Breadth-first search

Breadth-first search

A breadth-first search (BFS) in a graph visits the nodes in the following order:

- First it visits some node (the *start node*)
- Then all the start node's immediate neighbours
- Then *their* neighbours
- and so on

So it visits the nodes in order of how far away they are from the start node

Implementing breadth-first search

We maintain a *queue* of nodes that we are going to visit soon

- Initially, the queue contains the start node

We also remember which nodes we've already added to the queue

Then repeat the following process:

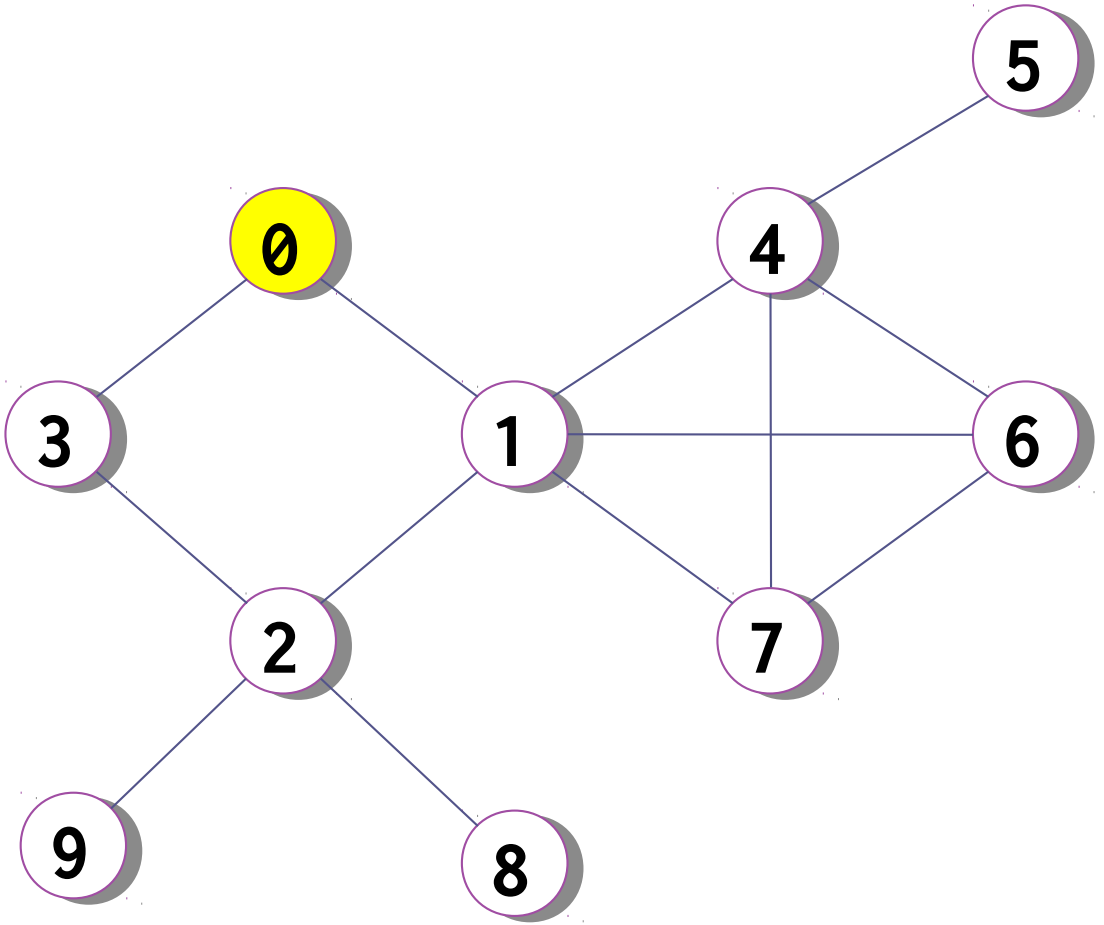
- Remove a node from the queue
- Visit it
- Find all adjacent nodes and add them to the queue, *unless* they've previously been added to the queue

Example of a breadth-first search

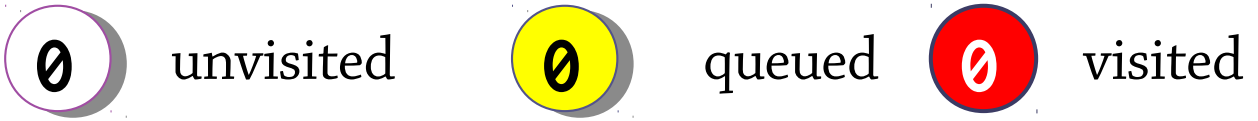
Queue:

0

Visit order:



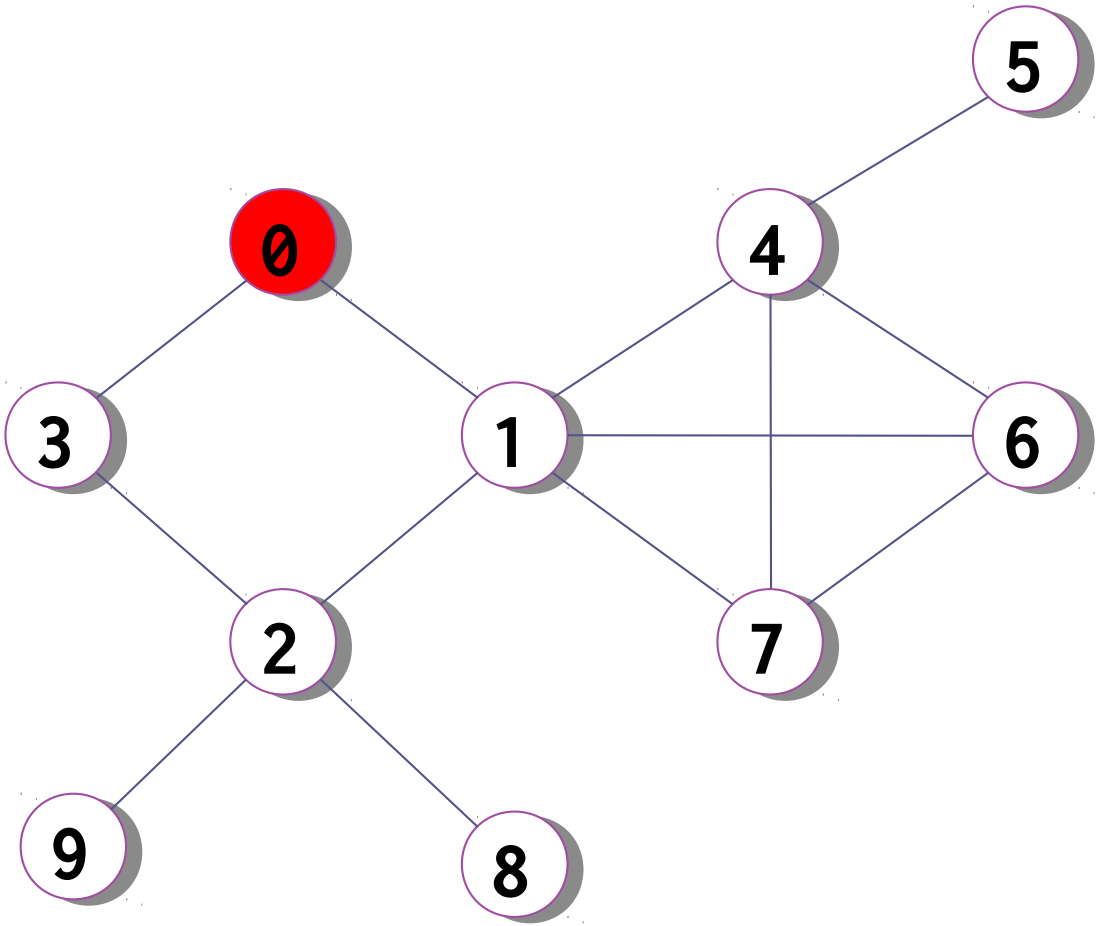
Initially,
queue contains
start node



Example of a breadth-first search

Queue:

Visit order:
0



Step 1:
remove node
from queue
and visit it

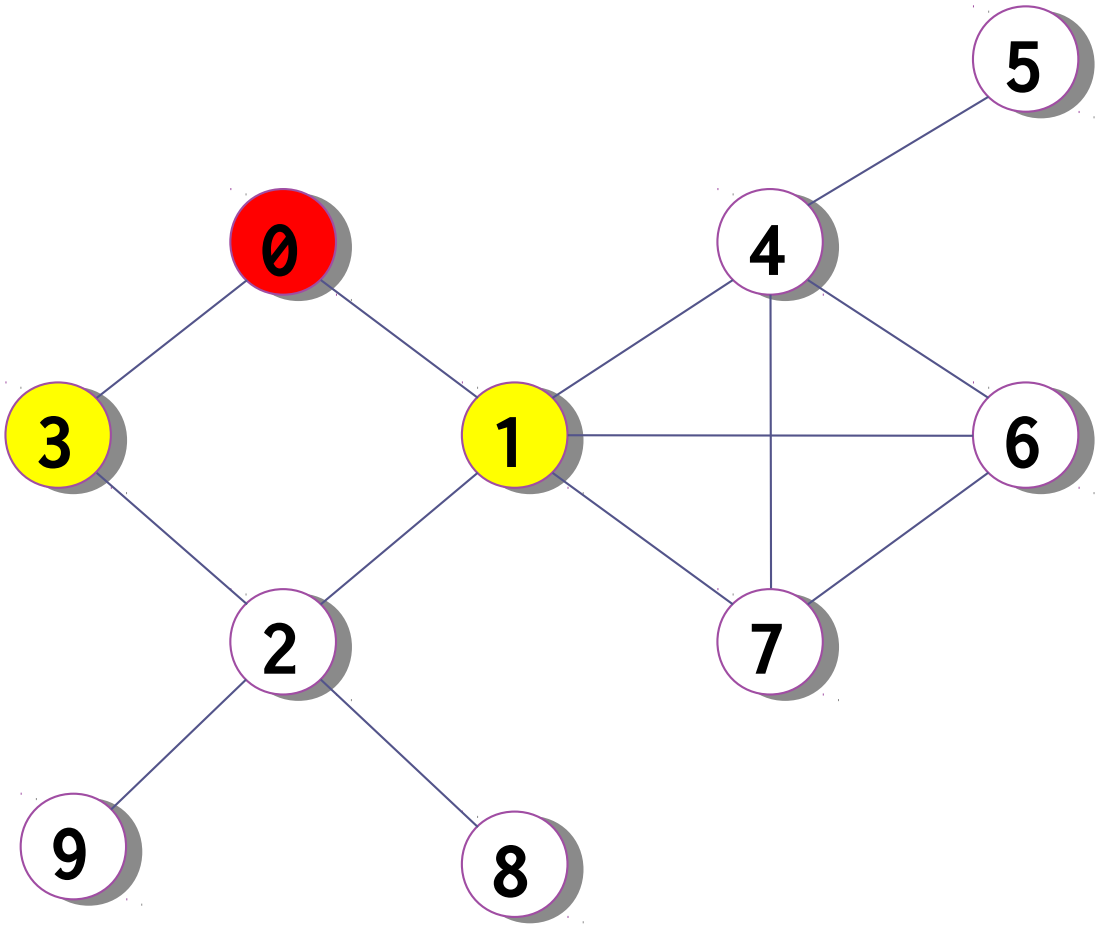
Example of a breadth-first search

Queue:

3 1

Visit order:

0



Step 2:
add adjacent nodes
to queue
(only unvisited ones)

0 unvisited 0 queued 0 visited

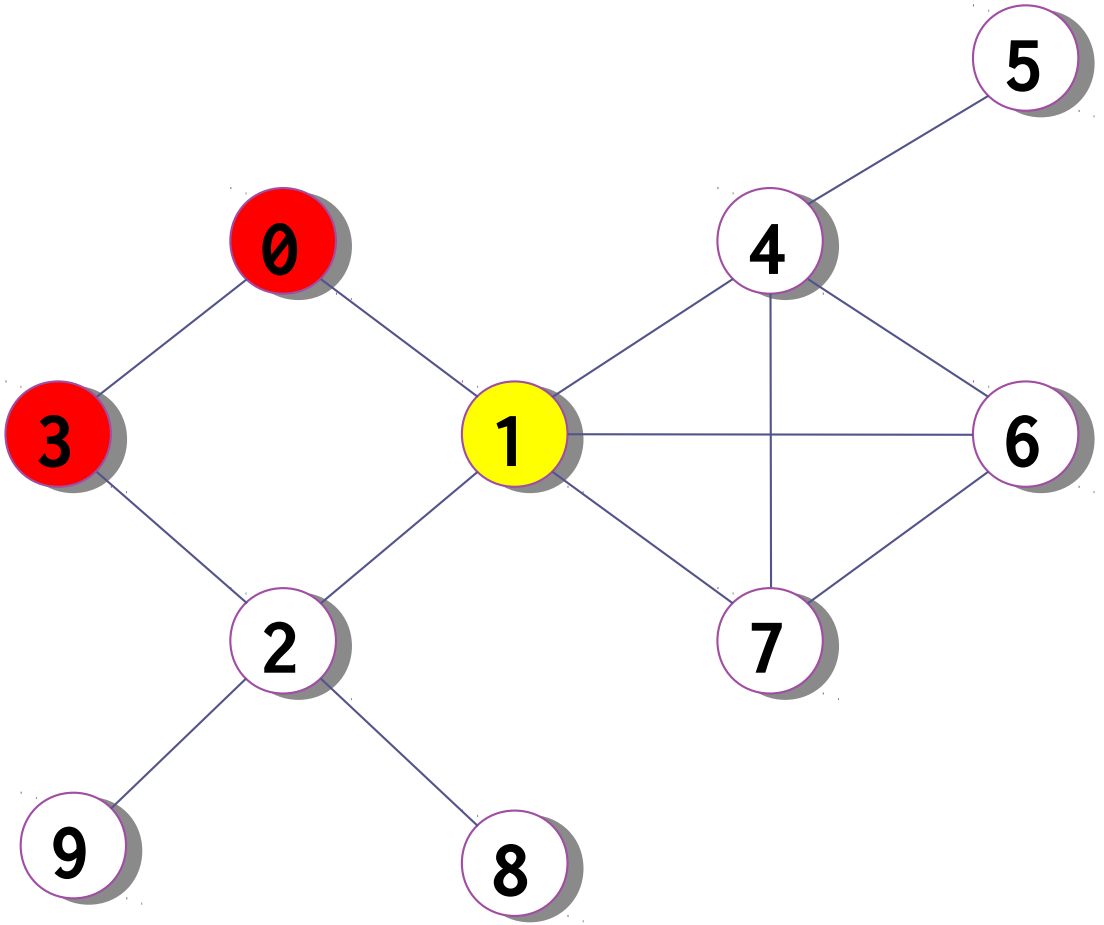
Example of a breadth-first search

Queue:

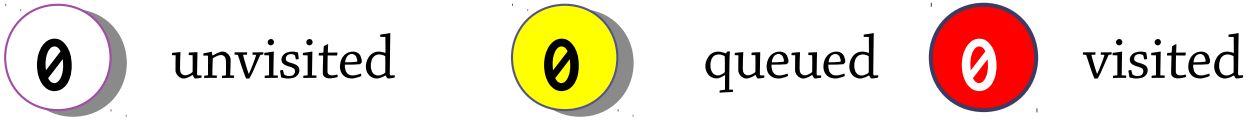
1

Visit order:

0 3



Step 1:
remove node
from queue
and visit it



Example of a bre arch

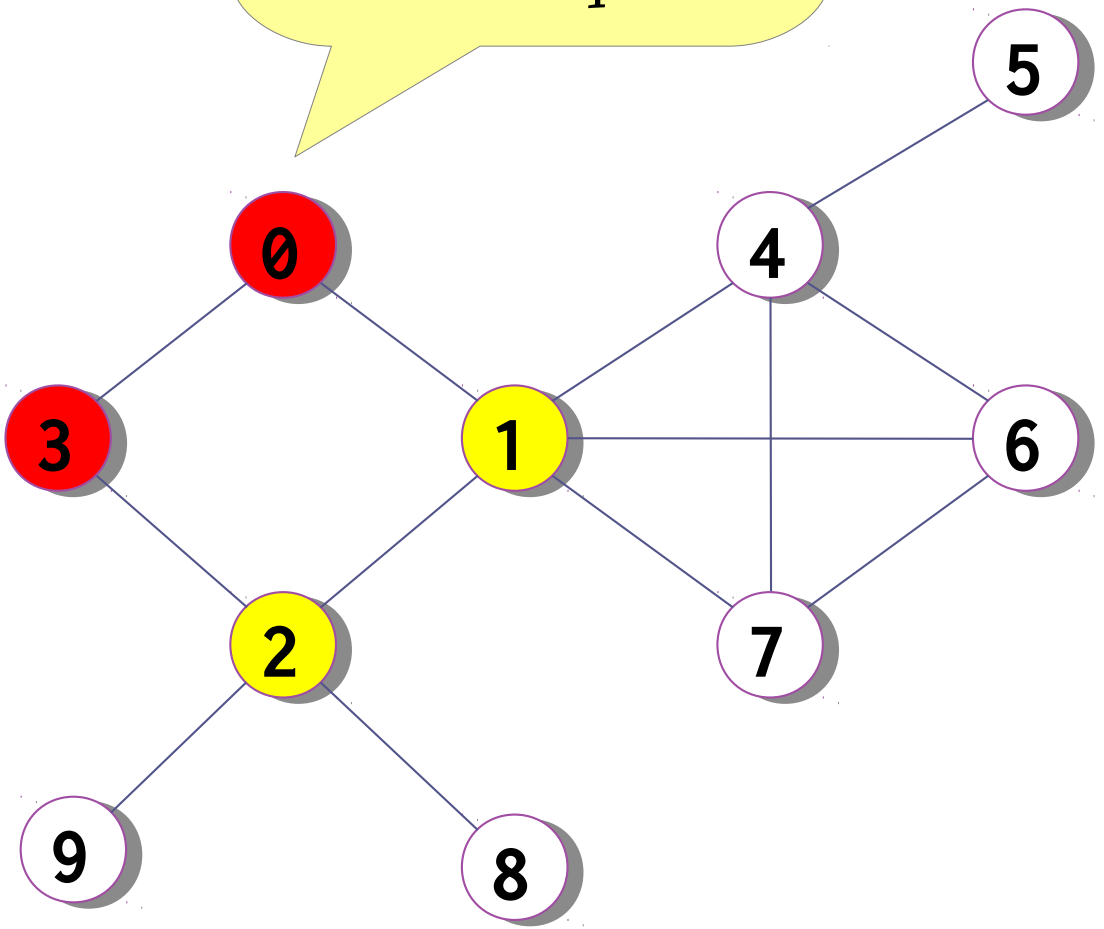
0 is already visited, so we don't add it to the queue

Queue:

1 2

Visit order:

0 3



Step 2:
add adjacent nodes to queue
(only unvisited ones)

0 unvisited 0 queued 0 visited

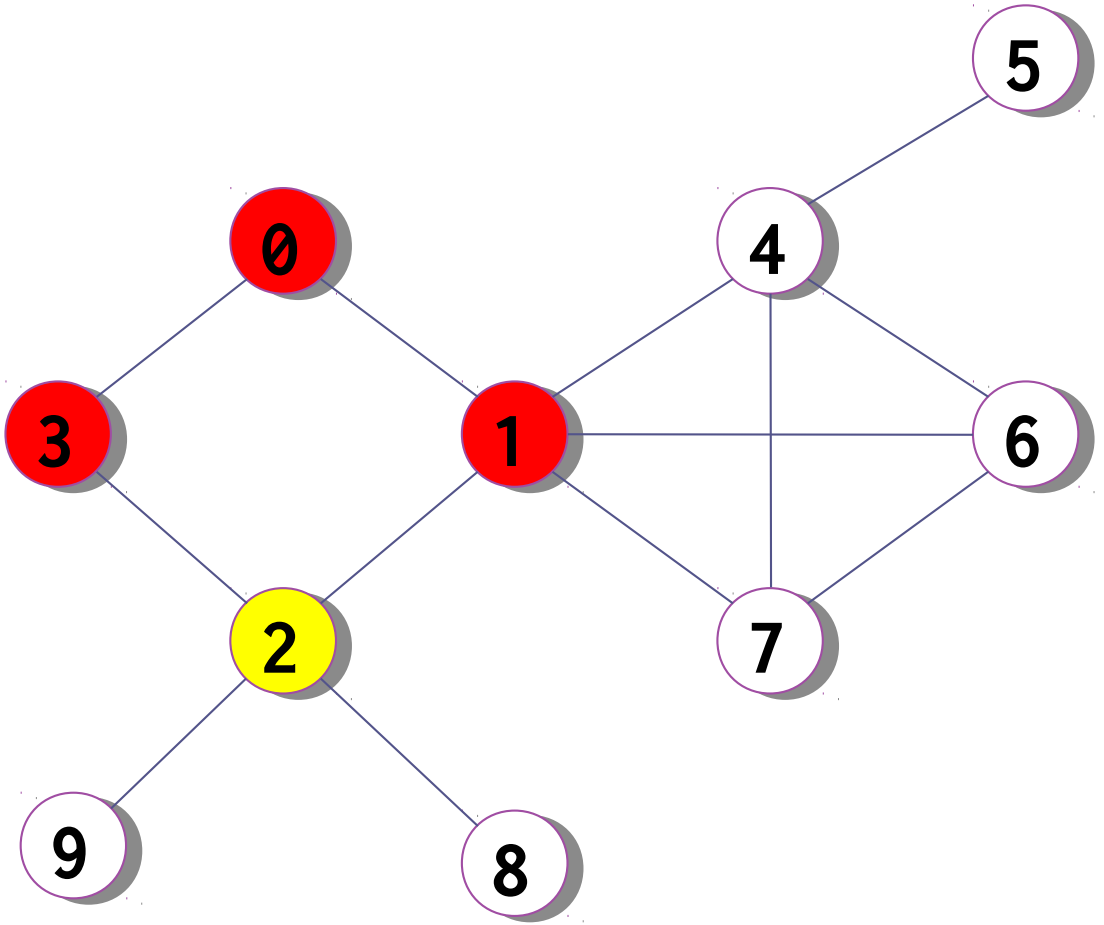
Example of a breadth-first search

Queue:

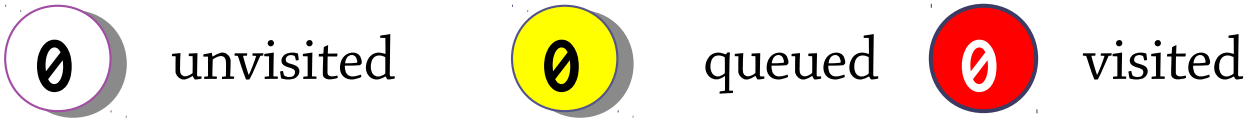
2

Visit order:

0 3 1



Step 1:
remove node
from queue
and visit it



Example of a bre arch

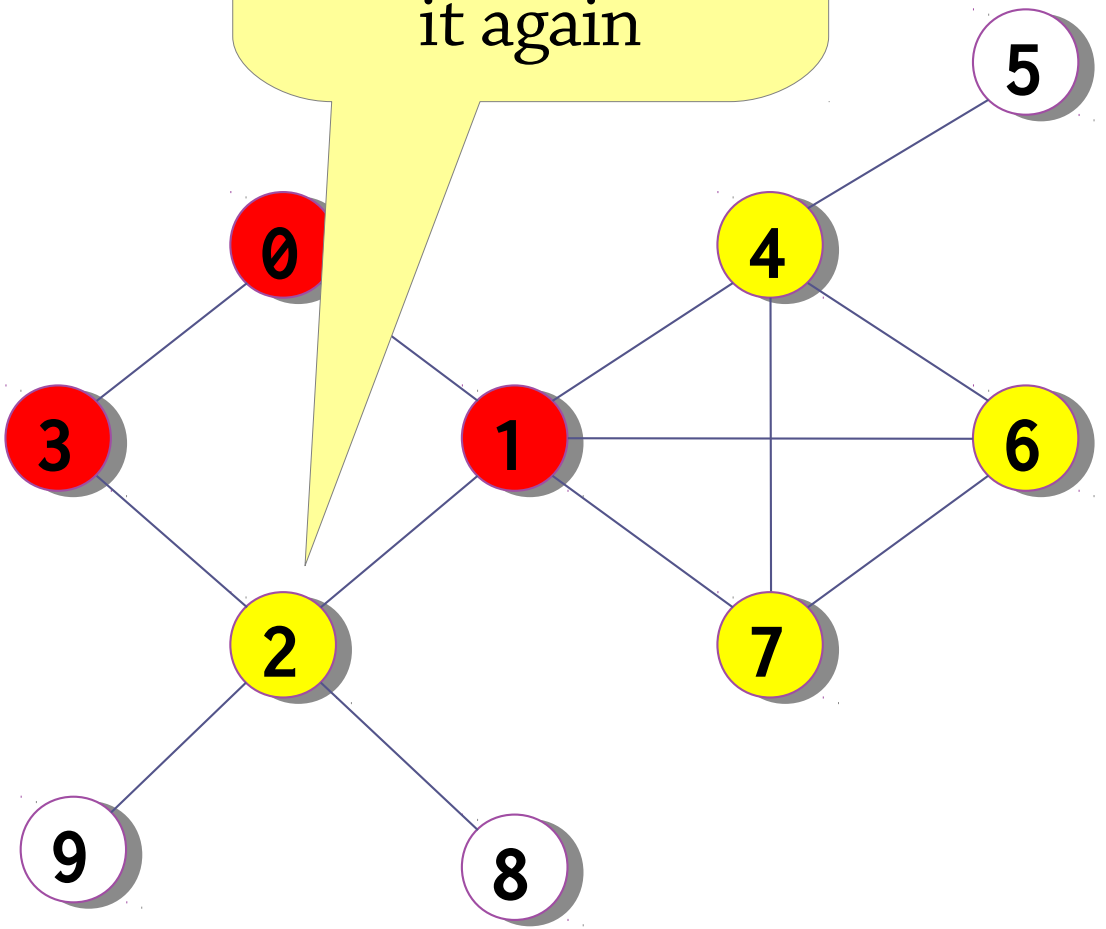
Queue:

2 4 6 7

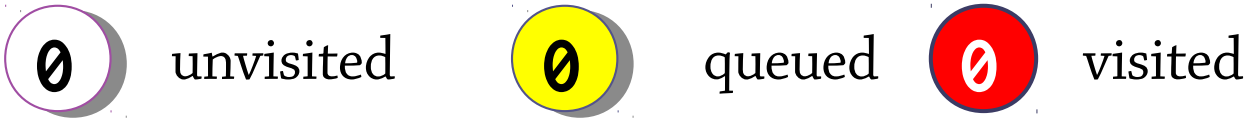
Visit order:

0 3 1

2 is already in the queue, so we don't add it again



Step 2:
add adjacent nodes
to queue
(only unvisited ones)



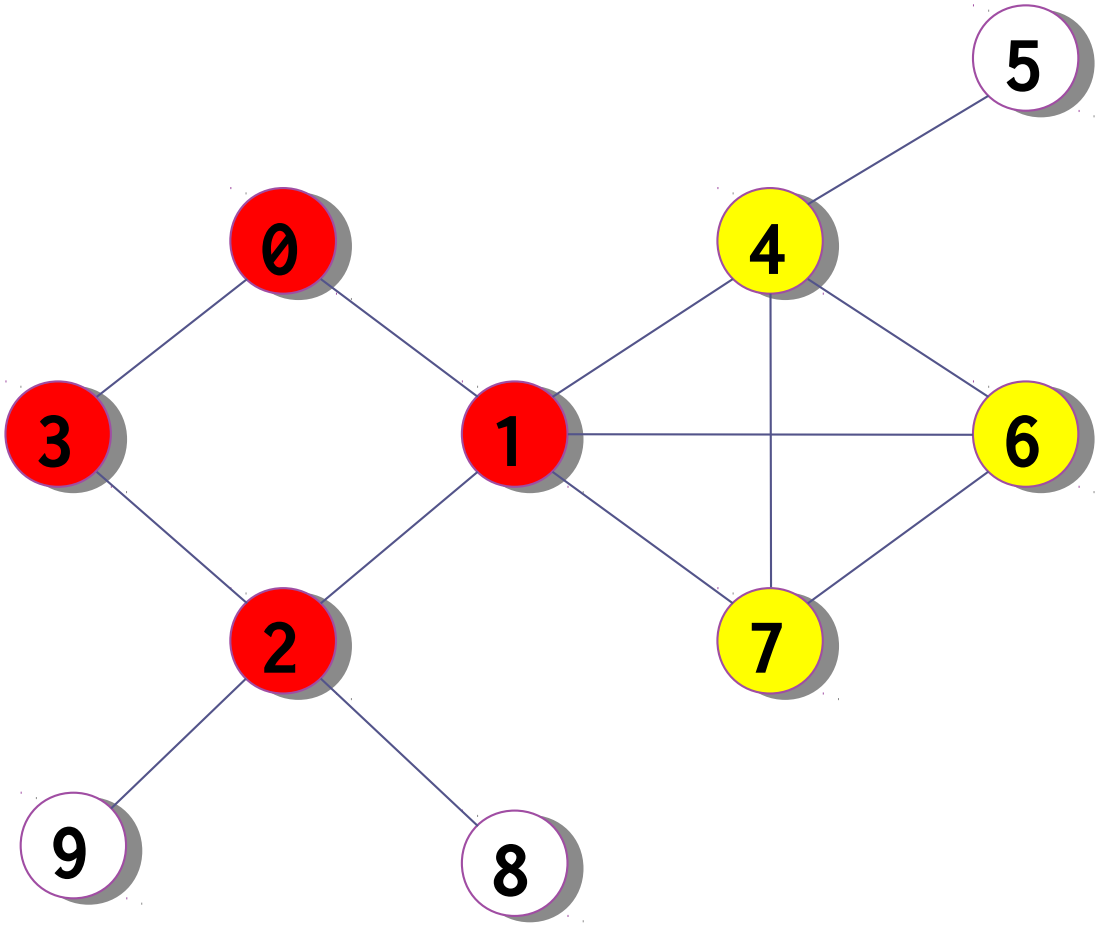
Example of a breadth-first search

Queue:

4 6 7

Visit order:

0 3 1 2



Step 1:
remove node
from queue
and visit it

Example of a breadth-first search

Queue:

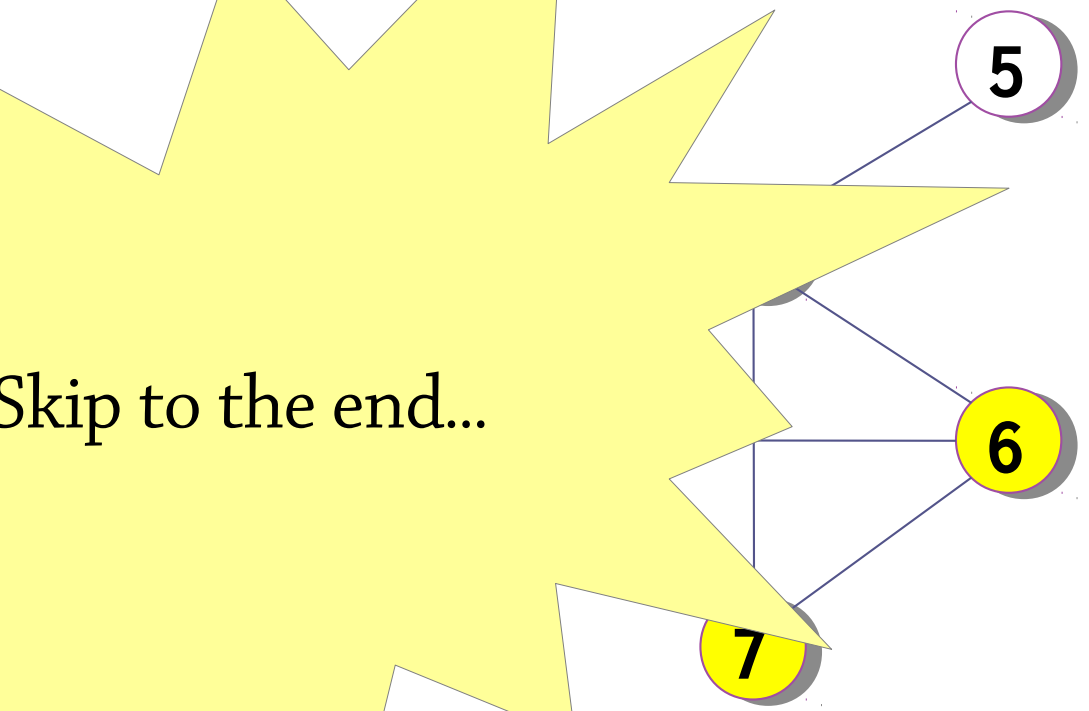
4 6 7 9 8

Visit orde

0 3 1 2

Skip to the end...

