Data structures and Algorithms

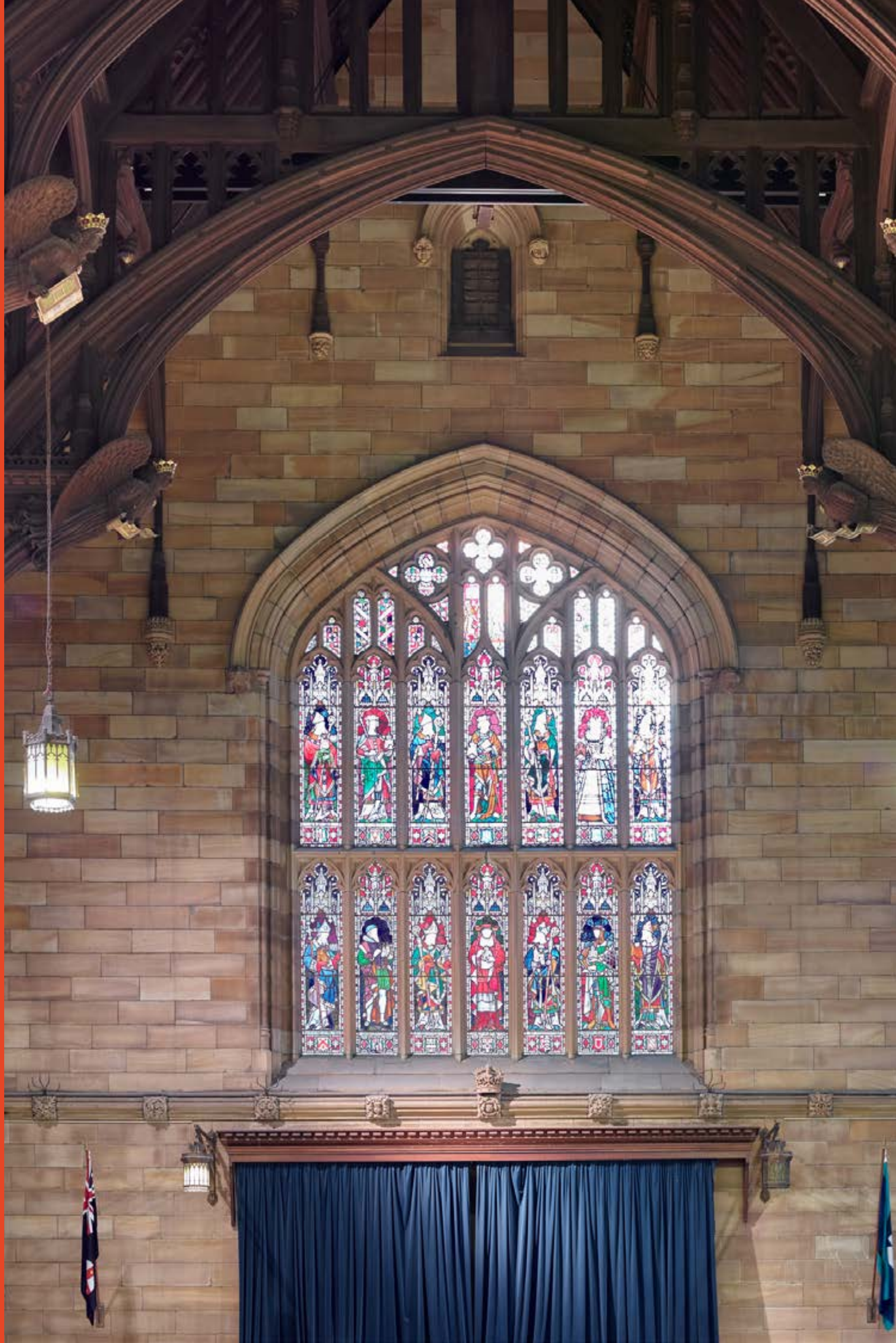
Lecture 9: Graph Algorithms
[GT 14.1-2, 15.1-3]

Lecturer: Dr. Karlos Ishac

Co-ordinator: Dr. Ravihansa Rajapakse
School of Computer Science

Some content is taken from the textbook publisher Wiley and previous Co-ordinator Dr. Andre van Renssen.

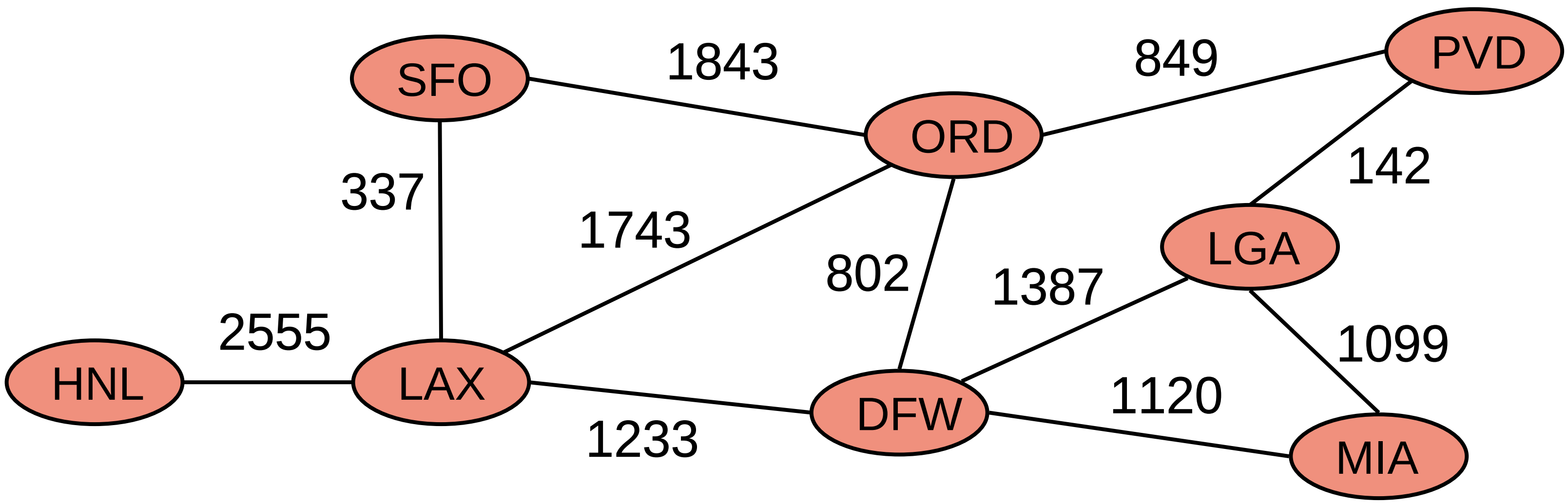
THE UNIVERSITY OF SYDNEY



Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.

Example: In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports

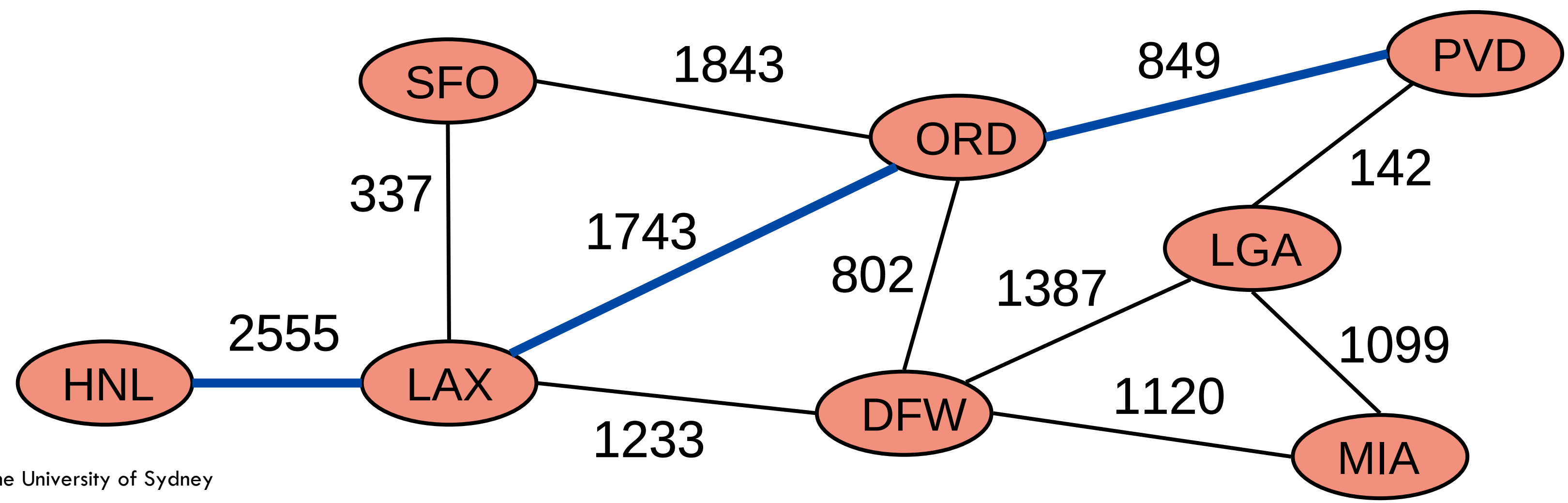


Shortest Paths

Given an edge weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v, where the weight of a path is the sum of the weights of its edges.

Applications: Internet packet routing, flight reservations and driving directions.

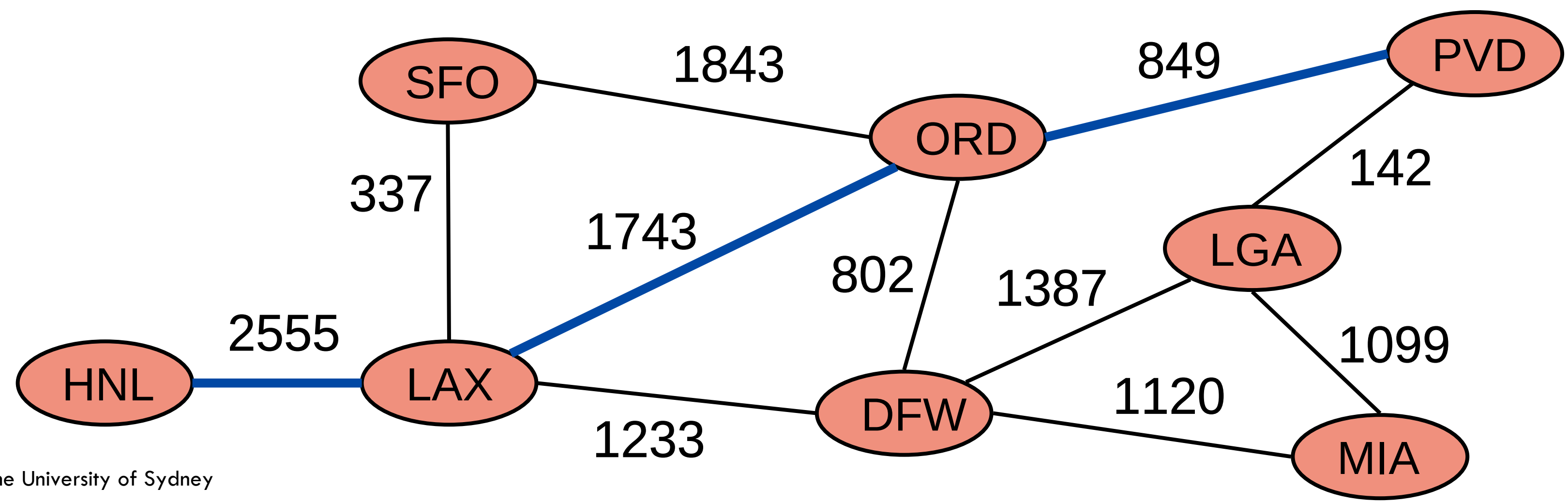
Example: Shortest path between Providence (PVD) and Honolulu (HNL)



Shortest Path Properties

Property: A subpath of a shortest path is itself a shortest path

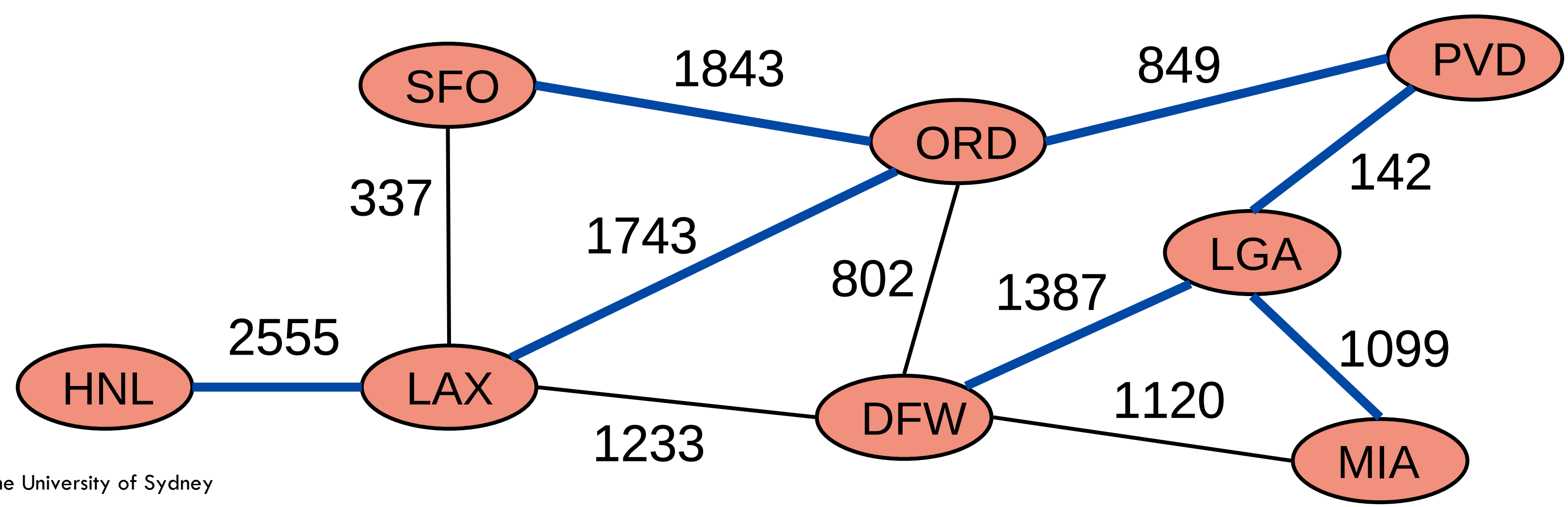
Example: Shortest path from Providence (PVD) to Honolulu (HNL) also contains a shortest path from Providence (PVD) to Los Angeles (LAX)



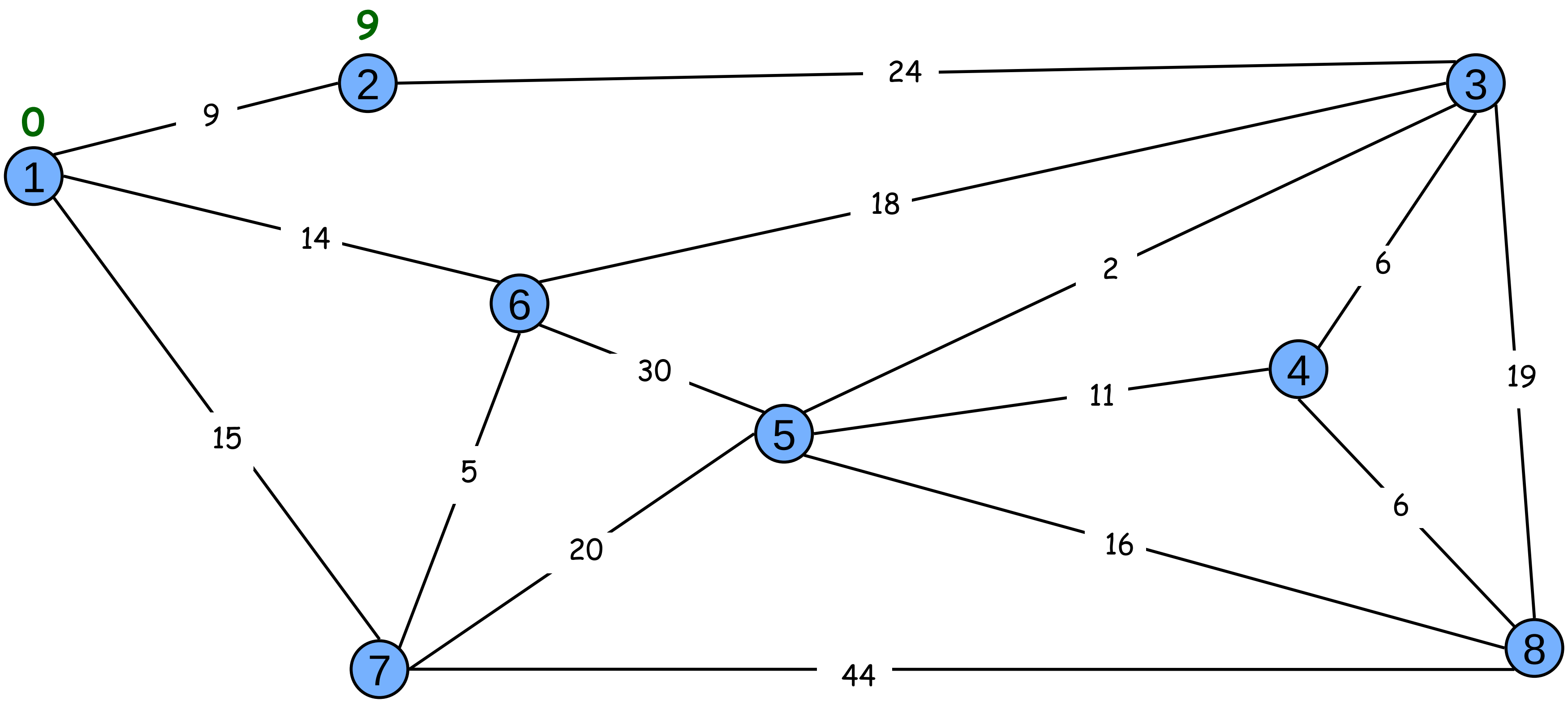
Shortest Path Properties

Property: There is a tree of shortest paths from a start vertex to all the other vertices (shortest path tree).

