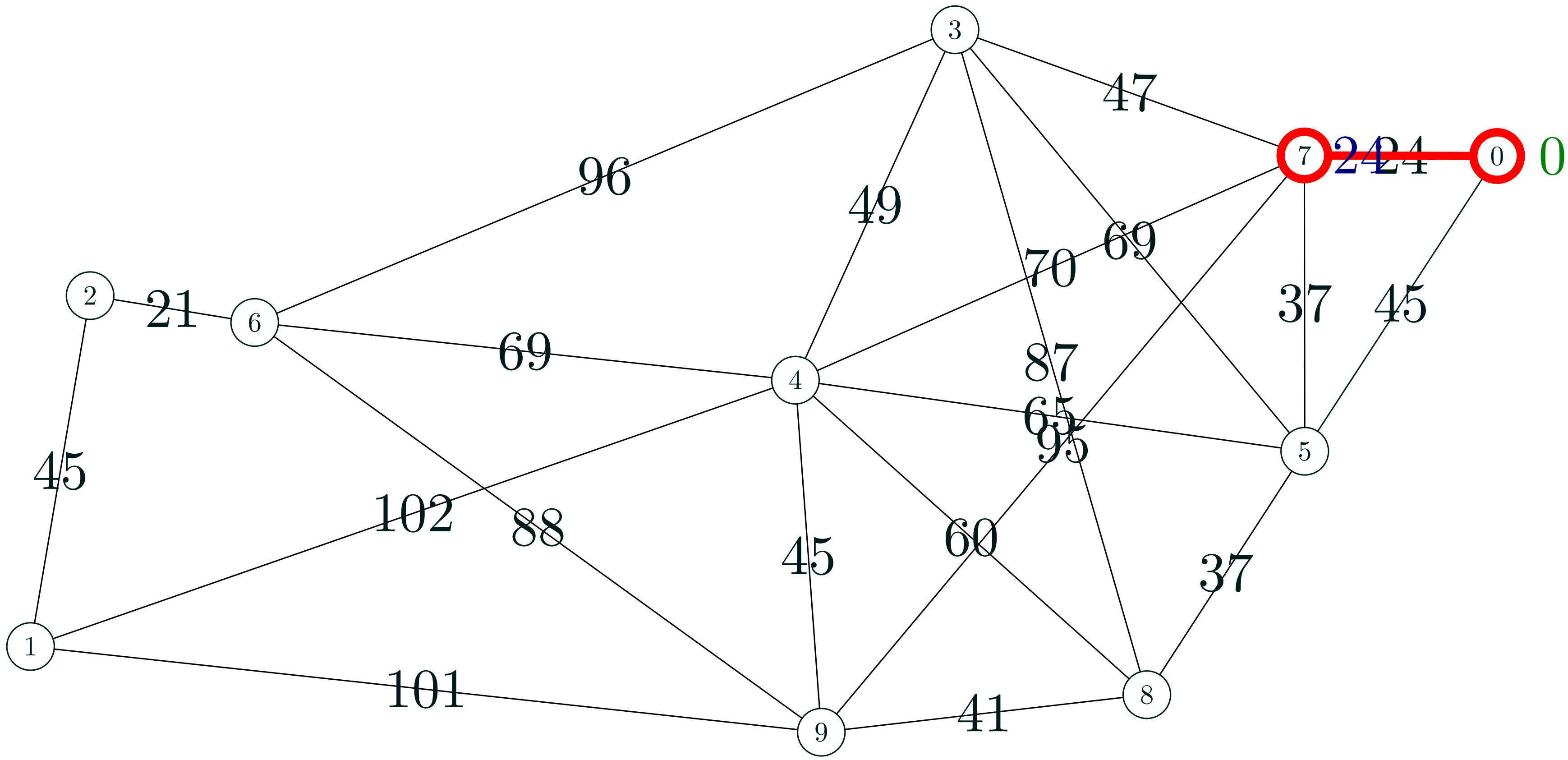
Dijkstra's Algorithm



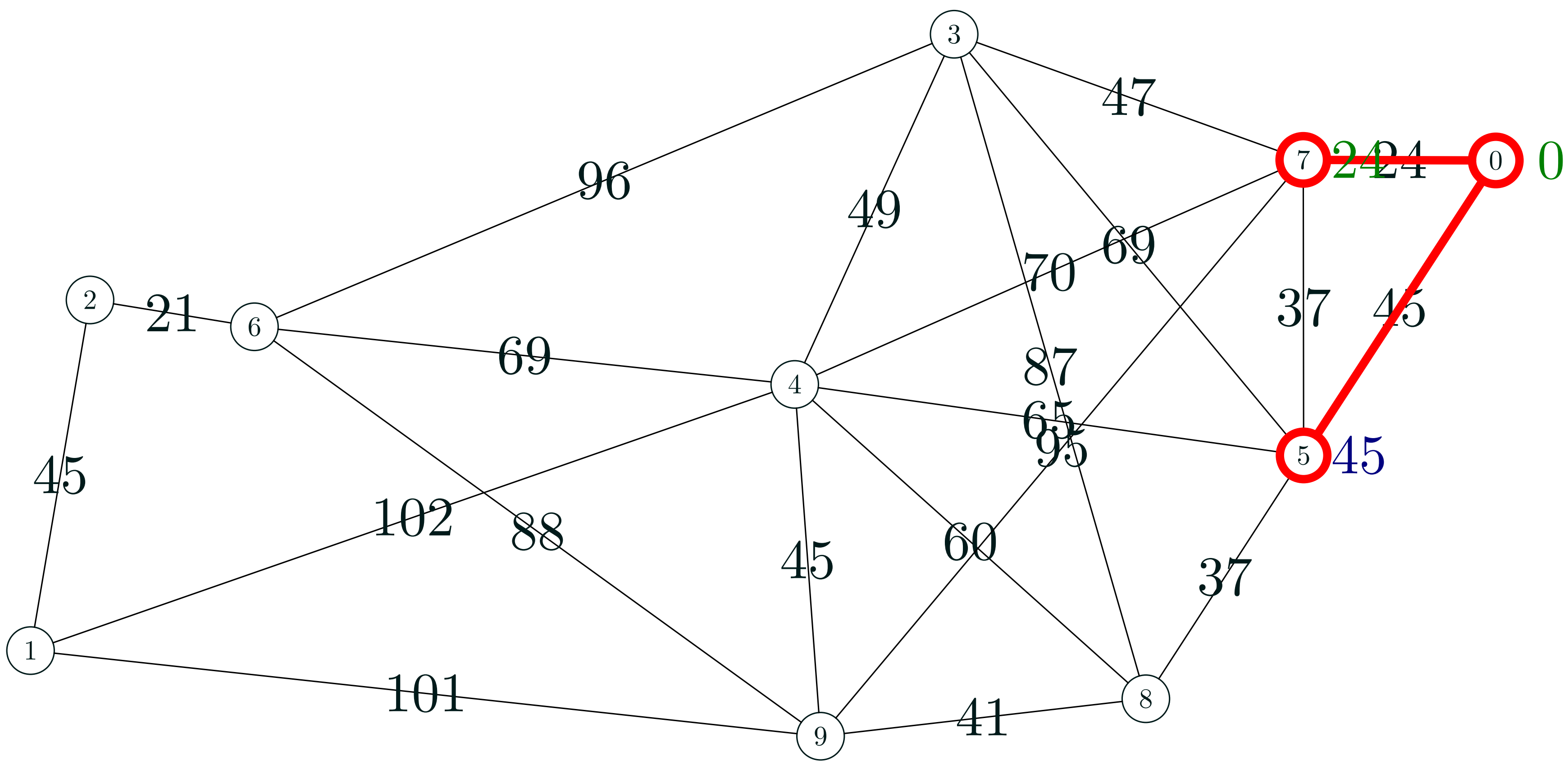
Dijkstra's Algorithm



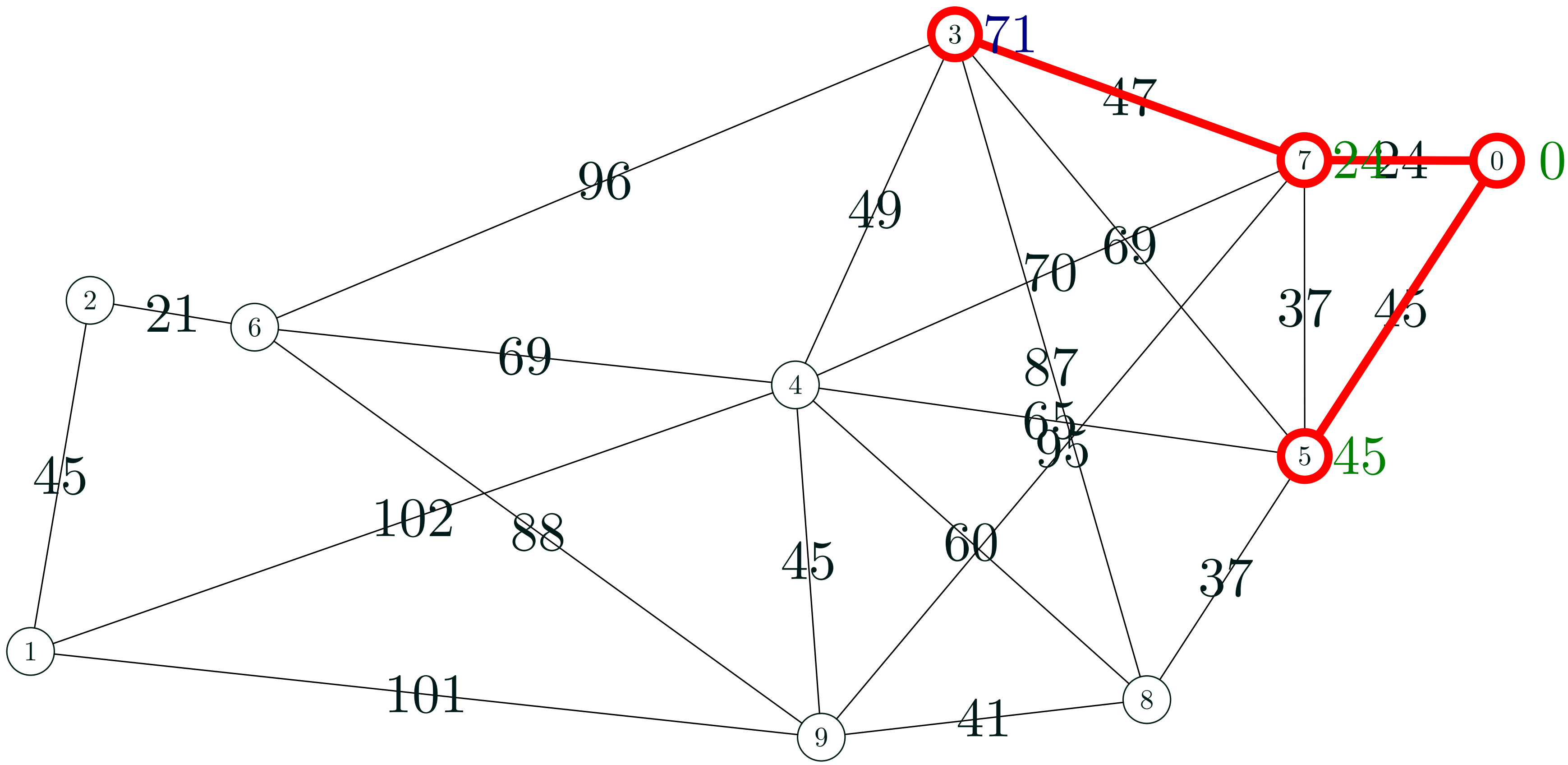
Dijkstra's Algorithm



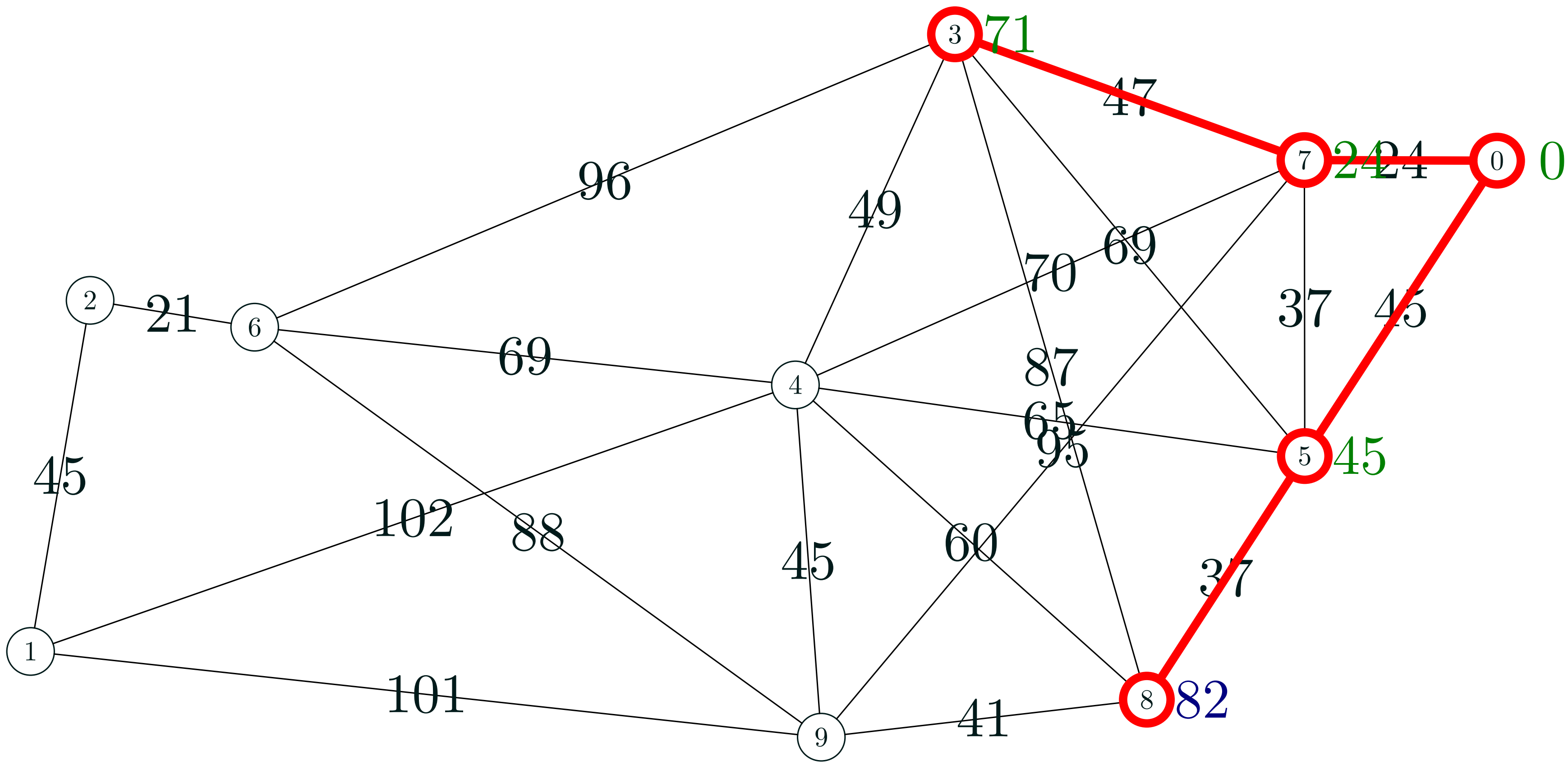
Dijkstra's Algorithm



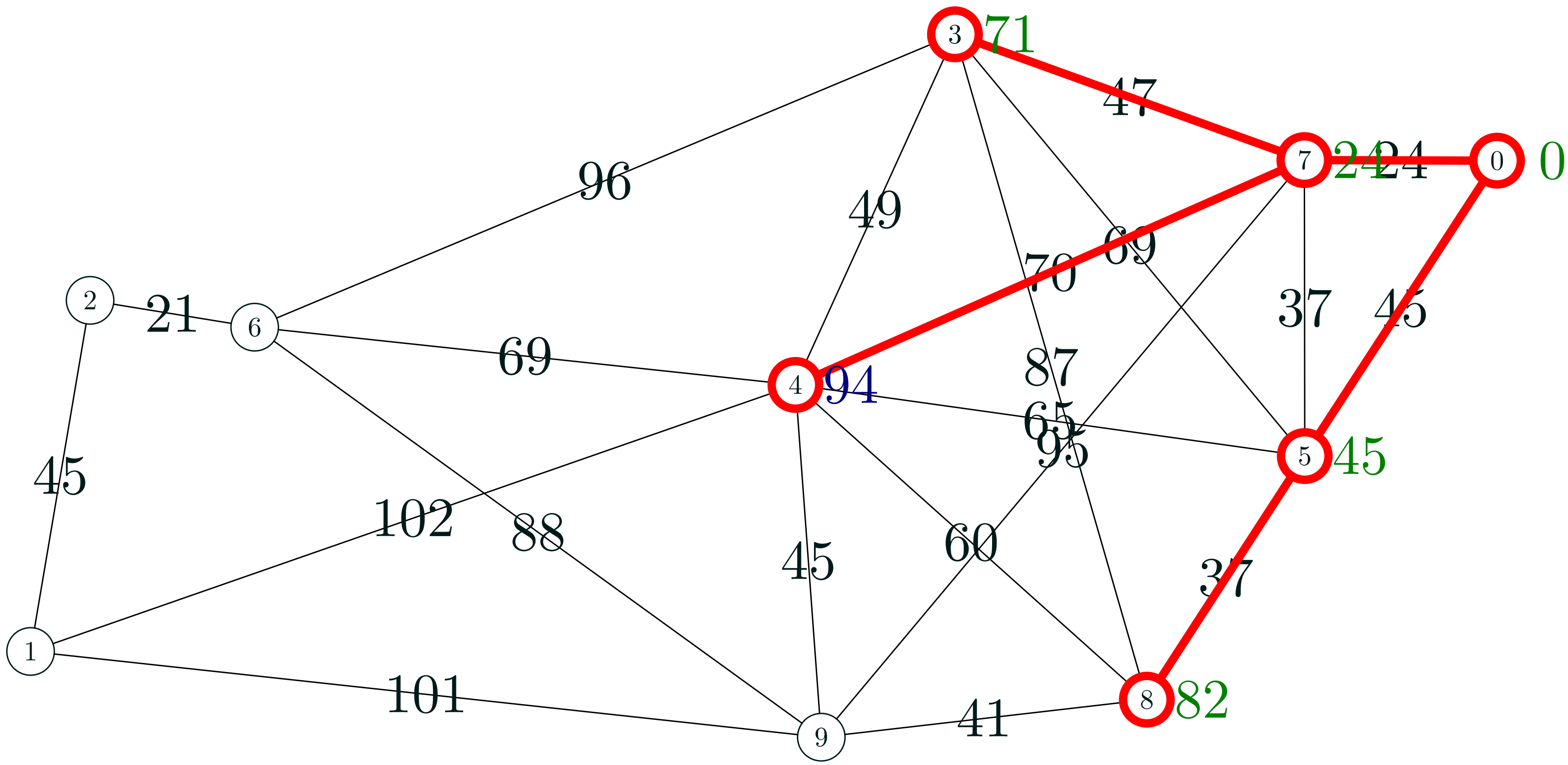
Dijkstra's Algorithm



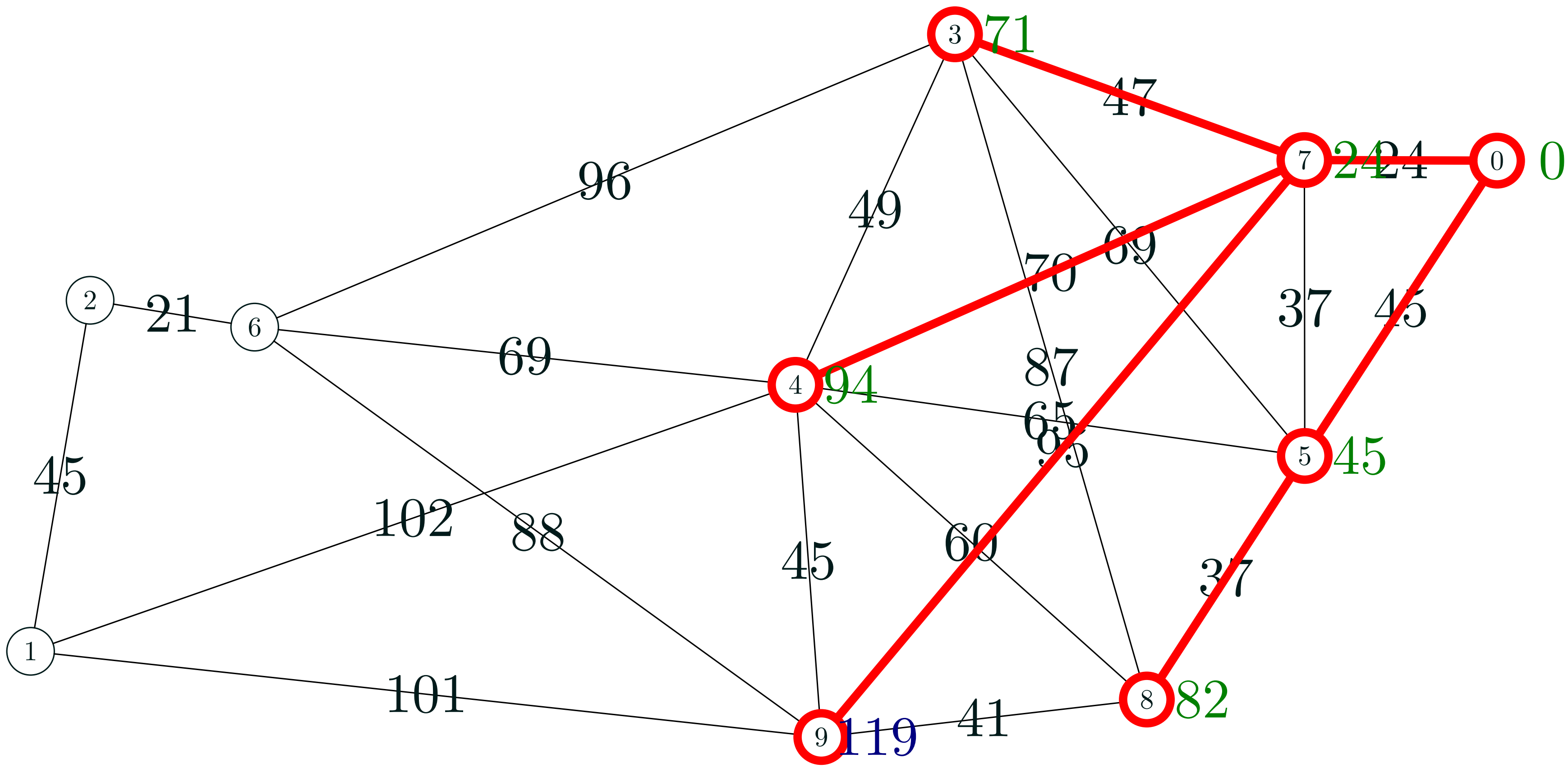