Step 2:
add adjacent nodes
to queue
(only unvisited ones)



0 unvisited 0 queued 0 visited

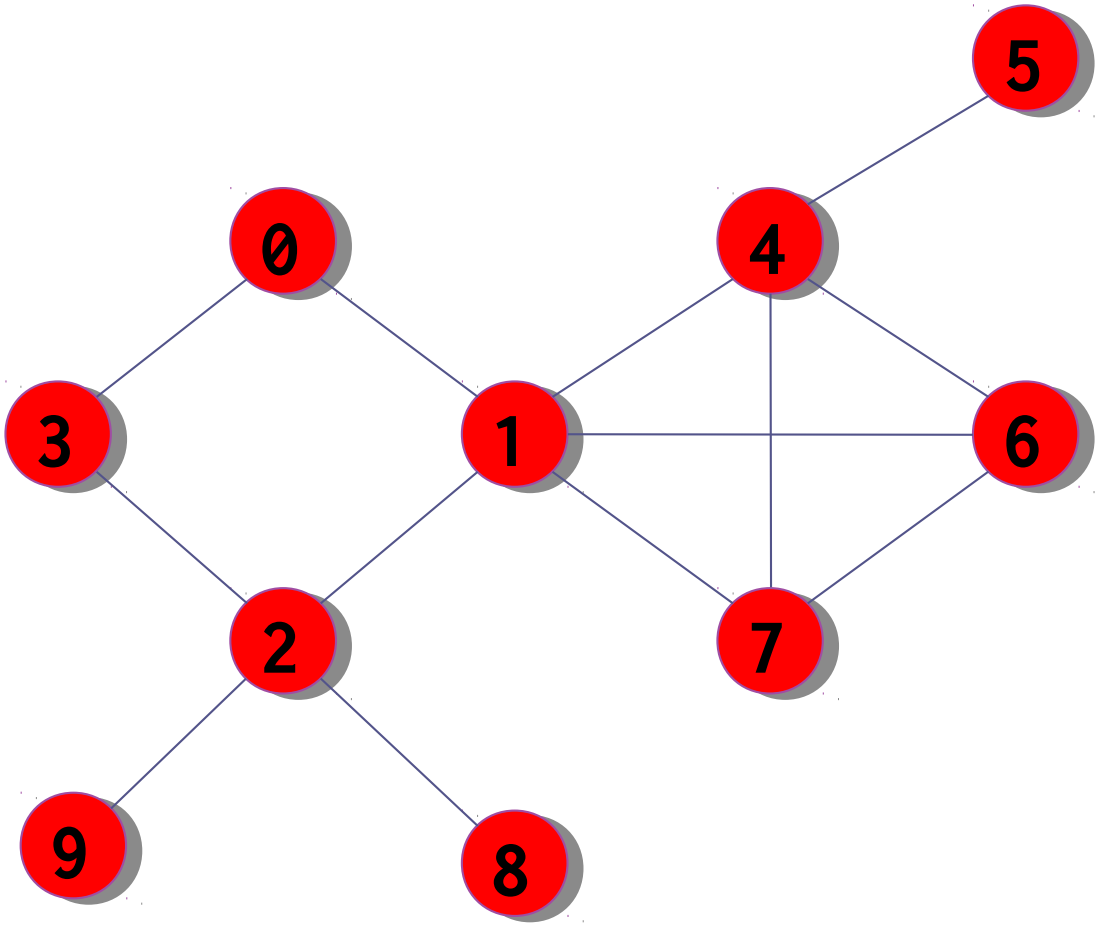
Example of a breadth-first search

Queue:

Visit order:

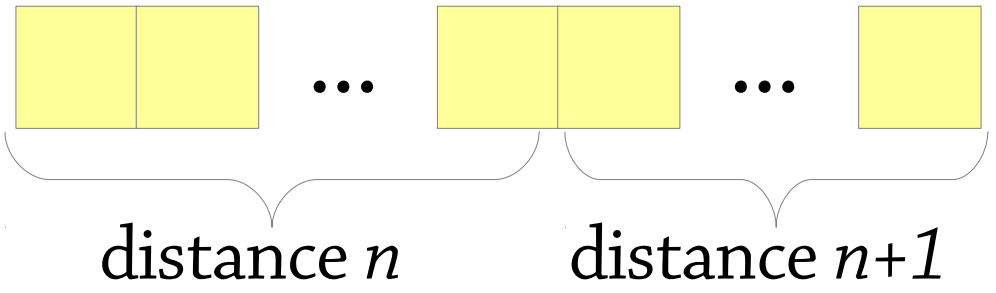
0 3 1 2 4
6 7 9 8 5

We reach step 1, but the queue is empty, and **we're finished!**



Why does using a queue work?

The queue in BFS always contains nodes that are *n* distance from the start node, followed by nodes that are *n+1* distance away:



When we remove the node from the head of the queue (distance *n*), we add its neighbours (distance *n+1*) to the end – so this situation remains true

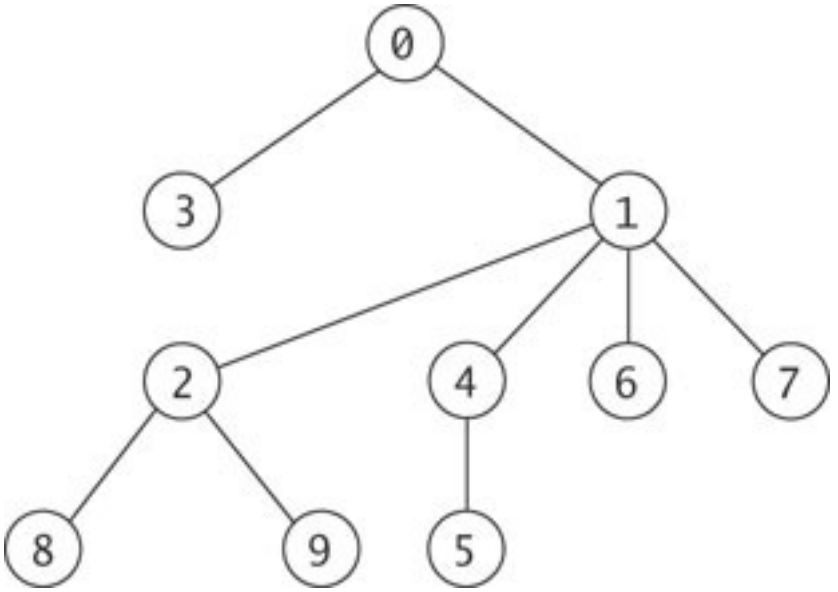
This means that we explore all nodes of distance *n* before getting to distance *n+1*

- Once we remove the first distance *n+1* node, the queue will contain nodes of distance *n+1* and *n+2*, so we go up in order of distance

Breadth-first search trees

While doing the BFS, we can record *which node we came from* when visiting each node in the graph

(we do this when adding a node to the queue)



We can use this information to find the *shortest path* from the start node to any other node

We can even build the *breadth-first search tree*, which shows how the graph was explored and tells you the shortest path to all nodes

Tree graphs

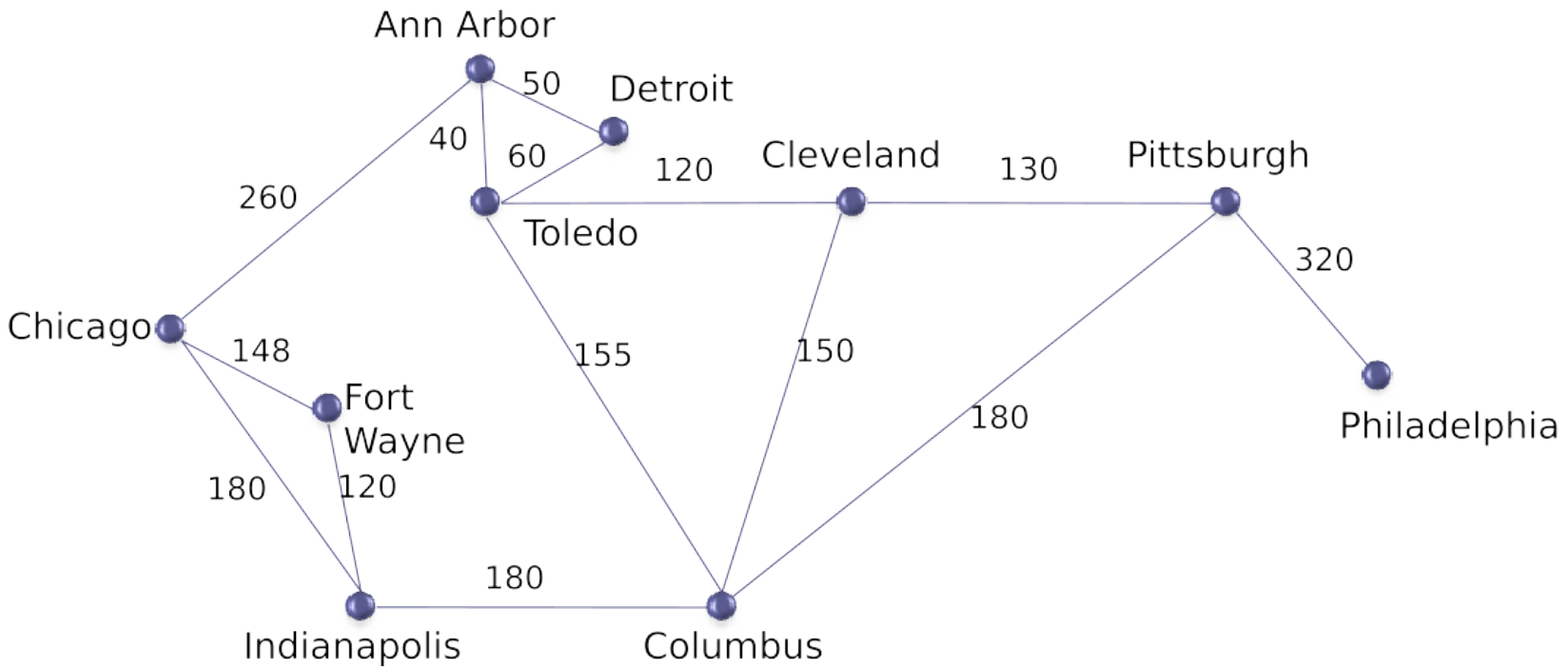
- A tree, seen as a special case of a graph, is a acyclic, connected graph.
- A rooted tree corresponds to a normal tree. A rooted tree is a tree where one node is identified as the root.
- What's the relation between the number of verteces and edges in a tree?

$$|V| = |E| + 1$$

Dijkstra's algorithm
Prim's algorithm

Weighted graphs

In a *weighted graph*, each edge is labelled with a *weight*, a number:



The weight typically represents the "cost" of following the edge

The (weighted) shortest path problem

Find the *path with least total weight* from point A to point B in a weighted graph

(If there are no weights:
can be solved with BFS)

Useful in e.g.,
route planning,
network routing

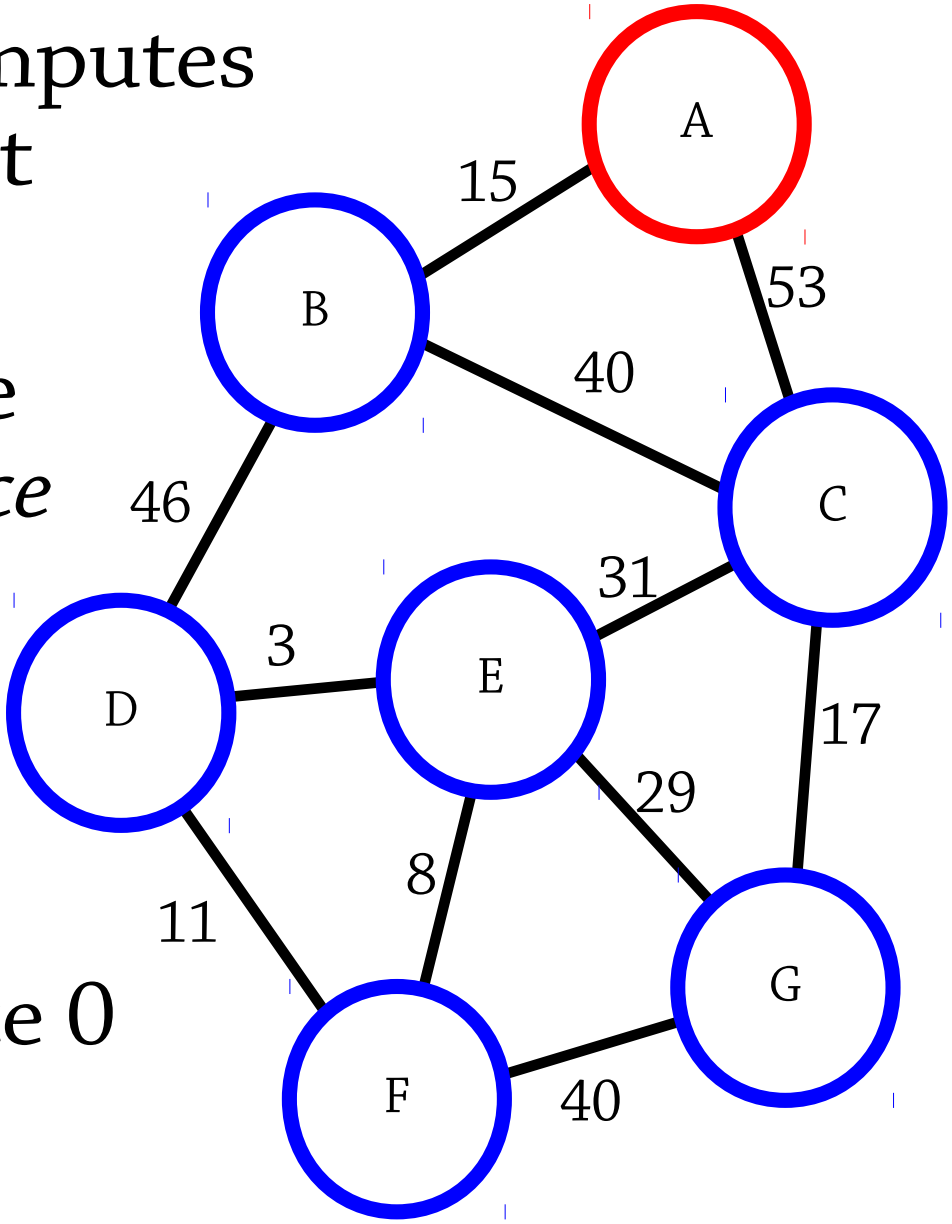
Most common approach:
Dijkstra's algorithm,
which works when all
edges have positive weight

Dijkstra's algorithm

Dijkstra's algorithm computes the distance from a start node to *all other nodes*

It visits the nodes of the graph in order of *distance from the start node*, and remembers their distance

We first visit the start node, which has distance 0

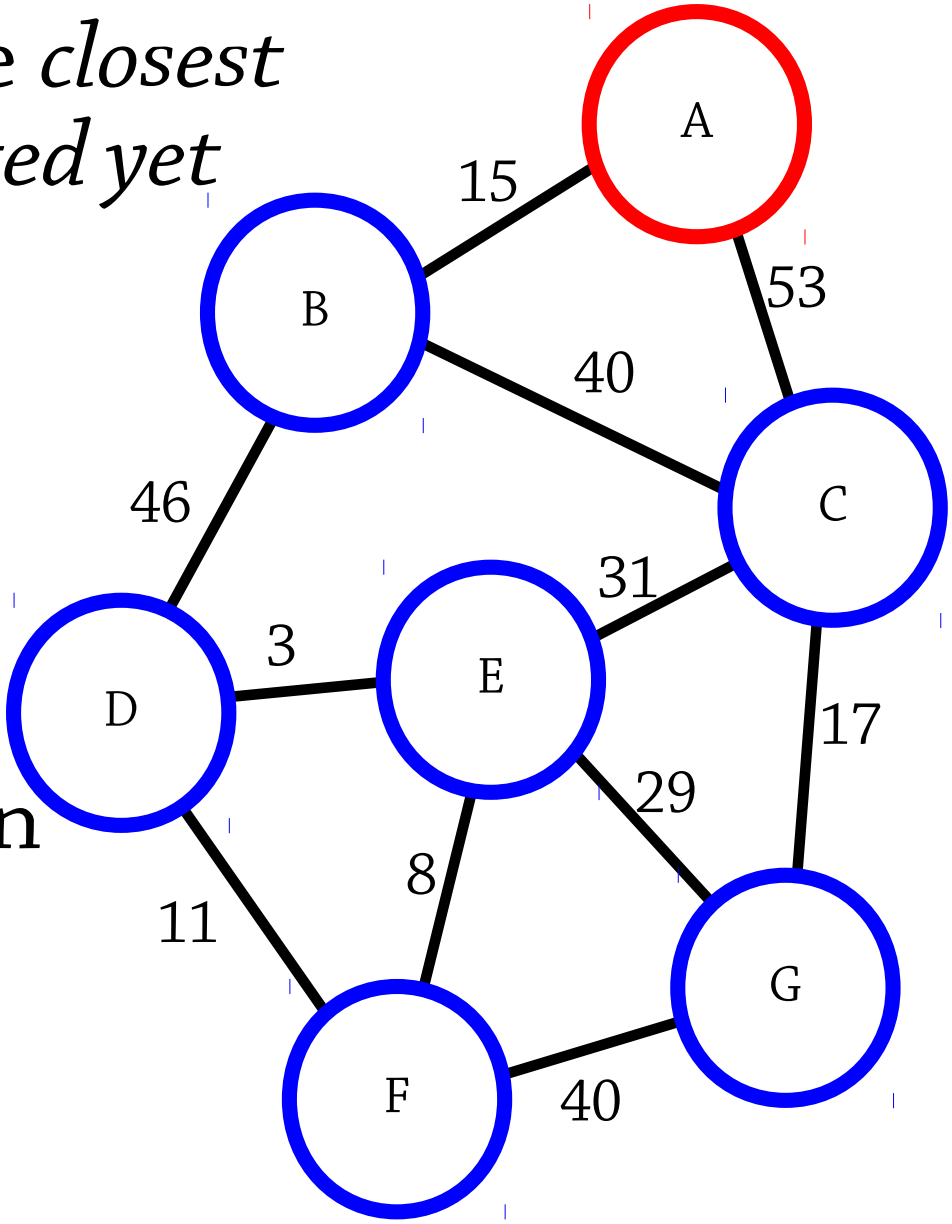


Dijkstra's algorithm

At each step we visit the *closest node that we haven't visited yet*

This node must be adjacent to a node we *have* visited (why?)

By looking at the outgoing edges from the visited nodes, we can find the closest unvisited node



Dijkstra's algorithm

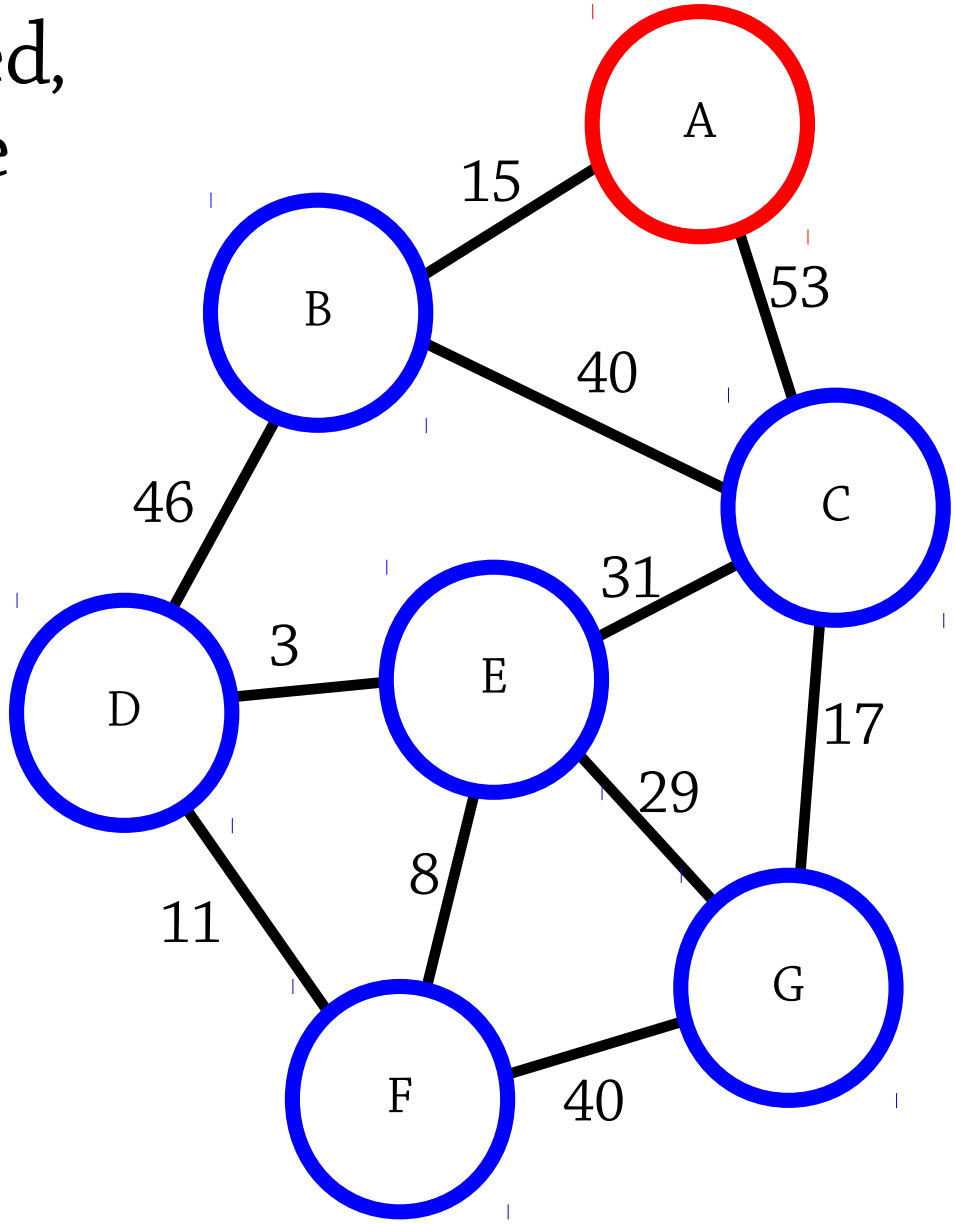
For each node *x* we've visited, and each edge x → y, where *y* is unvisited:

- Add the distance to *x* and the weight of the edge *x* → *y*

Whichever node *y* has the shortest total distance, visit it!

- This is the closest unvisited node

Repeat until there are no edges to unvisited nodes



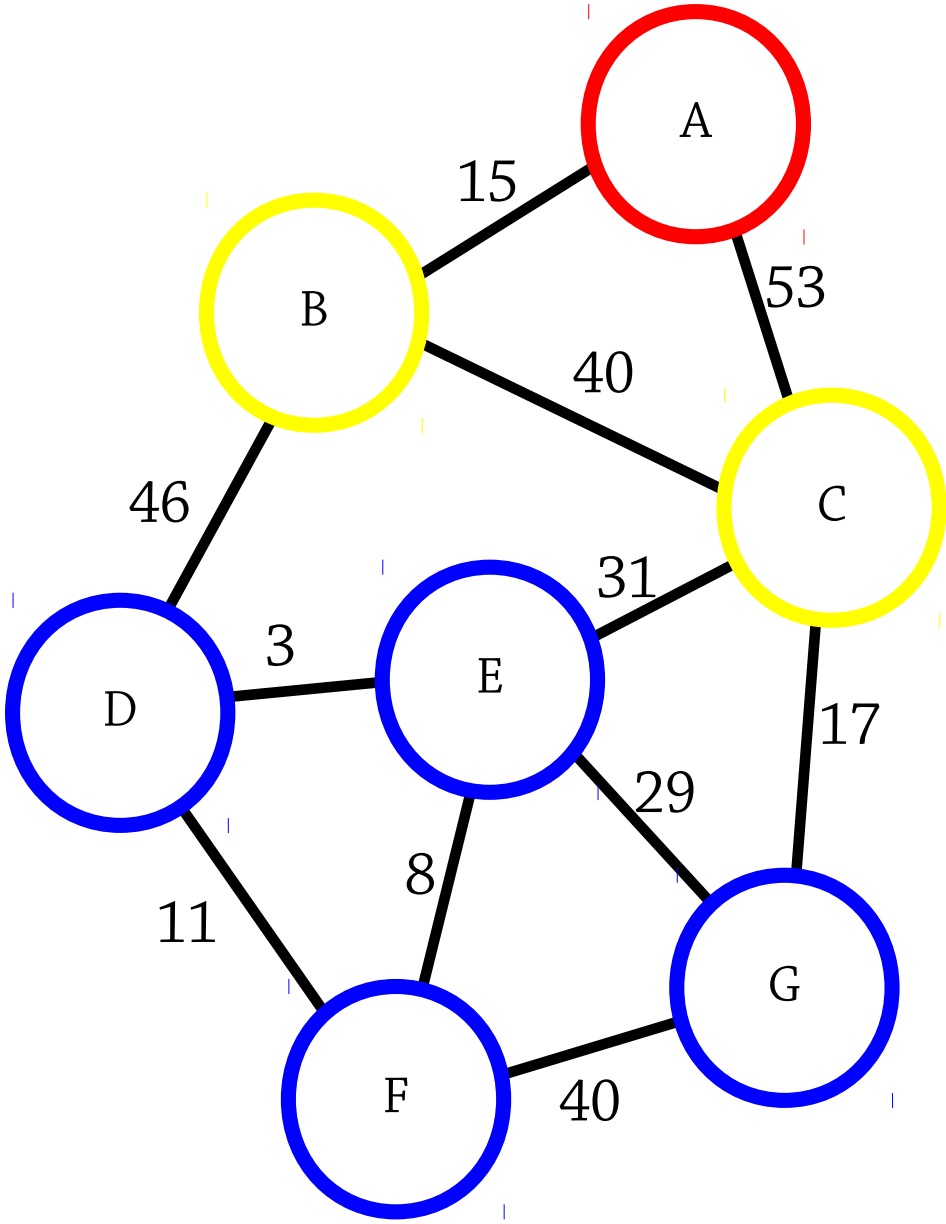
Dijkstra's algorithm

Visited nodes:
A distance 0

Neighbours of A are B (distance 15), C (distance 53)

So visit B (distance 15)

(Red = visited node, yellow = neighbour of visited node)



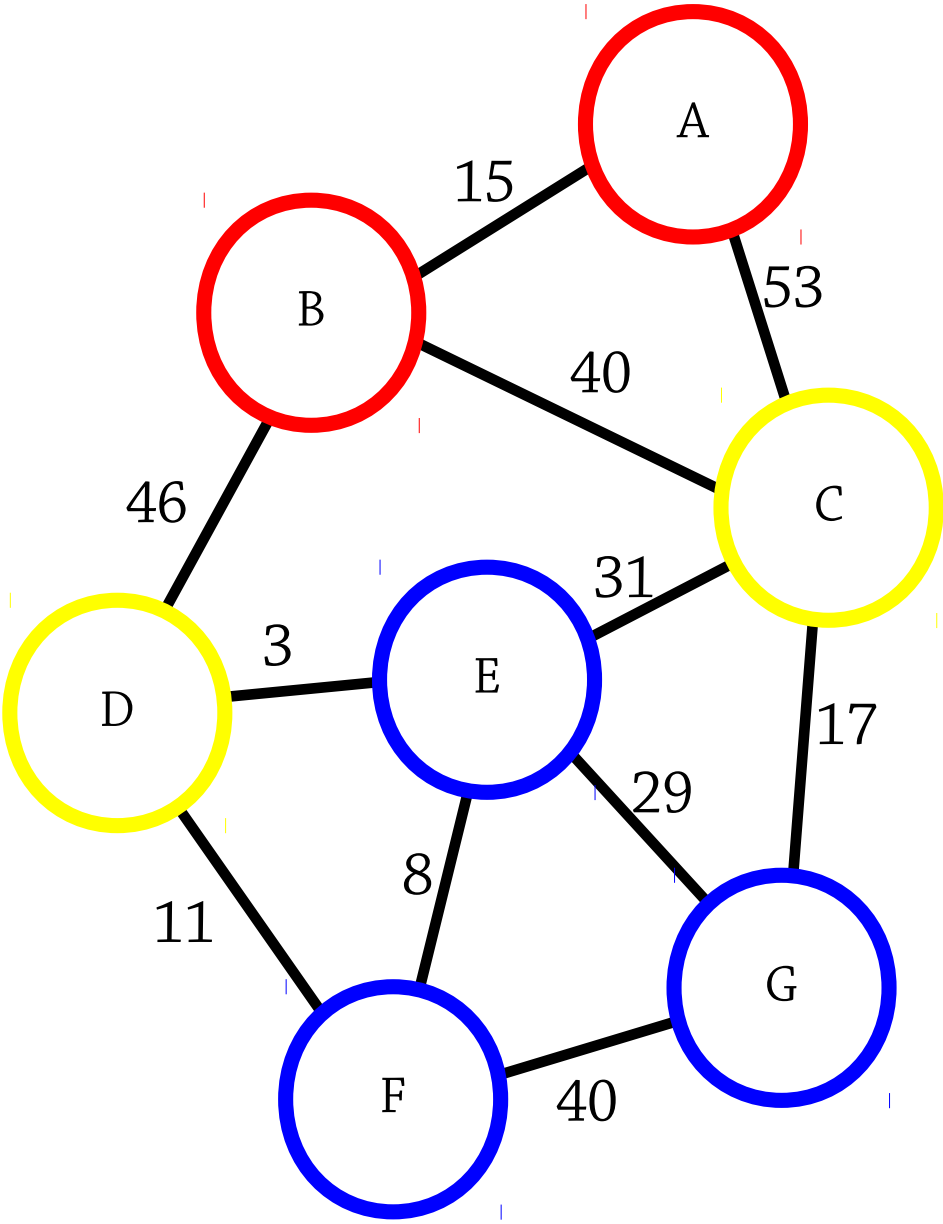
Dijkstra's algorithm

Visited nodes:
A distance 0
B distance 15

Neighbours are:

- D (distance 15 + 46 = 61)
- C (distance 53 – also via B 15 + 40 = 55)

So visit C (distance 53)



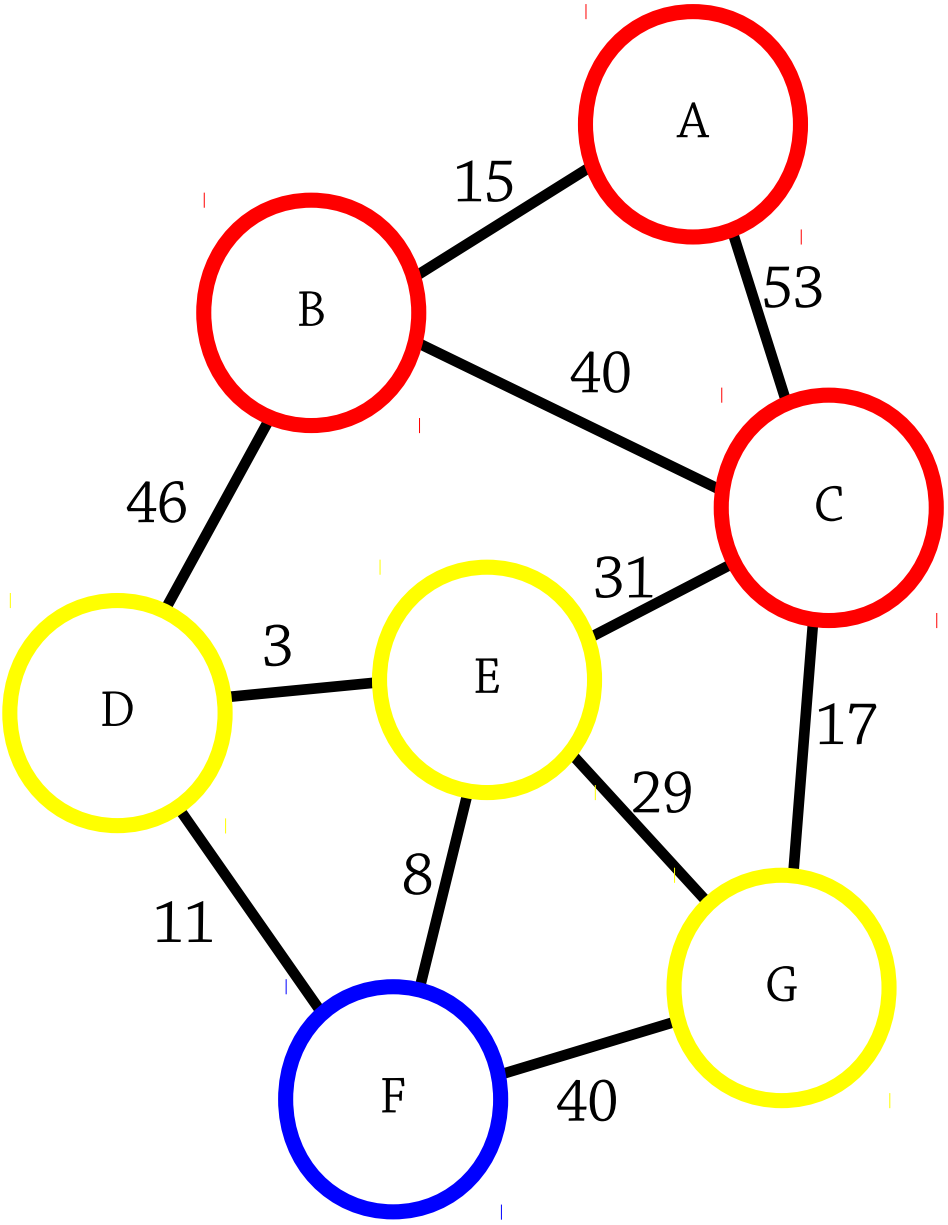
Dijkstra's algorithm

Visited nodes:
A distance 0
B distance 15
C distance 53

Neighbours are:

- D (distance 15 + 46 = 61)
- E (distance 53 + 31 = 84)
- G (distance 53 + 17 = 70)

So visit D (distance 61)



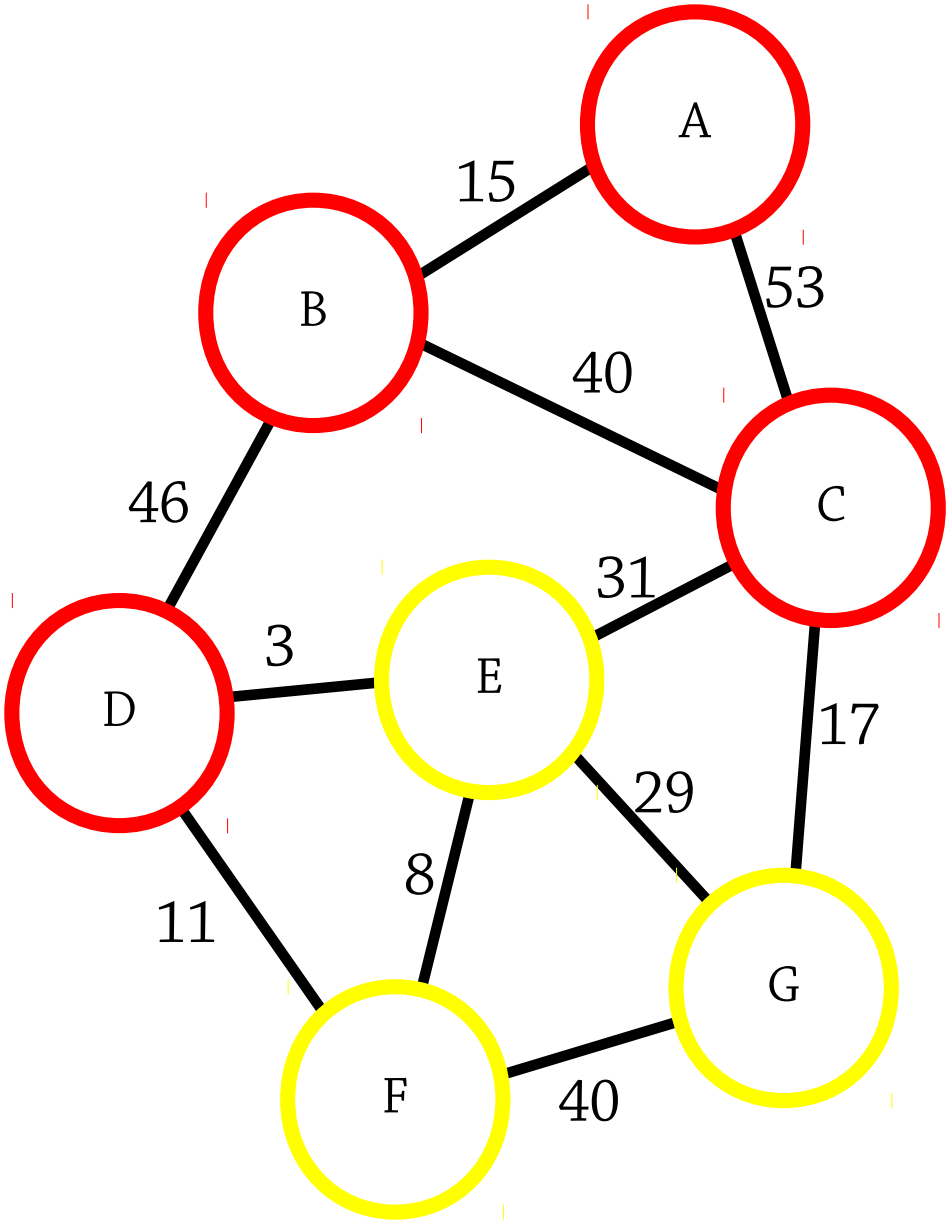
Dijkstra's algorithm

Visited nodes:
A distance 0
B distance 15
C distance 53
D distance 61

Neighbours are:

- E (distance
  61 + 3 = 64,
  also via C 55 + 29 = 84)
- G (distance 53 + 17 = 70)
- F (distance 61 + 11 = 72)

So visit E (distance 64)



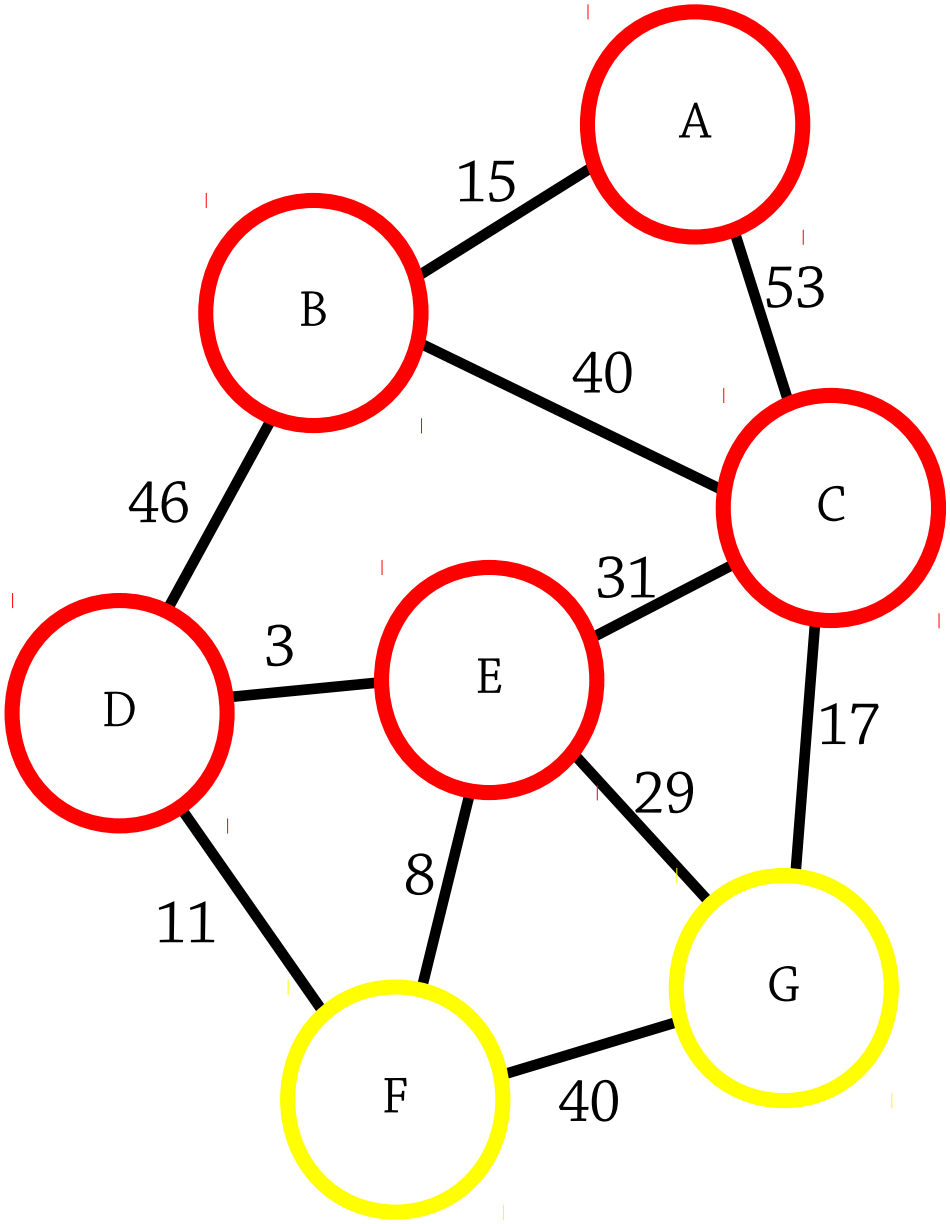
Dijkstra's algorithm

Visited nodes:
A distance 0
B distance 15
C distance 53
D distance 61
E distance 64

Neighbours are:

- G (distance 53 + 17 = 70, also via E 64 + 29 = 93)
- F (distance 61 + 11 = 72, also via E 64 + 8 = 72)

So visit G (distance 70)



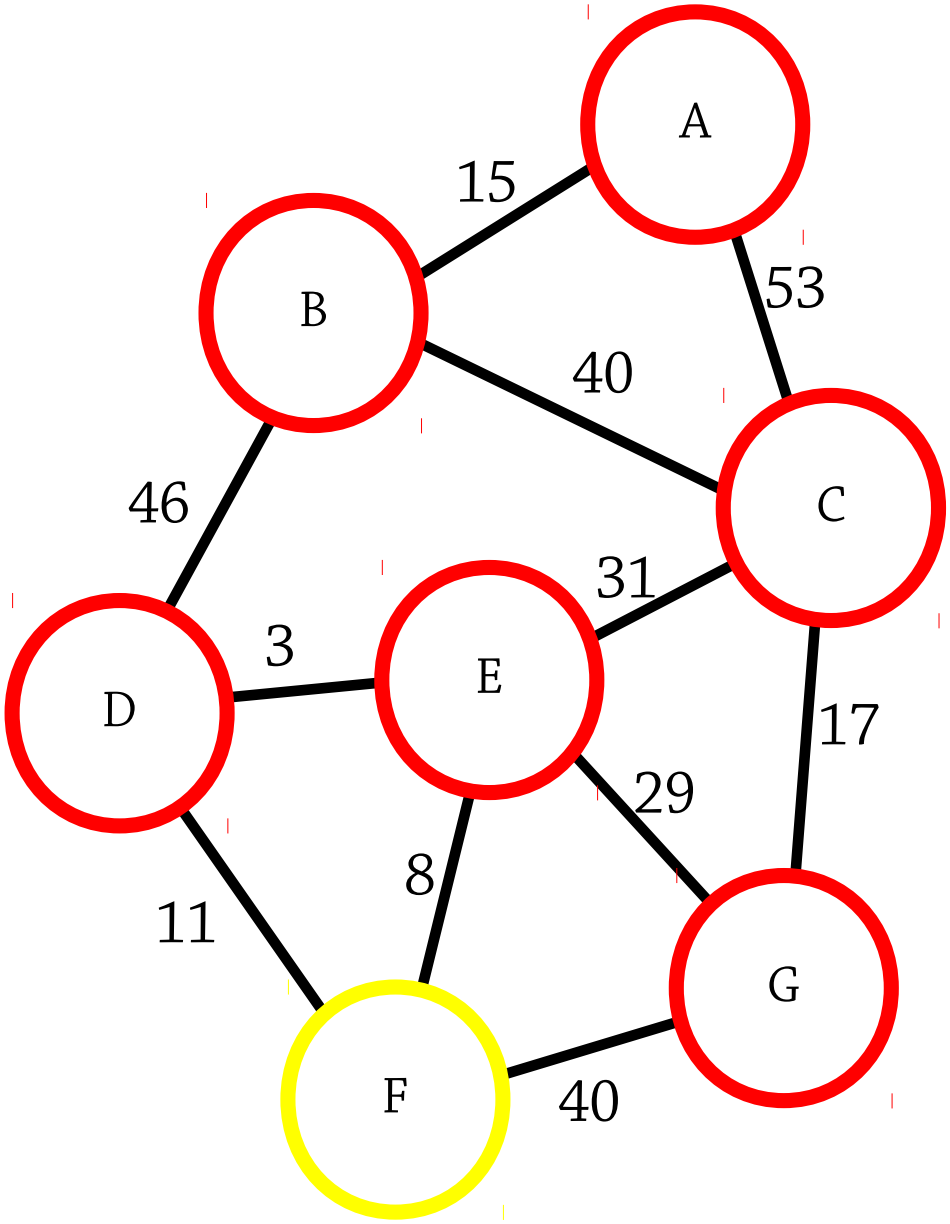
Dijkstra's algorithm

Visited nodes:
A distance 0
B distance 15
C distance 53
D distance 61
E distance 64
G distance 70

Neighbours are:

- F (distance 61 + 11 = 72,
  also via E 64 + 8 = 72,
  also via G 70 + 40 = 110)

So visit F (distance 72)

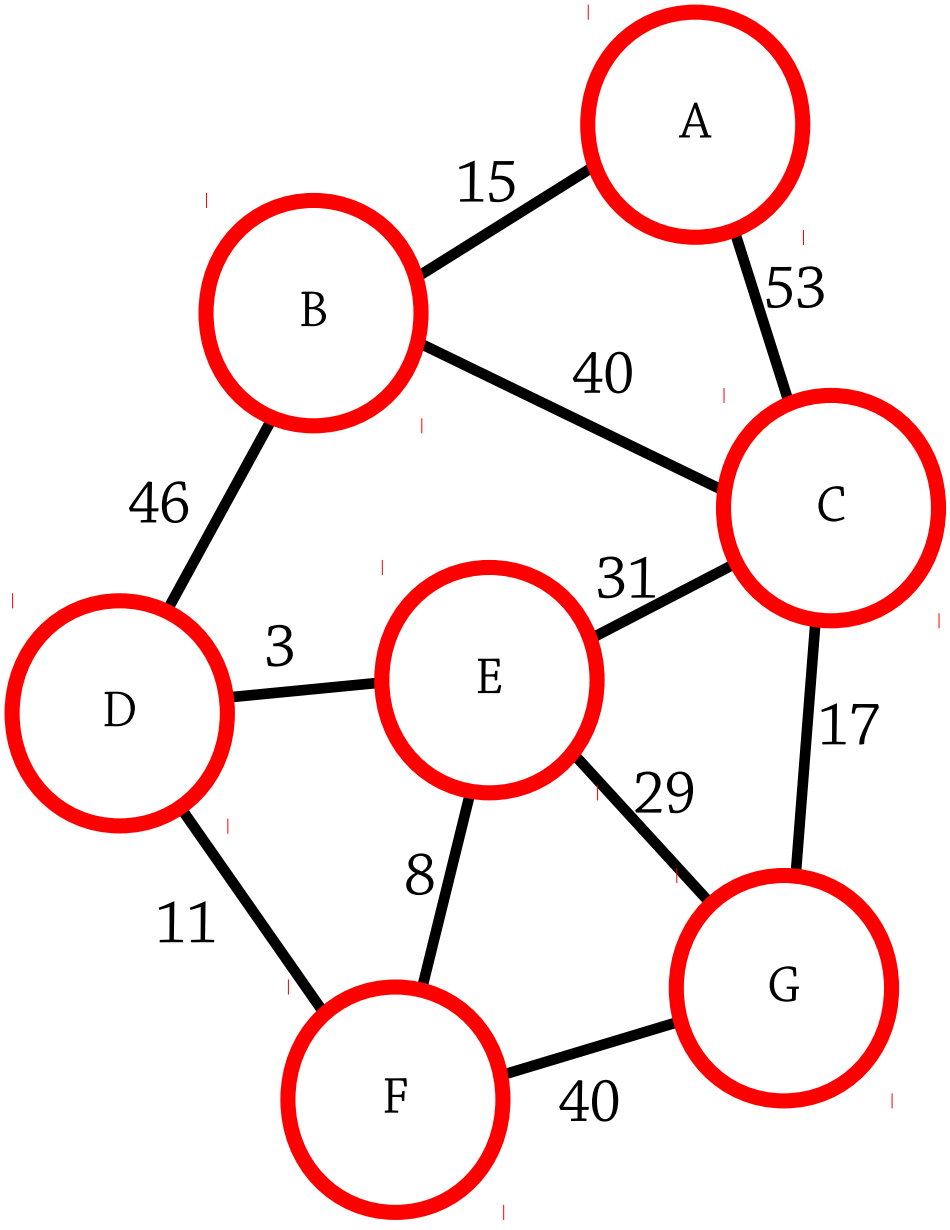


Dijkstra's algorithm

Visited nodes:

A distance 0
B distance 15
C distance 53
D distance 61
E distance 64
G distance 70
F distance 72

Finished!



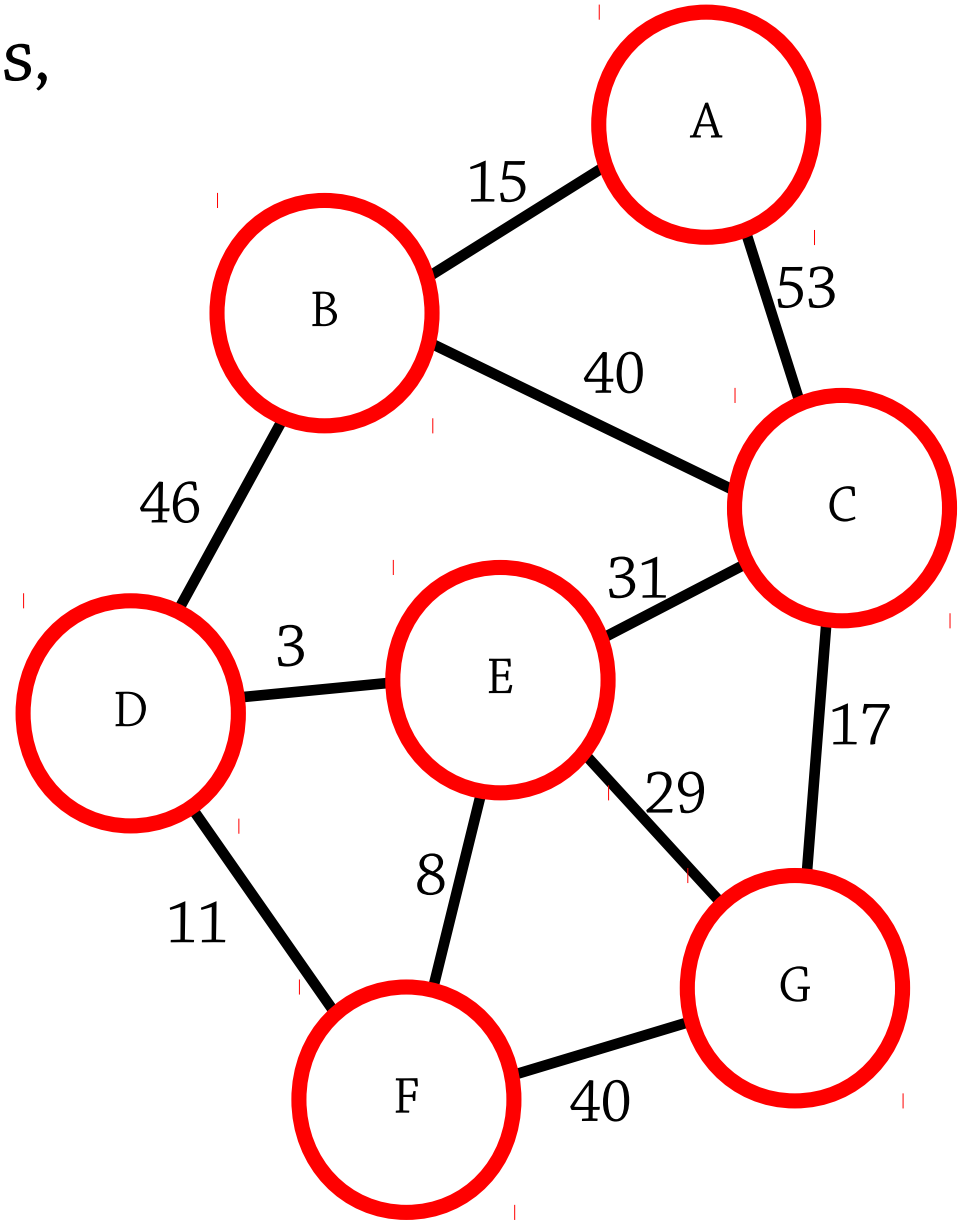
Dijkstra's algorithm

Once we have these distances, we can use them to find the shortest path to any node!

e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72



hm

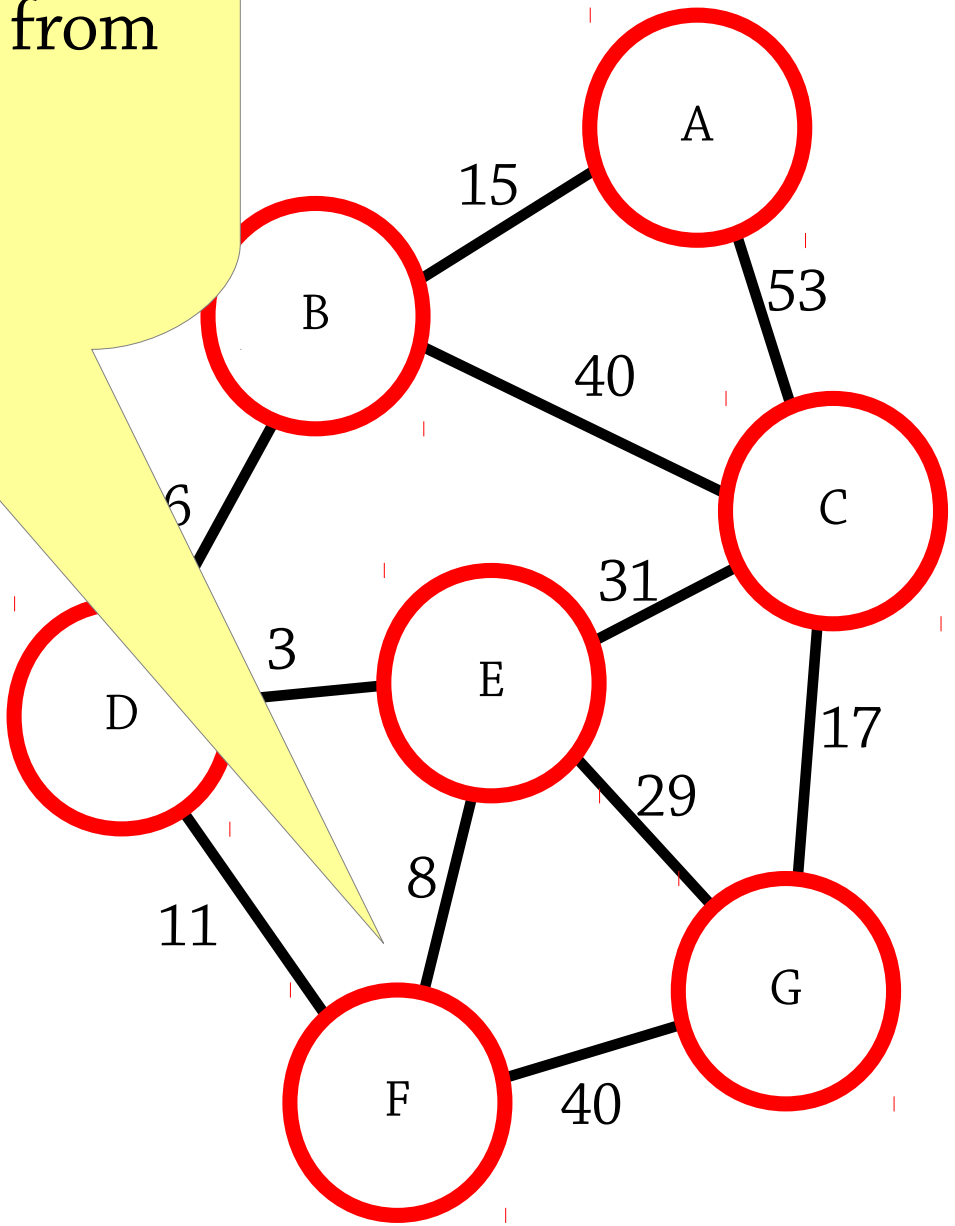
Once we
we can us
shortest p

e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72

To arrive at F, we must take the edge from D, E or G



m

Once we
we can us
shortest p

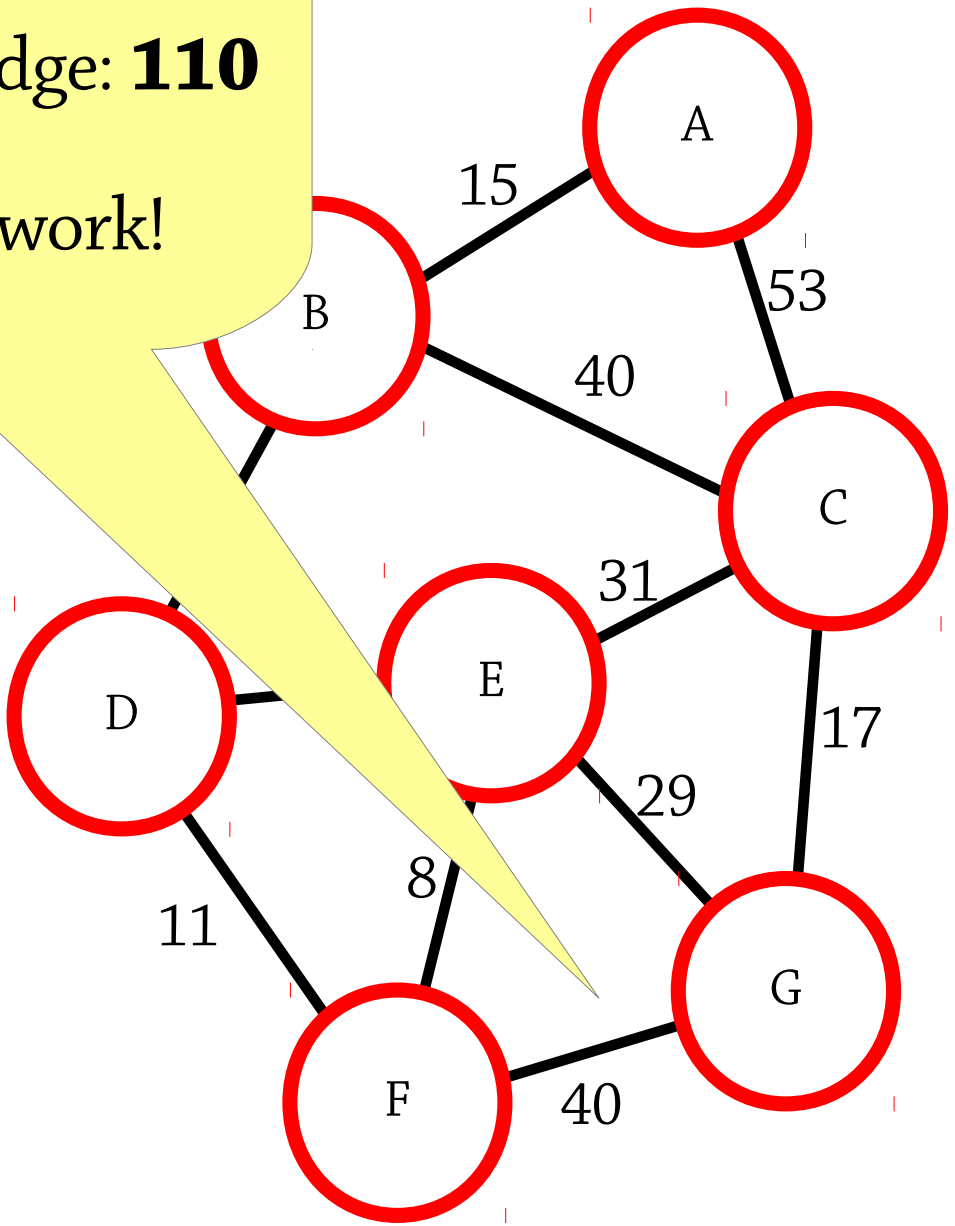
e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72