Example: Tree of shortest paths from Providence (PVD)



Shortest Paths



Dijkstra's Algorithm

Input:

- Graph $G = (V, E)$
- Edges weights $w : E \to R_+$
- Start vertex s

Output:

- Distance from s to all v in V
- Shortest path tree rooted at s

Assumptions:

- G is connected and undirected
- edge weights are nonnegative

High level idea:

- Maintain a distance estimate
  $D[v] \geq dist_w(s, v)$ for all v in V
- Keep track of a subset S of V s.t.
  $D[v] = dist_w(s, v)$ for all v in S

Initially:

- $D[s] = 0$
- $D[v] = \infty$ for all v in $V - s$

In each iteration we:

- add to S vertex u in $V \setminus S$ with smallest $D[u]$
- update D-values for vertices adjacent to u

Edge Relaxation

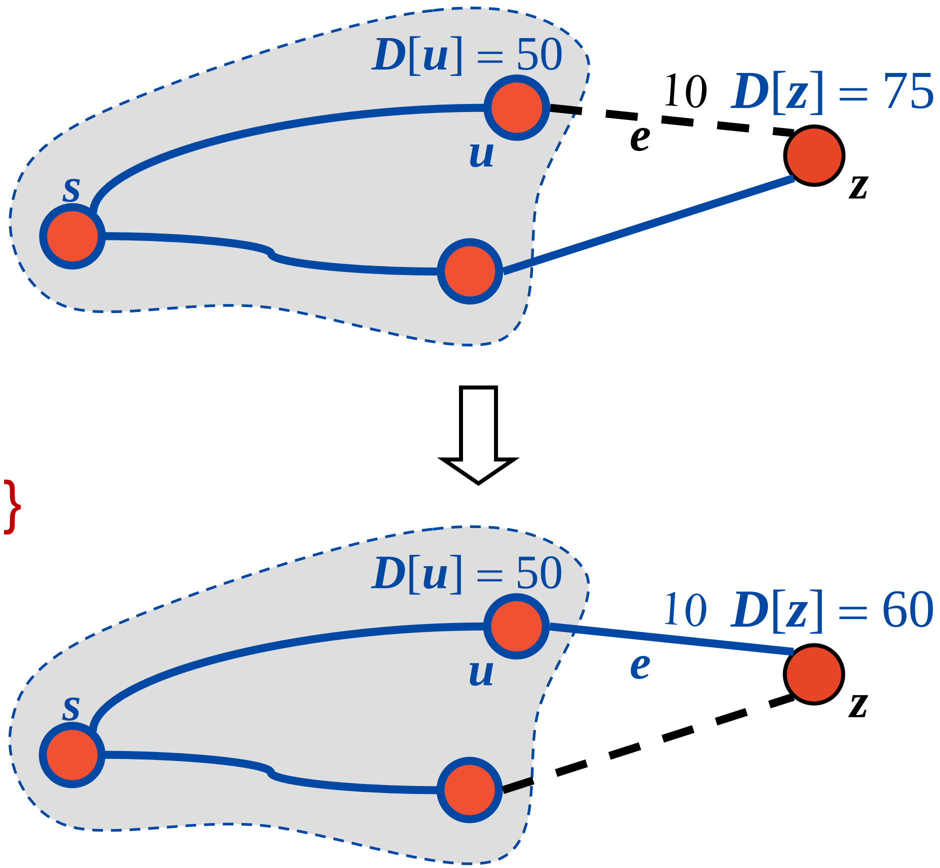
Consider edge e = (u, z) such that:

– u is the last vertex added to S

– z is not in S

The relaxation of edge (u, z) updates D[z] as follows:

$$D[z] \leftarrow \min\{D[z], D[u] + w(u, z)\}$$



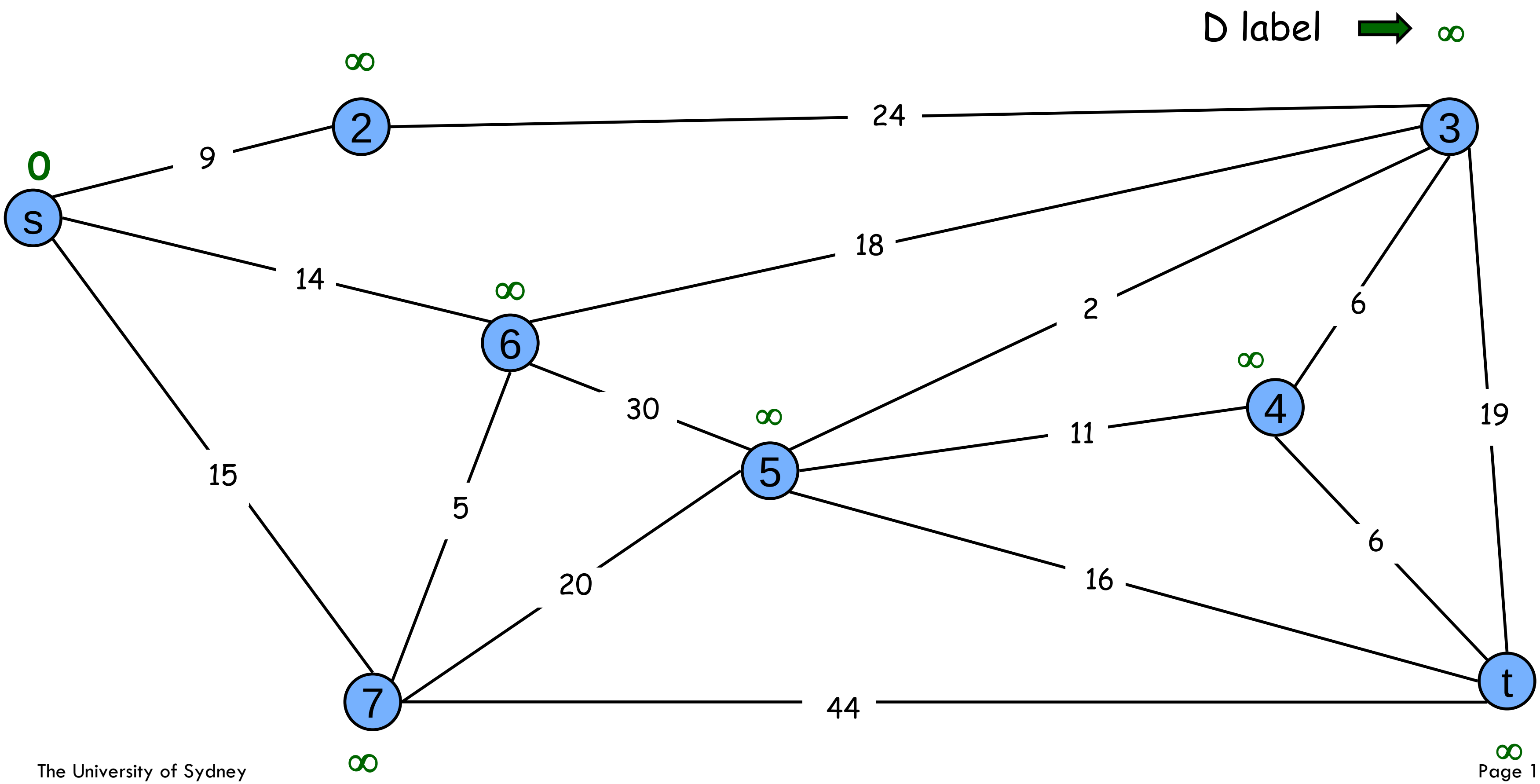
Dijkstra's Algorithm pseudocode

```
def Dijkstra(G, w, s):

 # initialize algorithm
 for v in V do
  D[v] ← ∞
  parent[v] ← ∅
 D[s] ← 0
 Q ← new priority queue for { (v, D[v]) : v in V }

 # iteratively add vertices to S
 while Q is not empty do
  u ← Q.remove_min()
  for z in G.neighbors(u) do
   if D[u] + w[u, z] < D[z] then
    D[z] ← D[u] + w[u, z]
    Q.update_priority(z, D[z])
    parent[z] ← u
 return D, parent
```

Dijkstra's Shortest Path Algorithm

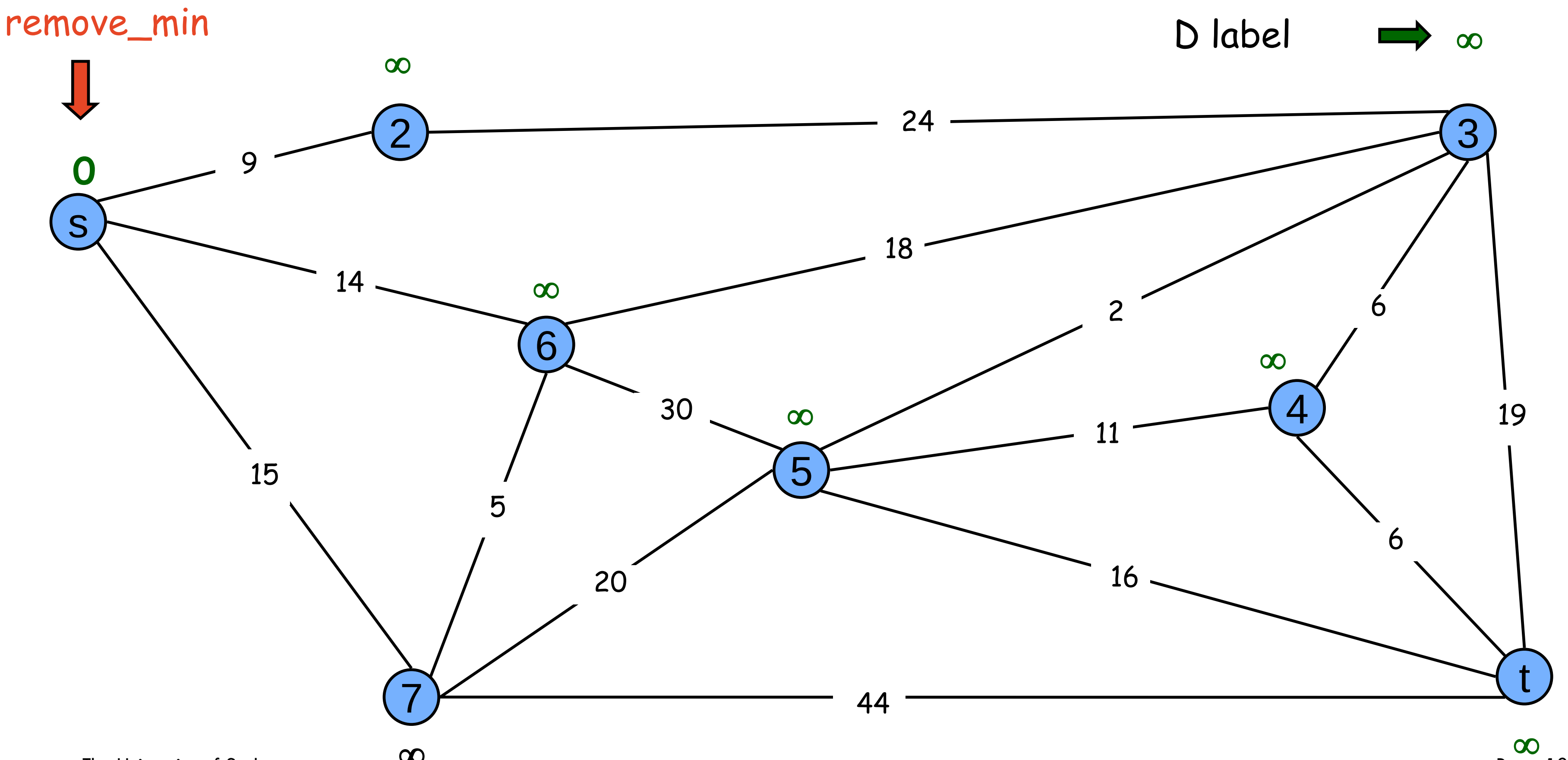
S = { }

PQ = { s, 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

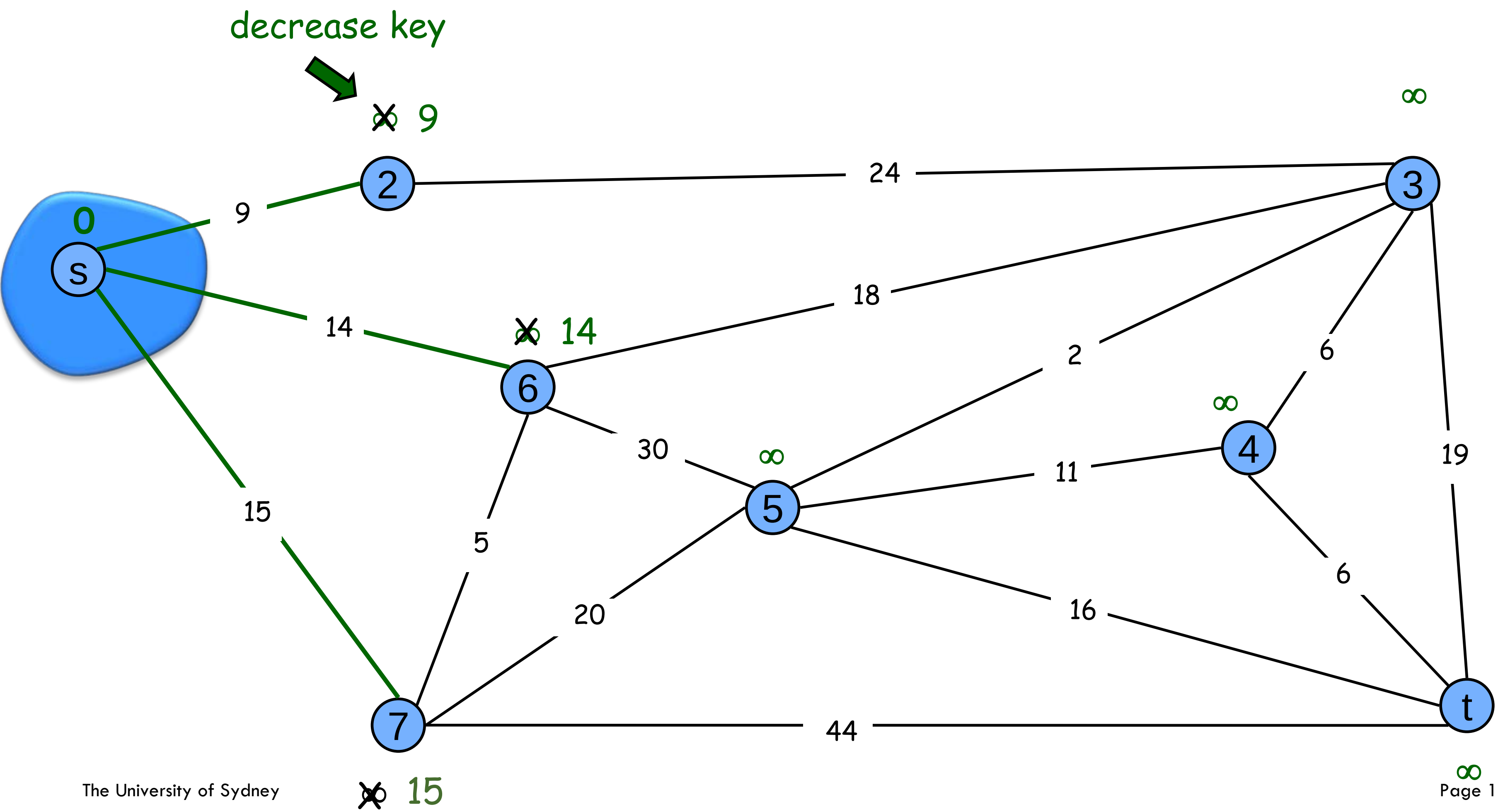
S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s }

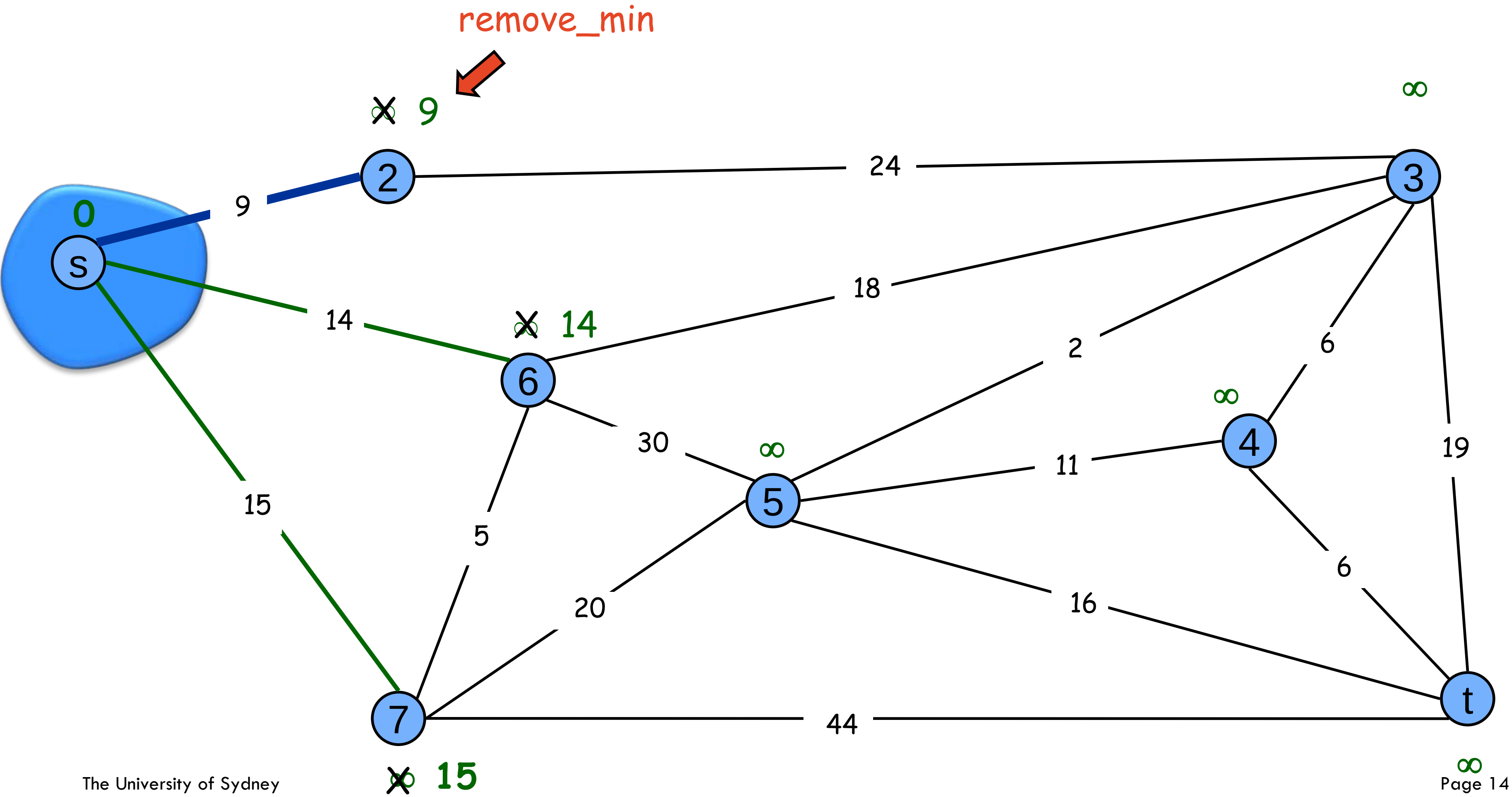
PQ = { 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s }