Dijkstra's Algorithm



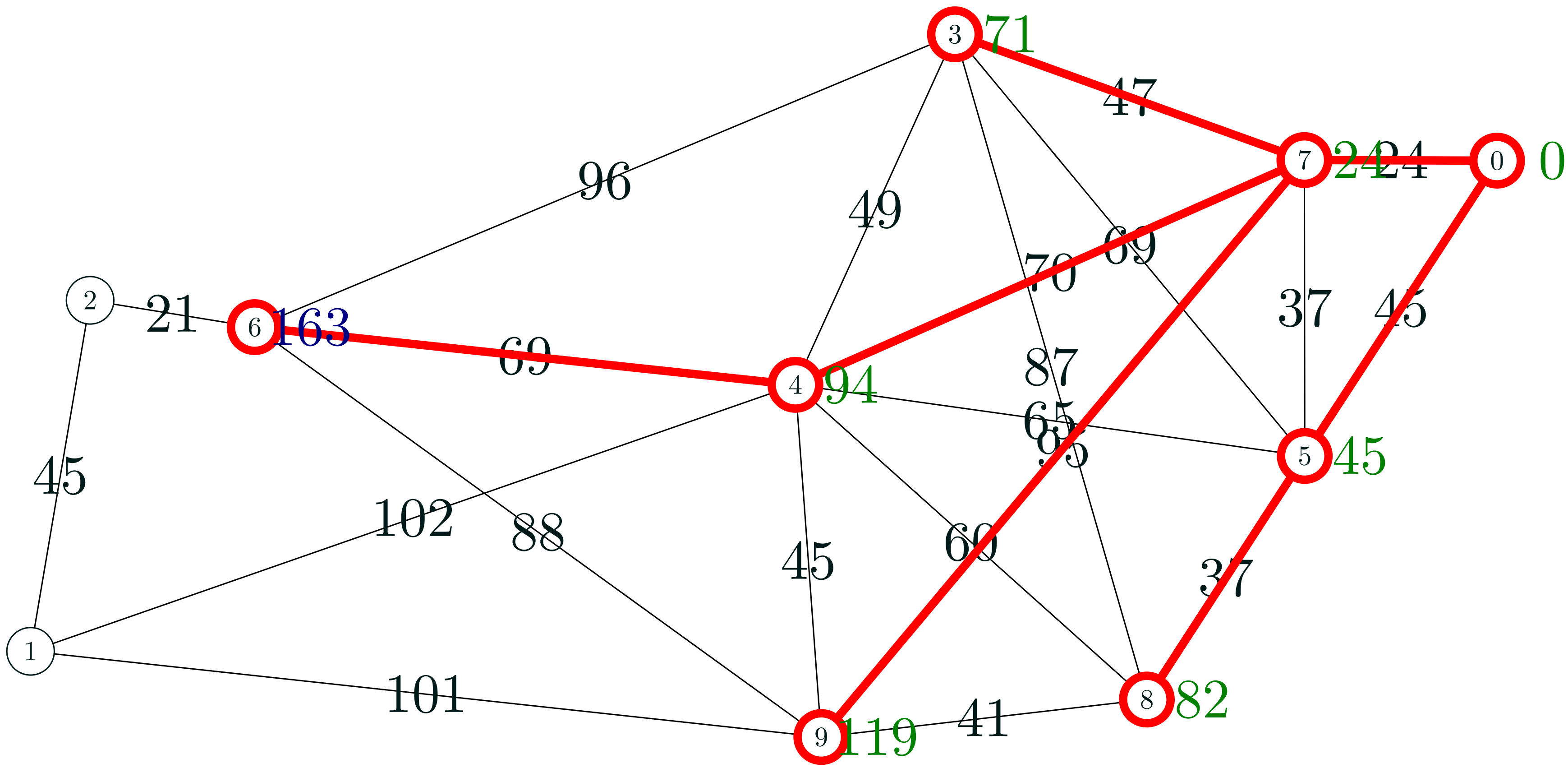
Dijkstra's Algorithm



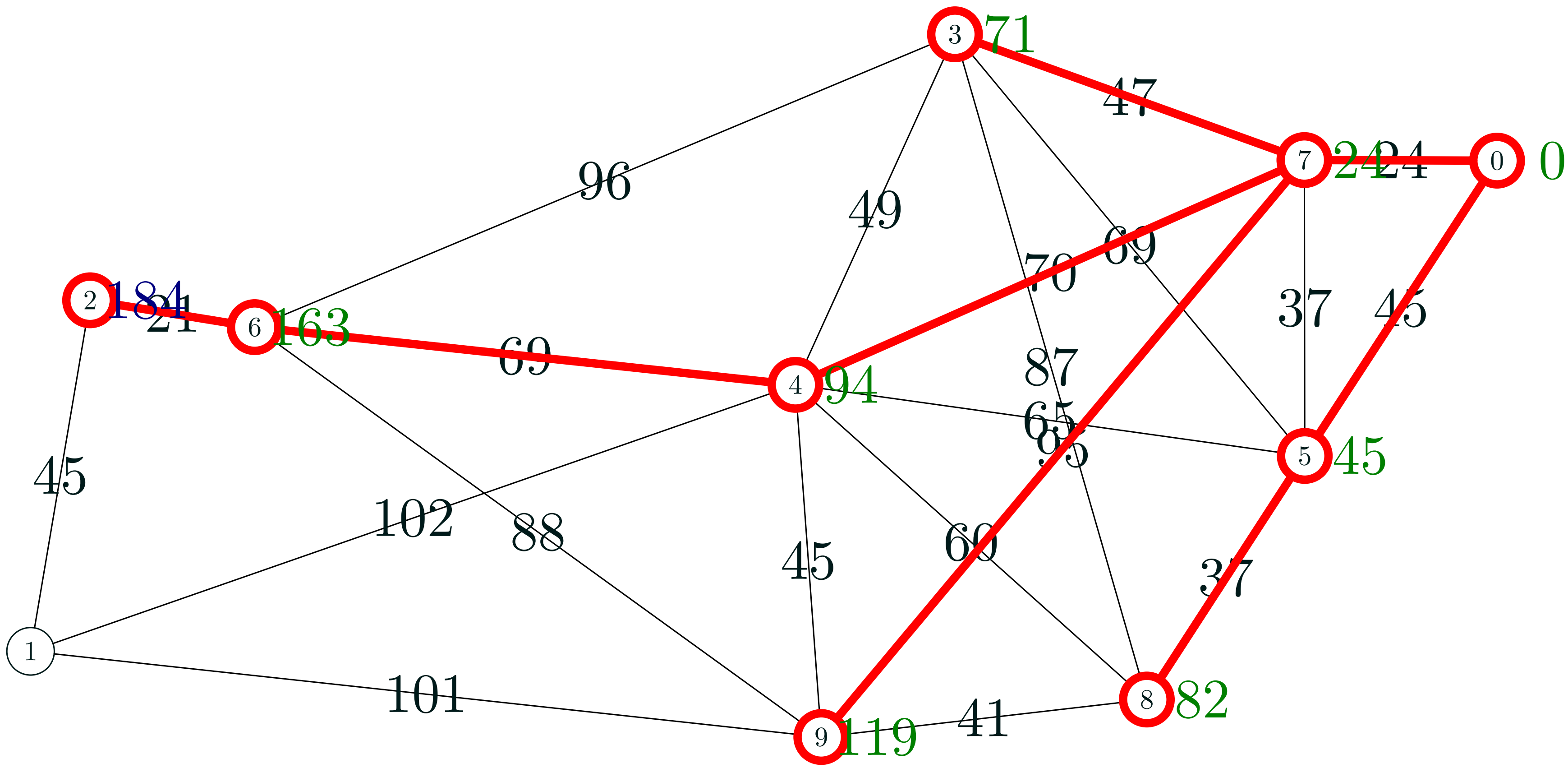
Dijkstra's Algorithm



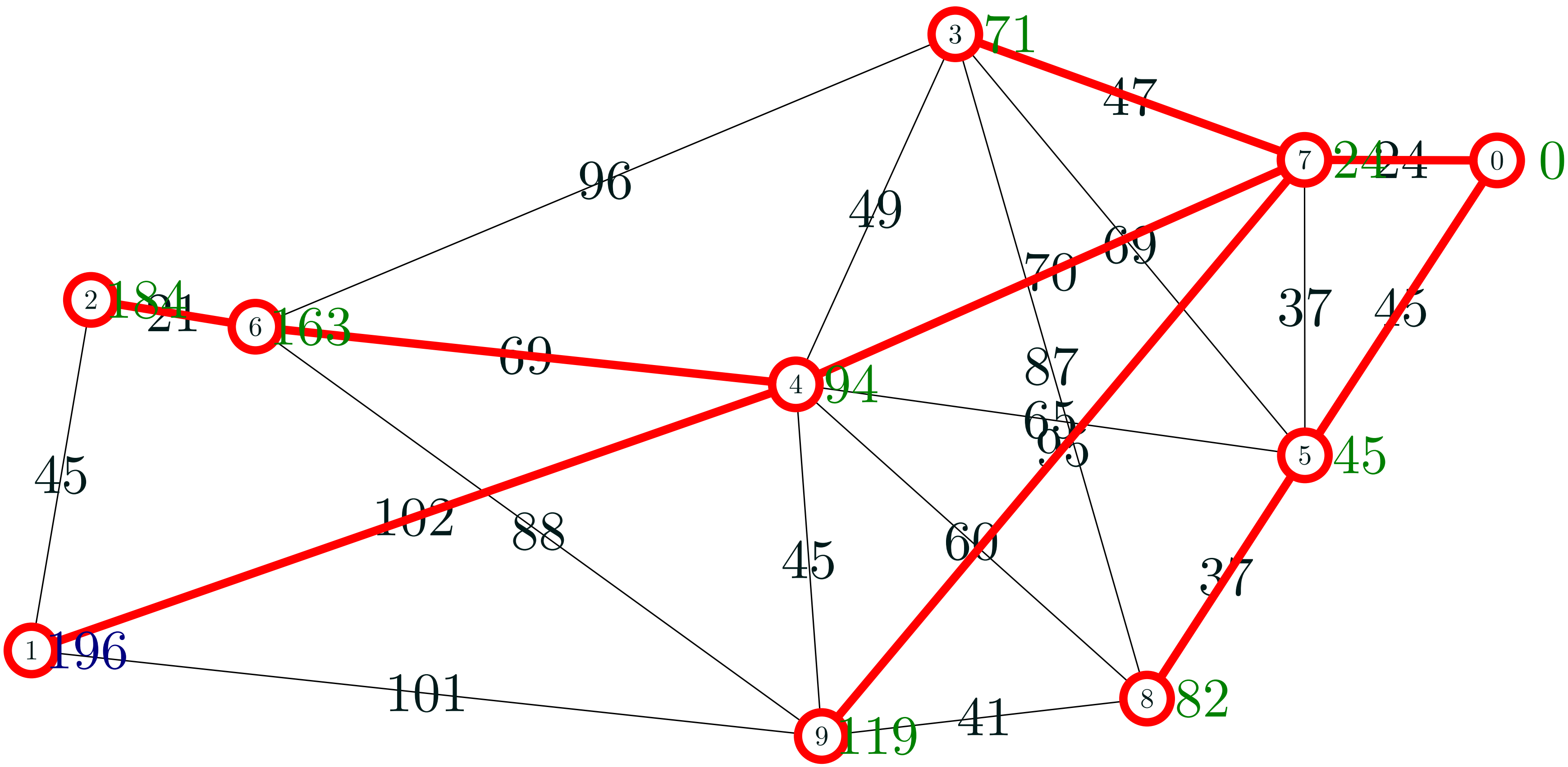
Dijkstra's Algorithm



Dijkstra's Algorithm



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 k ∈ {v ∈ V|(node, v) ∈ E}
         if ( w_node,k + d_node < d_k )