A → G: **70**
G → F edge: **40**

So coming via this edge: **110**
A → F: **72**
This route won't work!



m

Once we
we can us
shortest p

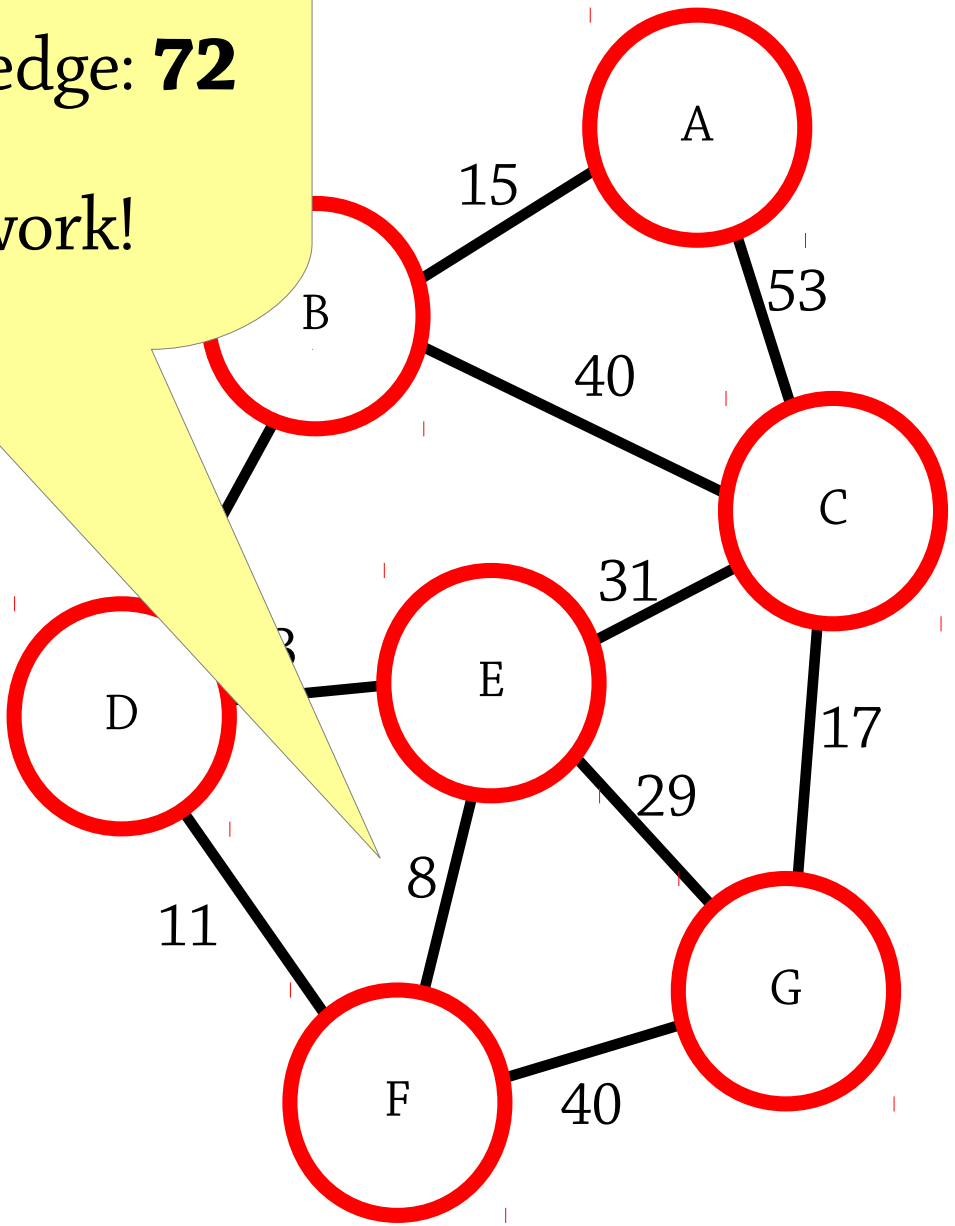
e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72

A → E: **64**
E → F edge: **8**

So coming via this edge: **72**
A → F: **72**
This route will work!



m

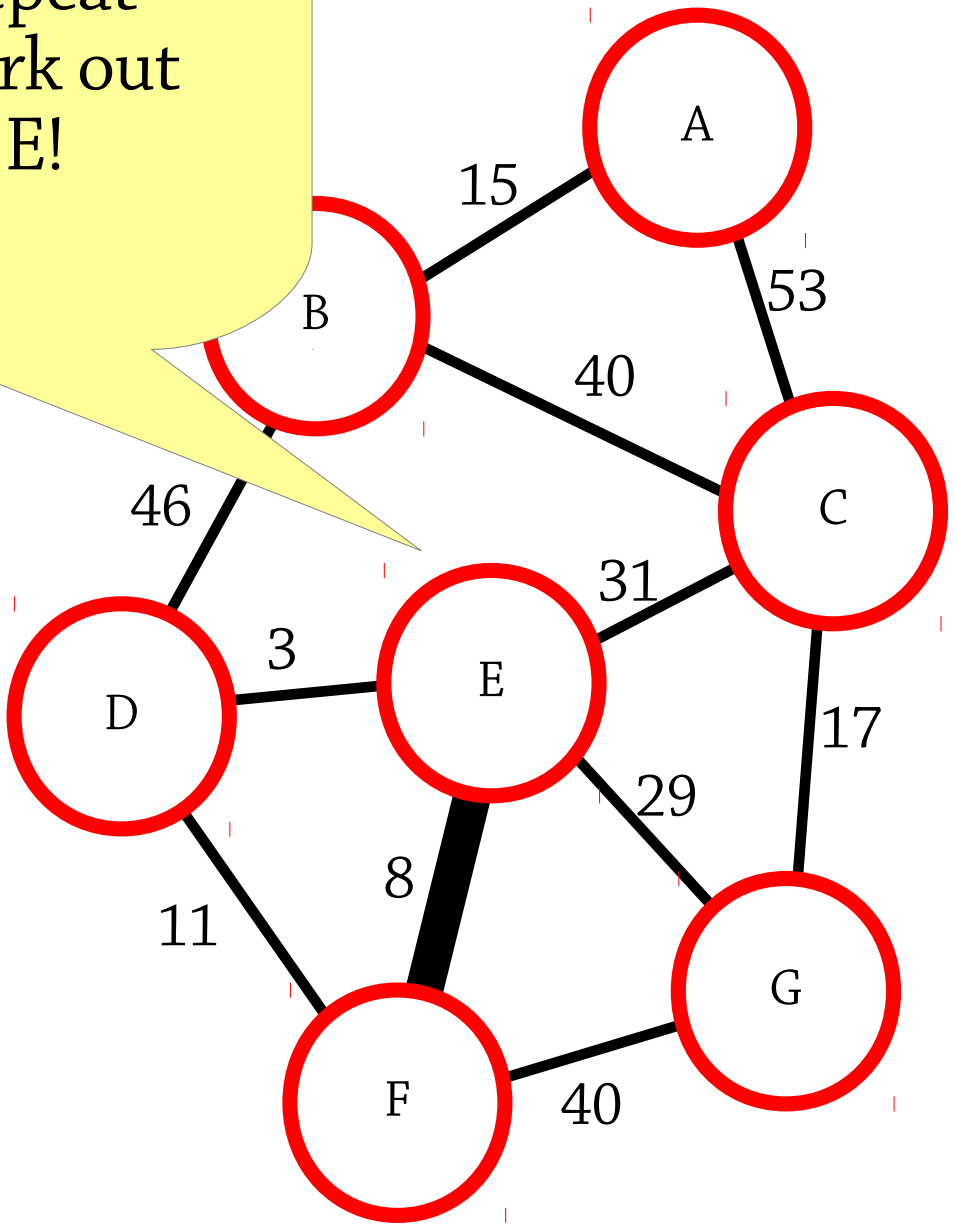
Once we
we can us
shortest p

e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72

Now we know we can come via E – so just repeat the process to work out how to get to E!



m

Once we
we can us
shortest p

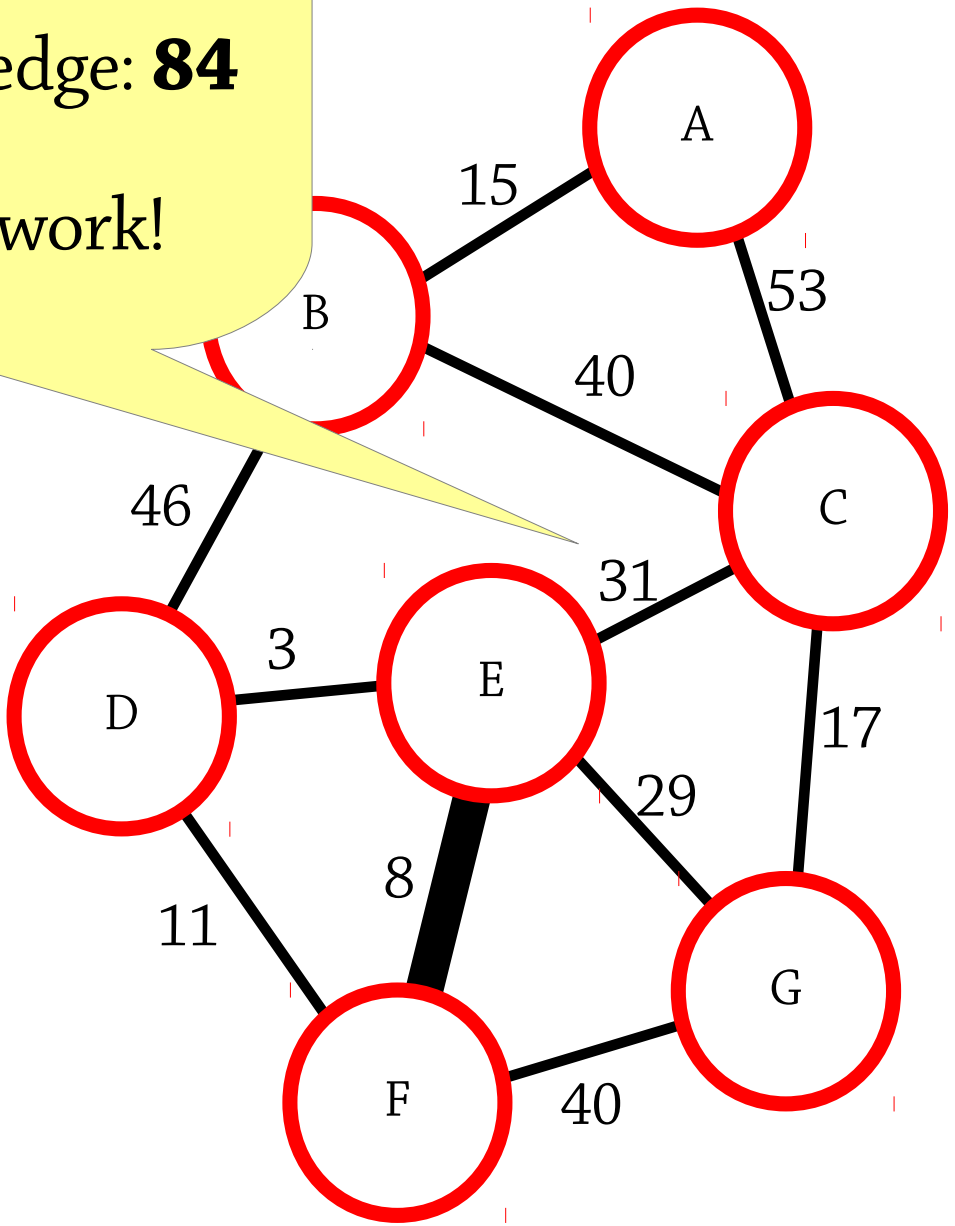
e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72

A → C: **53**
C → E edge: **31**

So coming via this edge: **84**
A → E: **64**
This route won't work!



m

Once we
we can us
shortest p

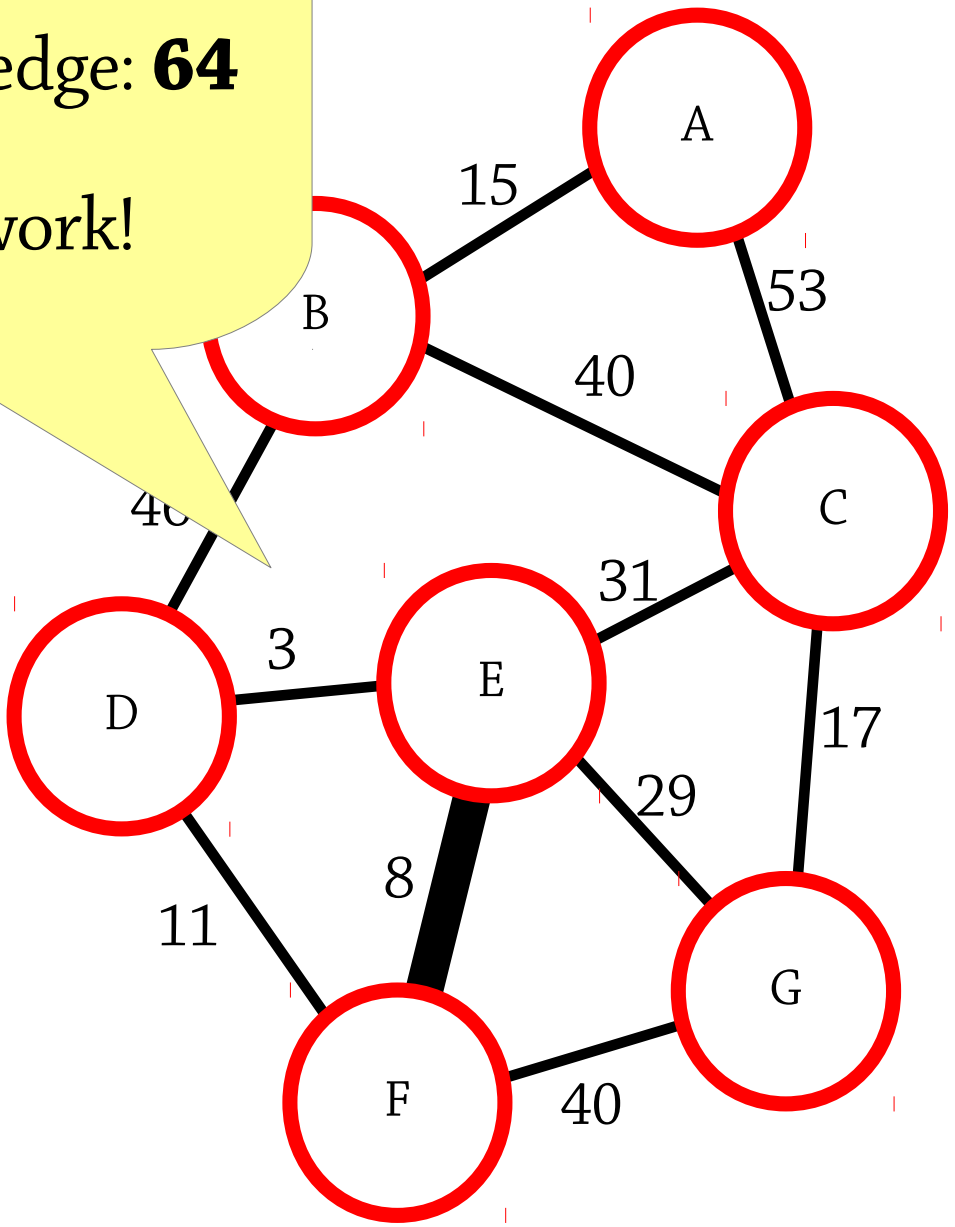
e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72

A → D: **61**
D → E edge: **3**

So coming via this edge: **64**
A → E: **64**
This route will work!



m

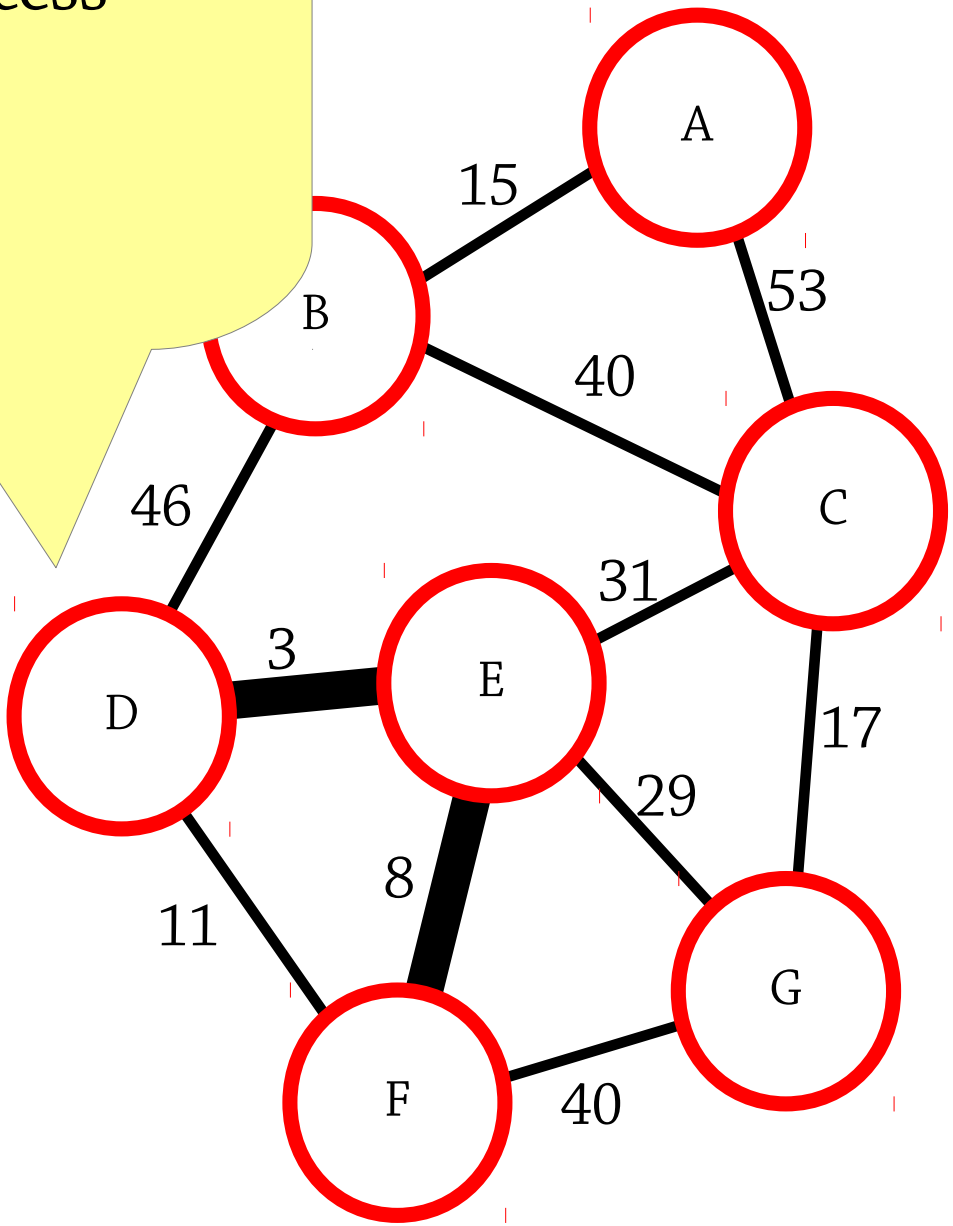
Repeat the process for D

Once we
we can us
shortest p

e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72



m

Once we
we can us
shortest p

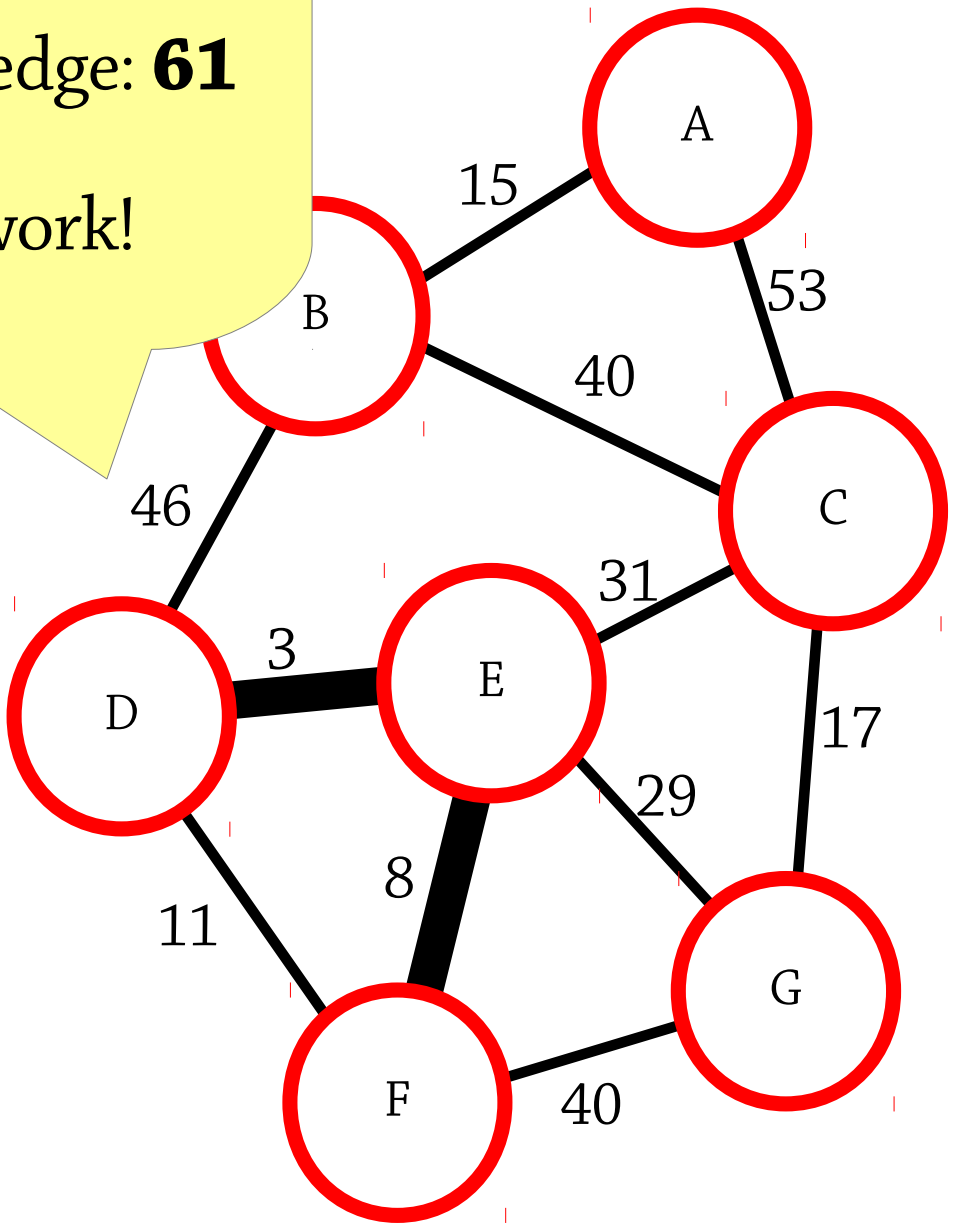
e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72

A → B: **15**
B → D edge: **46**

So coming via this edge: **61**
A → D: **61**
This route will work!



algorithm

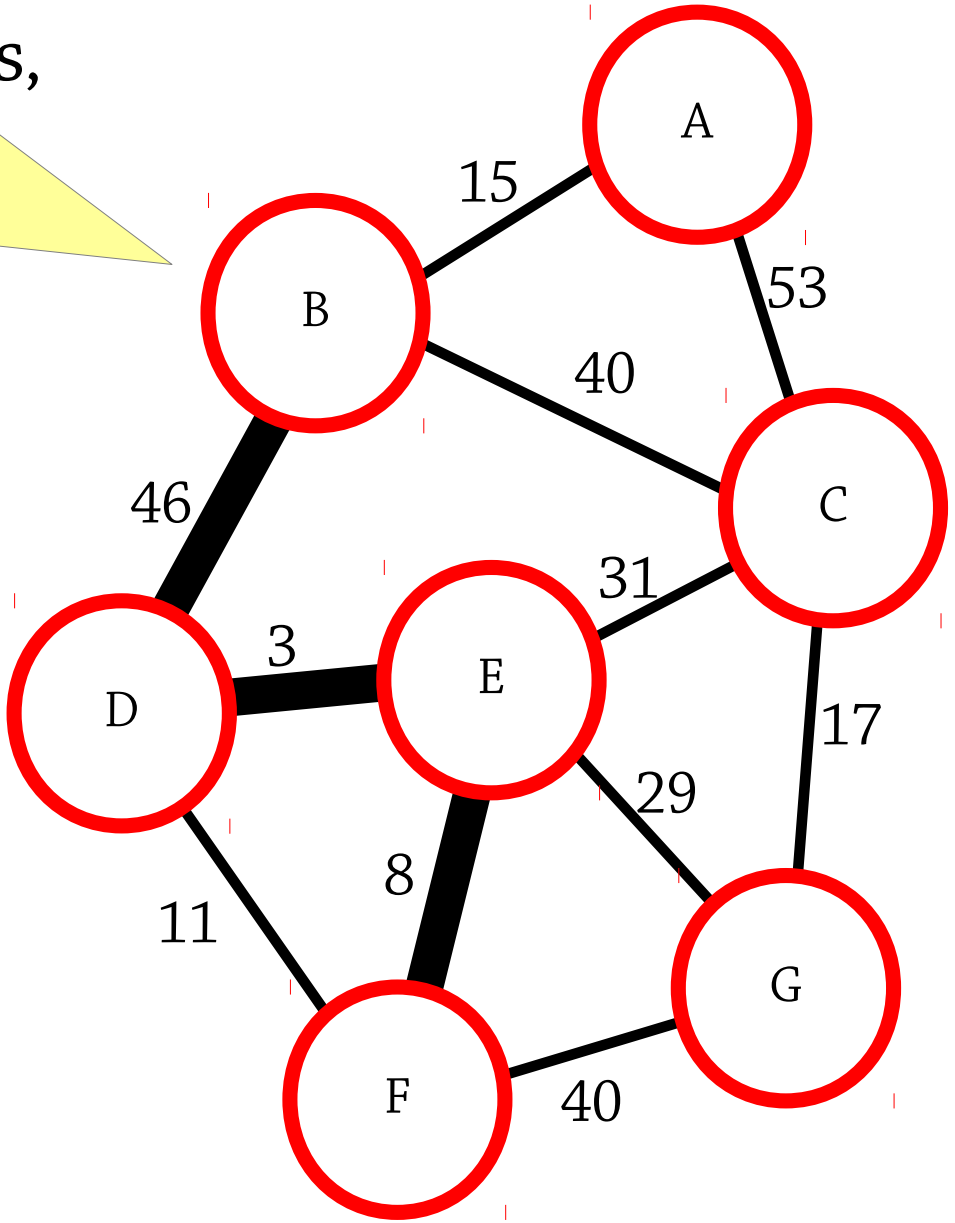
Repeat the process for B

ces,

e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72



algorithm

A → C: **53**
C → B edge: **40**

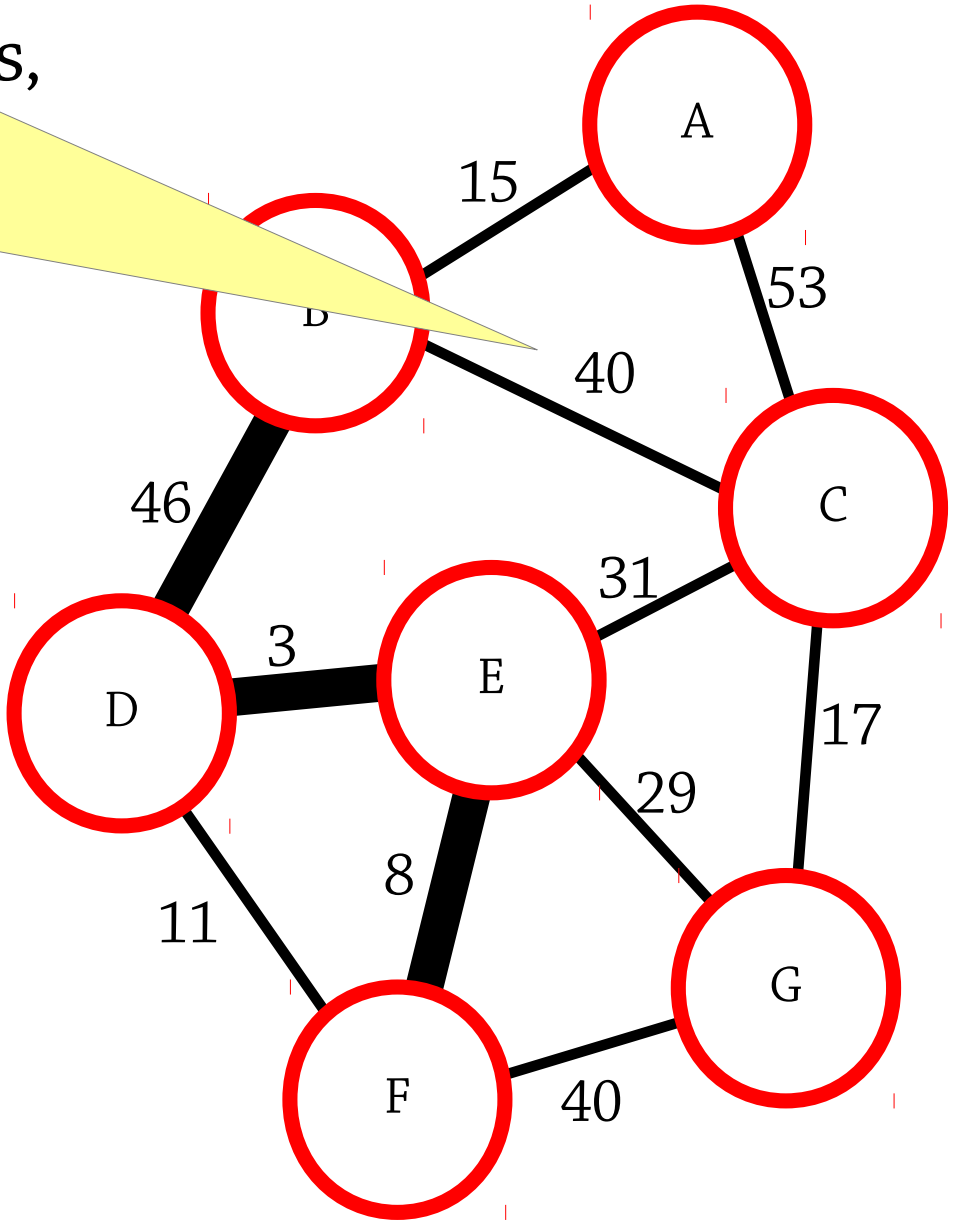
So coming via this edge: **93**
A → B: **15**
This route won't work!