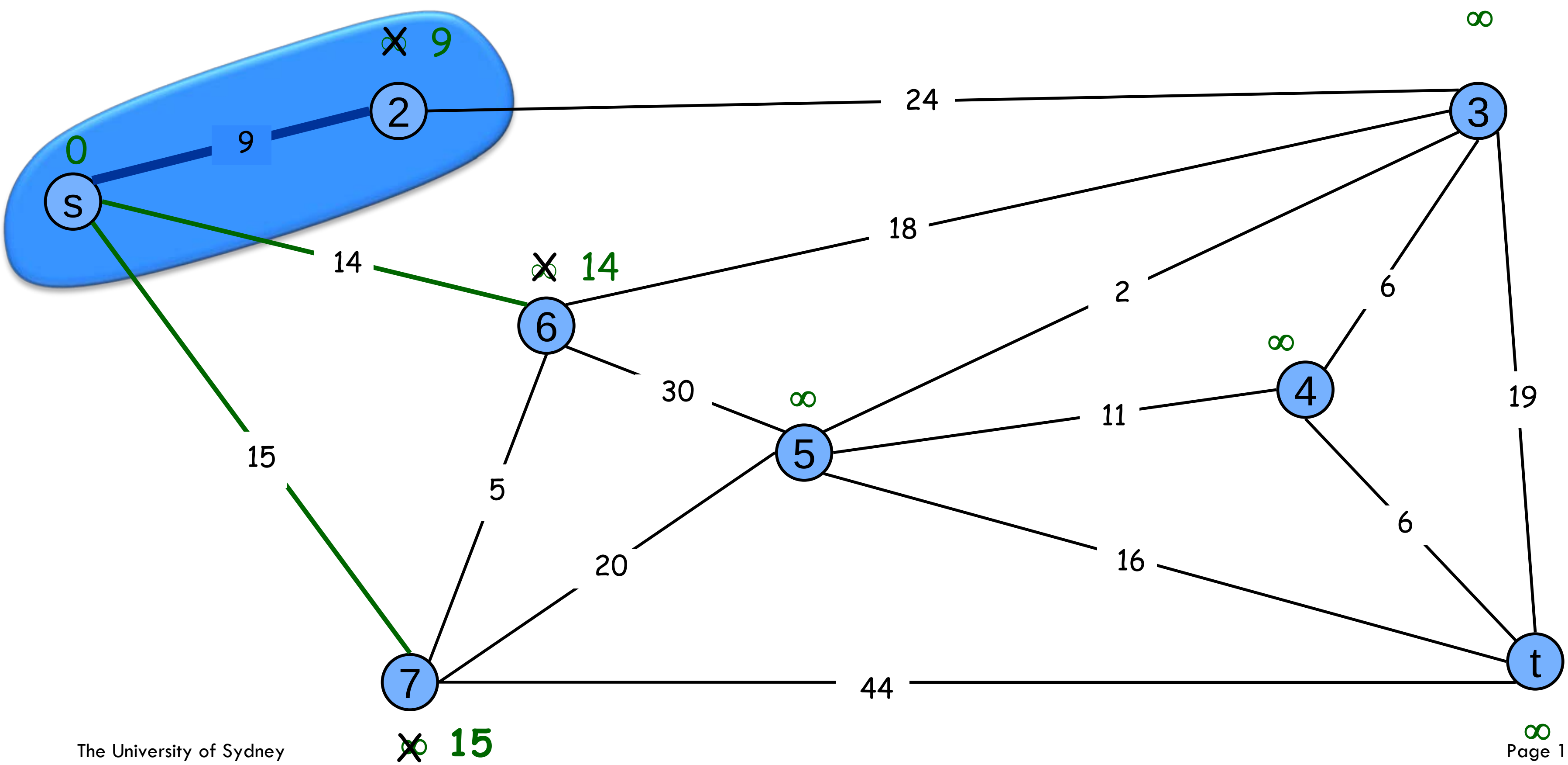
PQ = { 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2 }

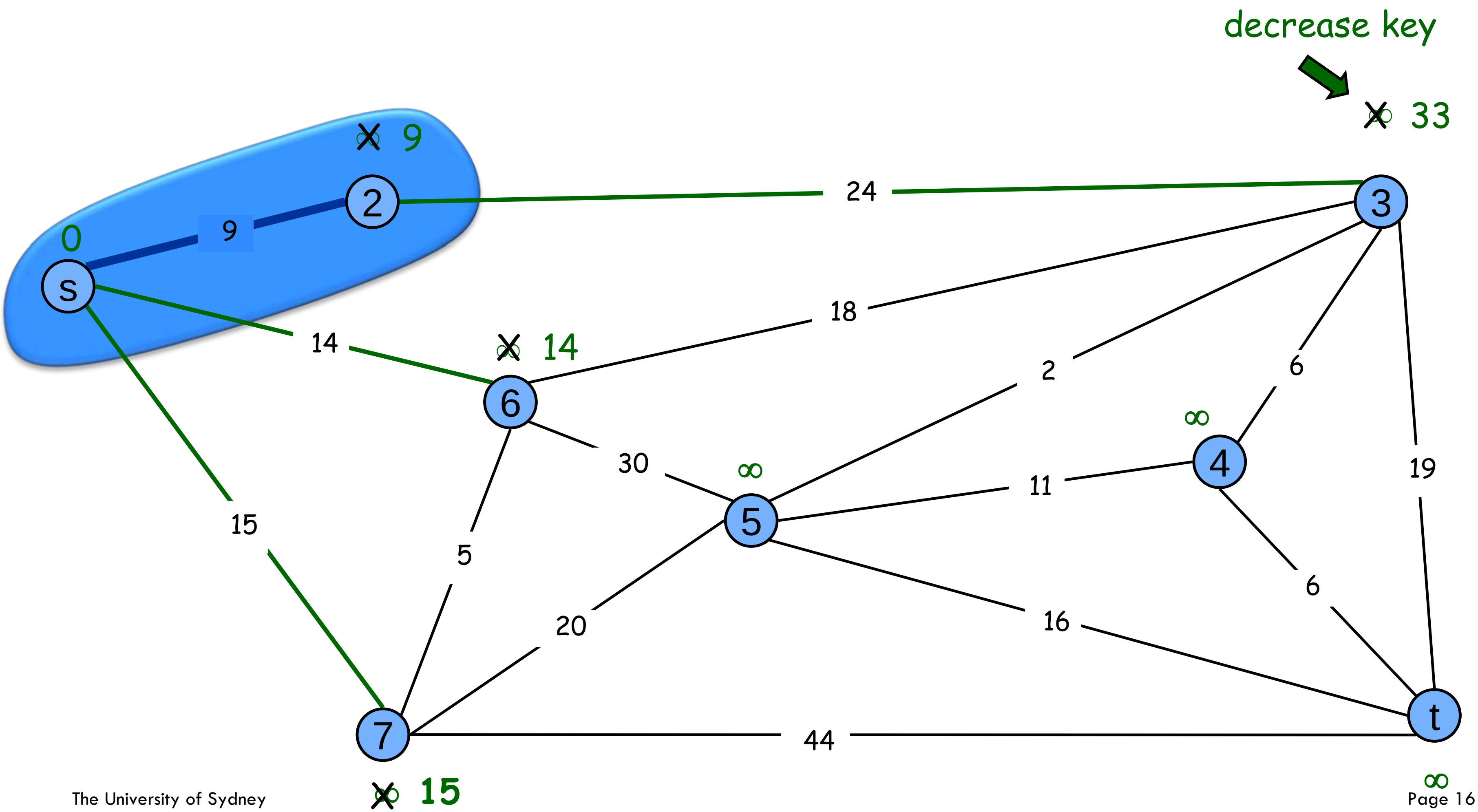
PQ = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2 }

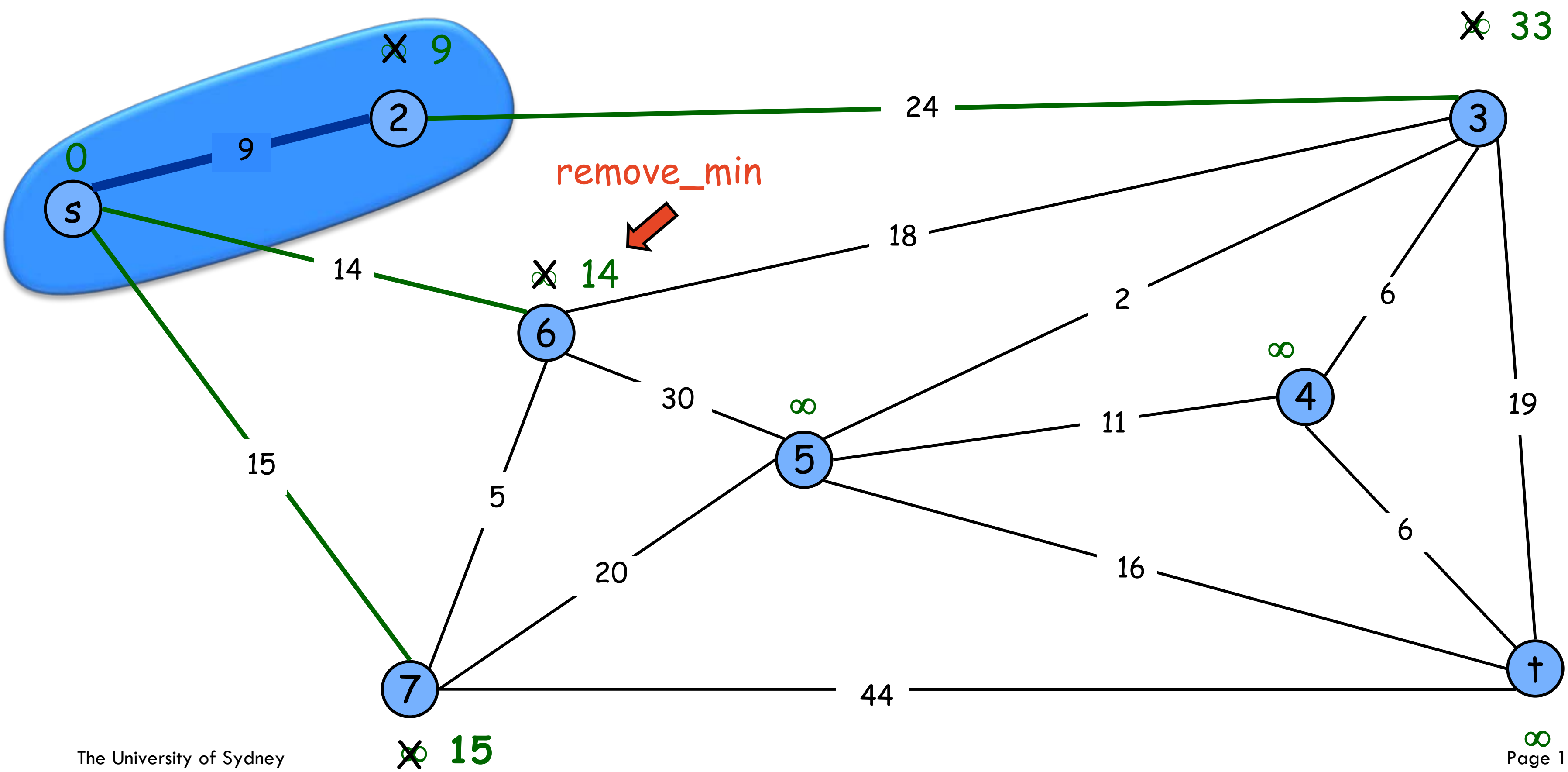
PQ = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2 }

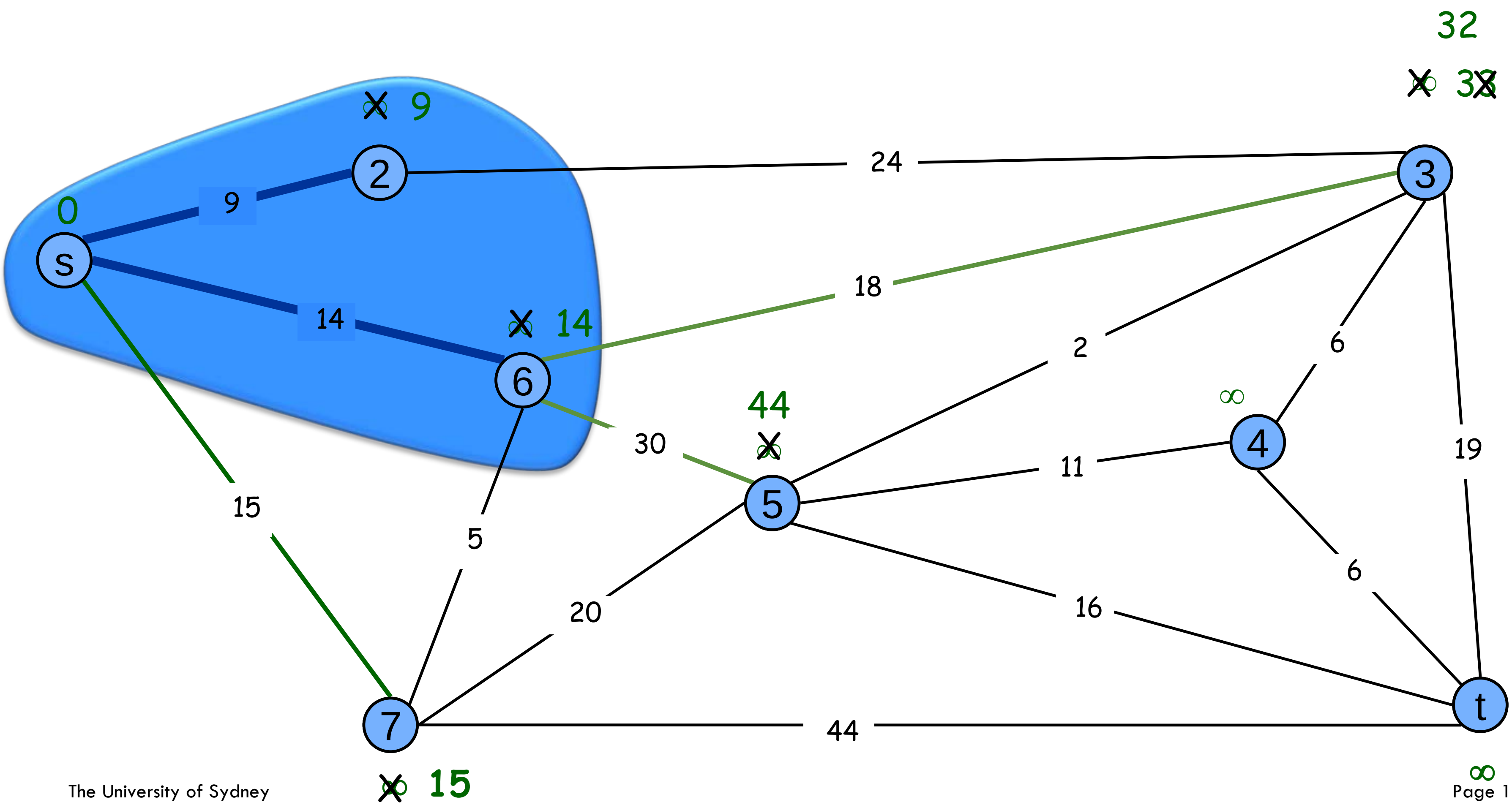
PQ = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

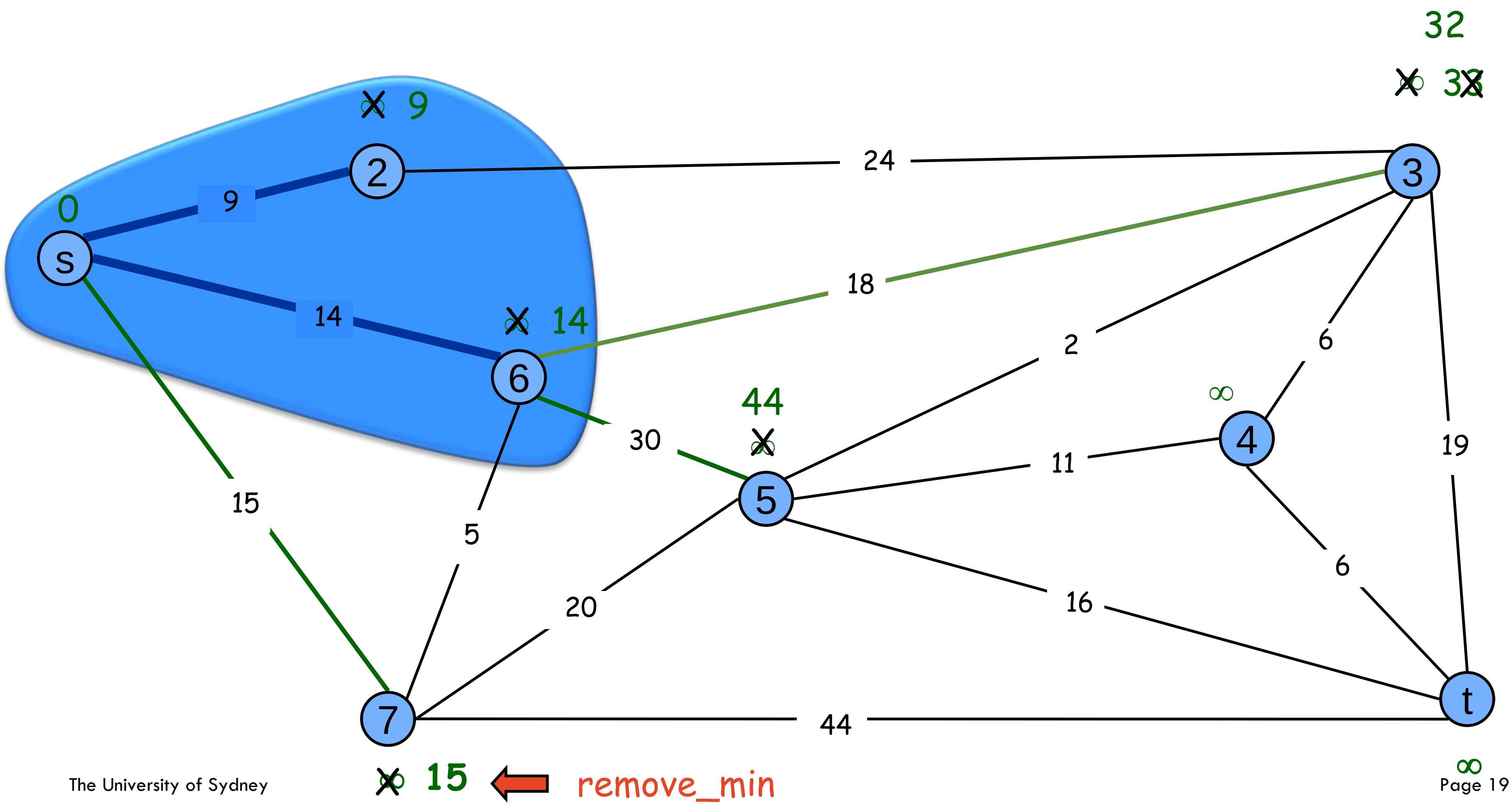
S = { s, 2, 6 }

PQ = { 3, 4, 5, 7, t }



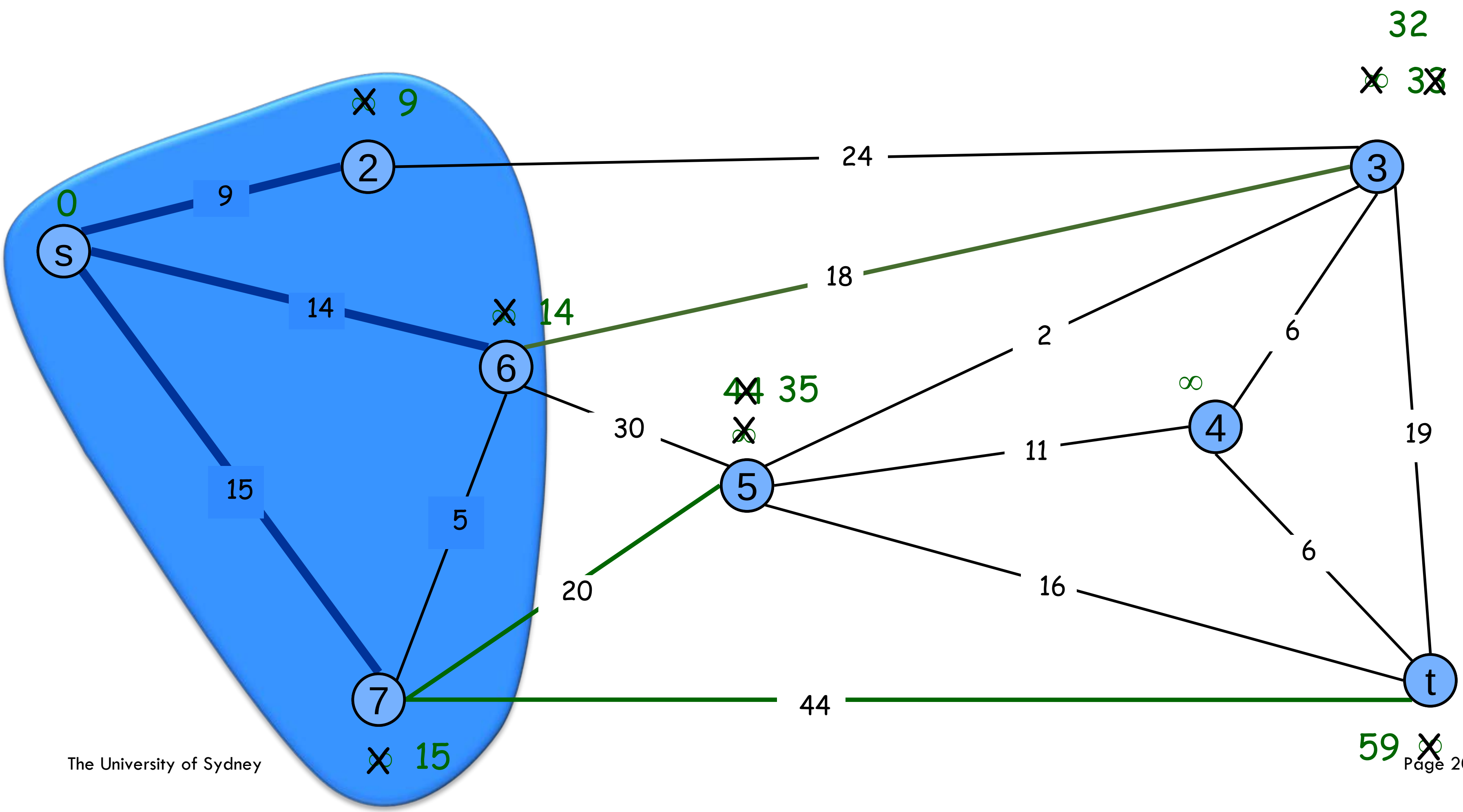
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