            d_k ←w_node,k + d_node
            PQ.add( (d_k, (node, k)) )
         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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,k + d_node < d_k  )
        d_k ←w_node,k + d_node
        PQ.add( (d_k, (node, k)) )
      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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,k + d_node < d_k )
        d_k ←w_node,k + d_node
        PQ.add( (d_k, (node, k)) )
      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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_{node,k} + d_node < d_k  )
        d_k ←w_{node,k} + d_node
        PQ.add( (d_k, (node, k)) )
      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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_{node,k} + d_node < d_k )
        d_k ←w_{node,k} + d_node
        PQ.add( (d_k, (node, k)) )
      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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,k + d_node < d_k )
        d_k ←w_node,k + d_node
        PQ.add( (d_k, (node, k)) )
      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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,k + d_node < d_k )
        d_k ←w_node,k + d_node
        PQ.add( (d_k, (node, k)) )
      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
}
```

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 k ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_{node,k} + d_node < d_k )
        d_k ←w_{node,k} + d_node
        PQ.add( (d_k, (node, k)) )
      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 k ∈ {v ∈ V|(node, v) ∈ E}
         if ( w_node,k < d_k  )
            d_k ←w_node,k
            PQ.add( (d_k, (node,k)) )
         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)

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)

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)

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

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

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

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

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