ces,

e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72



algorithm

A → A: **0**
A → B edge: **15**

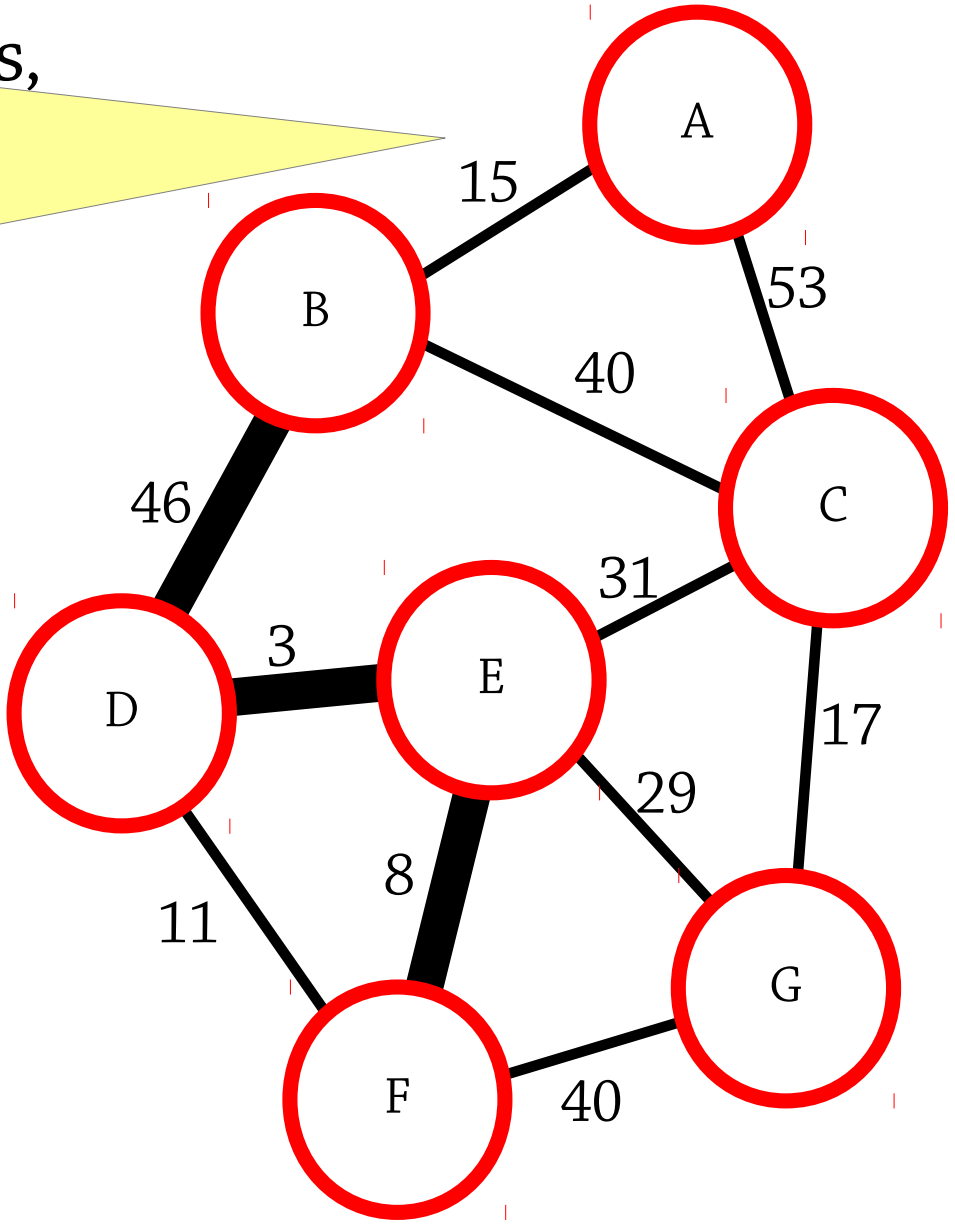
So coming via this edge: **15**
A → B: **15**
This route will work!

ces,

e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72



algorithm

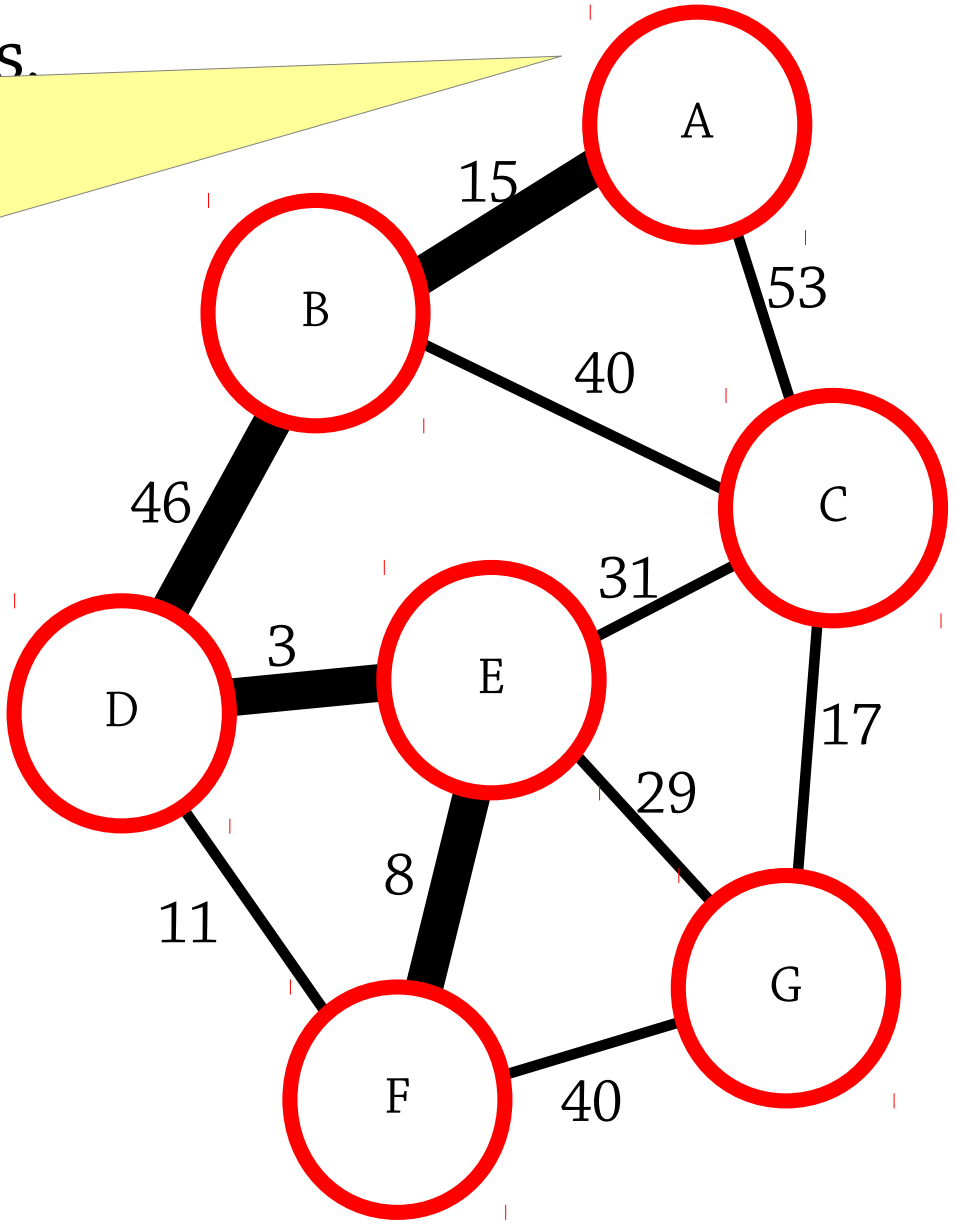
Now we have found our way back to the start node and have the shortest path!

ces.

e.g. take F

Idea: work out which edge we should take on the final leg of the journey

A → 0,
B → 15,
C → 53,
D → 61,
E → 64,
G → 70,
F → 72



Dijkstra's algorithm

Formally, we maintain a set S, which contains all visited nodes and their distances (really a map)

Let S = {start node → 0}
While not all nodes are in S,

- For each node *x* in S, and each neighbour *y* of *x*, calculate
  d = distance to x + cost of edge from x to y
- Find the node *y* which has the smallest value for *d*
- Add that *y* and its distance *d* to S

This computes the shortest distance to each node, from which we can reconstruct the shortest path to any node

What is the efficiency of this algorithm?

ra's algori

Each time through the outer loop, we loop through all edges in S, which by the end contains |E| edges

We add one node to S each time through the loop – loop runs |V| times

n a set S, whic

ances (really a ma

Let S = {star de → 0}

While not all no les are in S

- For each node *x* in S, an each neighbour *y* of *x*, calculate
  d = distance to x + cost of edge from x to y
- Find the node *y* which has the malles or *d*
- Add that *y* and its distance *d* to

This computes the shorte node, from which we can reconst to any node

What is the efficiency of

Total:
O(|V| × |E|)!

Dijkstra's algorithm, made efficient

The algorithm so far is $O(|V| \times |E|)$

This is because this step:

- For all nodes adjacent to a node in S, calculate their distance from the start node, and pick the closest one

takes $O(|E|)$ time, and we execute it once for every node in the graph

How can we make this faster?

Dijkstra's algorithm, made efficient

Answer: use a priority queue!

To find the closest unvisited node, we store all *neighbours* of visited nodes in a priority queue, together with their distances

Instead of searching for the nearest unvisited node, we can just ask the priority queue for the node with the smallest distance

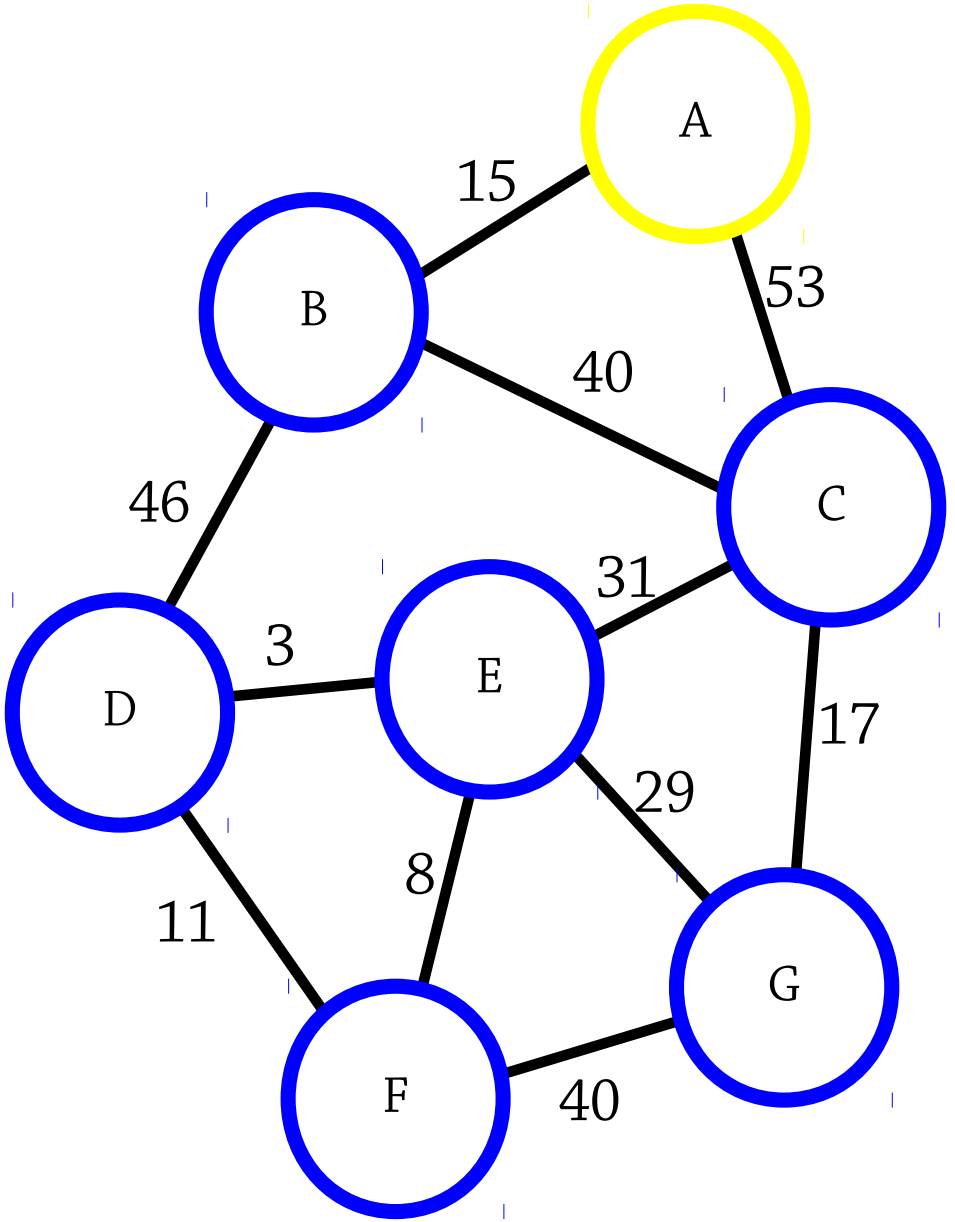
Whenever we visit a node, we will add each of its unvisited neighbours to the priority queue
We can also store the previous node. This allows us to calculating the shortest path by following the references backwards to the start node.

Dijkstra's algorithm

S = {}

Q = {A 0 -},

The distance to A itself is 0 and it has no predecessor.
Remove the smallest element of Q, "A 0 -".
Add it to S,
and add A's neighbours to Q.

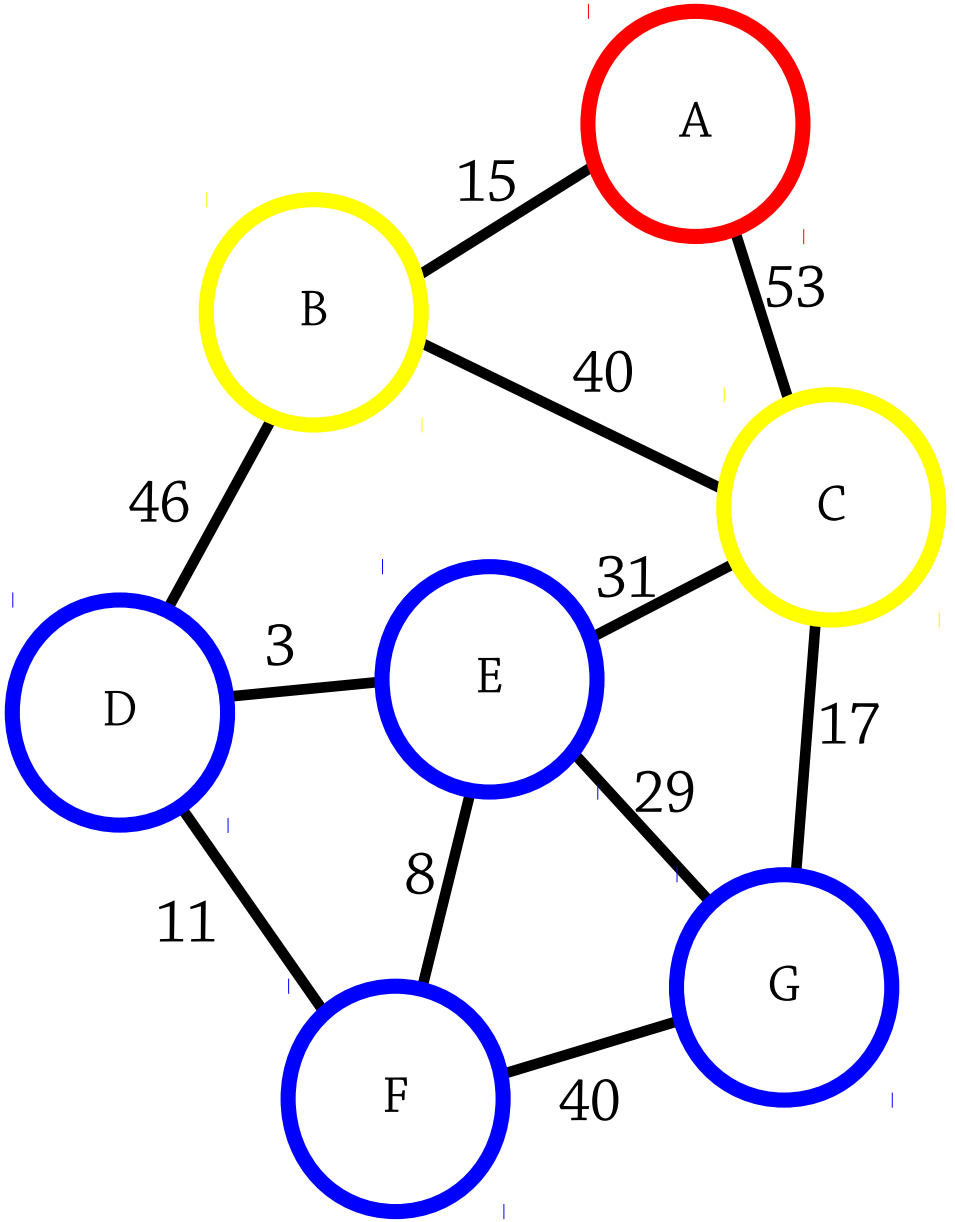


Dijkstra's algorithm

S = {A 0 -}

Q = {B 15 A,
C 53 A}

Remove the smallest element of Q, "B 15 A".
Add it to S, and add B's neighbours to Q.

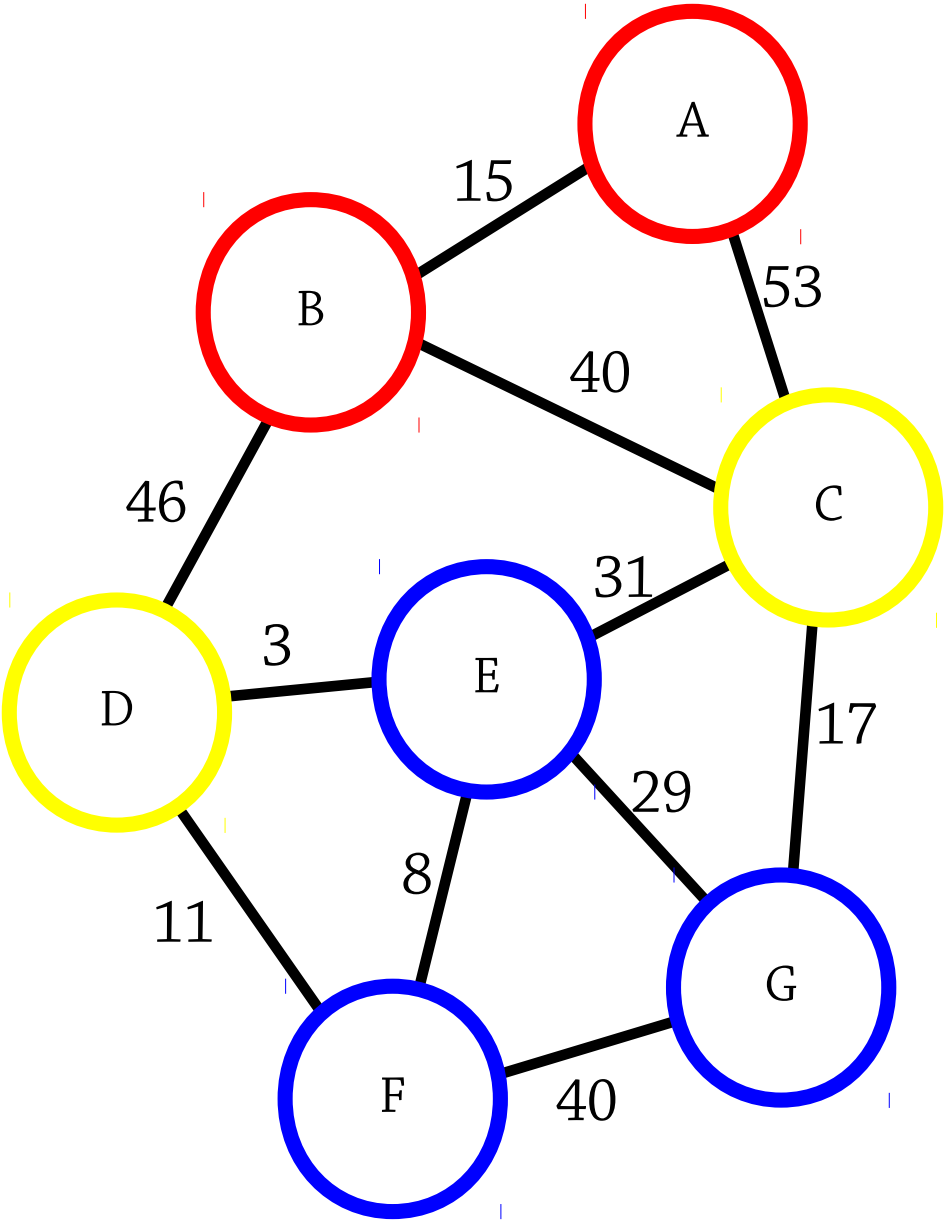


Dijkstra's algorithm

S = {A 0 -,
B 15 A}

Q = {C 53 A,
D 61 B,
C 55 B}

Remove the smallest element of Q, "C 53 A".
Add it to S, and add C's neighbours to Q.

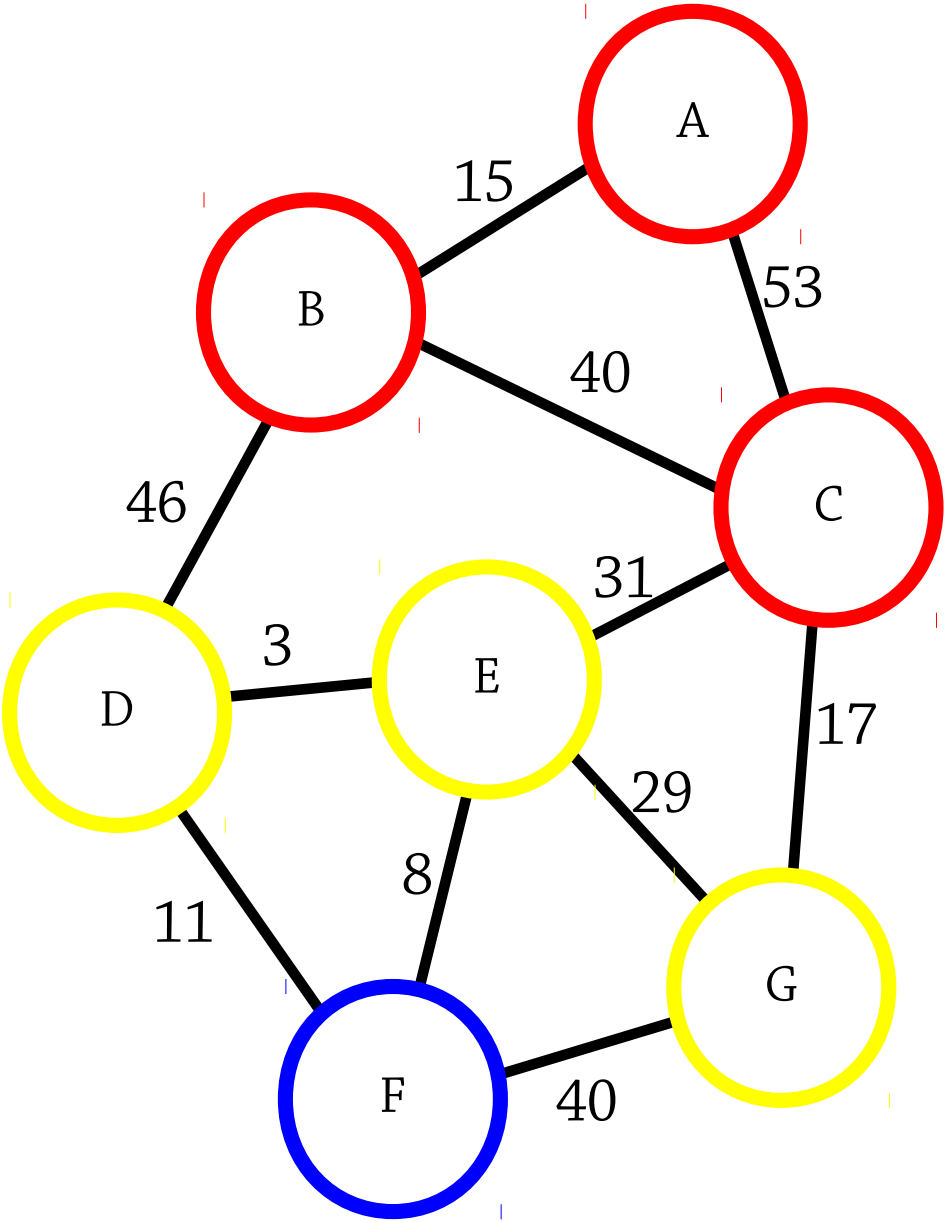


Dijkstra's algorithm

S = {A 0 -,
B 15 A,
C 53 A}

Q = {D 61 B,
C 55 B,
E 84 C,
G 70 C}

Remove the smallest element of Q, “C 55 B”.
Oh! C is already in S.
So just ignore it.

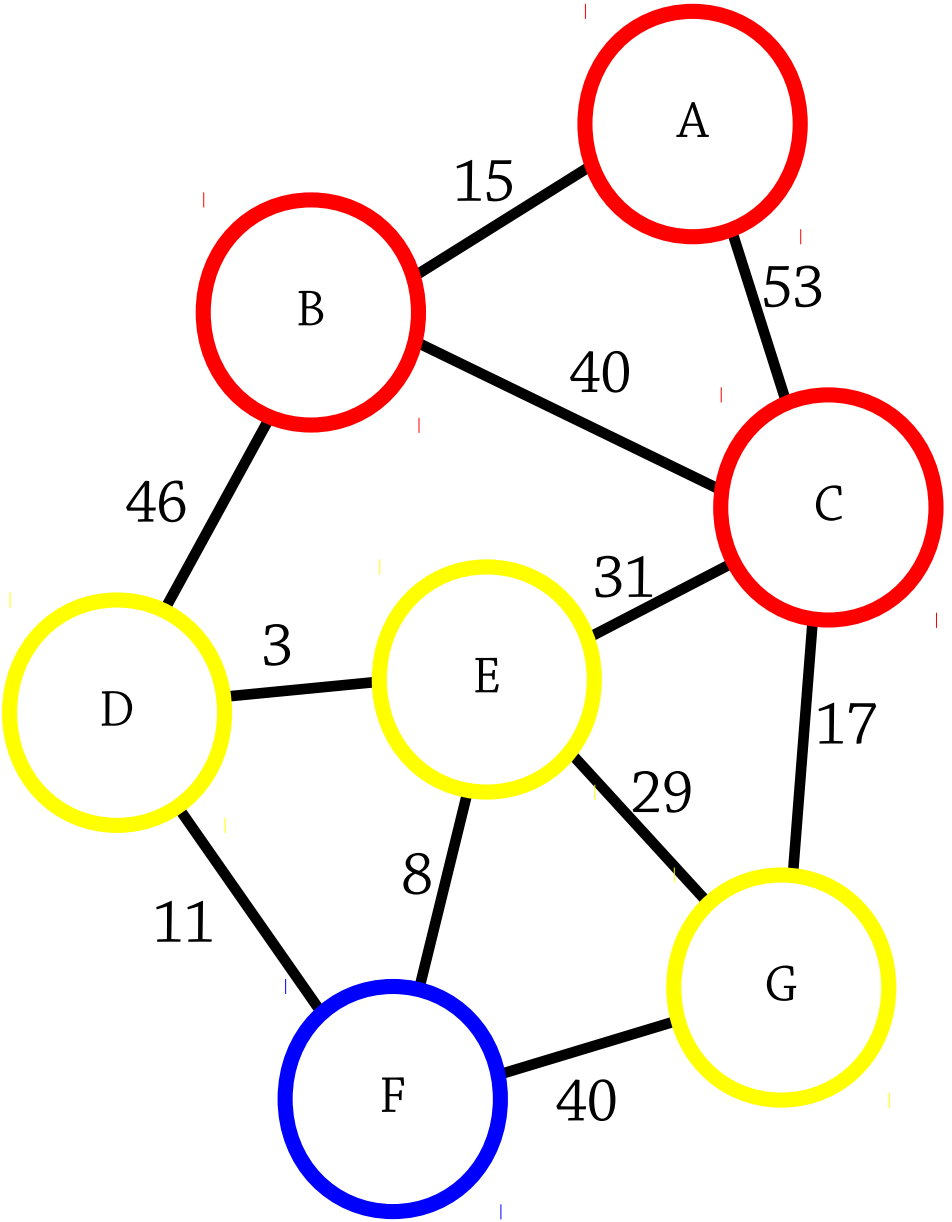


Dijkstra's algorithm

S = {A 0 -,
B 15 A,
C 53 A}

Q = {D 61 B,
E 84 C,
G 70 C}

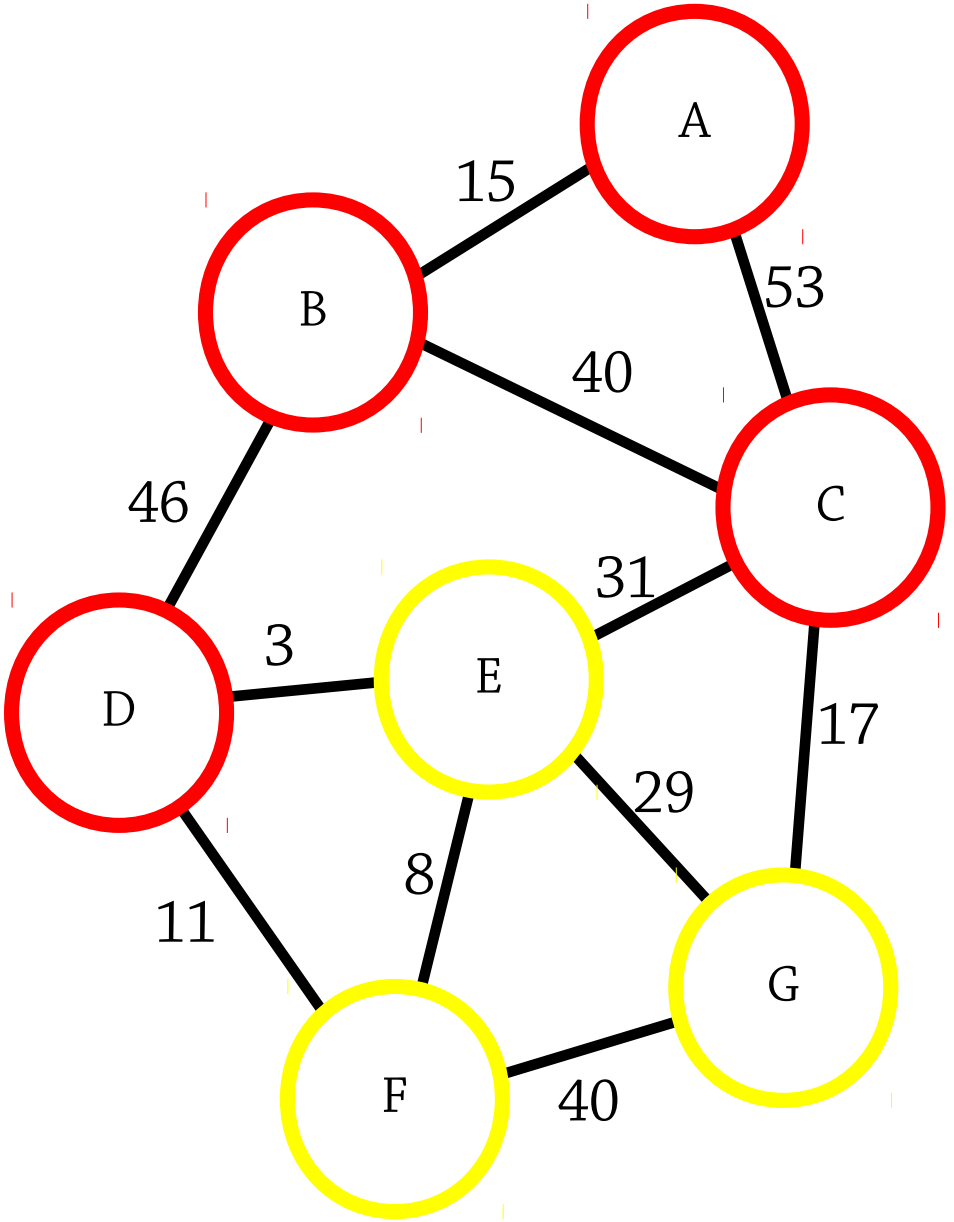
Remove the smallest element of Q, "D 61 B".
Add it to S,
and add D's neighbours to Q.