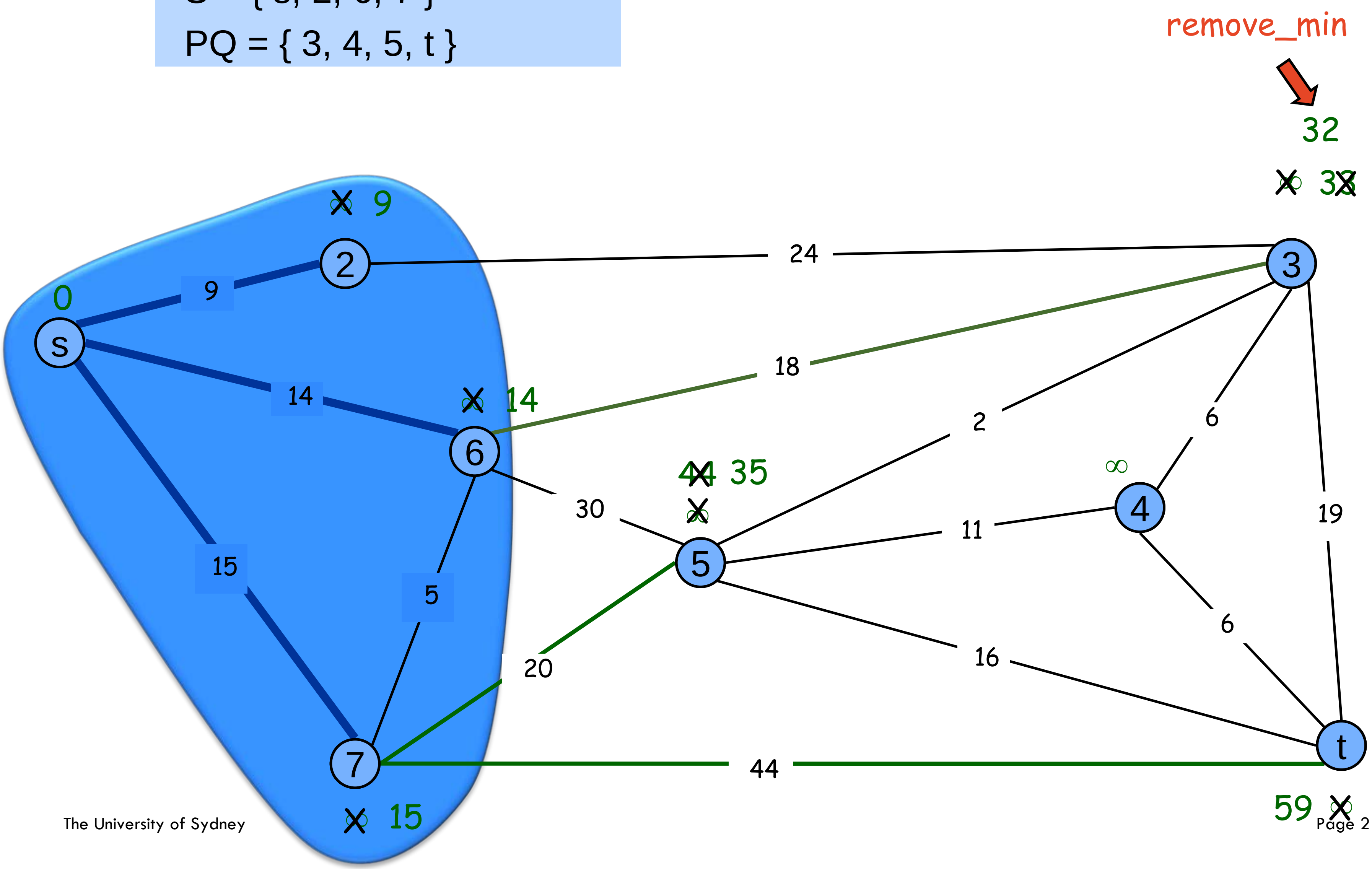
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



Dijkstra's Shortest Path Algorithm

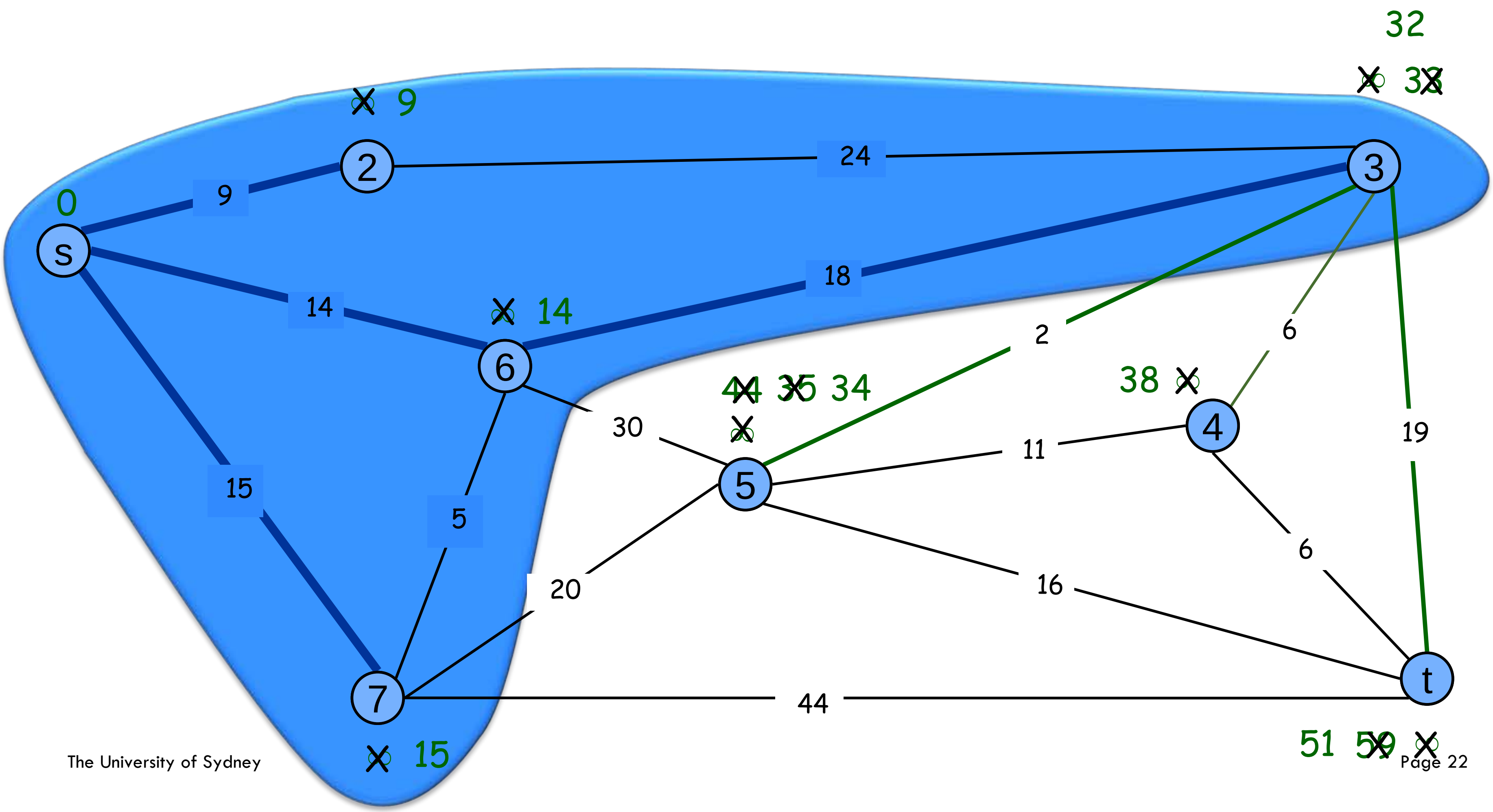
S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }

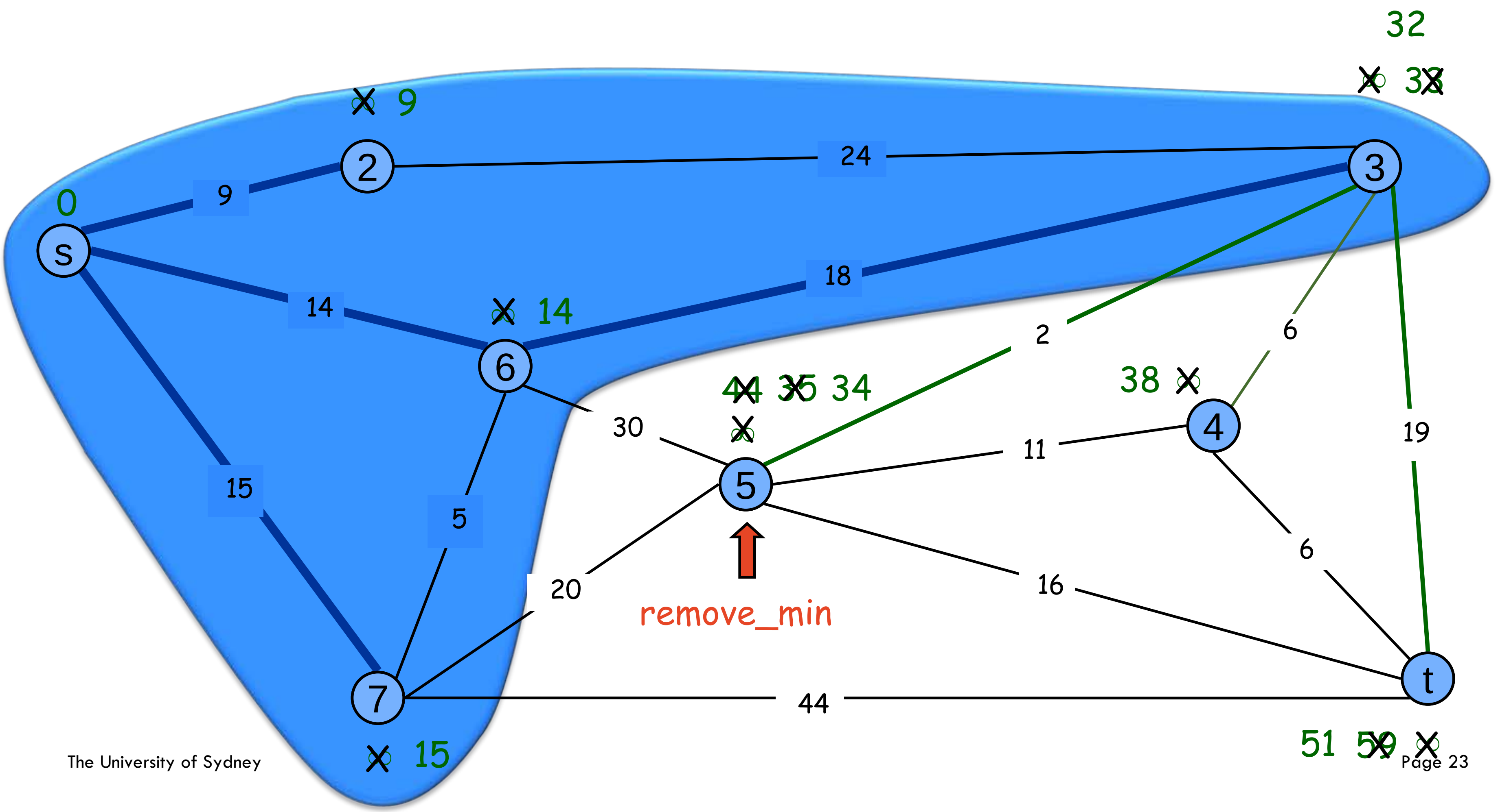
PQ = { 4, 5, t }



Dijkstra's Shortest Path Algorithm

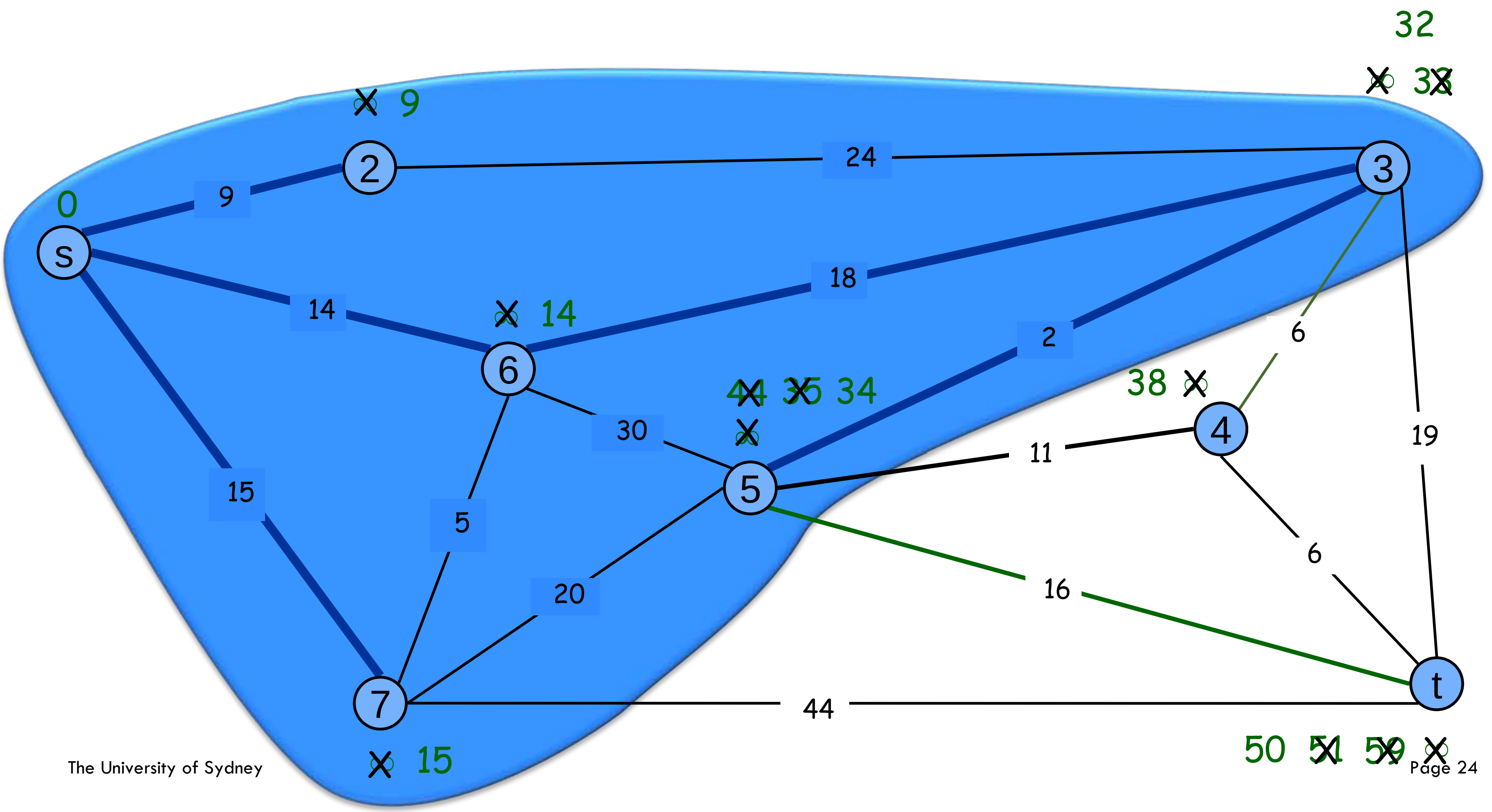
S = { s, 2, 3, 6, 7 }

PQ = { 4, 5, t }



Dijkstra's Shortest Path Algorithm

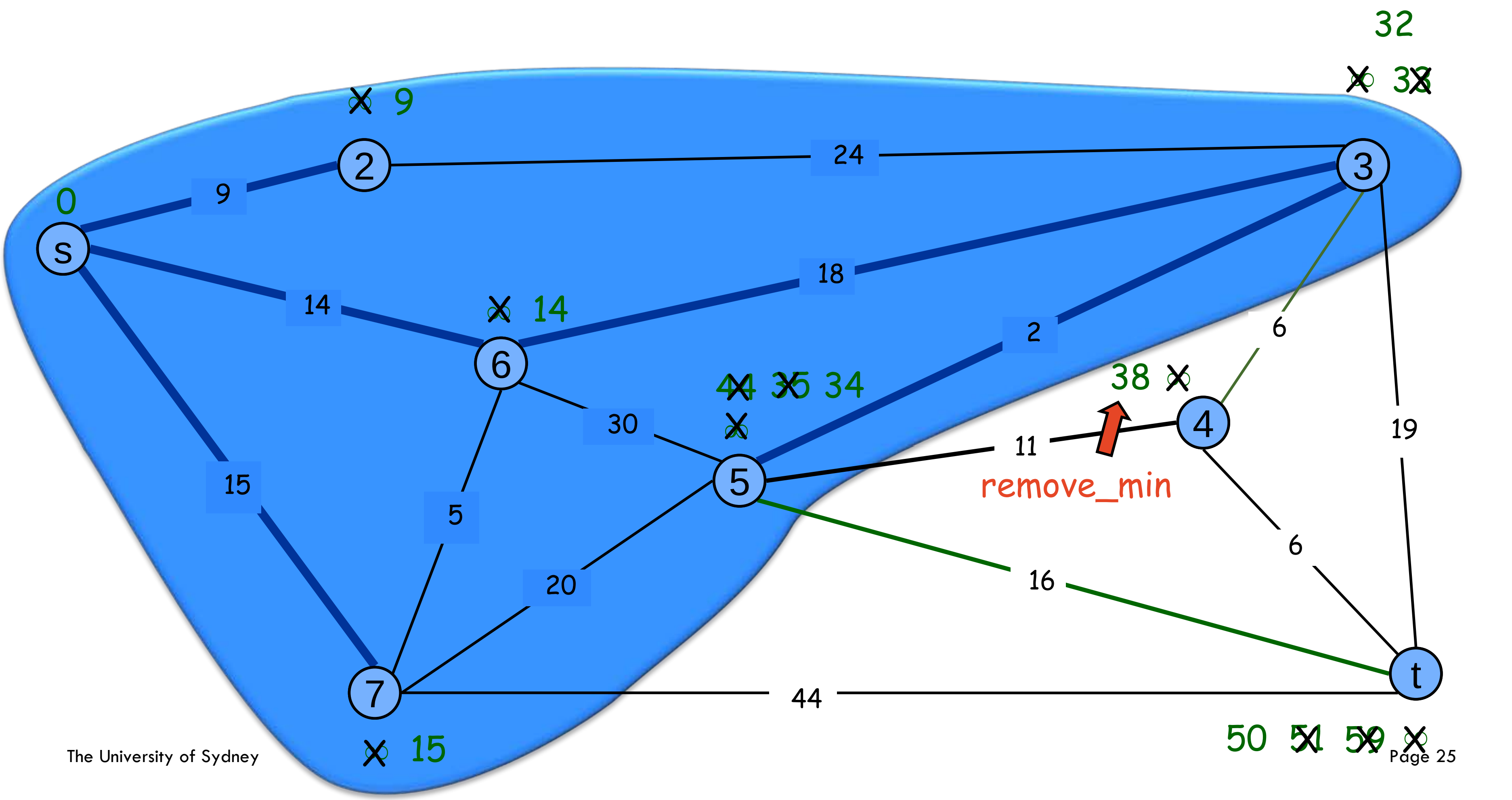
S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }

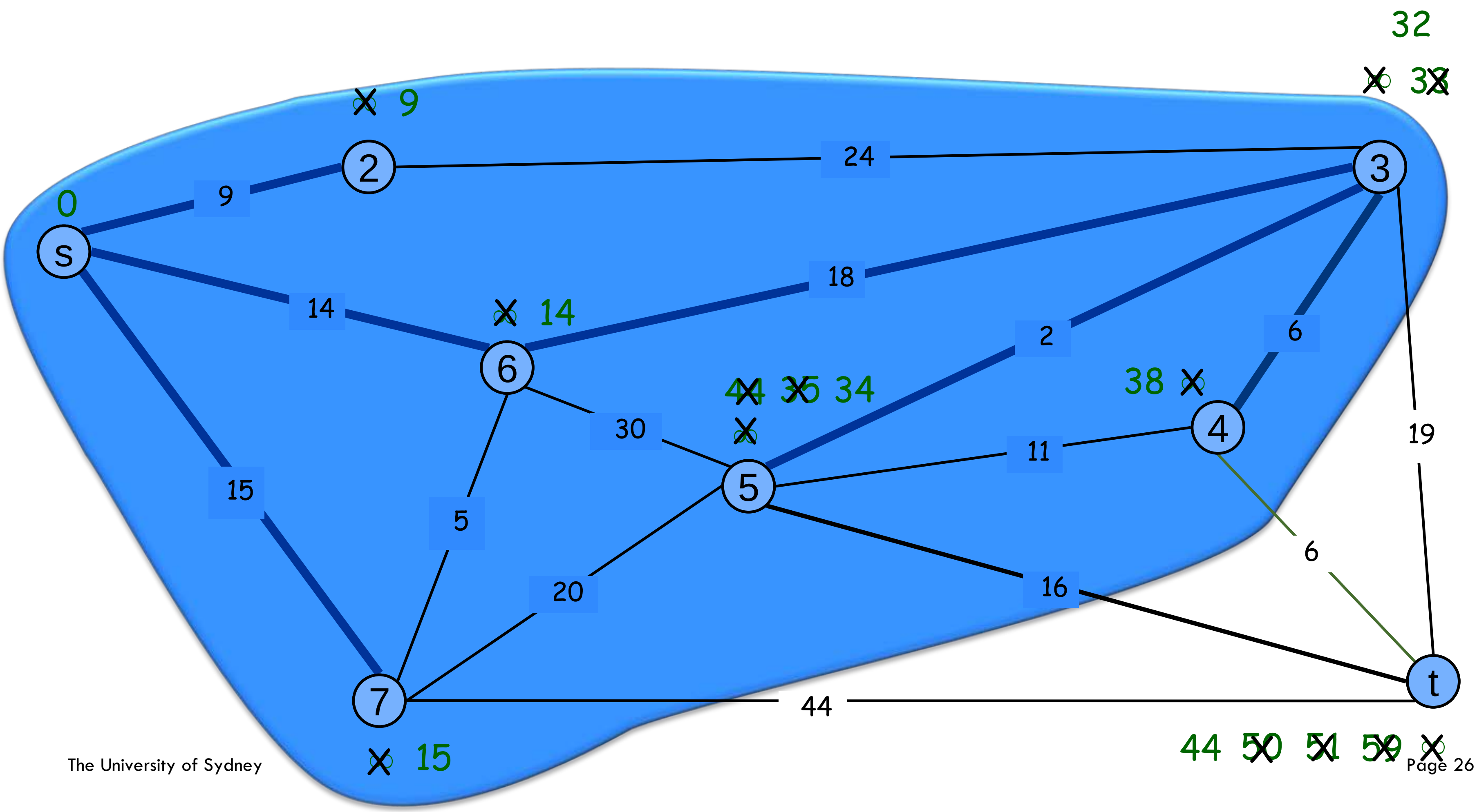
PQ = { 4, t }



Dijkstra's Shortest Path Algorithm

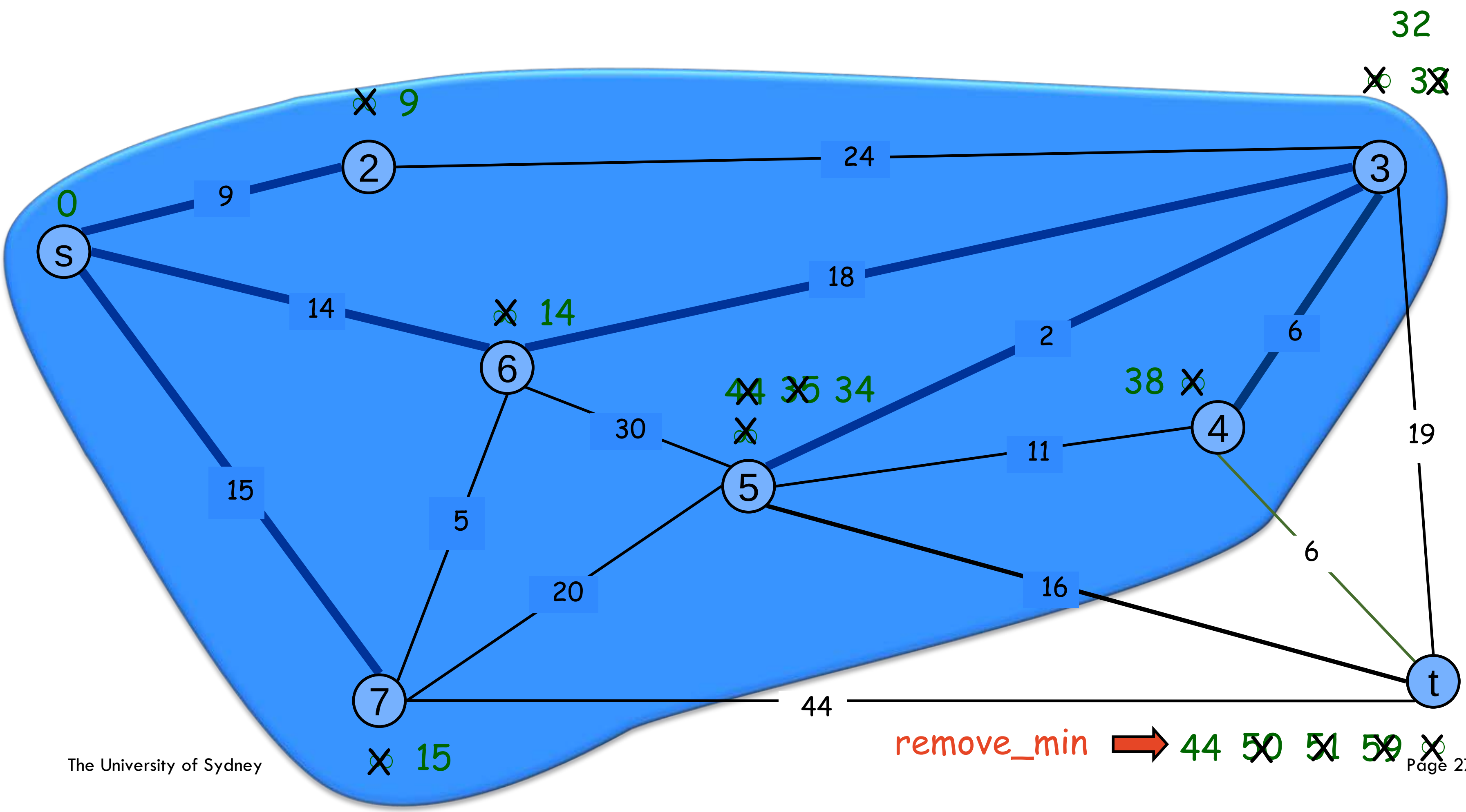
S = { s, 2, 3, 4, 5, 6, 7 }

PQ = { t }



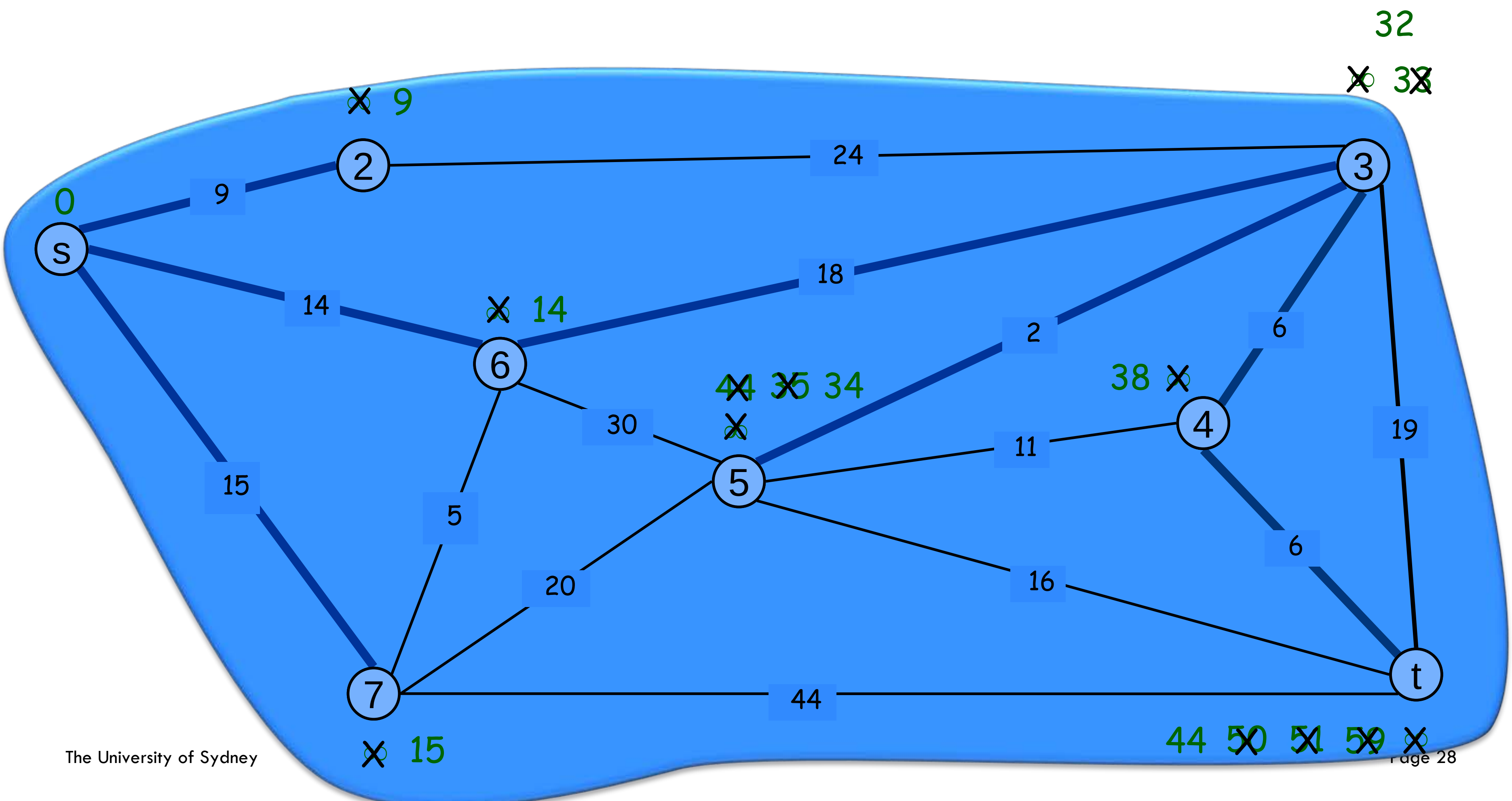
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



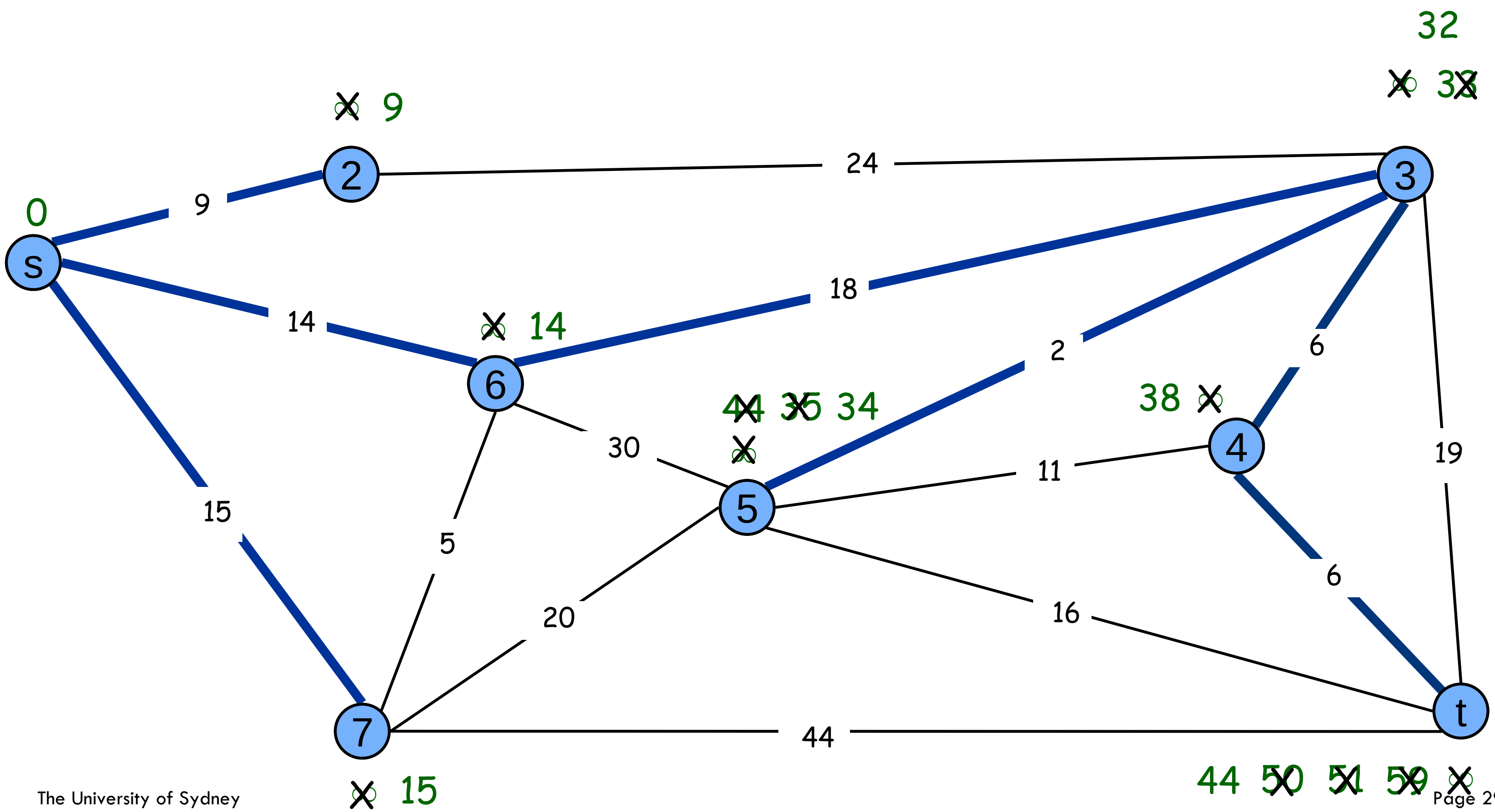
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Dijkstra complexity analysis <u>except PQ ops</u>

```
def Dijkstra(G, w, s):

  # initialize algorithm
  for v in V do
    D[v] ← ∞
    parent[v] ← Ø
  D[s] ← 0
  Q ← new priority queue for { (v, D[v]) : v in V }

  # iteratively add vertices to S
  while Q is not empty do
    u ← Q.remove_min()
    for z in G.neighbors(u) do
      if D[u] + w[u, z] < D[z] then
        D[z] ← D[u] + w[u,z]
        Q.update_priority(z, D[z])
        parent[z] ← u
  return D, parent
```

O(n)

O(deg(u)) for each u in V
plus update_priority work

Dijkstra's Algorithm complexity analysis

Assuming the graph is connected (so $m \geq n-1$), the algorithm spends $O(m)$ time on everything except PQ operations

Priority queue operation counts:

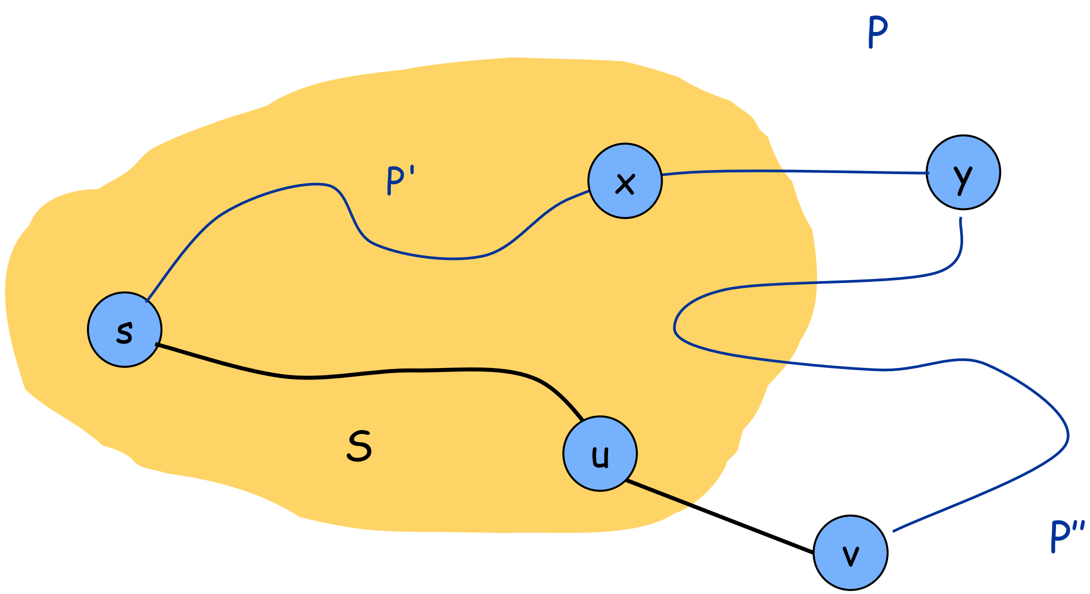
- insert: n
- decrease_key: m
- remove_min: n

Fact: Using a heap for PQ, Dijkstra runs in $O(m \log n)$ time

Fibonacci heap is a PQ that can carry out decrease key in $O(1)$ amortized time. Using that instead we get $O(m + n \log n)$ time.

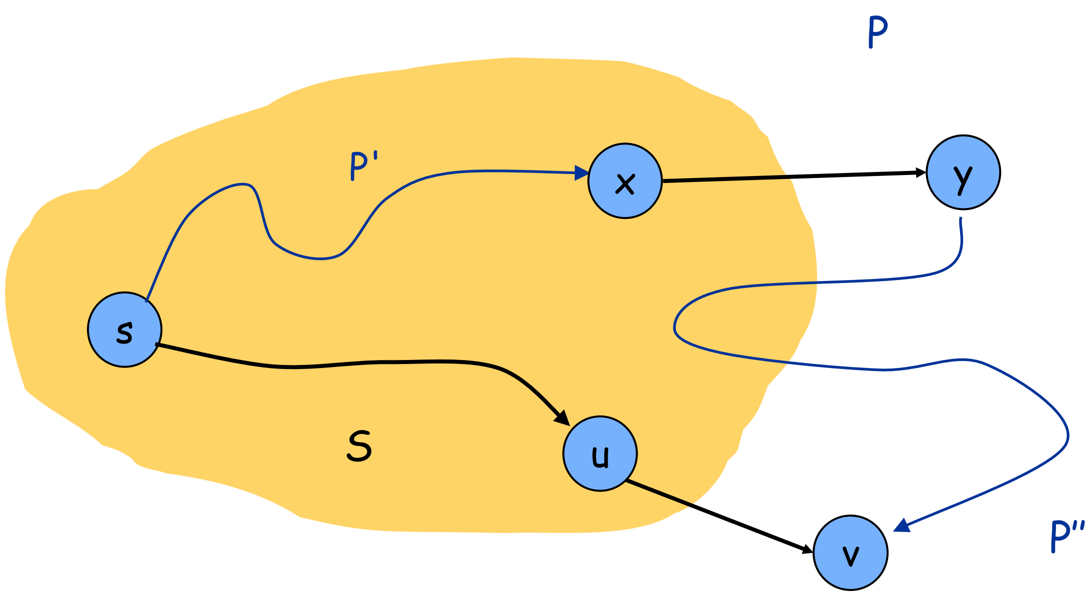
Warning: Dijkstra may not work for negative-weight edges

In the proof of correctness, even if D[v] is the smallest label, it may be that $dist_w(s, v) < D[v]$ if $w(P'') < 0$



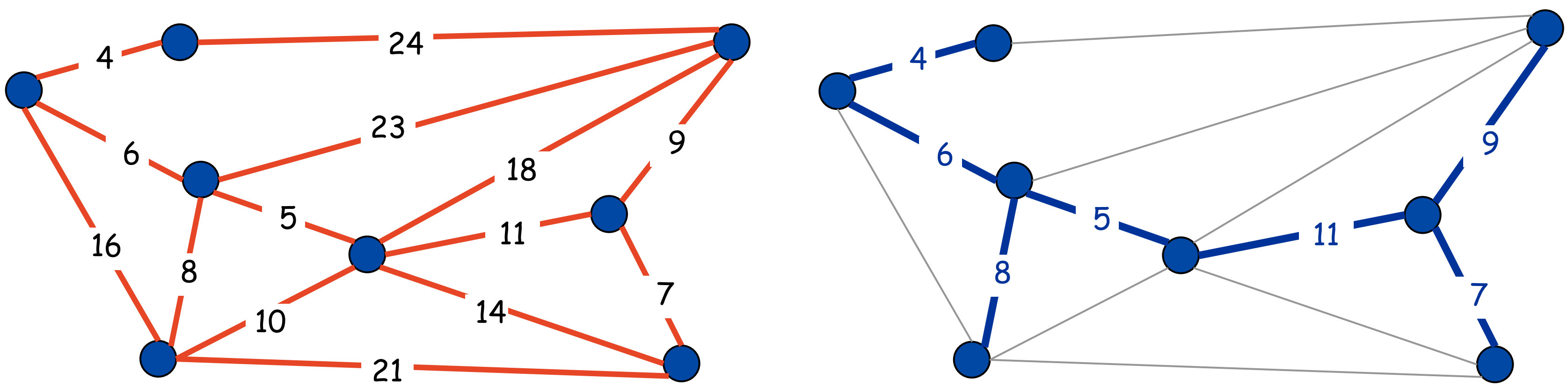
Dijkstra works for directed graph

In the proof of correctness, we need to use directed paths instead



Minimum Spanning Tree

Given a connected graph $G = (V, E)$ with real-valued edge weights c_e, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized.



$G = (V, E)$ $\qquad$ T with $\sum_{e \in T} c_e = 50$

Applications

MST is fundamental problem with diverse applications.

Network design: Telephone, electrical, hydraulic, TV cable, computer, road

Approximation algorithms for NP-hard problems: traveling salesperson problem, Steiner tree

Indirect applications.

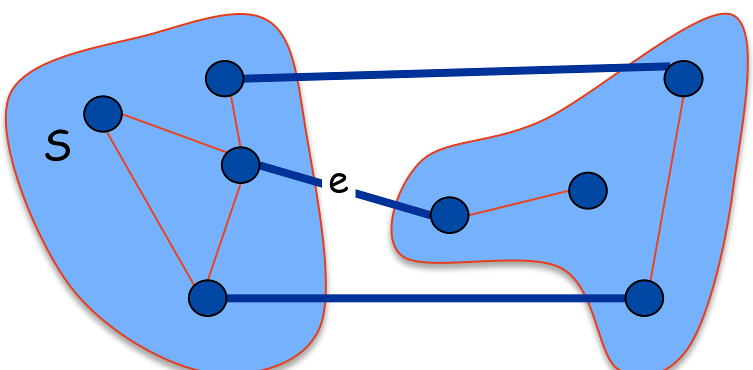
- max bottleneck paths
- LDPC codes for error correction
- image registration with Renyi entropy
- learning salient features for real-time face verification
- reducing data storage in sequencing amino acids in a protein
- …

MST properties

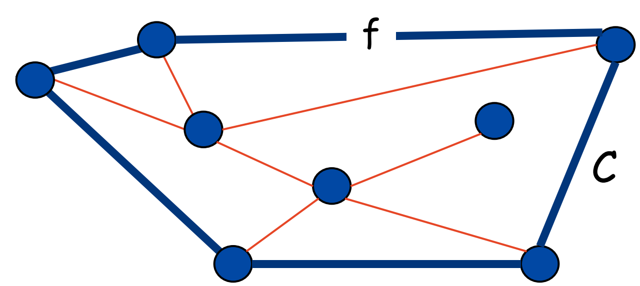
Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



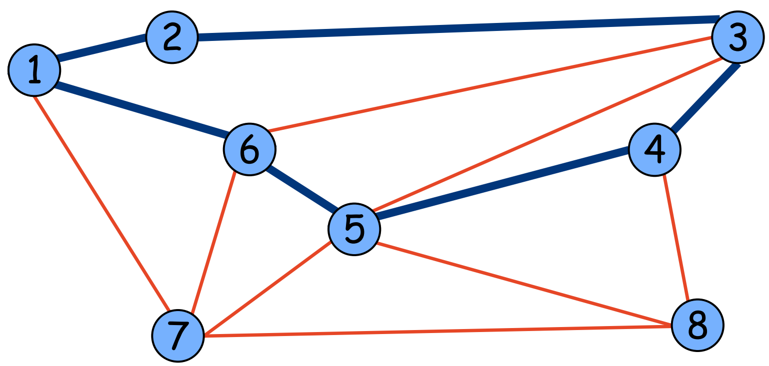
e is in the MST



f is not in the MST

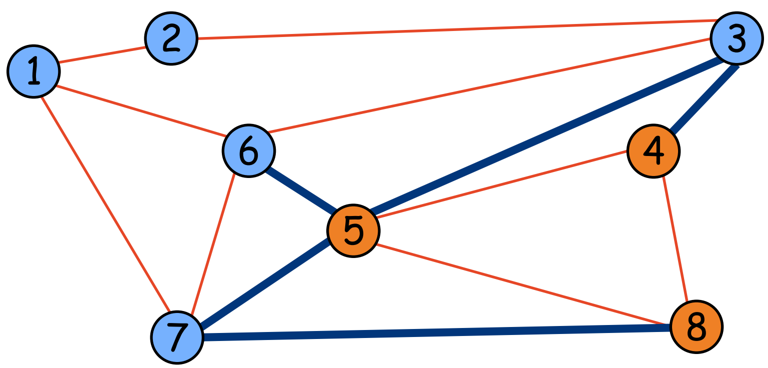
Cycles and Cuts

Cycle. Set of edges of the form a-b, b-c, c-d, …, y-z, z-a.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

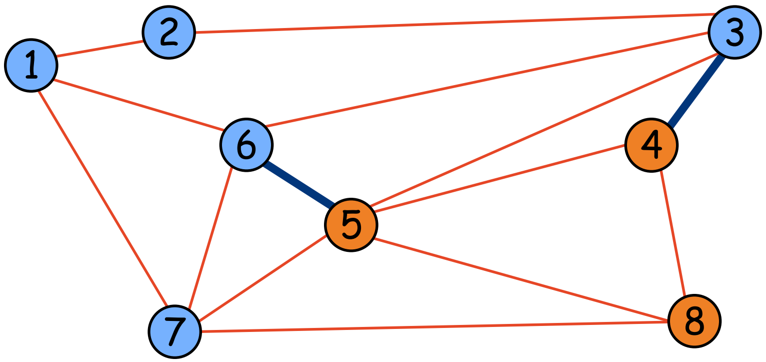
Cutset. A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

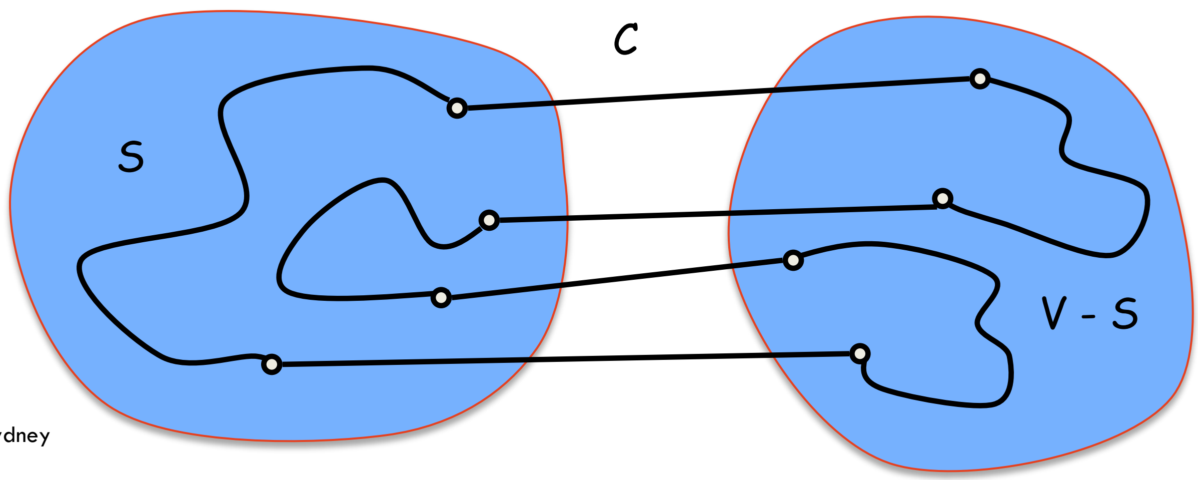
Cycle-Cut Intersection

Claim. A cycle and a cutset intersect in an even number of edges.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

Proof:



Prim's Algorithm

```
def prim(G, c):
  u ← arbitrary vertex in V
  S ← { u }
  T ← Ø
  while |S| < |V| do
    (u, v) ← min cost edge s.t. u in S and v not in S
    add (u, v) to T
    add v to S
  return T
```

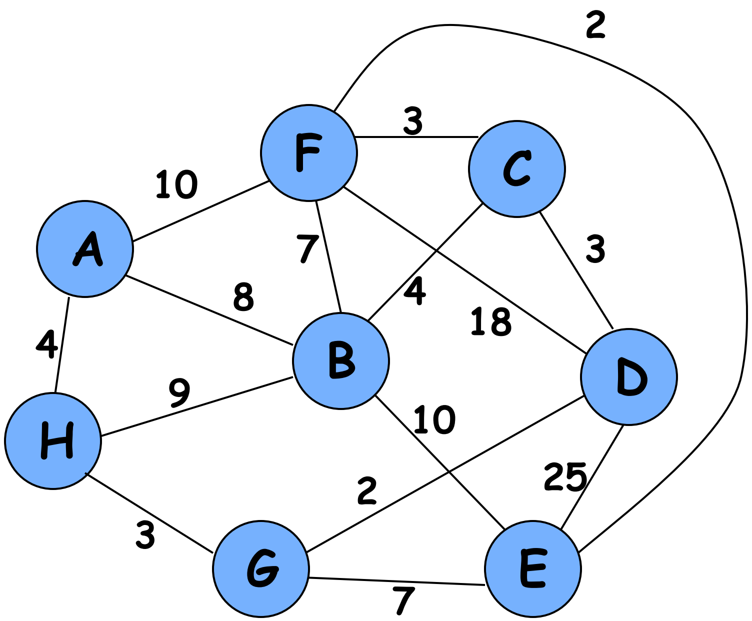
Implementation: Prim's Algorithm

```
def prim(G, c) {

 for v in V do
  d[v] ← ∞
  parent[v] ← ∅
 u ← arbitrary vertex in V
 d[u] ← 0
 Q ← new PQ with items { (v, d[v]) for v in V }
 S ← ∅

 while Q is not empty do
  u ← delete min element from Q
  add u to S
  for (u, v) incident to u do
   if v ∉ S and c_{u,v} < d[v] then
     parent[v] ← u
     decrease priority d[v] to c_{u,v}
 return parent
```

Main idea: for every v in V \ S we keep

- d[v] = distance to closest neighbor in S
- parent[v] = closest neighbor in S

Walk-Through



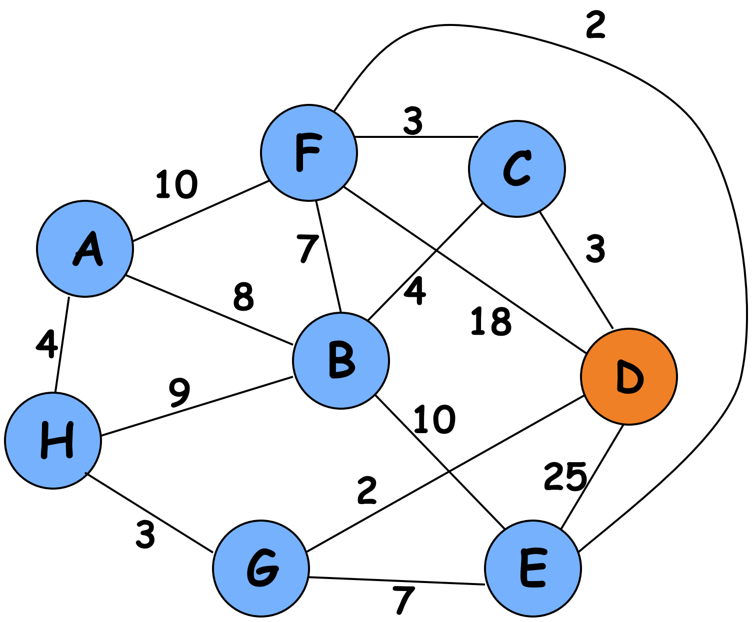
Initialize array

	S	d_v	p_v
A	F	∞	–
B	F	∞	–
C	F	∞	–
D	F	∞	–
E	F	∞	–
F	F	∞	–
G	F	∞	–
H	F	∞	–

Set S

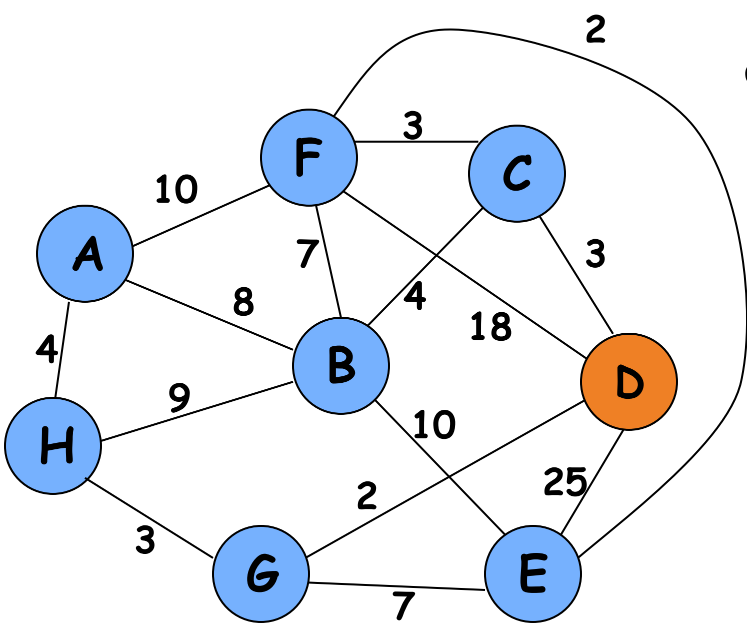
Min distance to S

Closest vertex in S



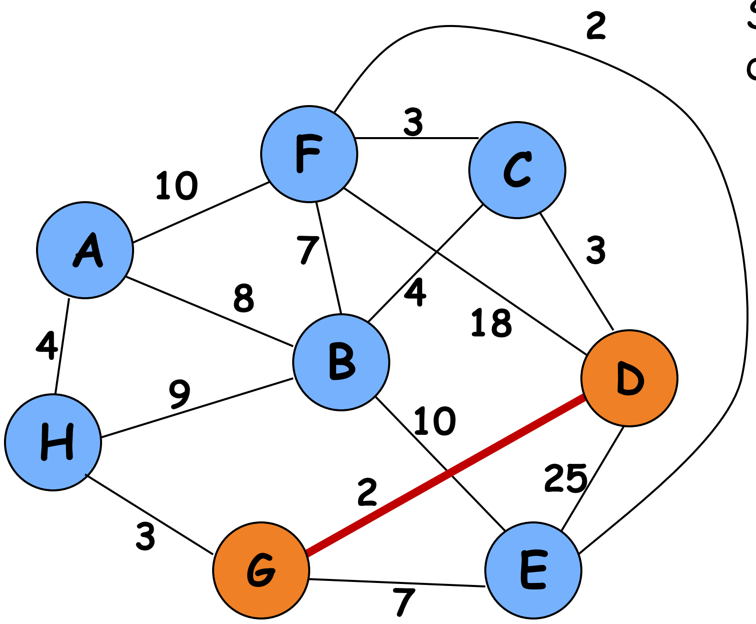
Start with any node, say D

	S	d_v	p_v
A			
B			
C			
D	T	0	–
E			
F			
G			
H			



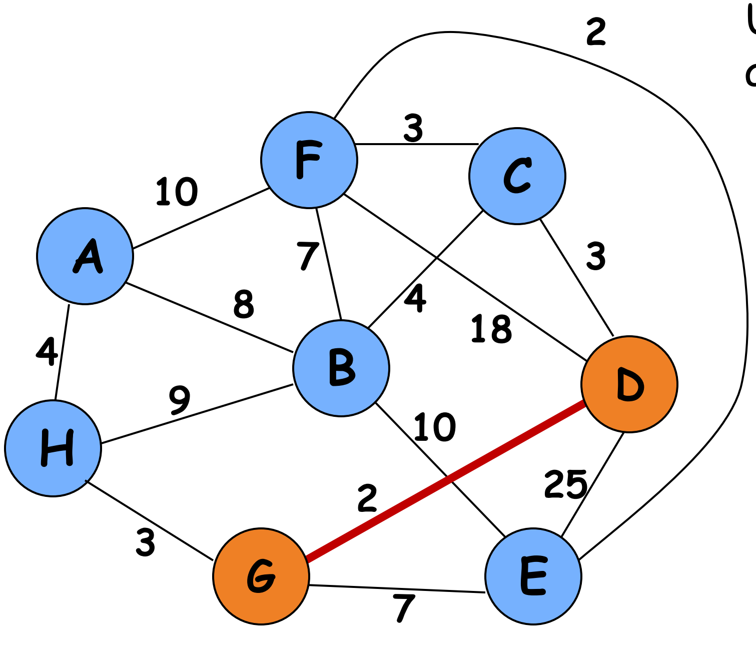
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G		2	D
H			