Dijkstra's algorithm

S = {A 0 -,
B 15 A,
C 53 A,
D 61 B}

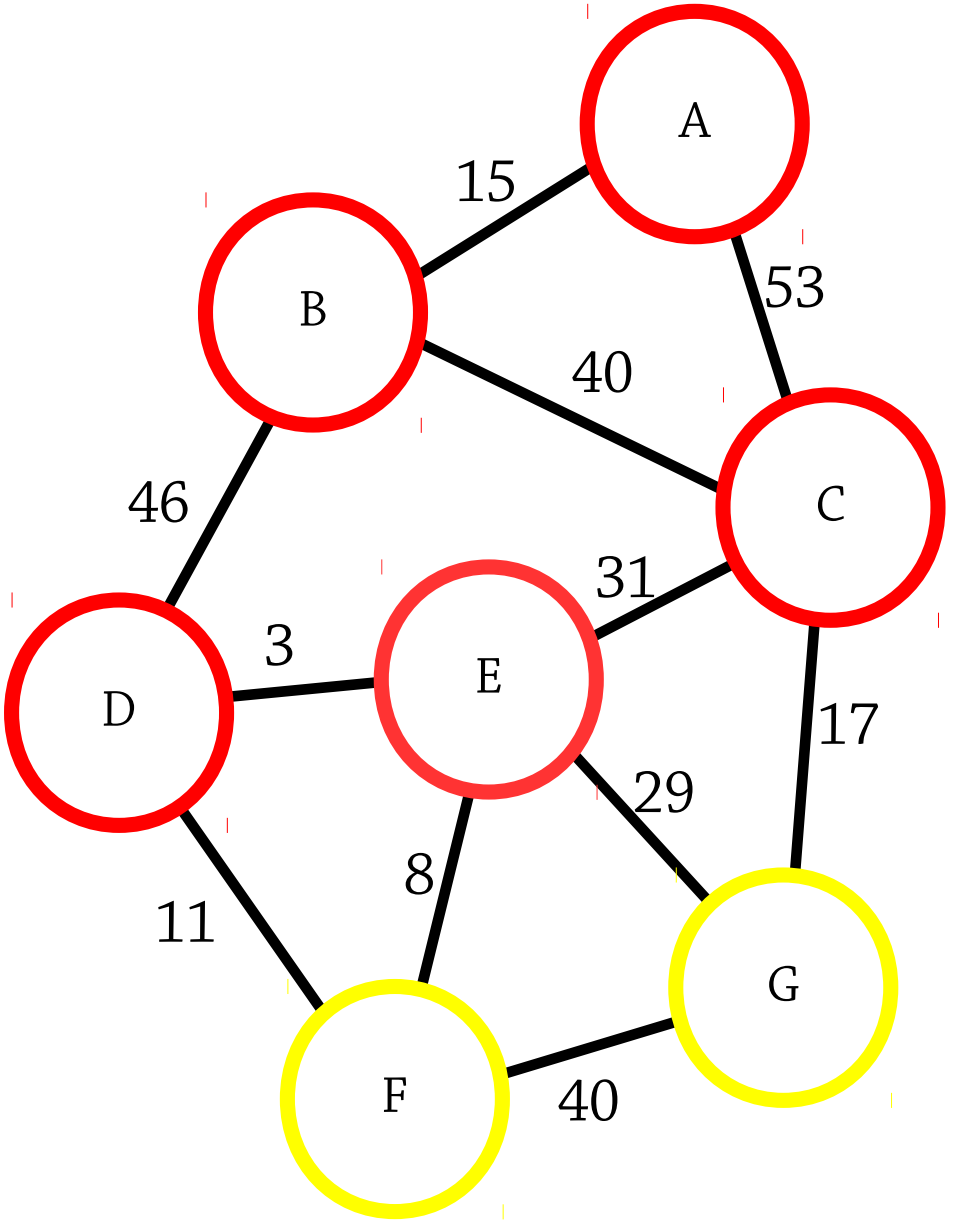
Q = {E 84 C,
G 70 C,
E 64 D,
F 72 D}



Dijkstra's algorithm

S = {A 0 -,
B 15 A,
C 53 A,
D 61 B,
E 64 D}

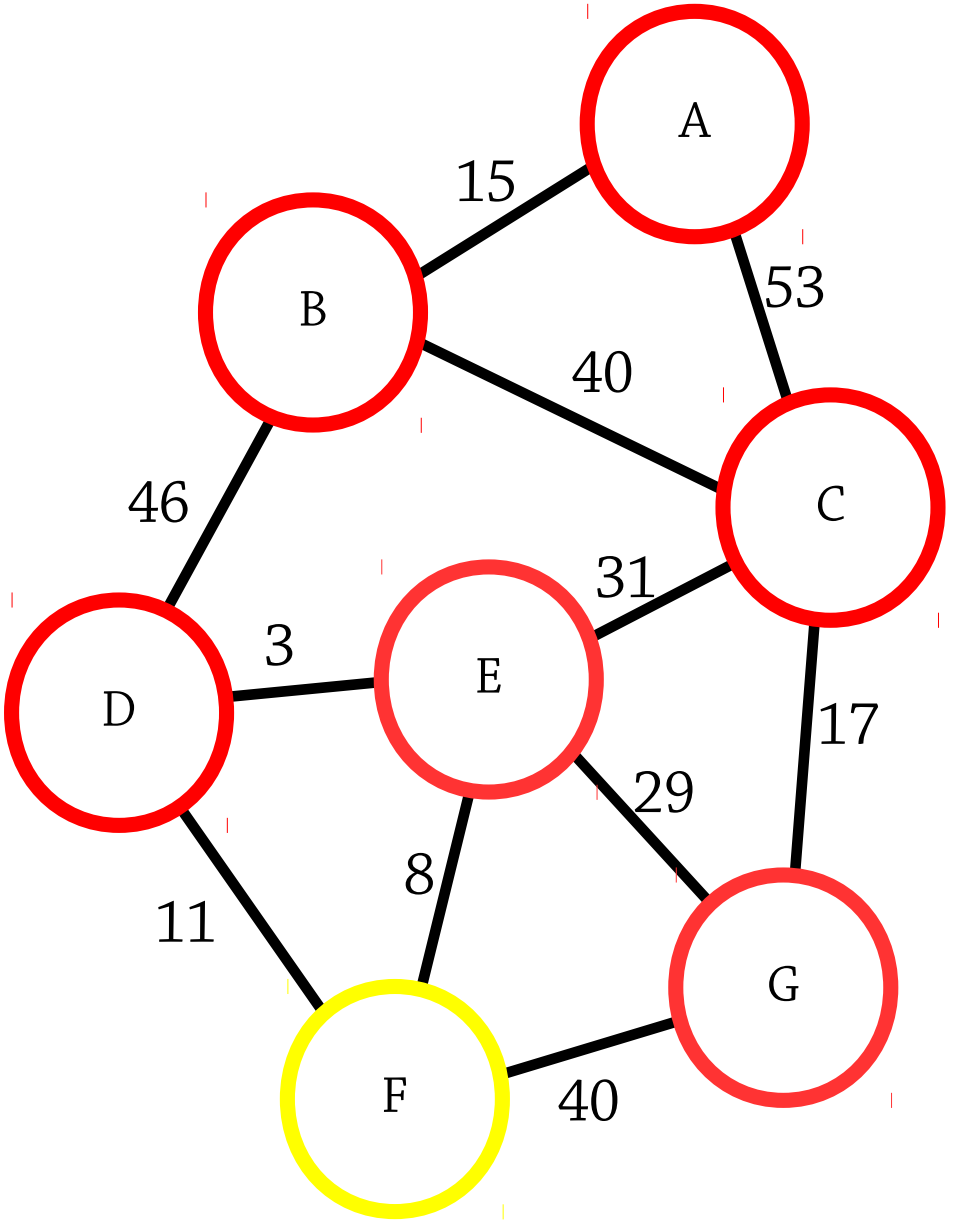
Q = {E 84 C,
G 70 C,
F 72 D,
F 72 E,
G 93 E}



Dijkstra's algorithm

S = {A 0 -,
B 15 A,
C 53 A,
D 61 B,
E 64 D,
G 70 C}

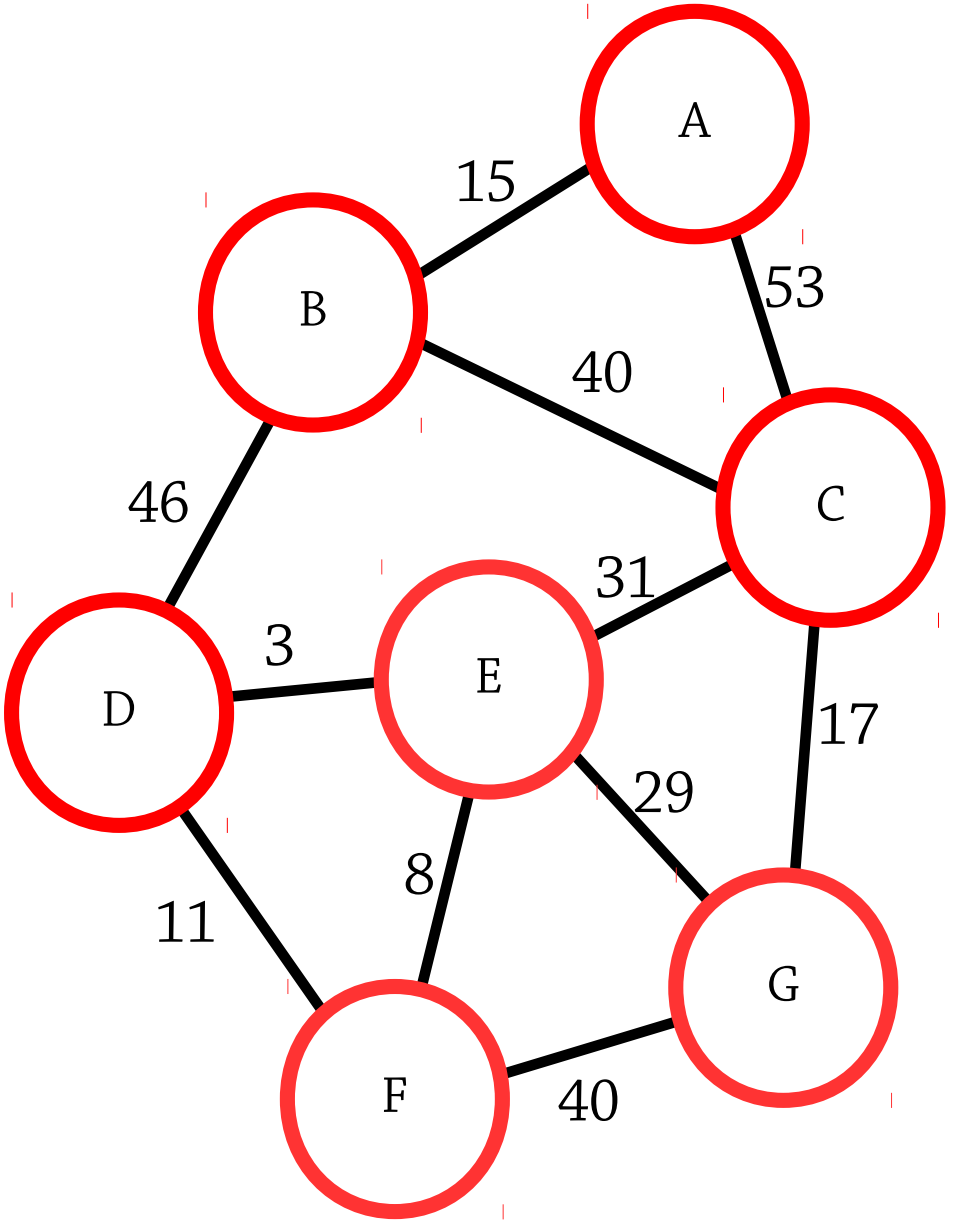
Q = {E 84 C,
F 72 D,
F 72 E,
G 93 E,
F 110 G}



Dijkstra's algorithm

S = {A 0 -,
B 15 A,
C 53 A,
D 61 B,
E 64 D,
G 70 C,
F 72 D}

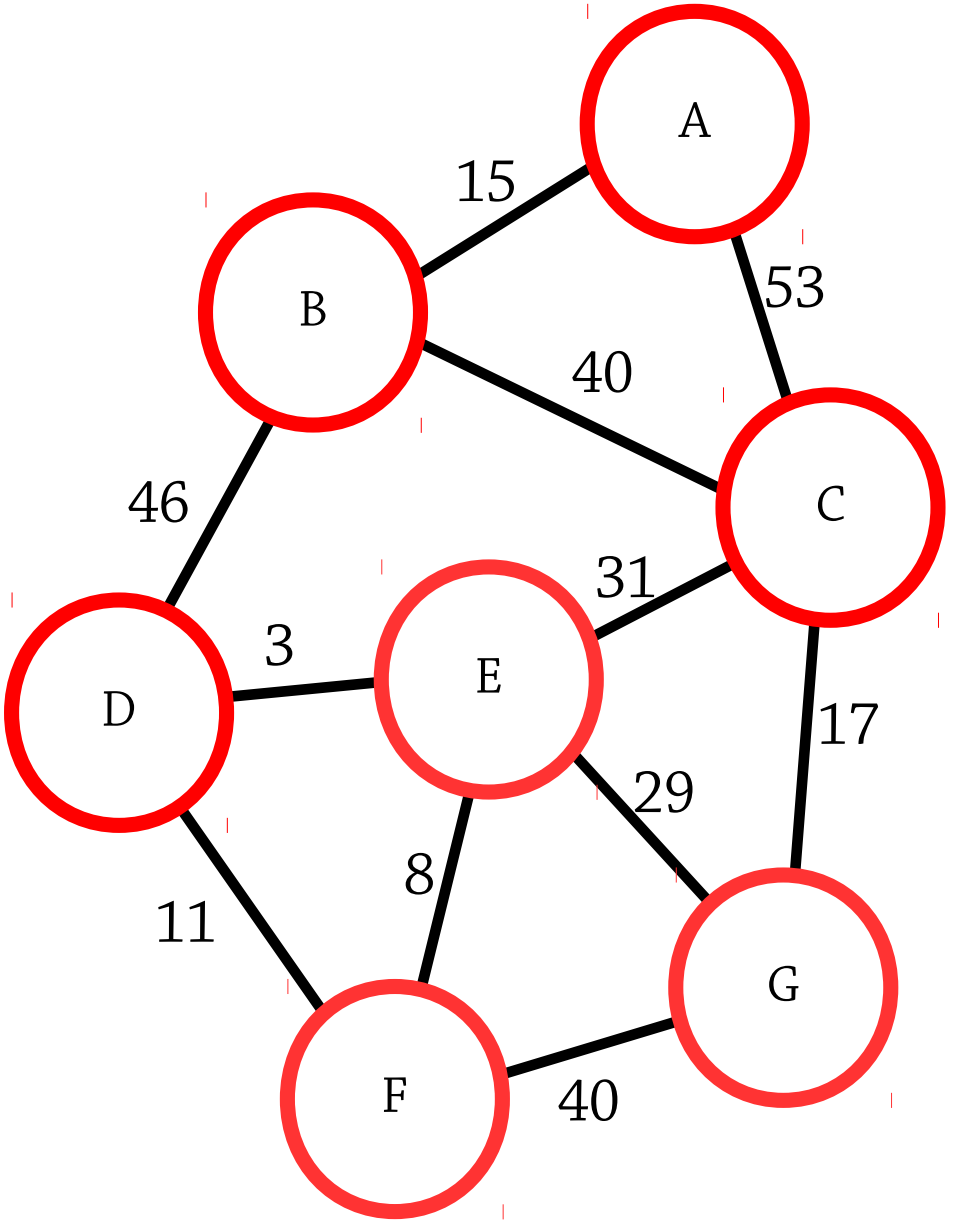
Q = {E 84 C,
F 72 E,
G 93 E,
F 110 G}



Dijkstra's algorithm

S = {A 0 -,
B 15 A,
C 53 A,
D 61 B,
E 64 D,
G 70 C,
F 72 D}

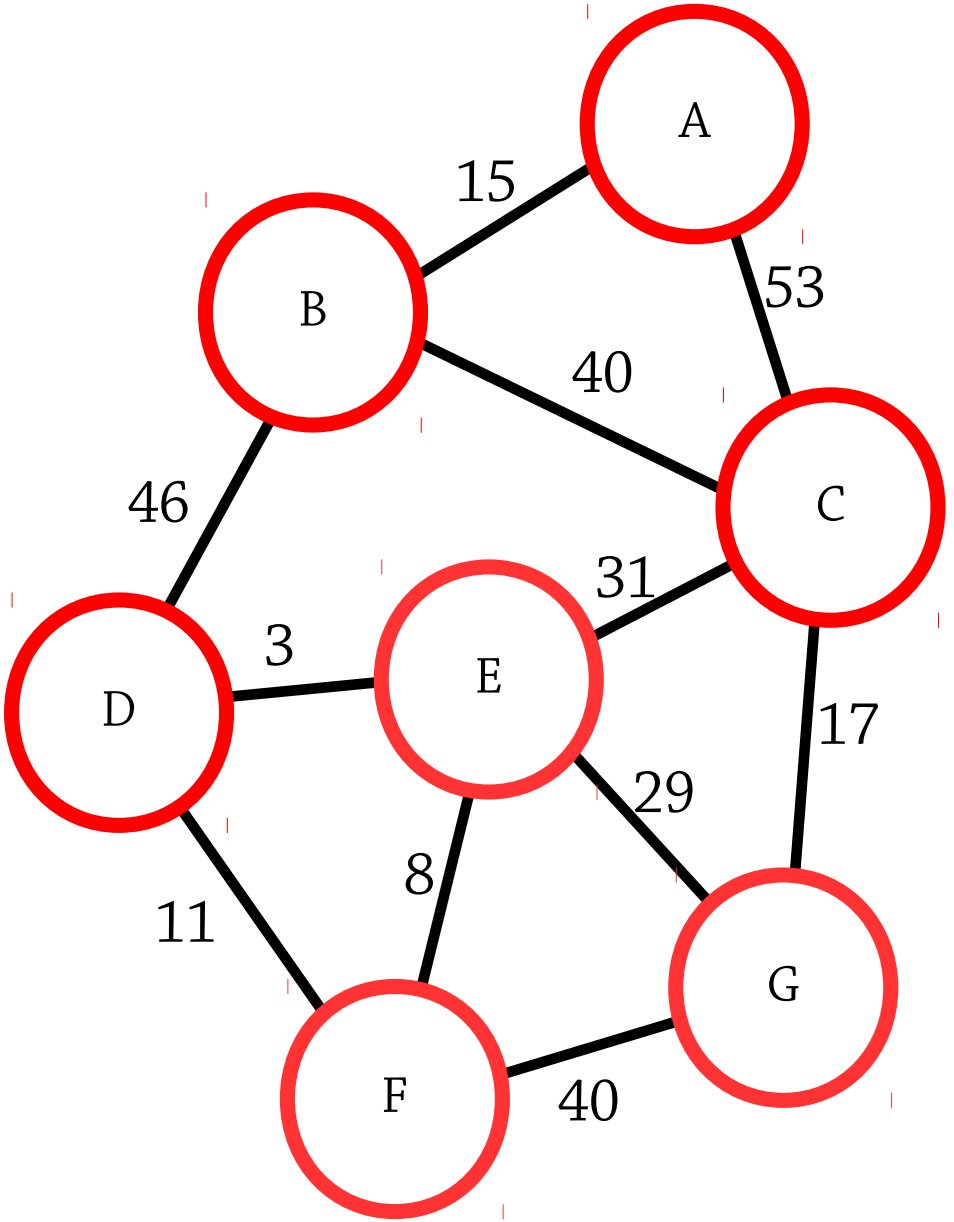
Q = {E 84 C,
G 93 E,
F 110 G}



Dijkstra's algorithm

S = {A 0 -,
B 15 A,
C 53 A,
D 61 B,
E 64 D,
G 70 C,
F 72 D}

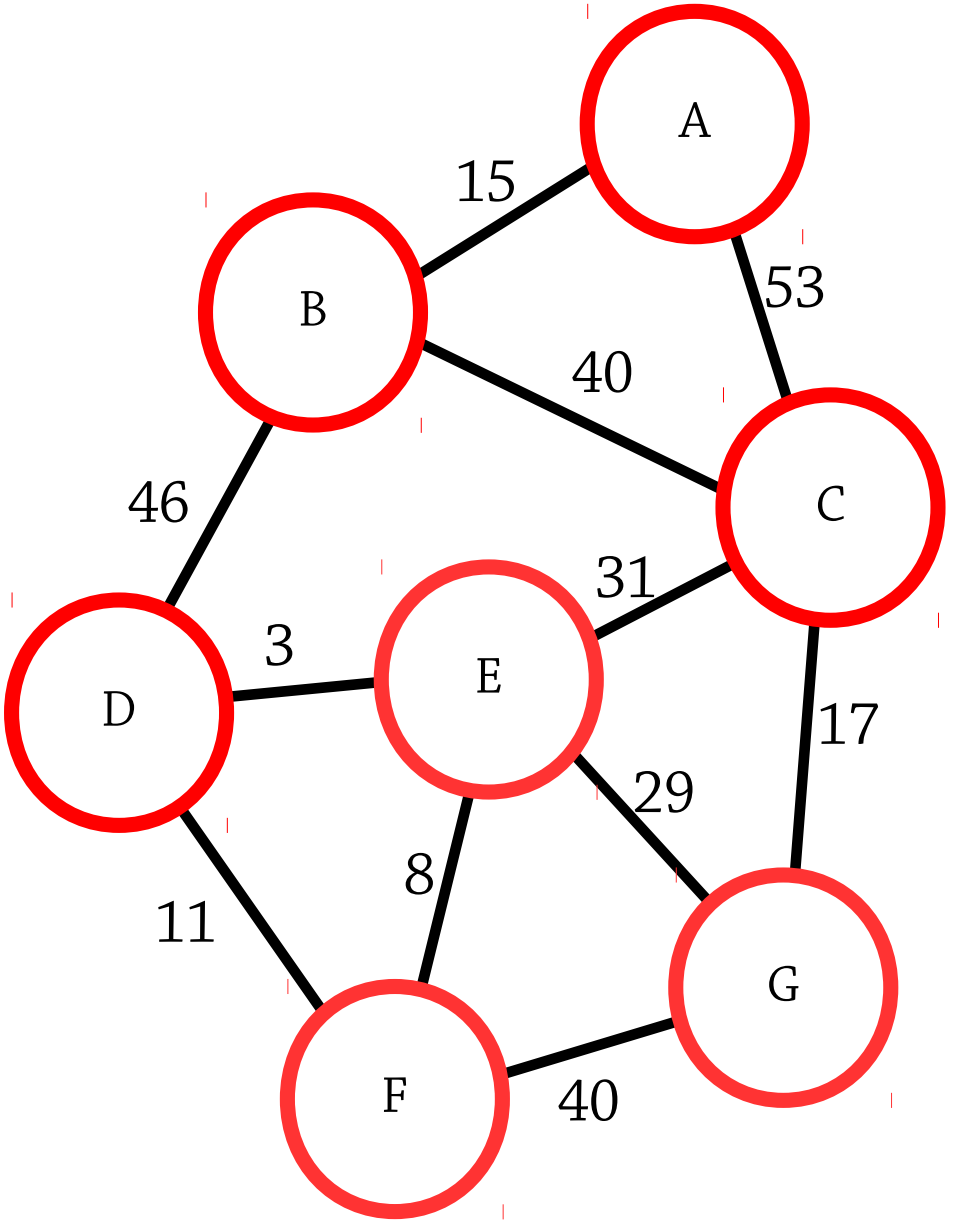
Q = {G 93 E,
F 110 G}



Dijkstra's algorithm

S = {A 0 -,
B 15 A,
C 53 A,
D 61 B,
E 64 D,
G 70 C,
F 72 D}

Q = {F 110 G}

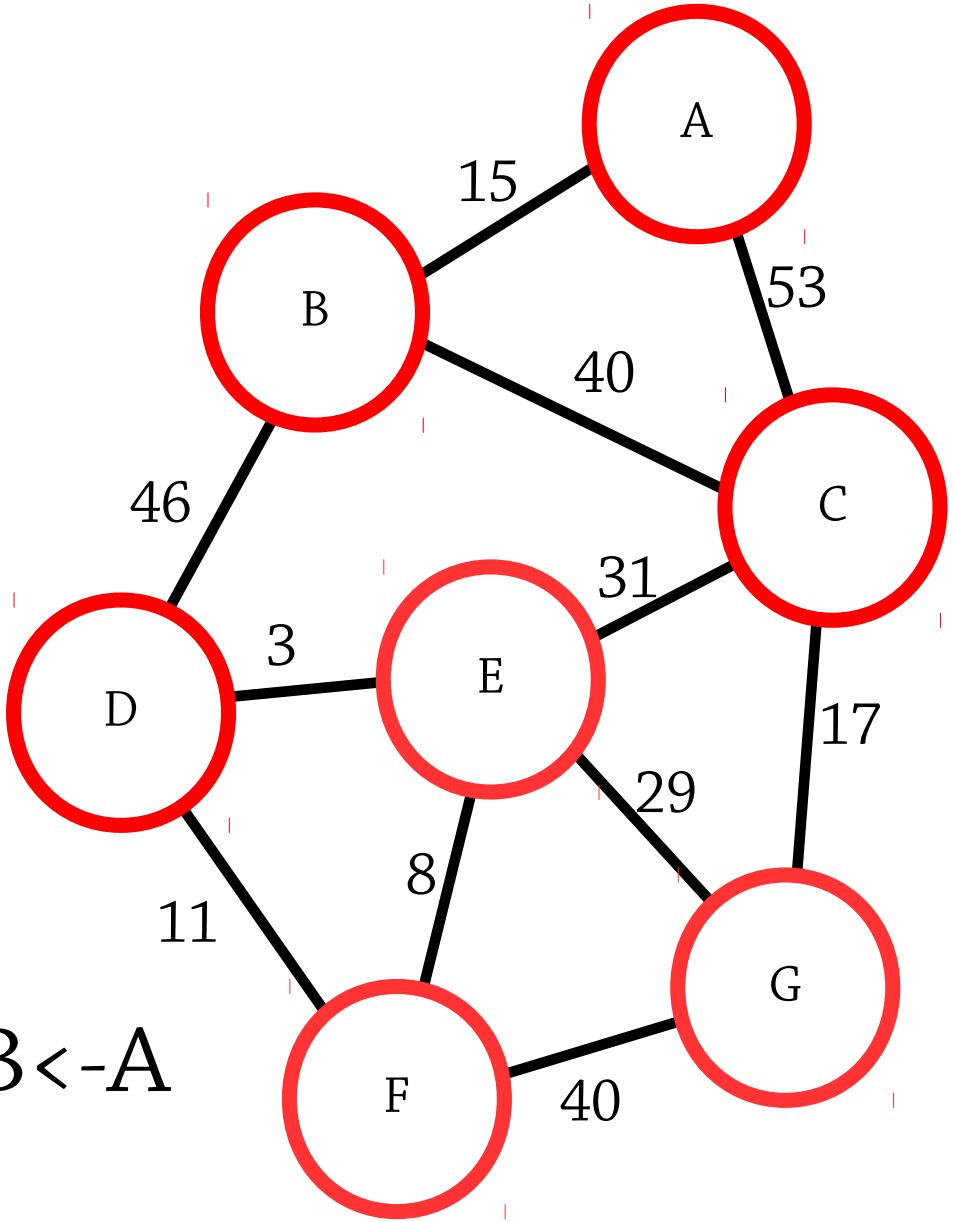


Dijkstra's algorithm

S = {A 0 -,
B 15 A,
C 53 A,
D 61 B,
E 64 D,
G 70 C,
F 72 D}

Q = {}

To get to e.g. to F we follow the backwards references: F<-D, D<-B, B<-A



Dijkstra's alg

Let S = {} and Q = {start node, 0}

While Q is not empty:

- Remove the entry (*x d z*) from Q that has the smallest priority (distance) *d*. *z* is the node's predecessor.
- If *x* is in S, do nothing
- Otherwise, add (*x d z*) to S and for each outgoing edge $x \rightarrow y$, add (*y* (*d* + *w*) *x*) to Q, where *w* is the weight of the edge

- The time complexity is $O(|E| \log |E|)$

The whole loop is repeated at most $|E|$ times.