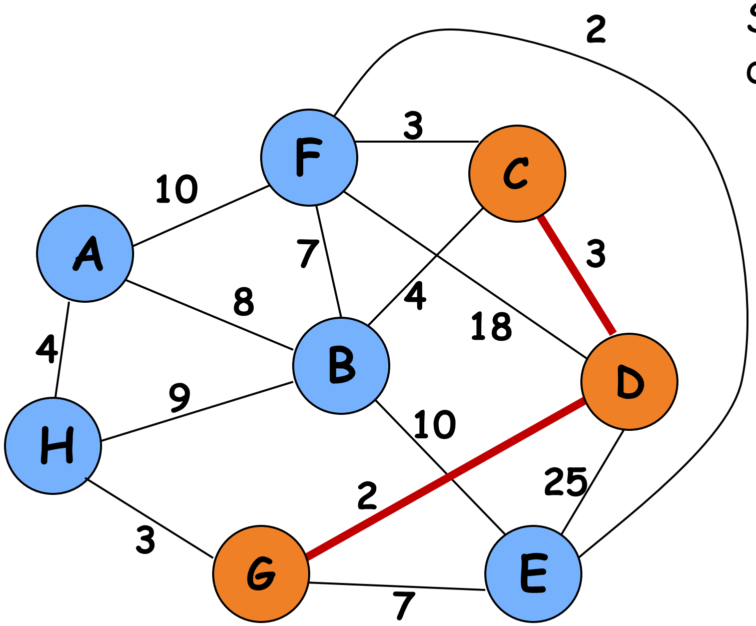
Select node with minimum distance

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G	T	2	D
H			



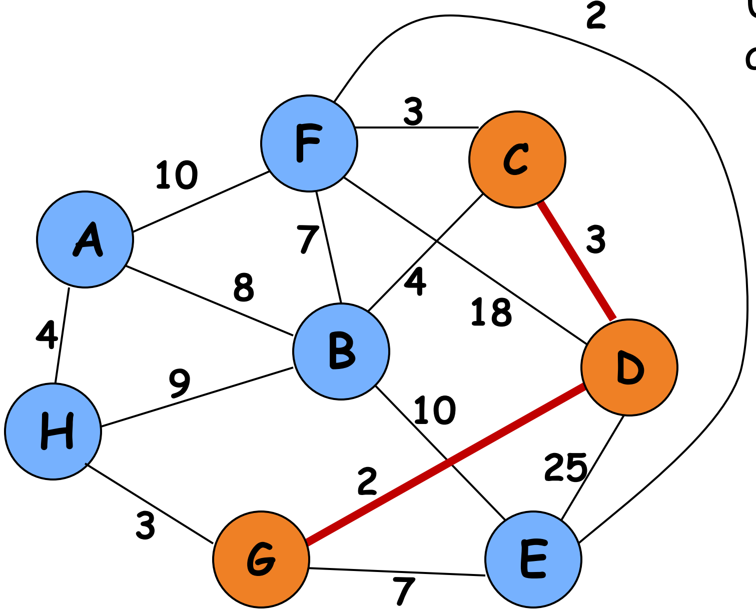
Update distances of
adjacent, unselected nodes

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



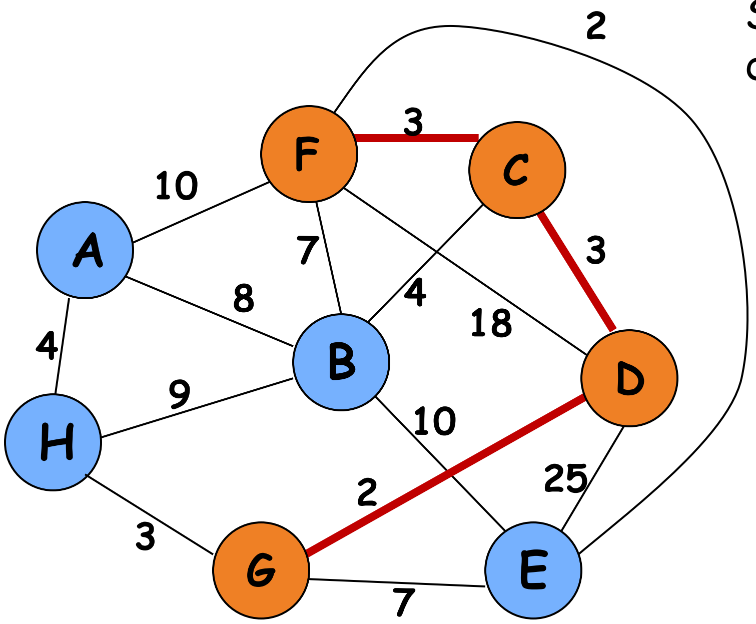
Select node with minimum distance

	S	d_v	p_v
A			
B			
C	T	3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



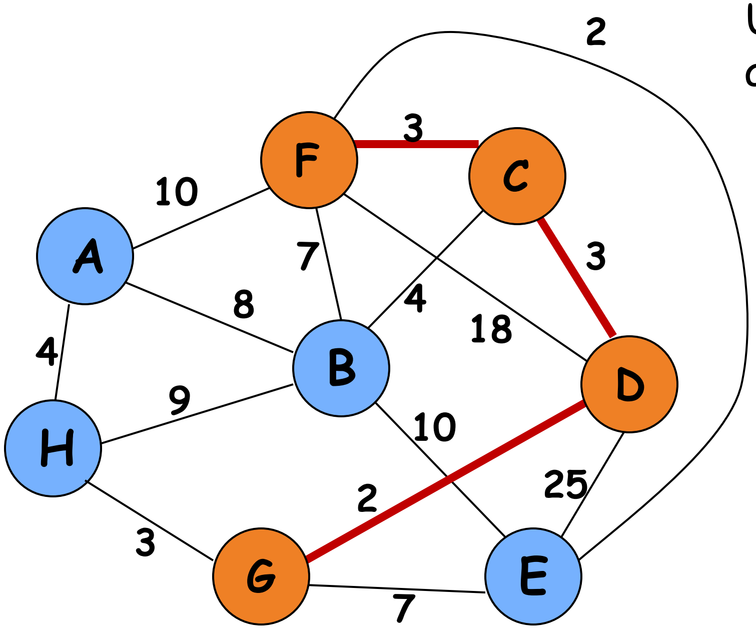
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F		3	C
G	T	2	D
H		3	G



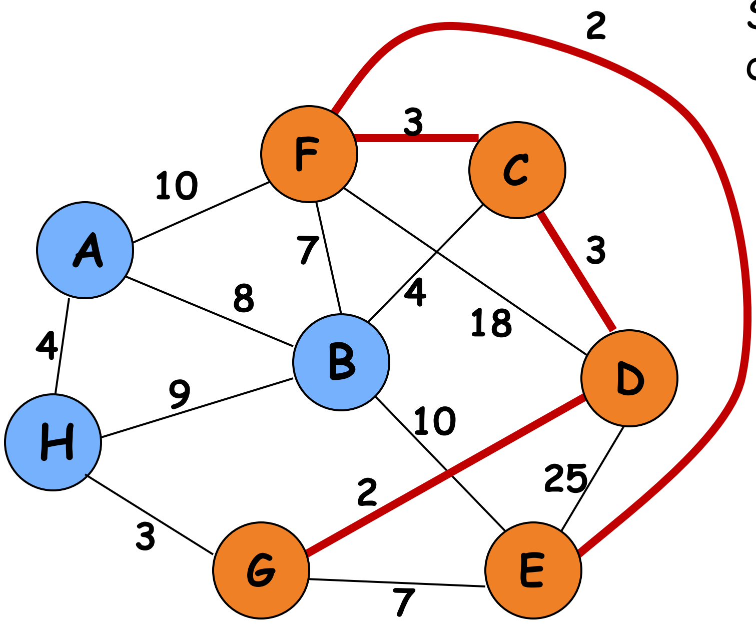
Select node with minimum distance

	S	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F	T	3	C
G	T	2	D
H		3	G



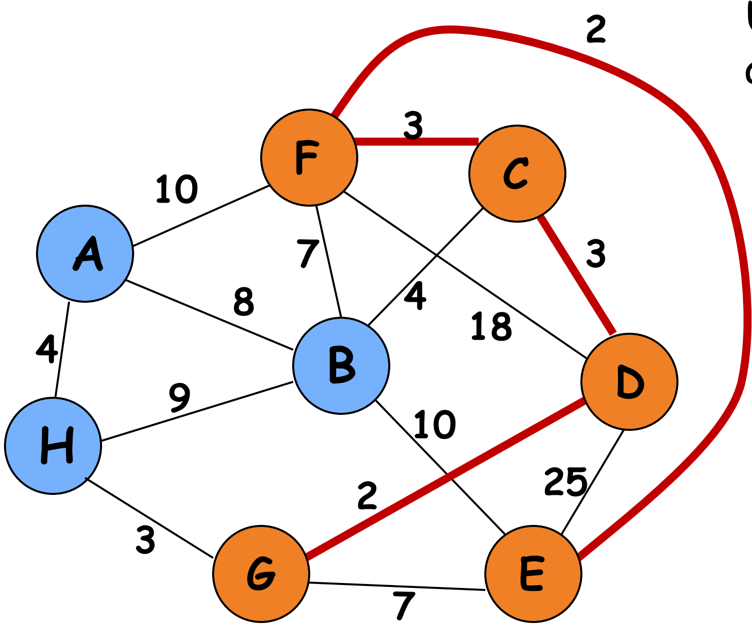
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E		2	F
F	T	3	C
G	T	2	D
H		3	G



Select node with minimum distance

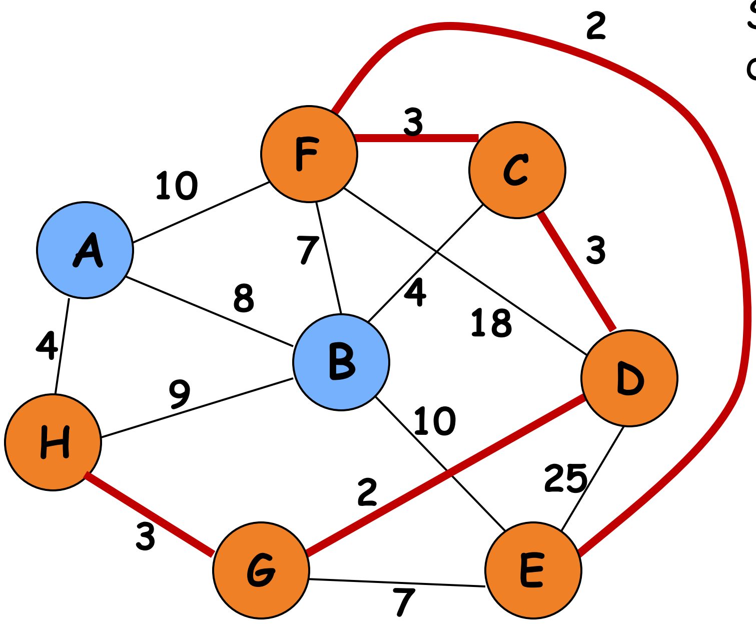
	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G



Update distances of adjacent, unselected nodes

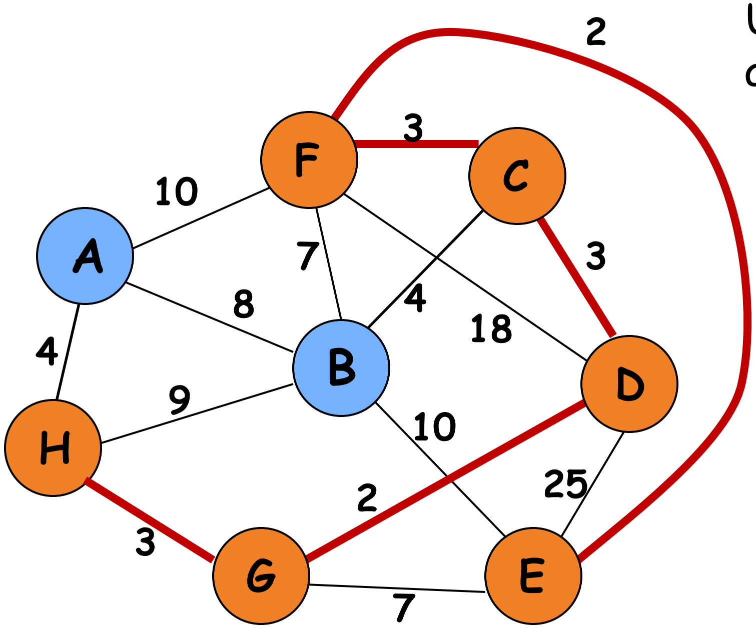
	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

Table entries unchanged



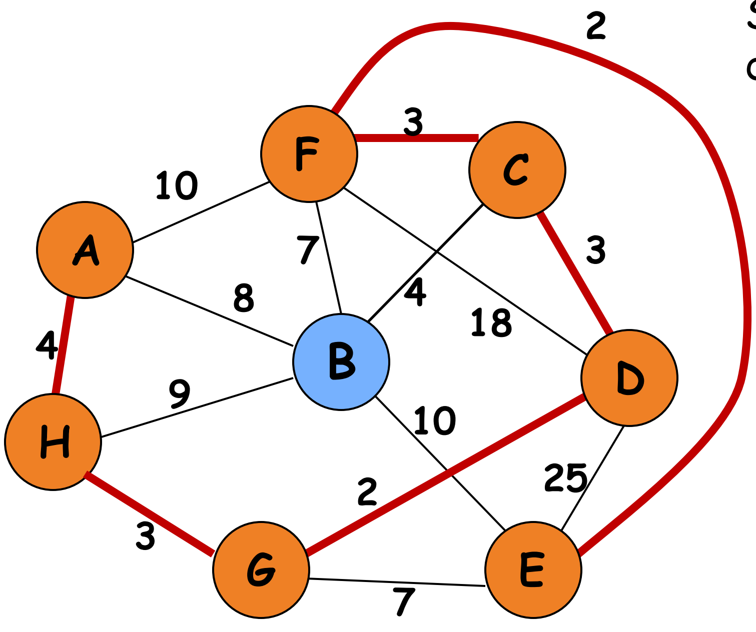
Select node with minimum distance

	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



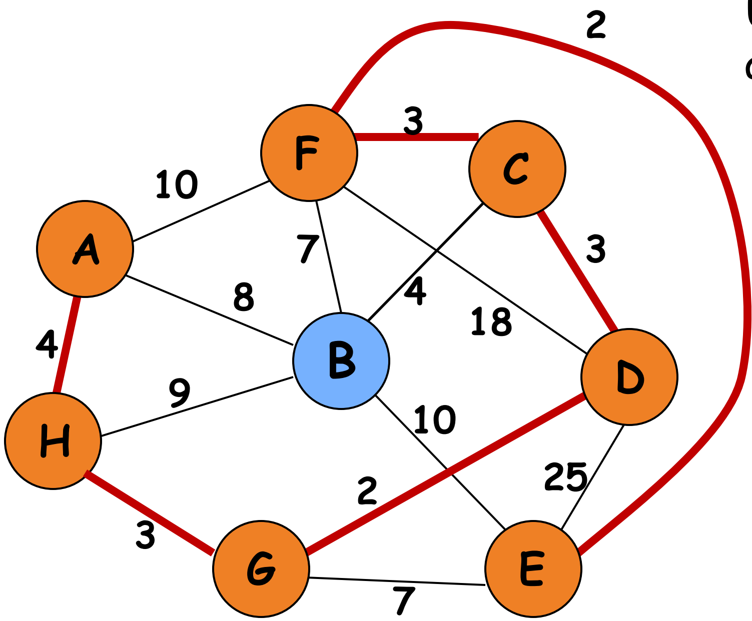
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A		4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Select node with minimum distance

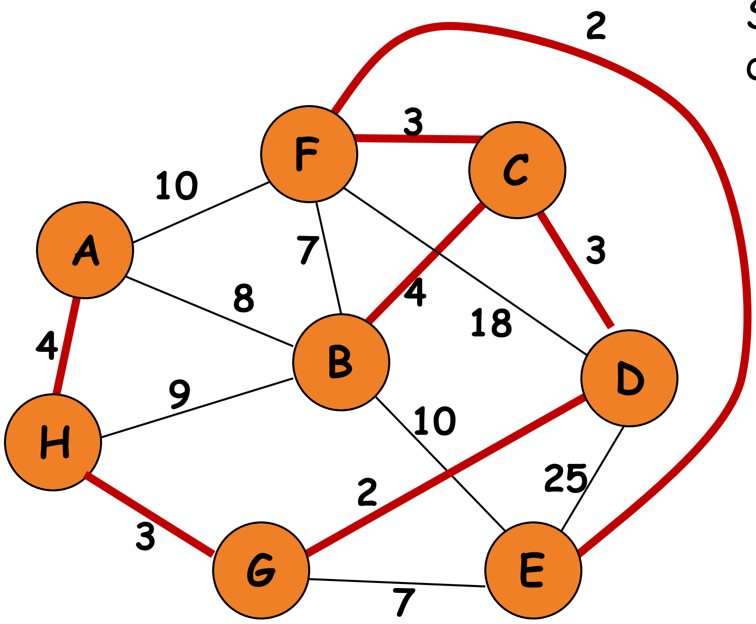
	S	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Update distances of adjacent, unselected nodes

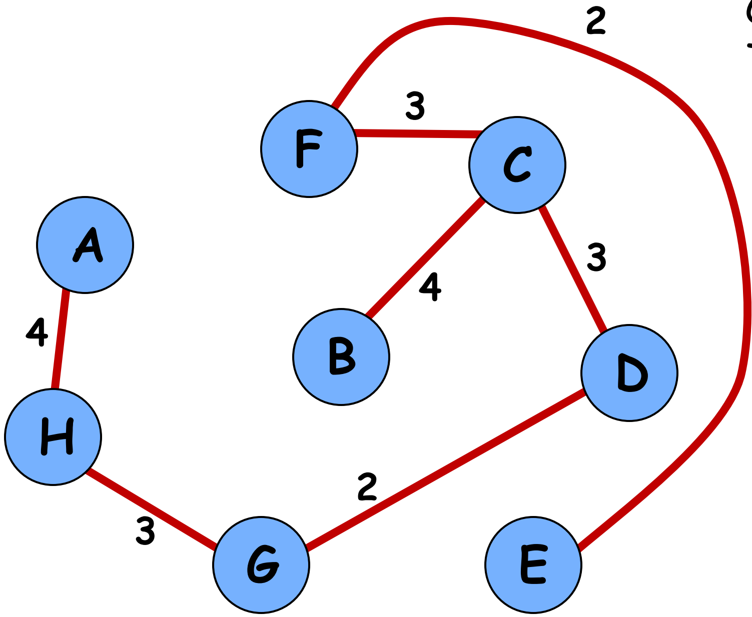
	S	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Table entries unchanged



Select node with minimum distance

	S	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Cost of Minimum Spanning Tree = $\Sigma\, d_v$ = 21

	S	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Done!