The size of the pq is at most $|E|$ so this takes $O(\log |E|)$

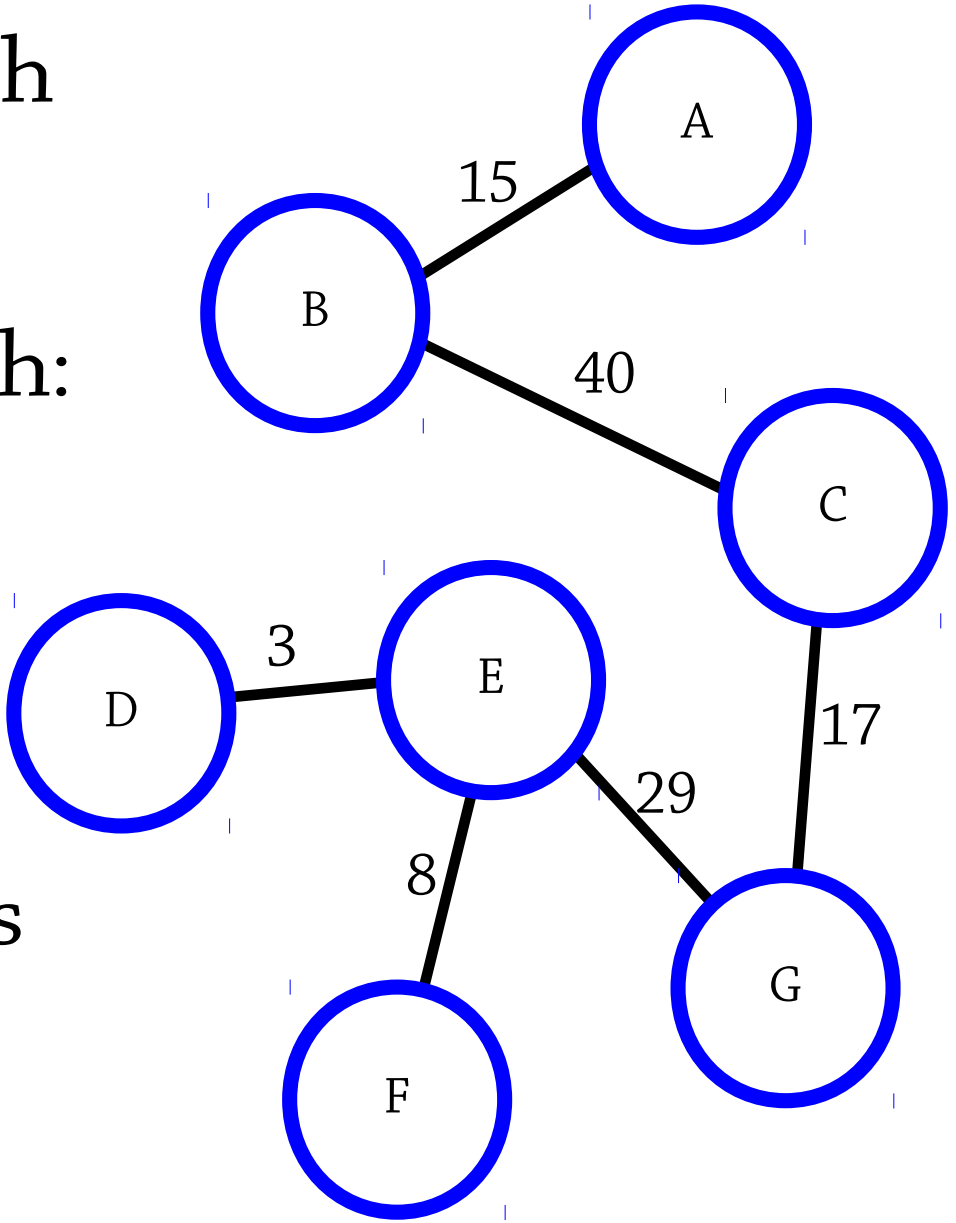
This is repeated at most $|E|$ times in total (i.e. for all iterations of the outer loop.) Each iteration is $O(\log |E|)$

Minimum spanning trees

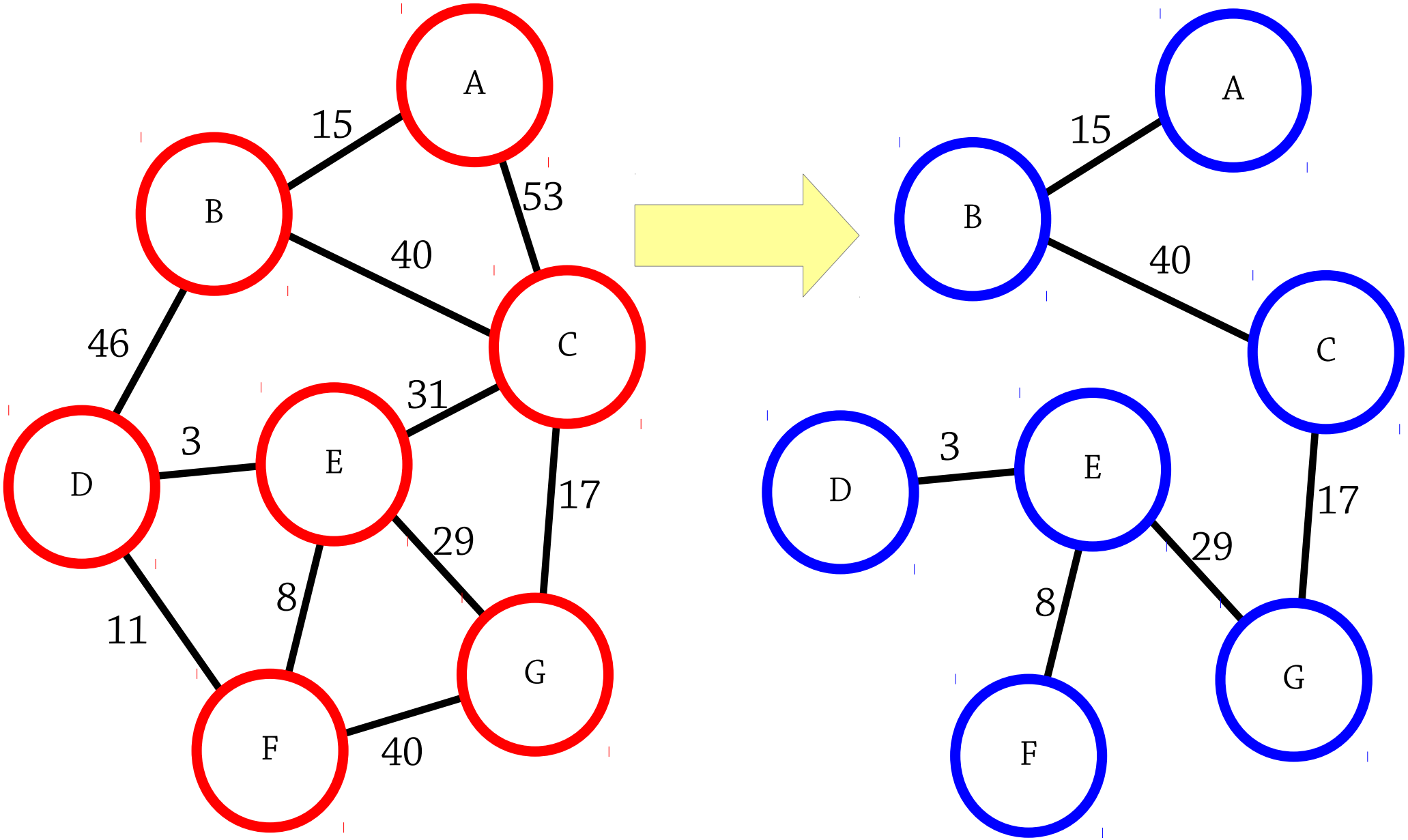
A *spanning tree* of a graph is a subgraph (a graph obtained by deleting some of the edges) which:

- is acyclic
- is connected

A *minimum* spanning tree is one where the total weight of the edges is as low as possible



Minimum spanning trees



Prim's algorithm

We will build a minimum spanning tree by starting with no edges and adding edges until the graph is connected

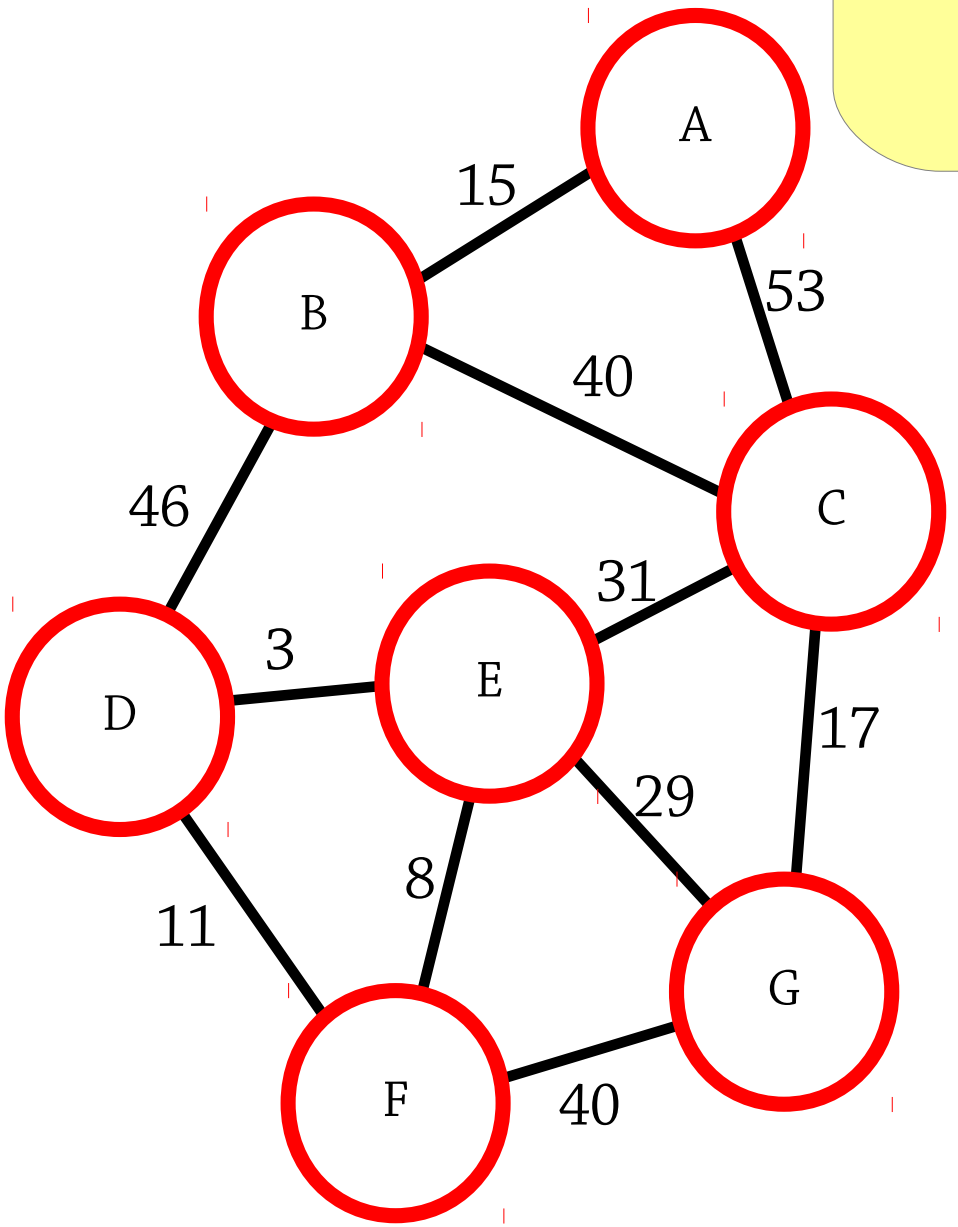
Keep a set S of all the nodes that are in the tree so far, initially containing one arbitrary node

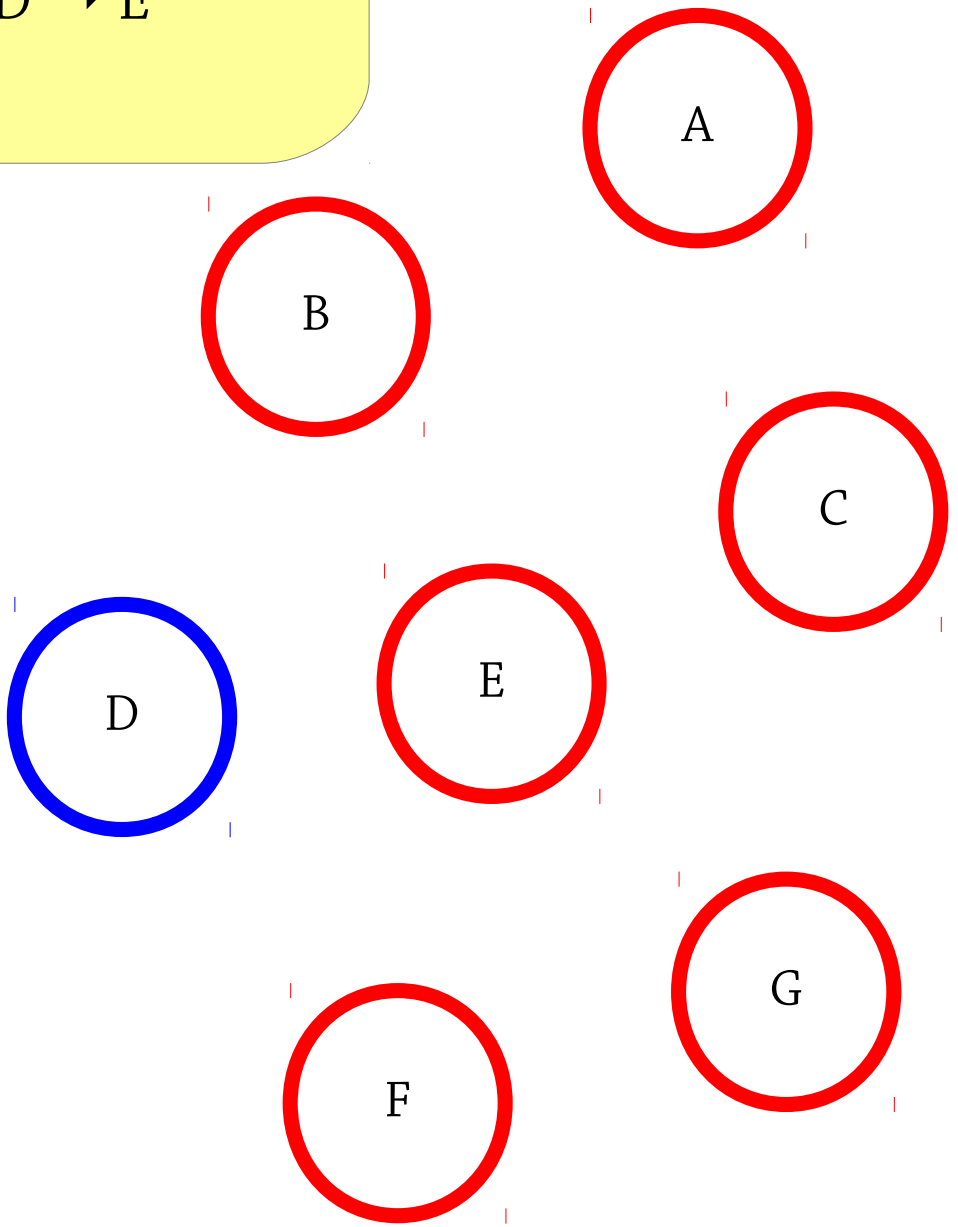
While there is a node not in S:

- Pick the *lowest-weight* edge between a node in S and a node not in S
- Add that edge to the spanning tree, and add the node to S

Minimun ees

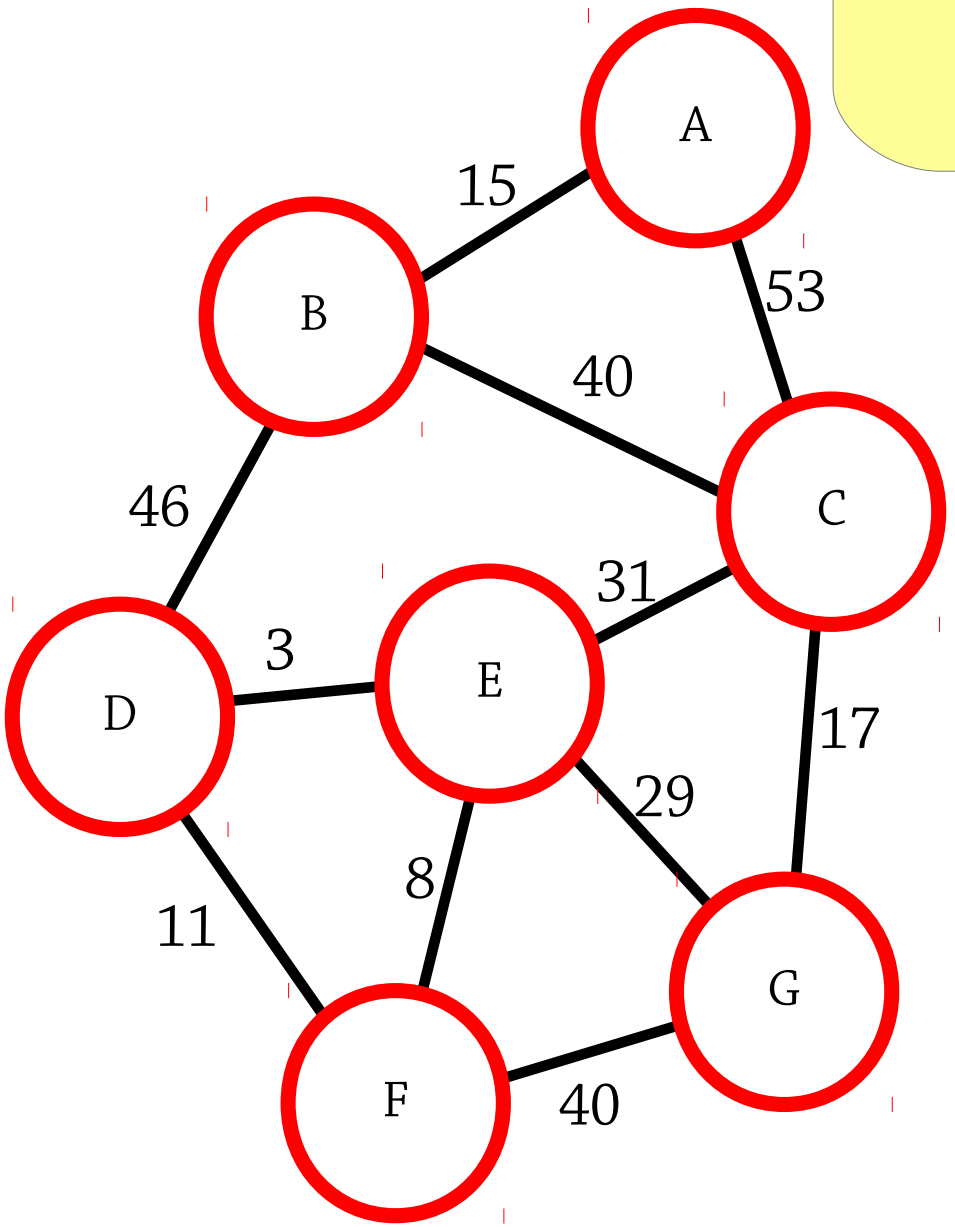
S = {D}
Lowest-weight edge
from S to not-S
is D → E

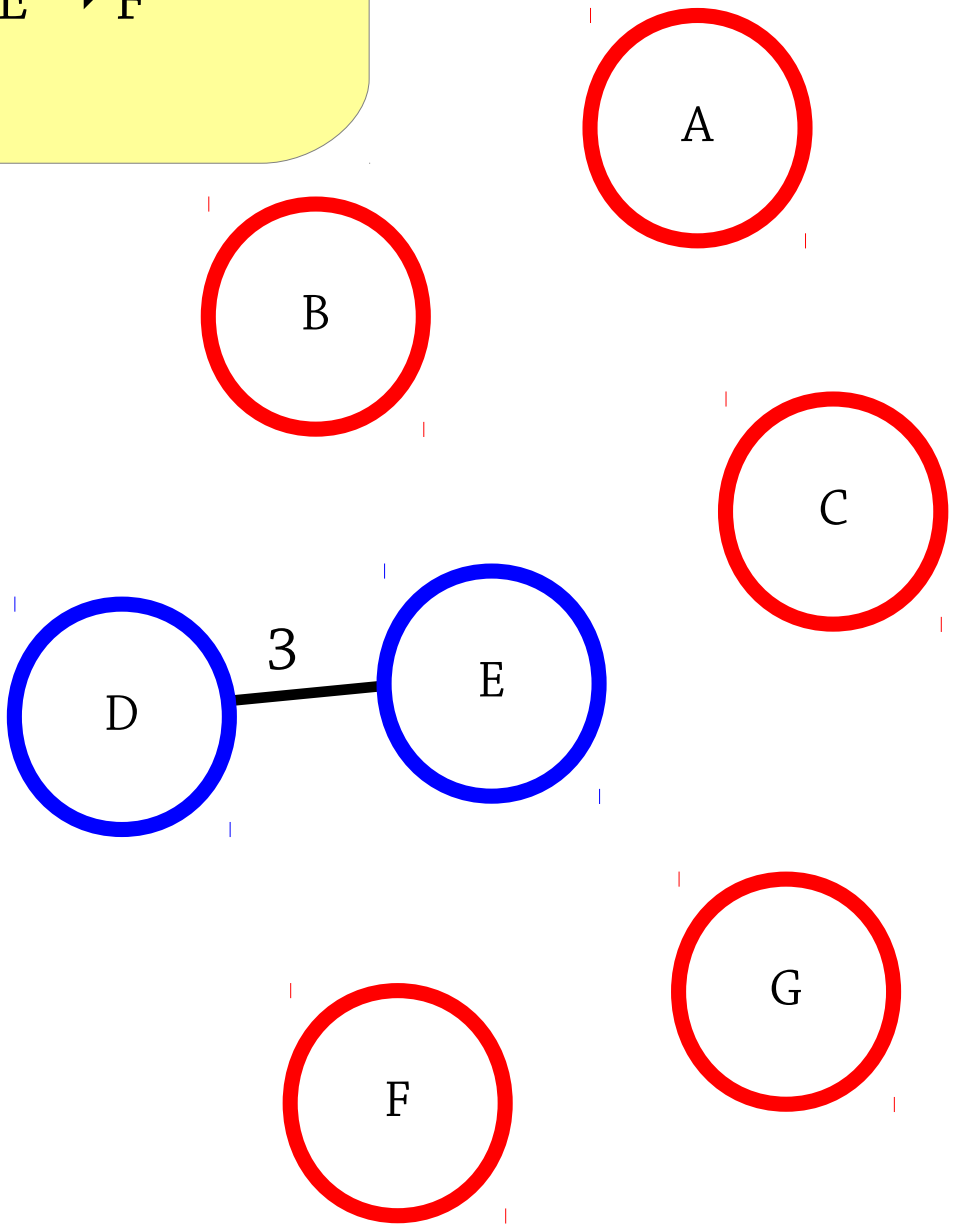




Minimun ees

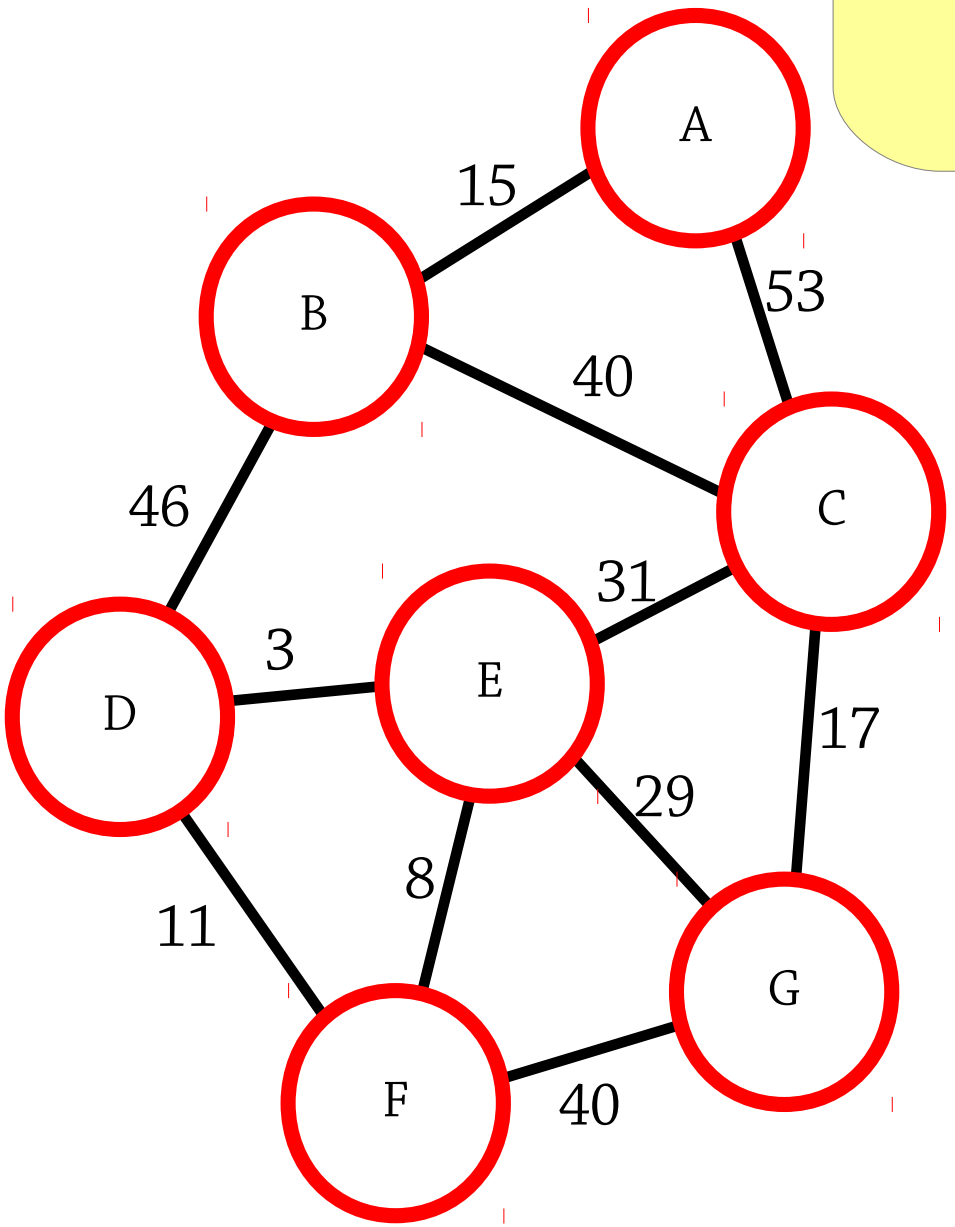
S = {D, E}
Lowest-weight edge
from S to not-S
is E → F

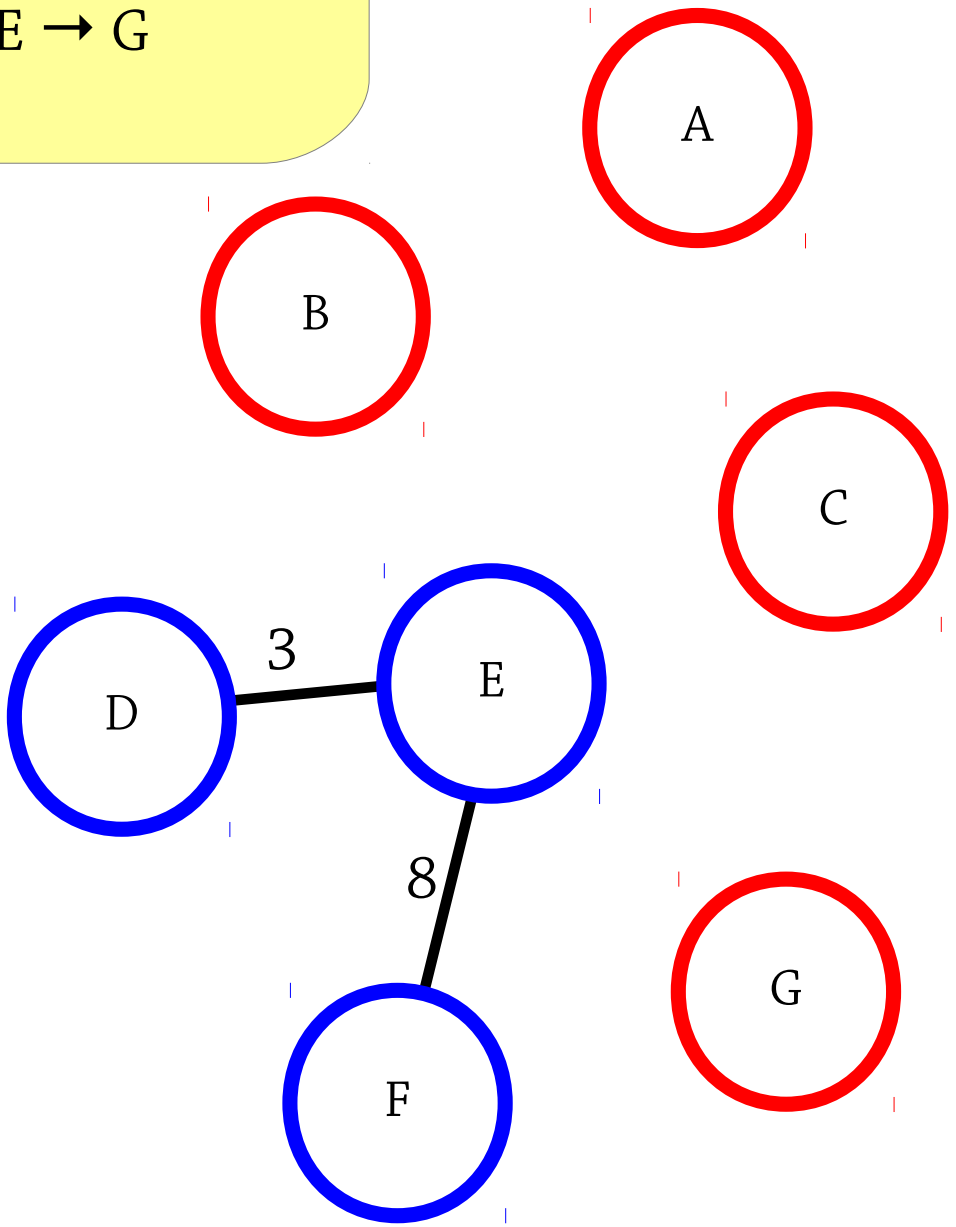




Minimun ees

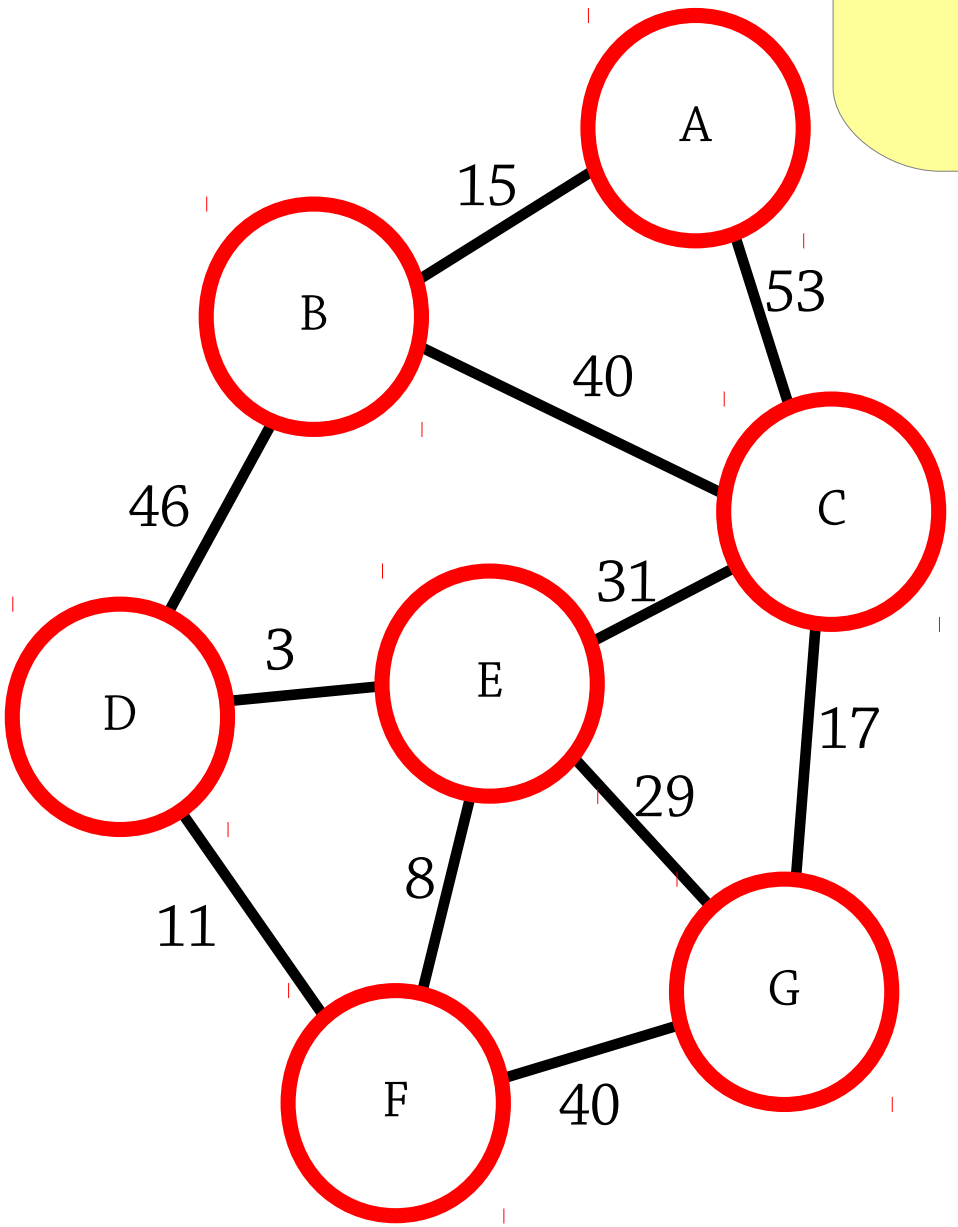
S = {D, E, F}
Lowest-weight edge from S to not-S is E → G

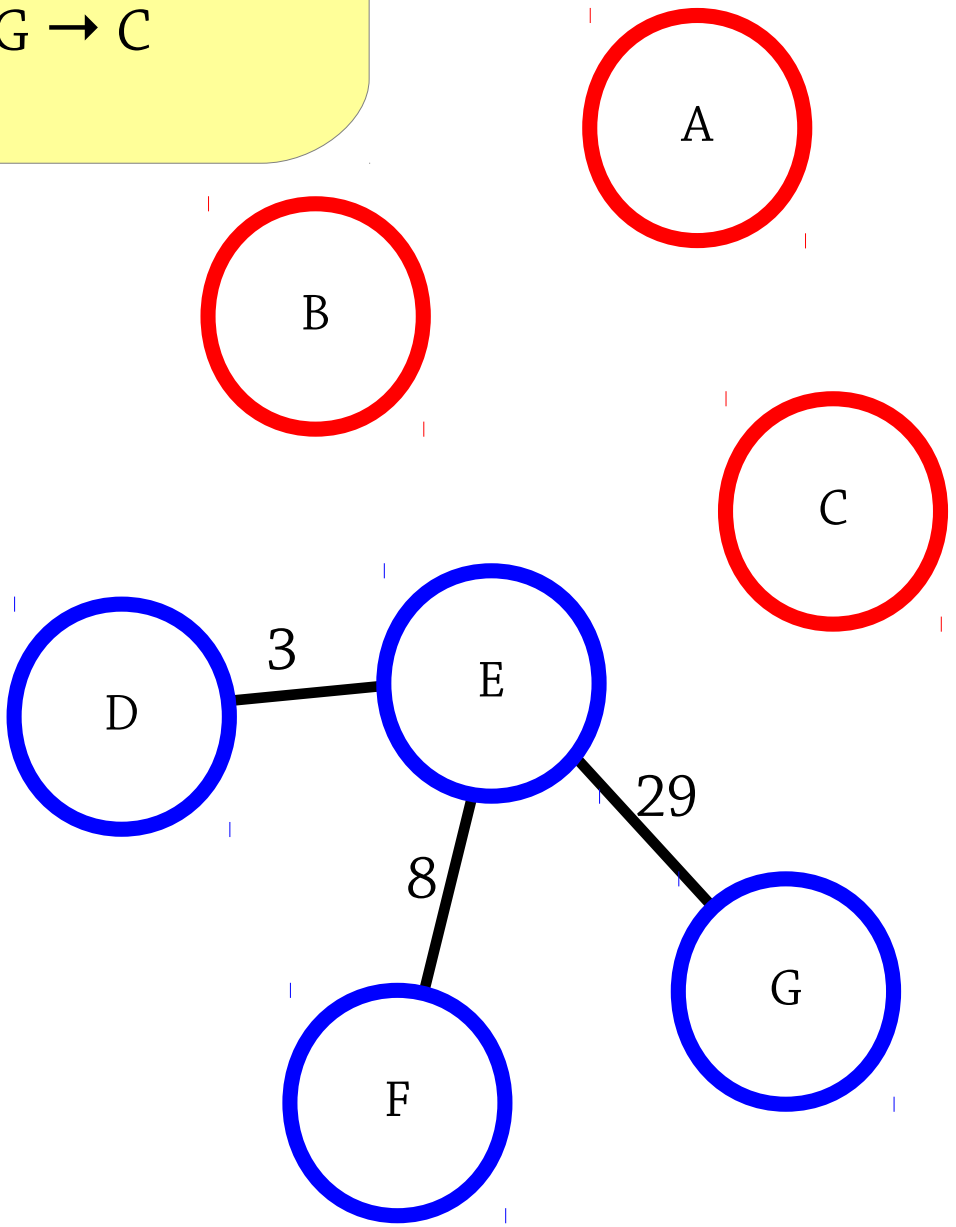




Minimun ees

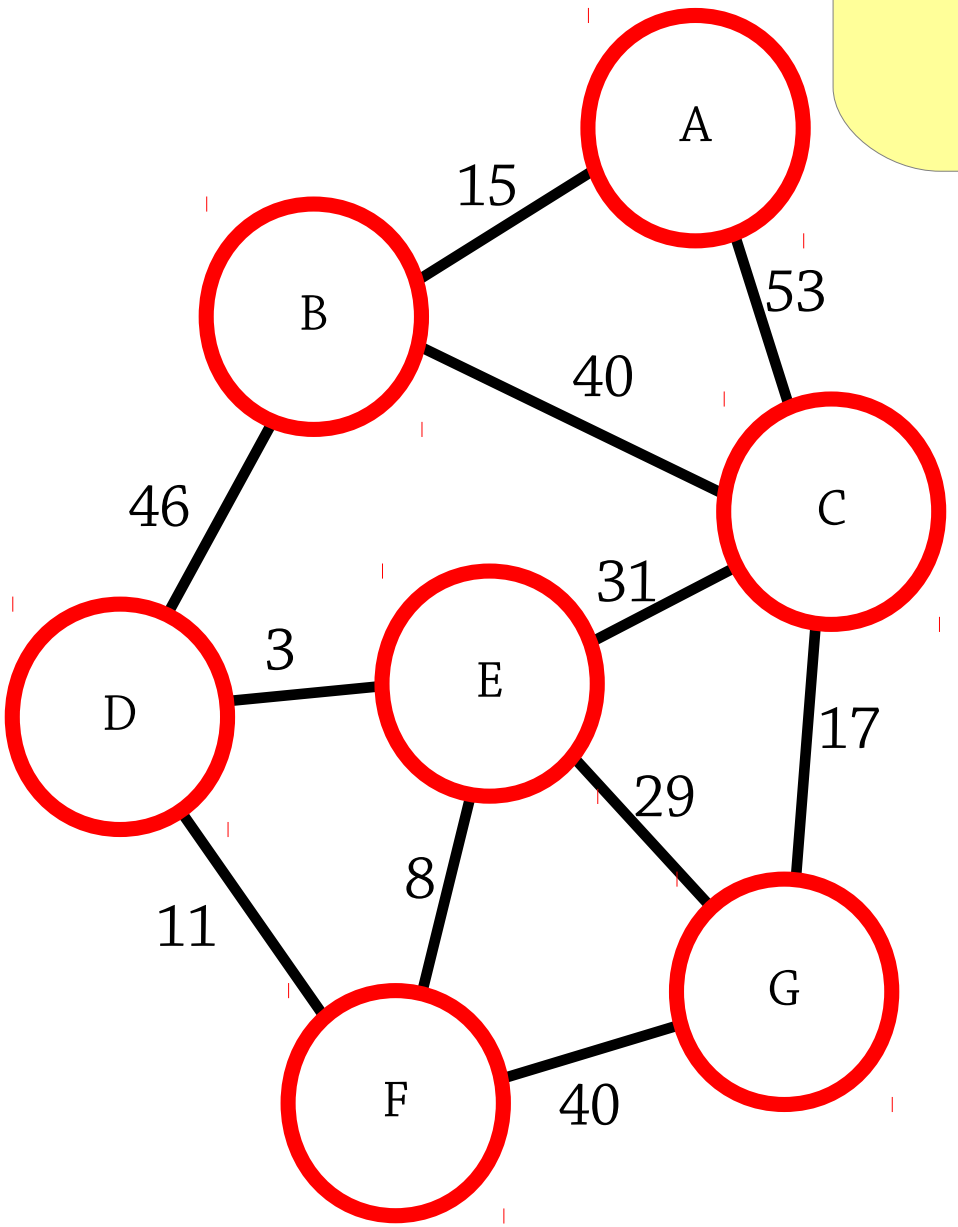
S = {D, E, F, G}
Lowest-weight edge from S to not-S is G → C

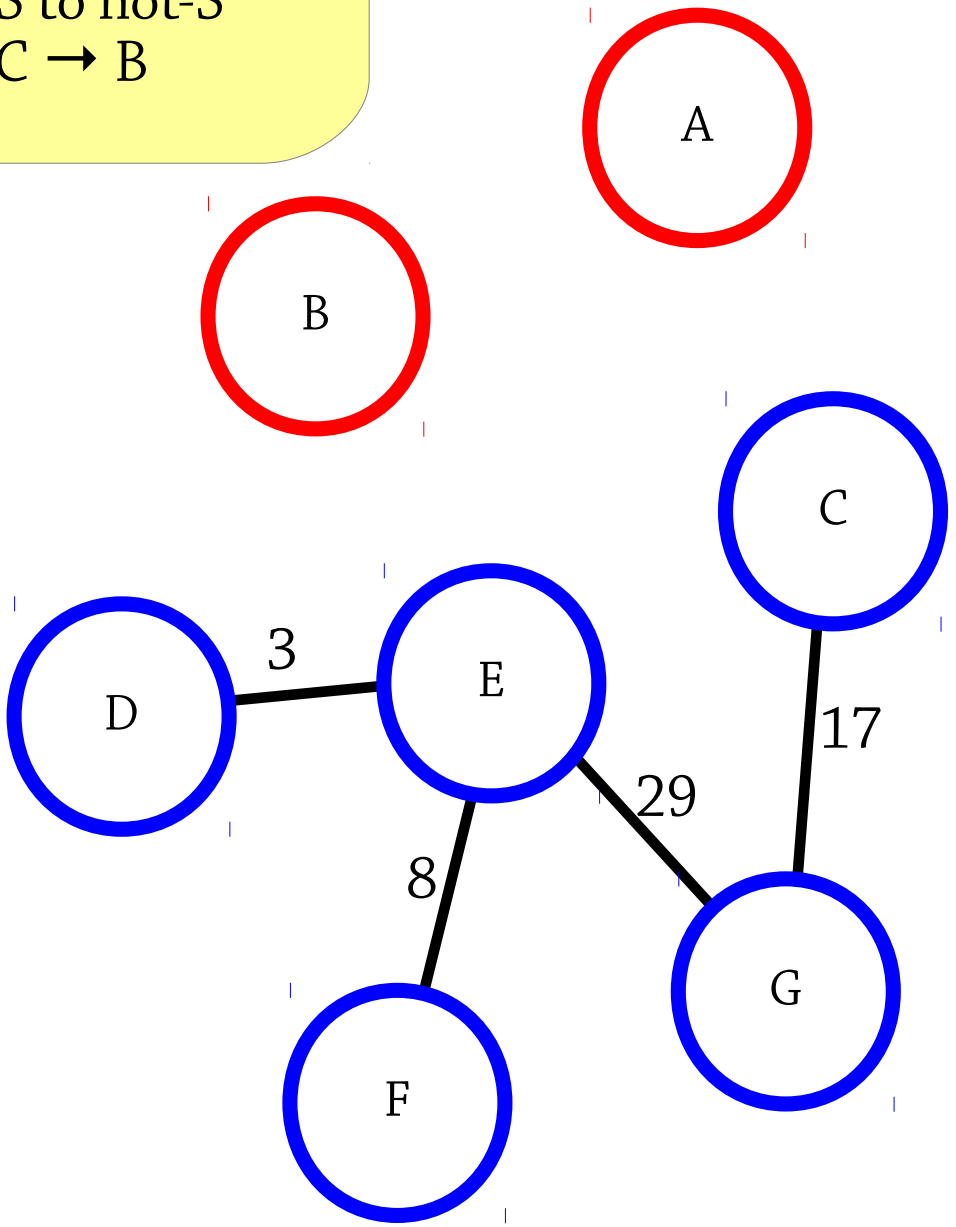




Minimum ees

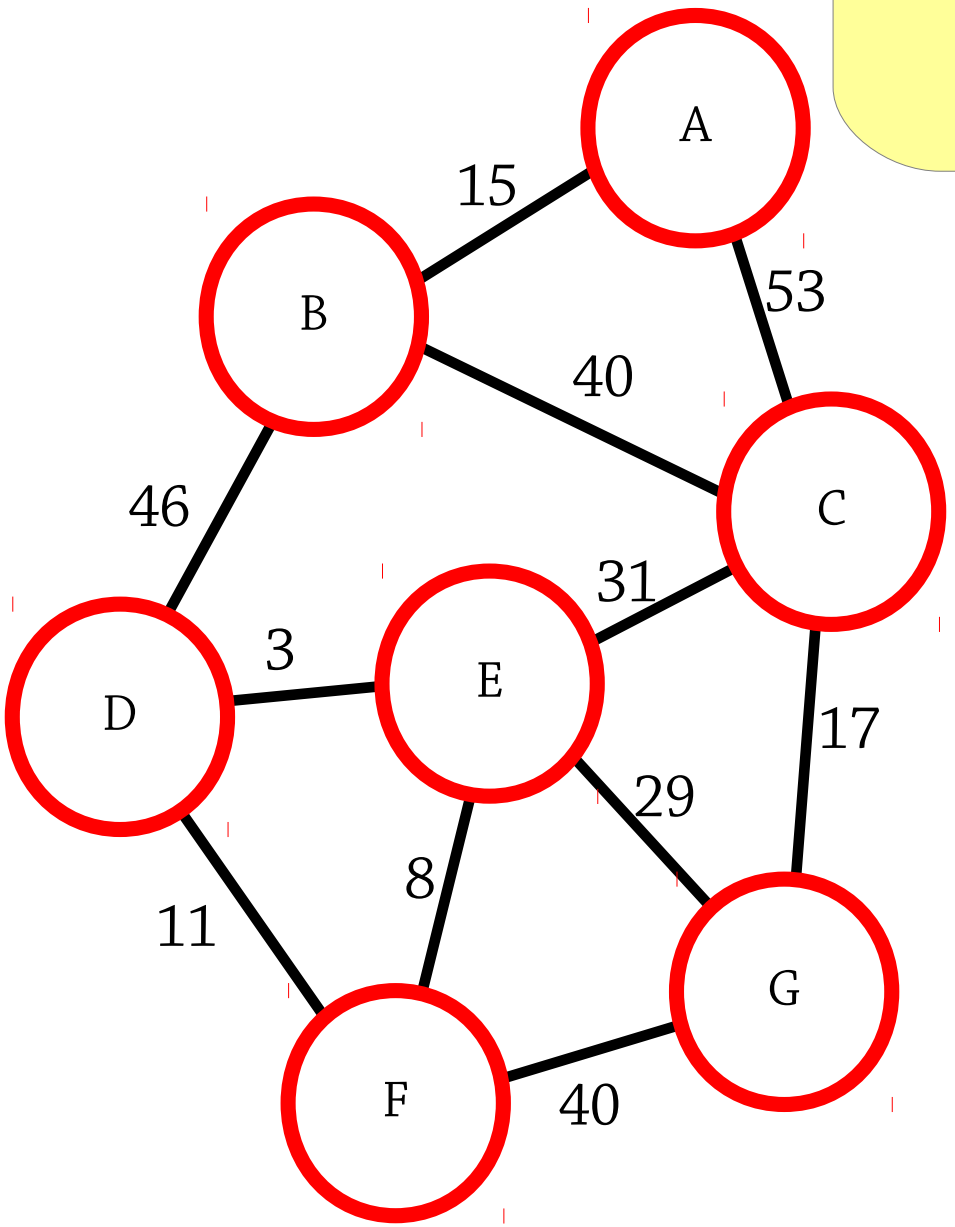
S = {D, E,
F, G,
C}
Lowest-weight edge
from S to not-S
is C → B

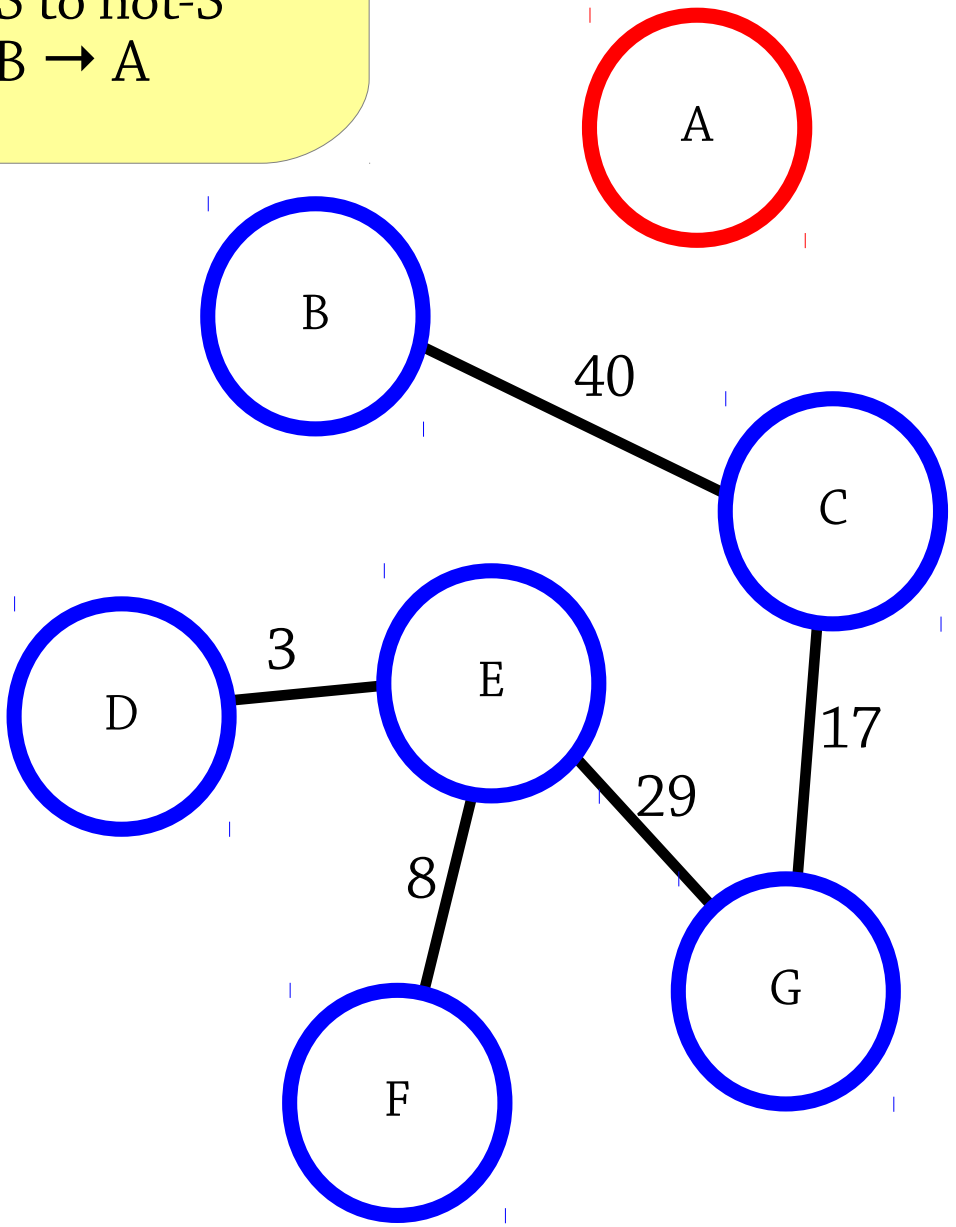




Minimum ees

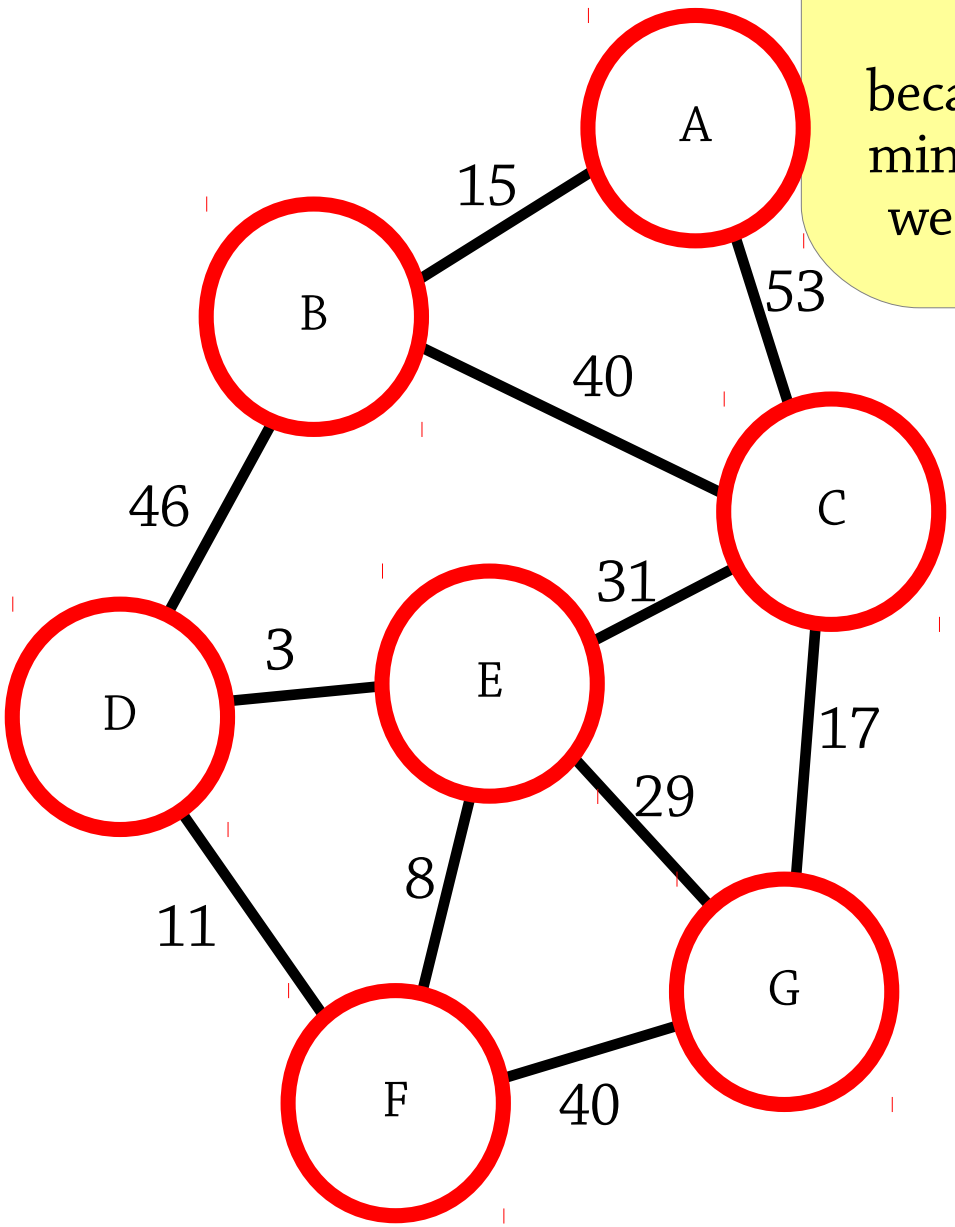
S = {D, E,
F, G,
C, B}
Lowest-weight edge
from S to not-S
is B → A

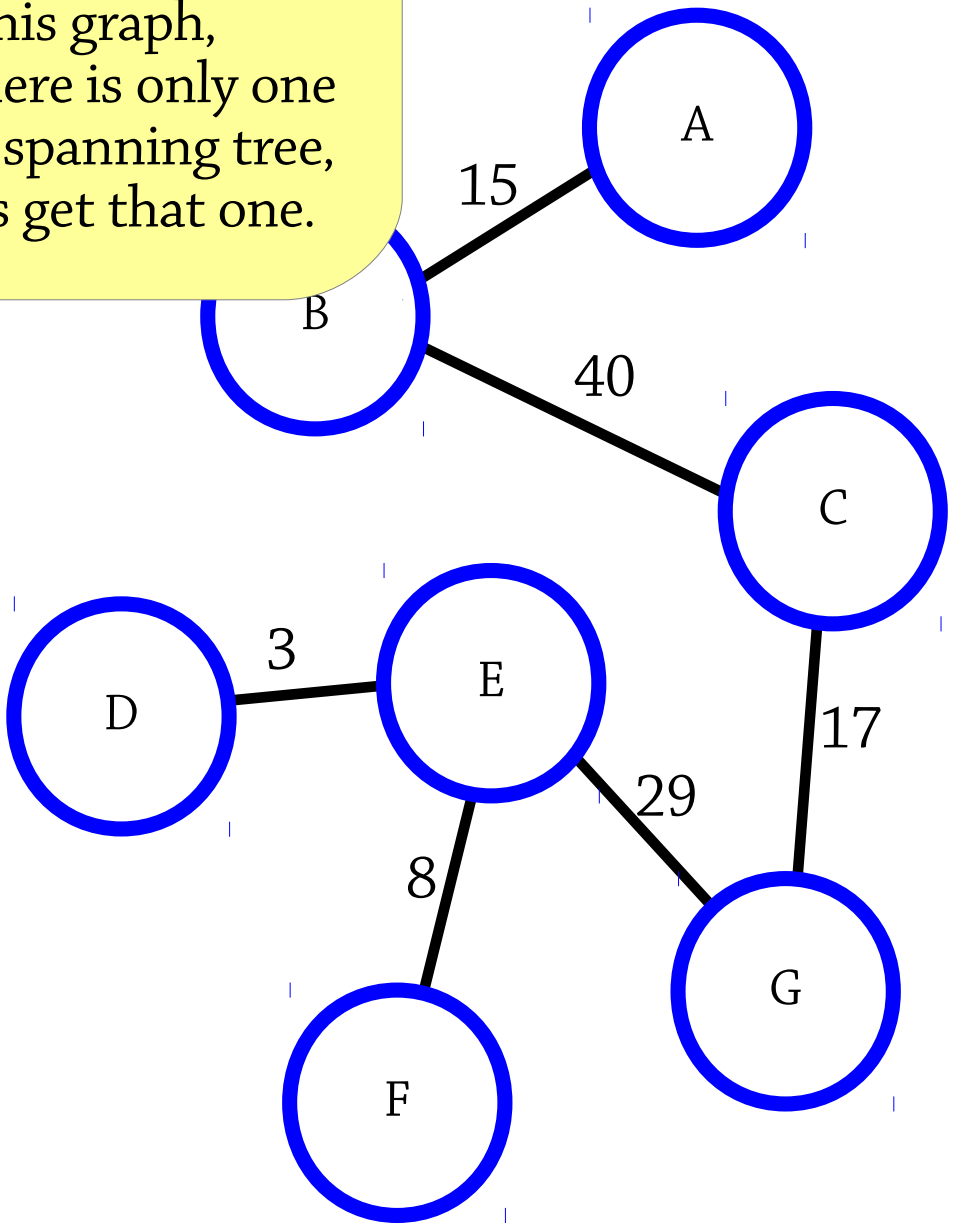




Minimum ees

Notice:
we get a minimum
spanning tree
whatever node we start at!
For this graph,
because there is only one
minimum spanning tree,
we always get that one.





Prim's algorithm, efficiently

The operation

- Pick the *lowest-weight* edge between a node in S and a node not in S

takes O(|E|) time if we're not careful! Then Prim's algorithm will be O(|V||E|)

To implement Prim's algorithm, use a priority queue containing all edges between S and not-S

- Whenever you add a node to S, add all of its edges to nodes in not-S to a priority queue
- To find the lowest-weight edge, just find the minimum element of the priority queue
- Just like in Dijkstra's algorithm, the priority queue might return an edge between two elements that are now in S: ignore it

New time: O(|E| log |E|)

Summary

Breadth-first search – finding shortest paths in unweighted graphs, using a queue

Dijkstra's algorithm – finding shortest paths in weighted graphs – some extensions for those interested:

- Bellman-Ford: works when weights are negative
- A* – faster – tries to move *towards* the target node, where Dijkstra's algorithm explores equally in all directions

Prim's algorithm – finding minimum spanning trees. If you're interested:

- Kruskal's algorithm finds minimum spanning forests (sets of trees) for unconnected graphs.

Both are *greedy algorithms* – they repeatedly find the "best" next element

- Common style of algorithm design

Both use a priority queue to get O(n log n)

- Dijkstra's algorithm is sort of BFS but using a priority queue instead of a queue

Many many many more graph algorithms