Prim's Algorithm complexity

```
def prim(G, c) {

 for v in V do
  d[v] ← ∞
  parent[v] ← ∅
 u ← arbitrary vertex in V
 d[u] ← 0
 Q ← new PQ with items { (v, d[v]) for v in V }
 S ← ∅

 while Q is not empty do
  u ← delete min element from Q
  S ← S ∪ { u }
  for (u, v) incident to u do
   if v ∉ S and c_{u,v} < d[v] then
    parent[v] ← u
    decrease priority d[v] to c_{u,v}
 return parent
```

Similar analysis to Dijkstra's algorithm:

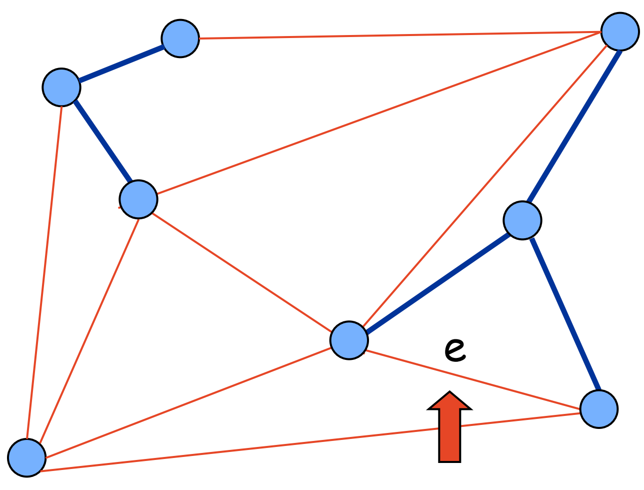
– $O(m \log n)$ using a heap
– $O(m + n \log n)$ using Fibonacci heap

Kruskal's Algorithm

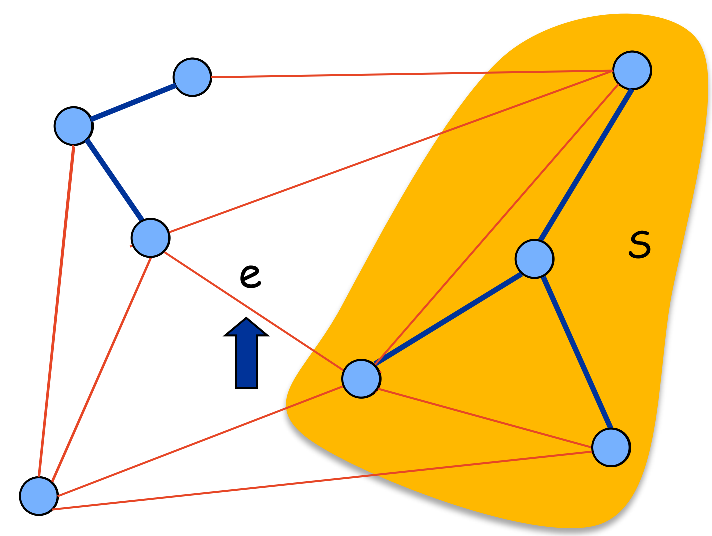
Consider edges in ascending order of weight.

Case 1: If adding e to T creates a cycle, discard e according to cycle property.

Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.

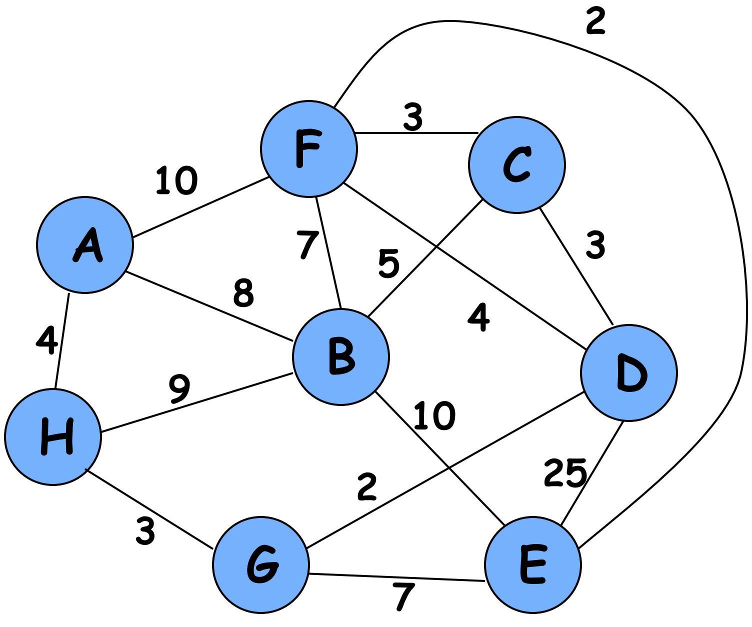


Case 1

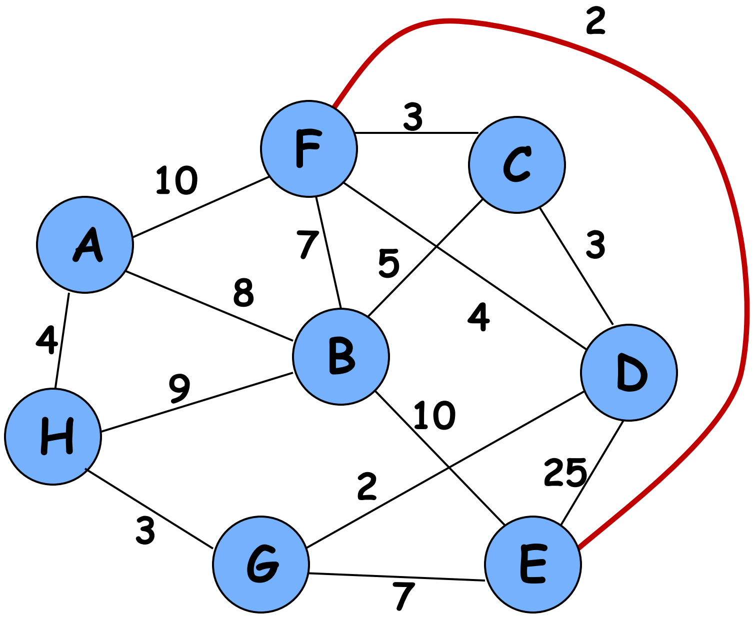


Case 2

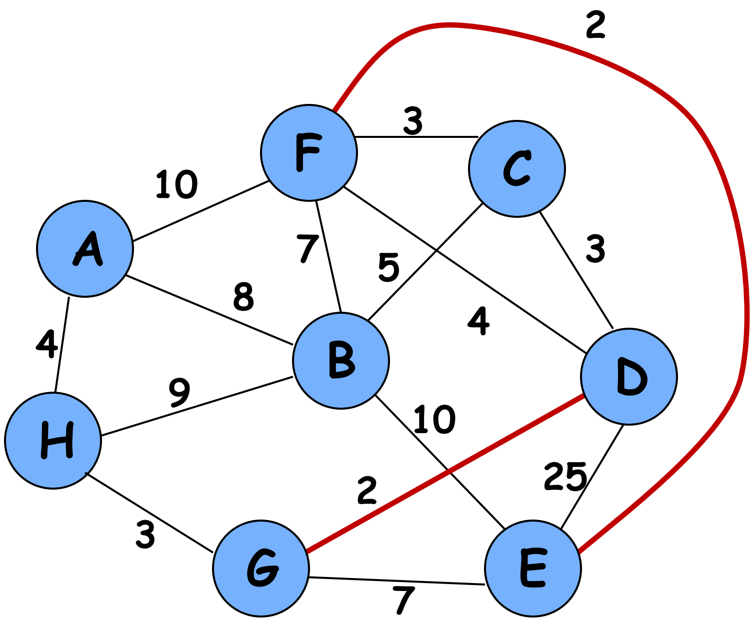
Walk-Through



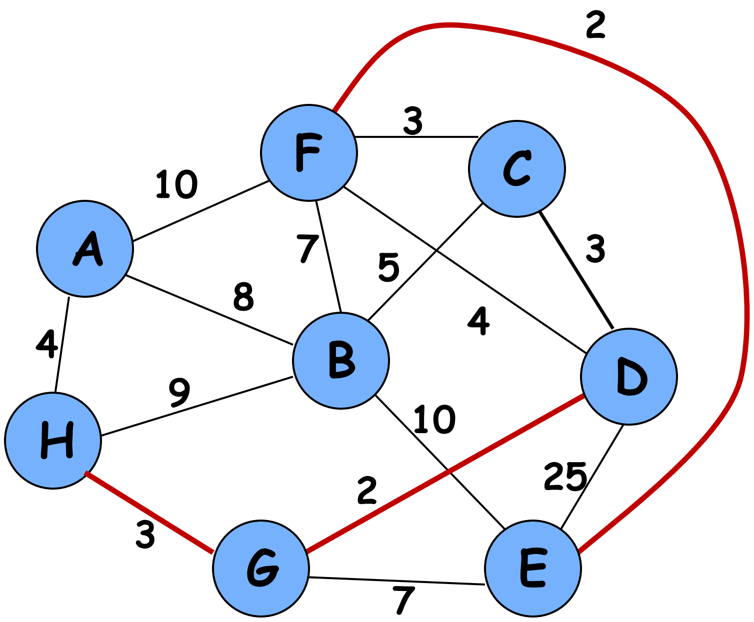
Walk-Through



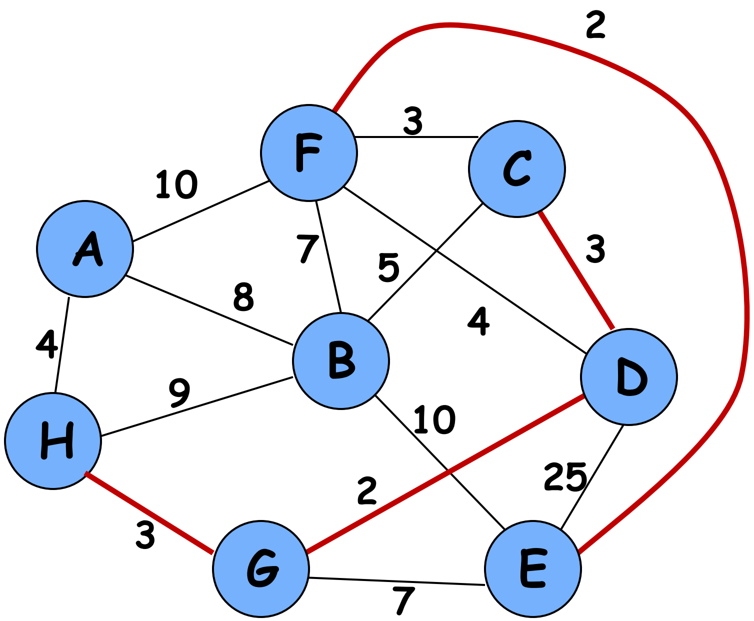
Walk-Through



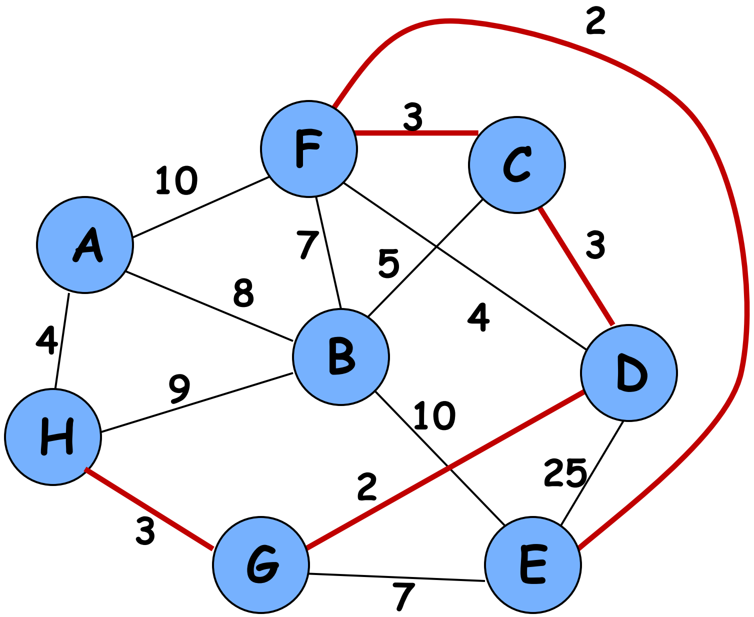
Walk-Through



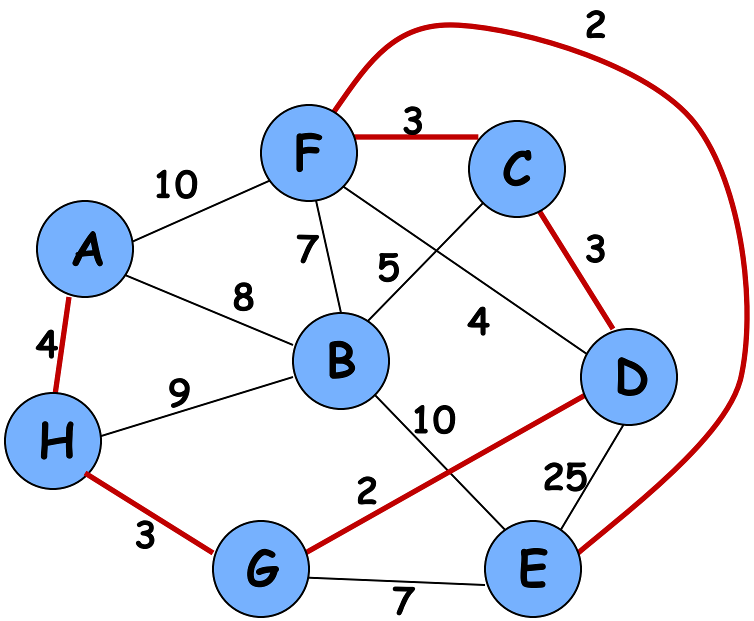
Walk-Through



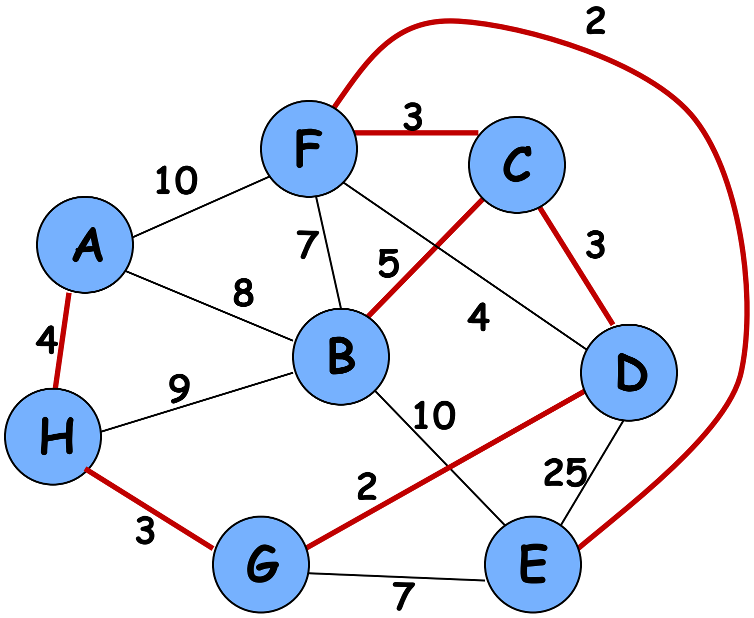
Walk-Through



Walk-Through



Walk-Through



Kruskal's Algorithm: Time complexity

Sorting edges takes O(m log m) time

We need to be able to test if adding a new edge creates a cycle, in which case we skip the edge

One option is to run DFS in each iteration to see if the number of connected components stays the same. This leads to O(m n) time for the main loop

Can we do better?

Yes, keep track of the connected components with a data structure

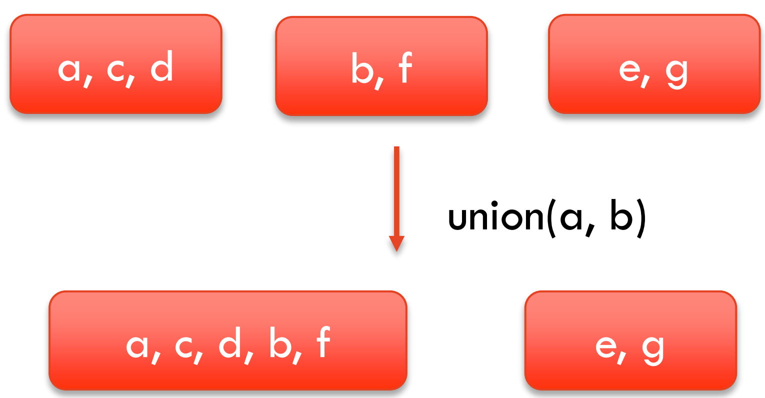
Union Find ADT

Data structure defined on a ground set of elements A

Used to keep track of an evolving partition of A

Supported operations:

- make_sets(A) : makes |A| singleton sets with elements in A
- find(a) : returns an id for the set element a belongs to
- union(a,b) : union the sets elements a and b belong to



Kurskal's algorithm implementation

```
def Kruskal(G,c):

  sort E in increasing c-value
  answer ← [ ]
  comp ← make_sets(V)
  for (u,v) in E do
    if comp.find(u) ≠ comp.find(v) then
      answer.append( (u,v) )
      comp.union(u, v)
  return answer
```

Union find operations:

- make_sets(A) : one call with $|A| = |V|$
- find(a) : 2m calls
- union(a,b) : n-1 calls

Simple union-find implementation

Sets are represented with lists. An array points to the set each element belongs to

- make_sets(A) creates and initialized the array
- find(u) is a simple lookup in the array
- union(u,v) adds elements in u's set to v's set

Time complexity:

- make_sets(A) takes O(n) time, where n = |A|
- find(u) takes O(1) time
- union(u,v) take O(n) time

Kruskal's algorithm would run in O(n^2) time after the edge weights are sorted.

Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

For example, assuming all costs are integral, if we add i/n^2 to each edge e_i then any MST under the perturbed weights is still an MST under the original weights.

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.