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
- but only visiting each node once

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 visited or added to the queue

Then repeat the following process:

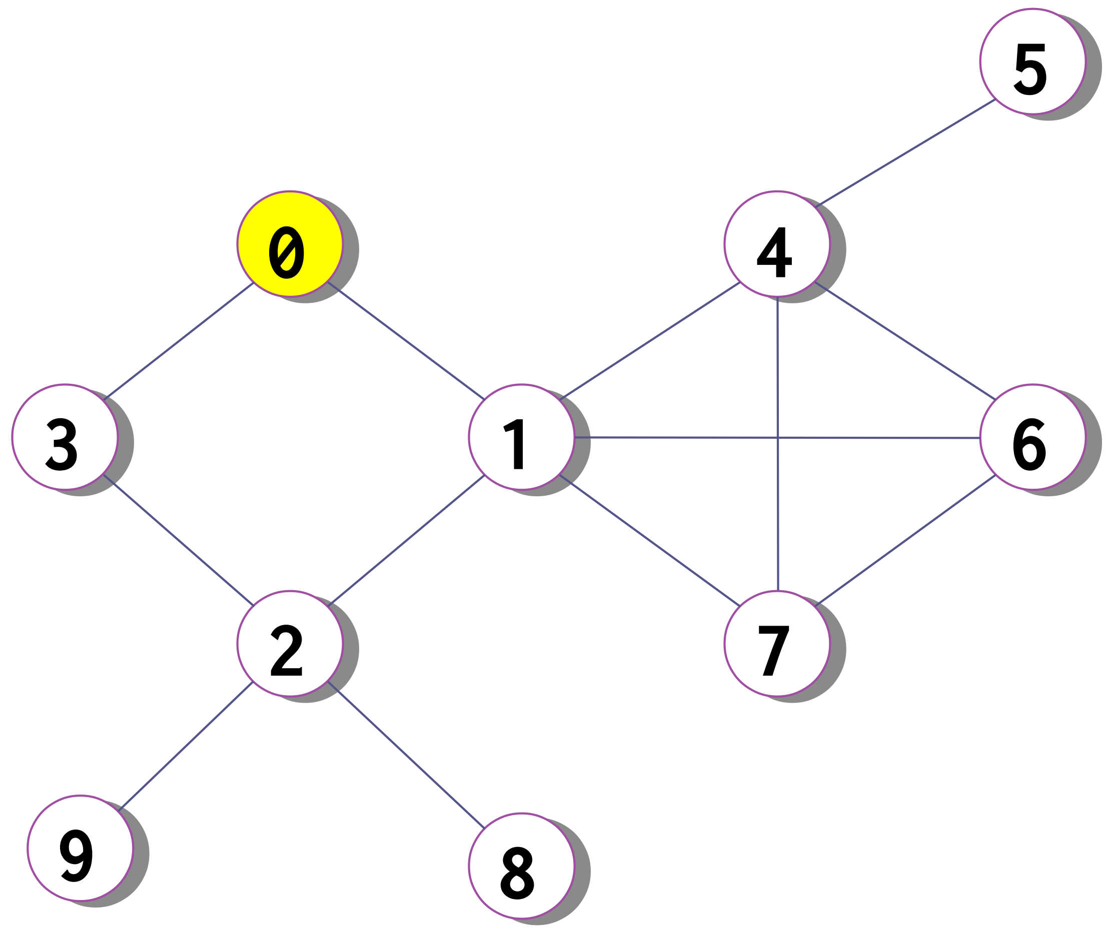
- Remove a node from the queue
- Visit it
- Find all adjacent nodes and add them to the queue, *unless* they've previously been added to the queue

Example of a breadth-first search

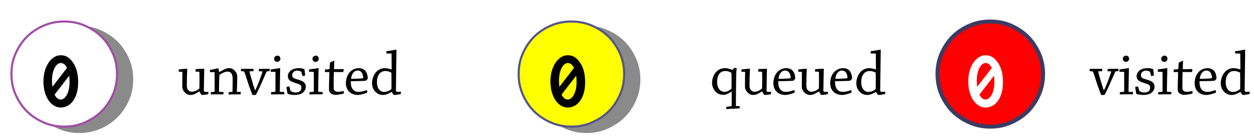
Queue:

0

Visit order:



Initially,
queue contains
start node

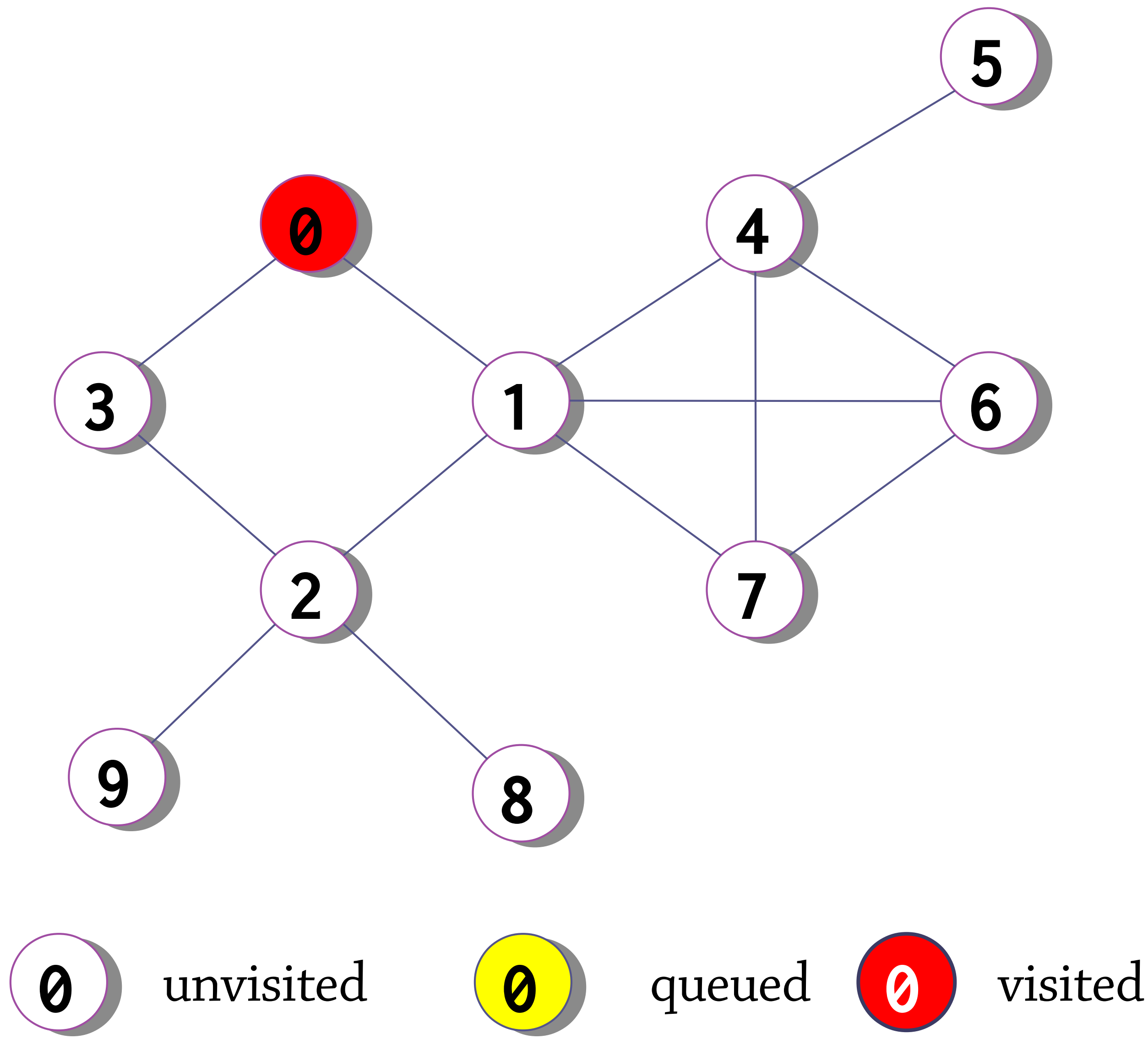


Example of a breadth-first search

Queue:

Visit order:

0



Step 1:
remove node
from queue
and visit it

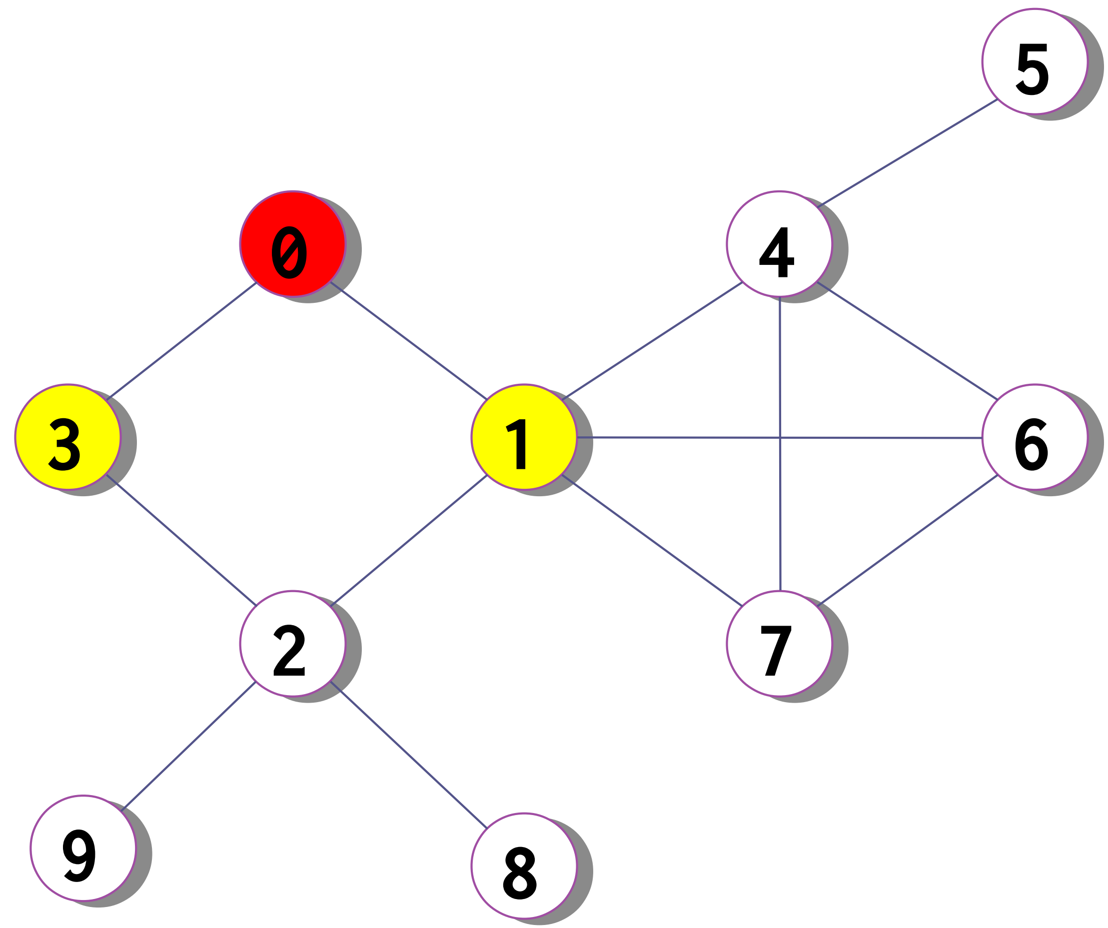
Example of a breadth-first search

Queue:

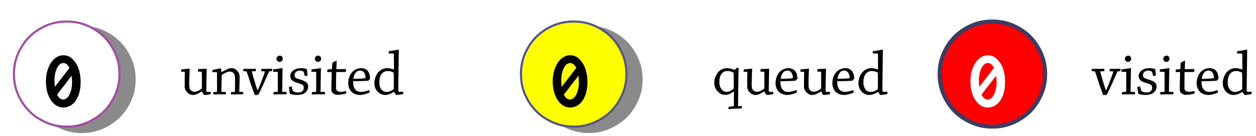
3 1

Visit order:

0



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



Example of a breadth-first search

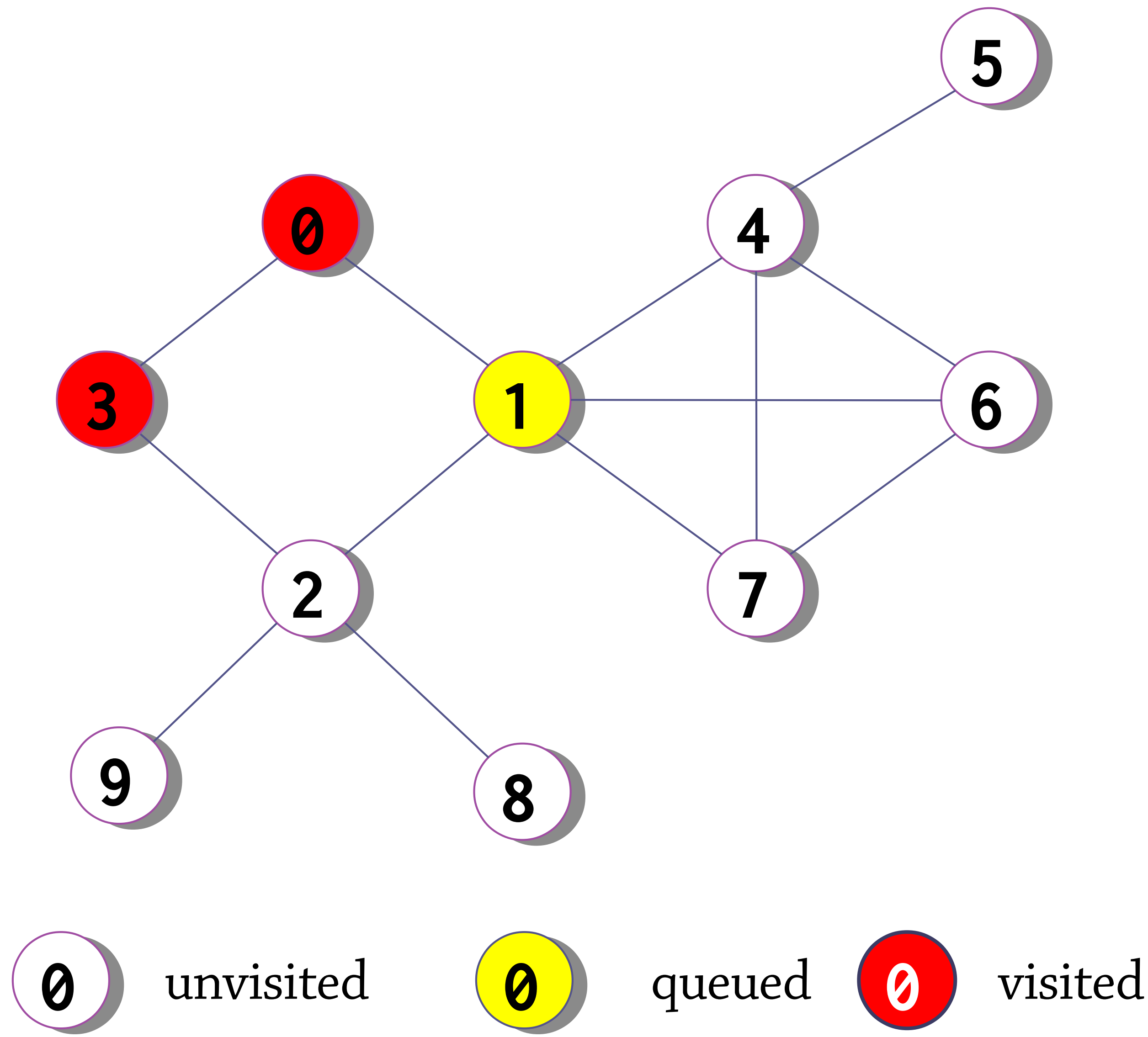
Queue:

1

Visit order:

0 3

Step 1:
remove node
from queue
and visit it



Example of a bre arch

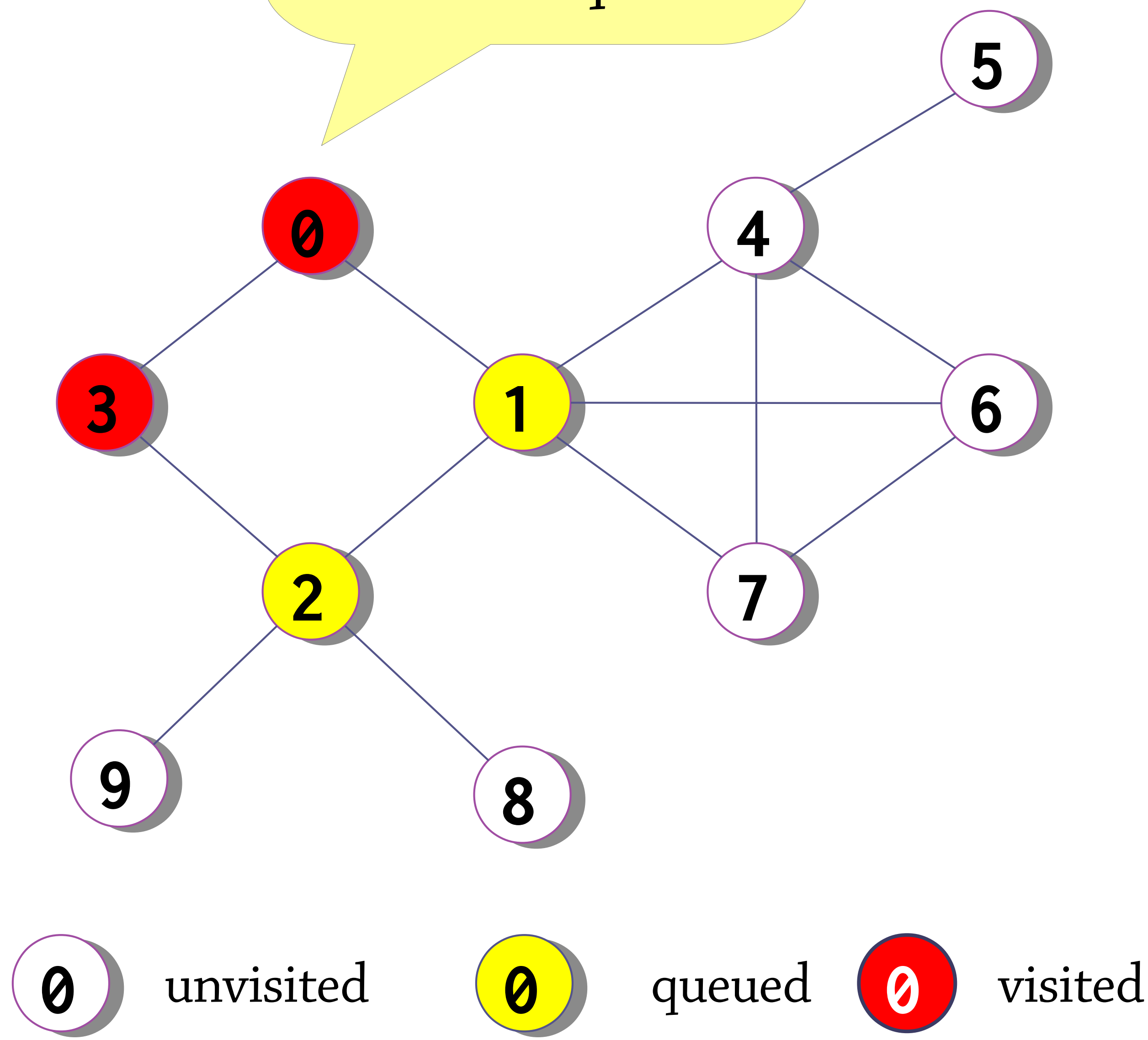
0 is already
visited, so
we don't add
it to the queue

Queue:

1 2

Visit order:

0 3



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

Example of a breadth-first search

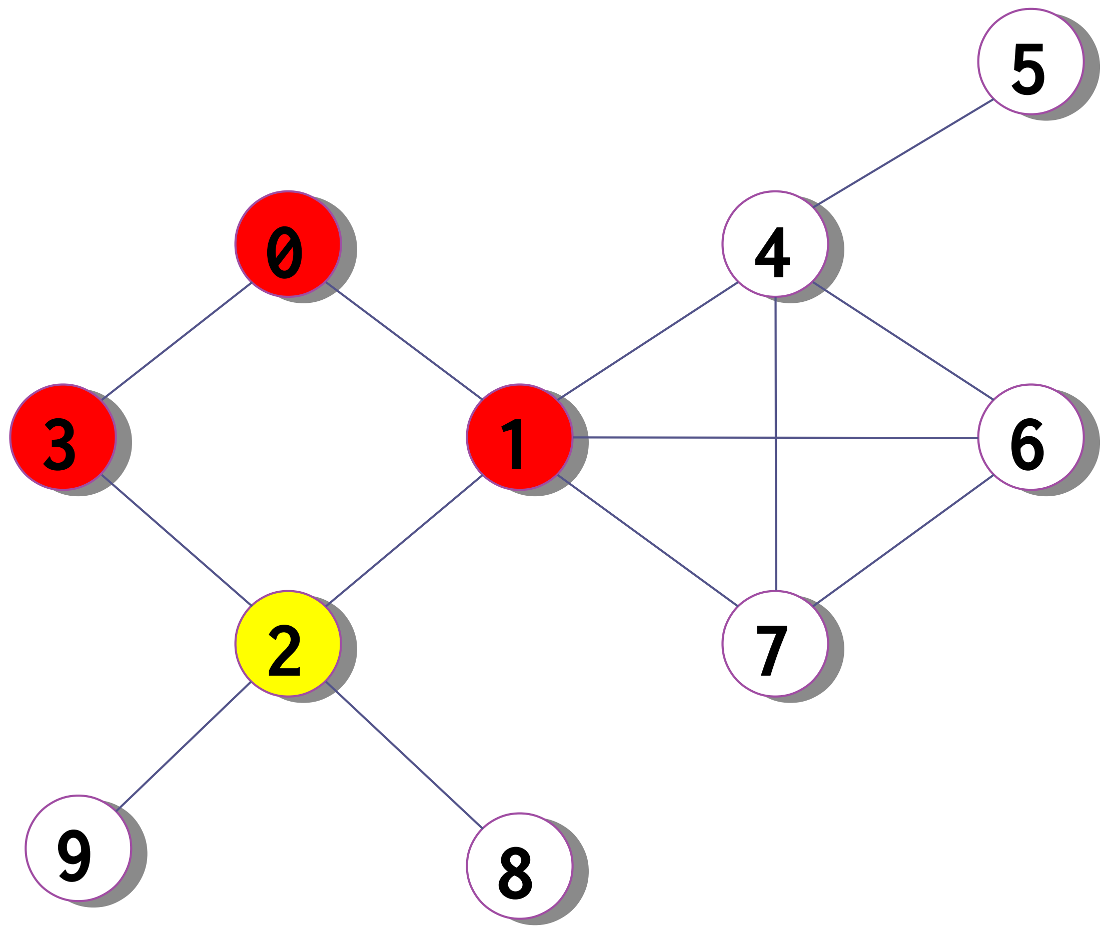
Queue:

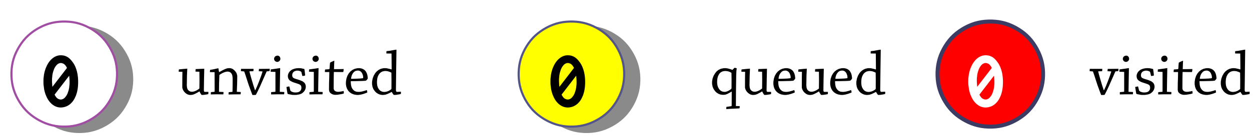
2

Visit order:

0 3 1

Step 1:
remove node
from queue
and visit it





Example of a bre arch

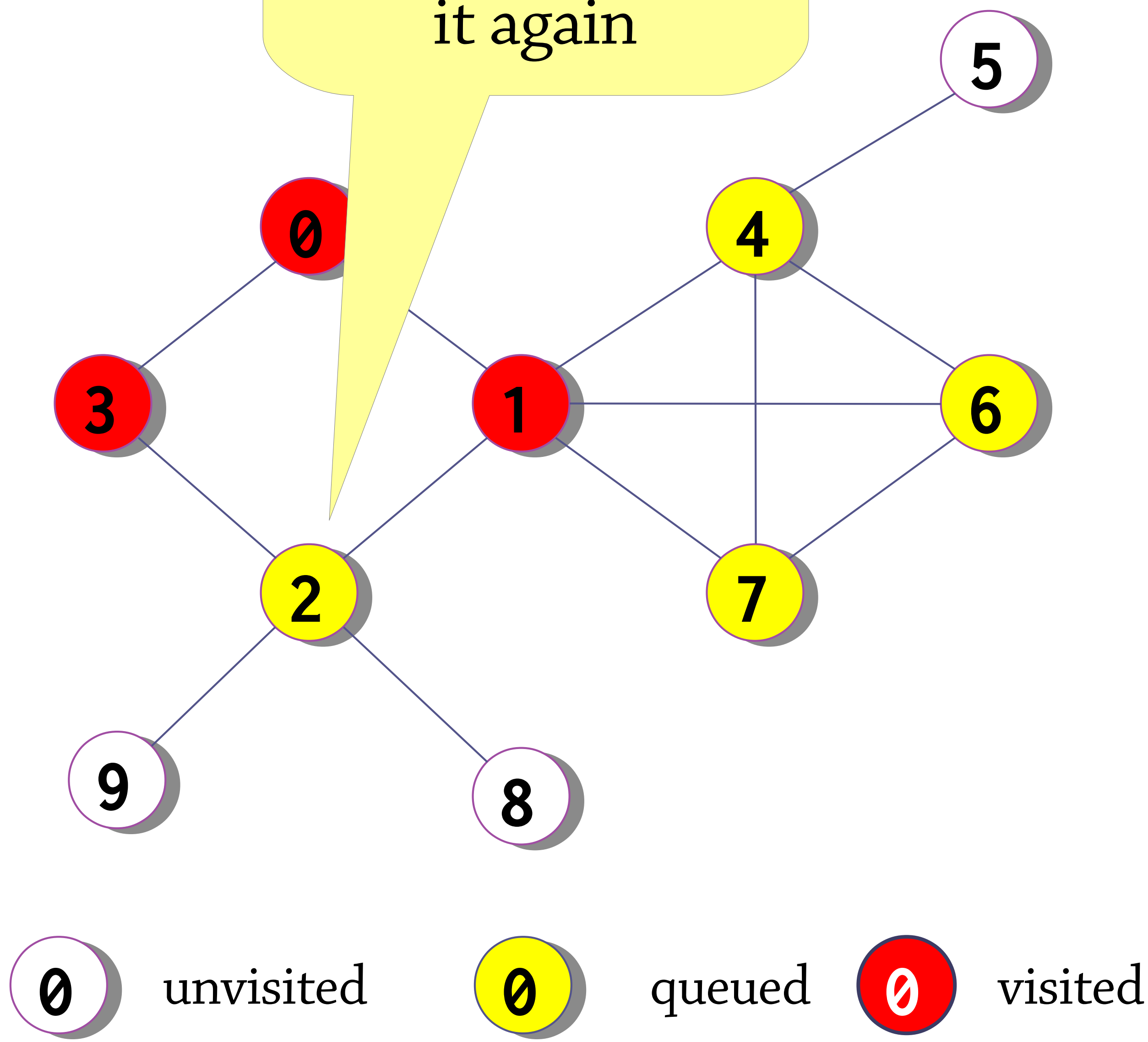
2 is already
in the queue, so
we don't add
it again

Queue:

2 4 6 7

Visit order:

0 3 1



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

Example of a breadth-first search

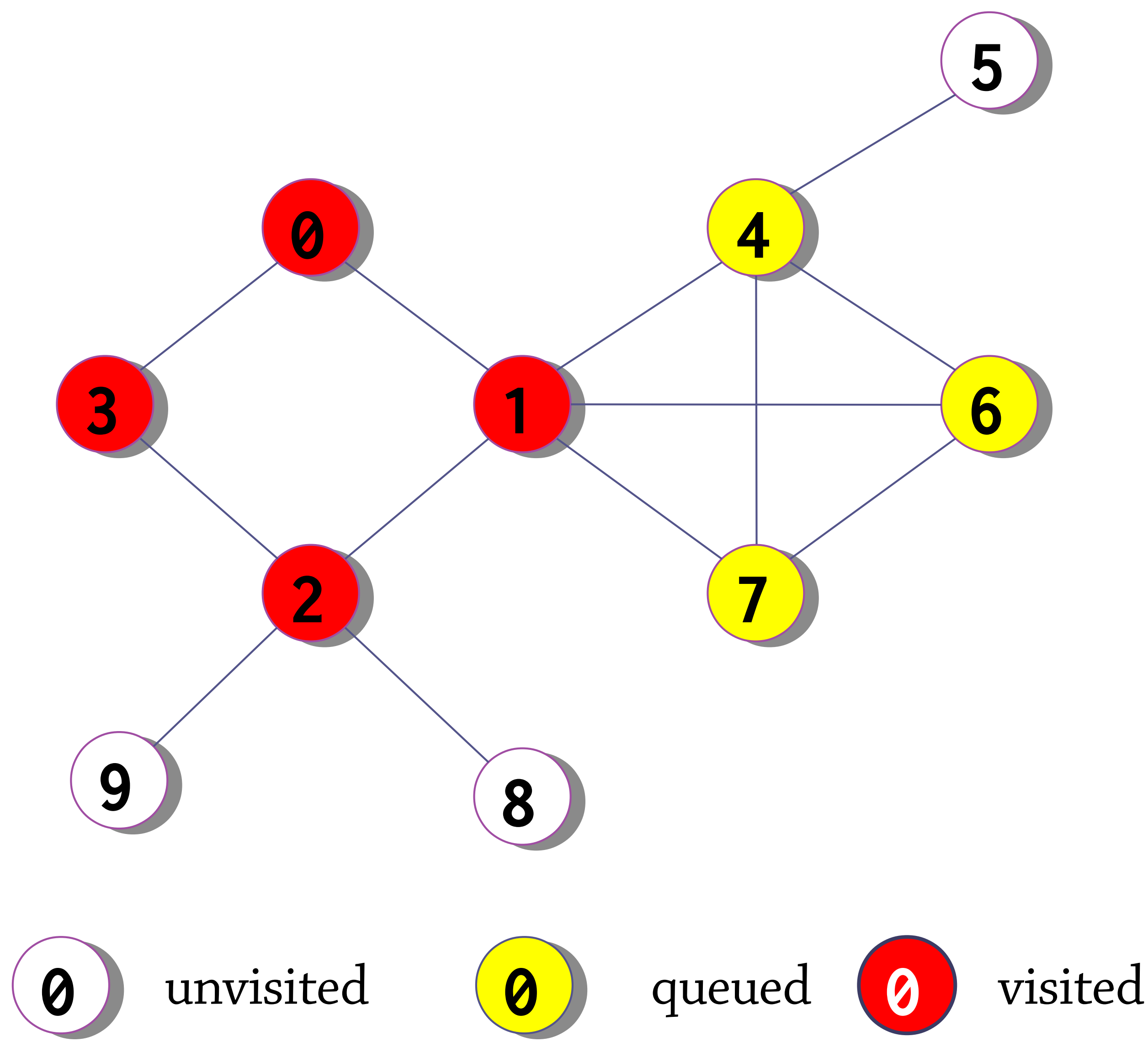
Queue:

4 6 7

Visit order:

0 3 1 2

Step 1:
remove node
from queue
and visit it



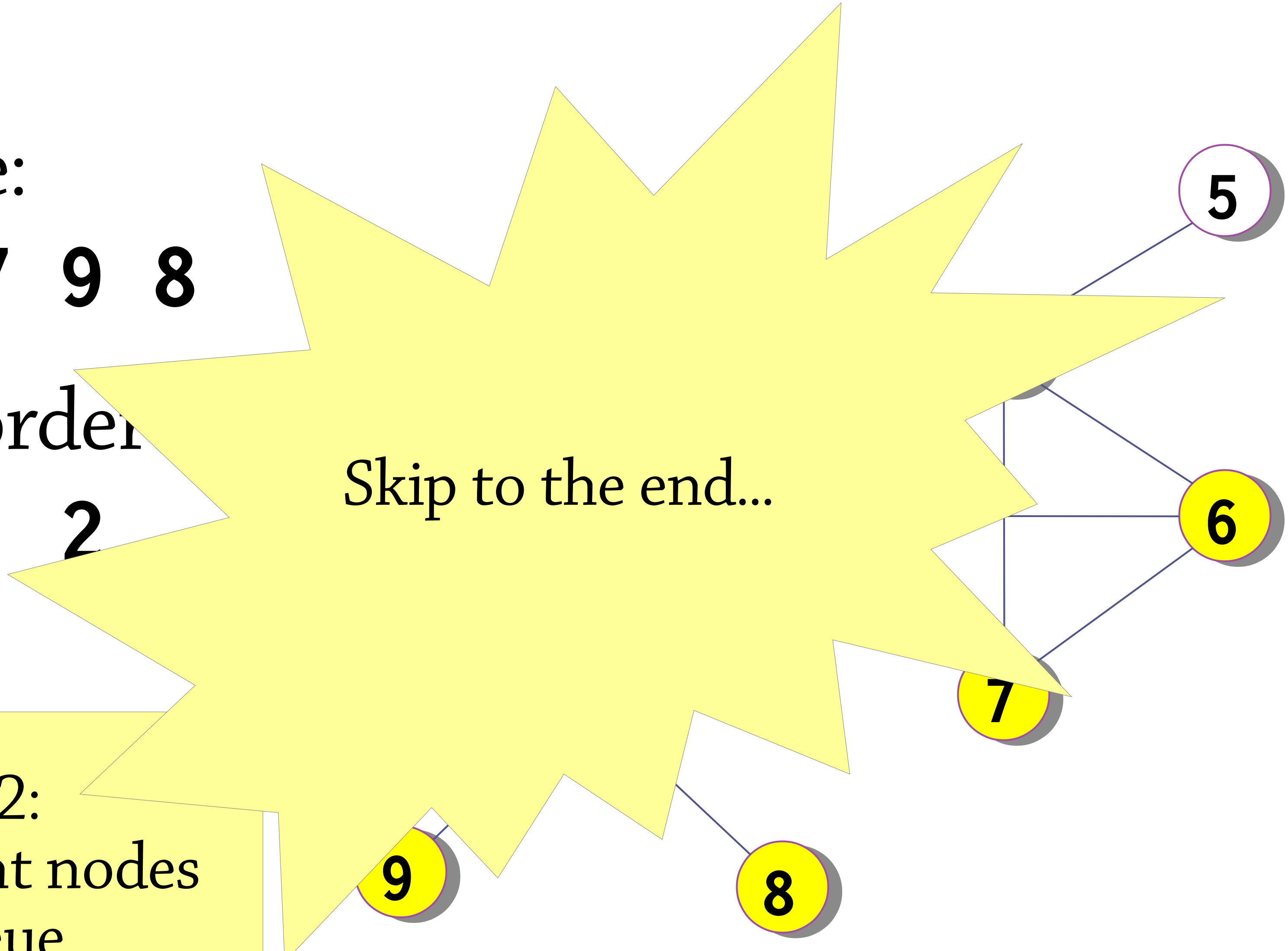
Example of a breadth-first search

Queue:

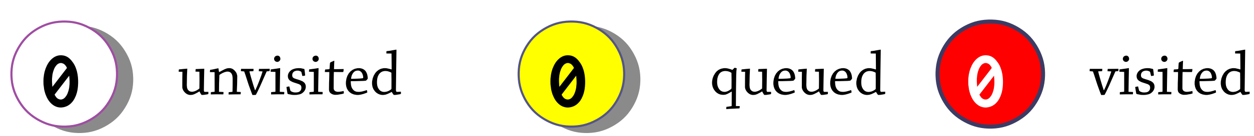
4 6 7 9 8

Visit order:

0 3 1 2



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



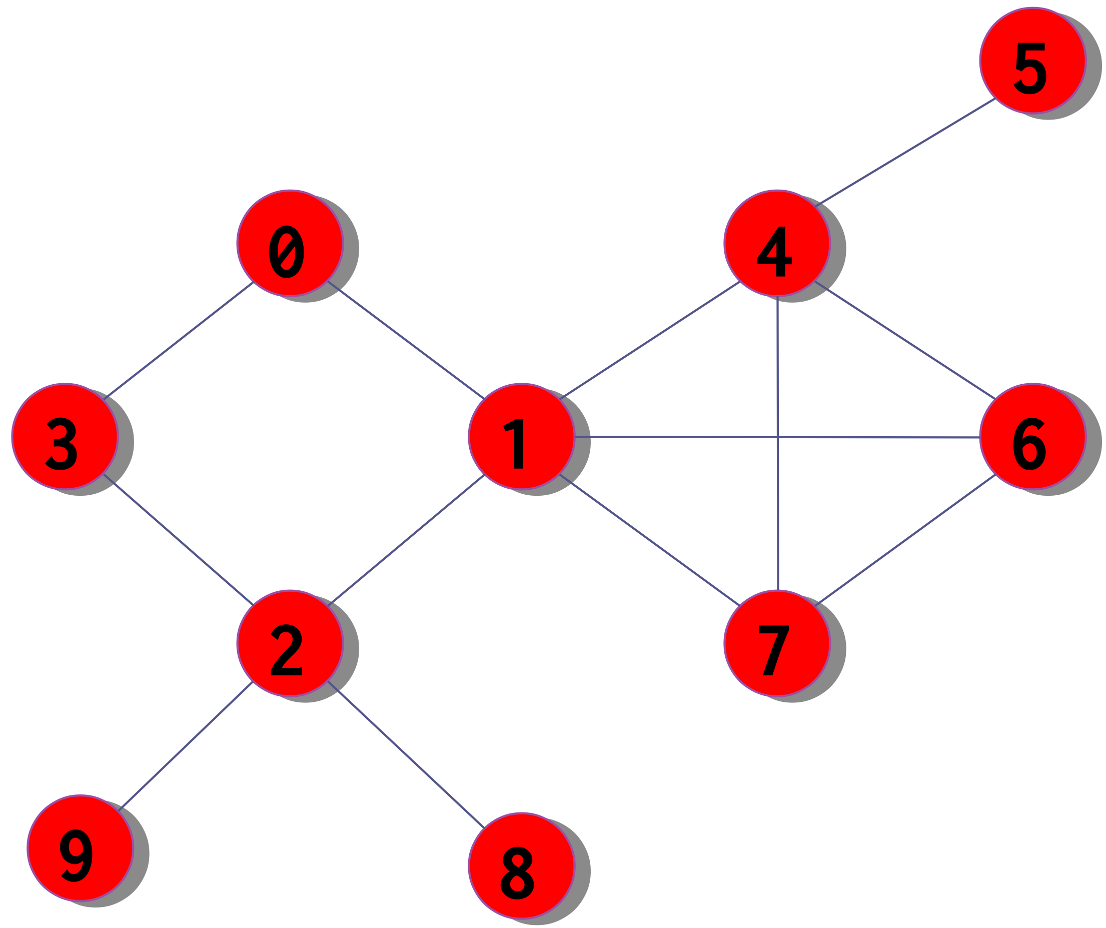
Example of a breadth-first search

Queue:

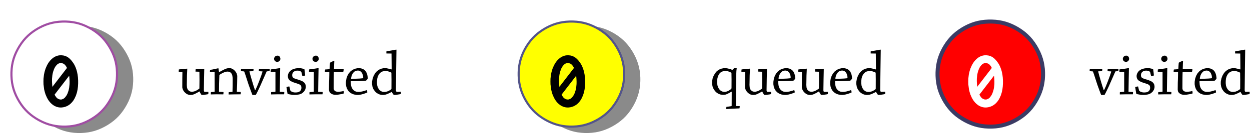
Visit order:

0 3 1 2 4

6 7 9 8 5

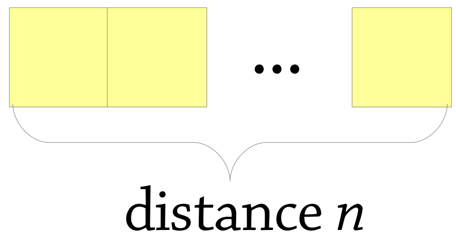


We reach step 1, but the queue is empty, and **we're finished!**

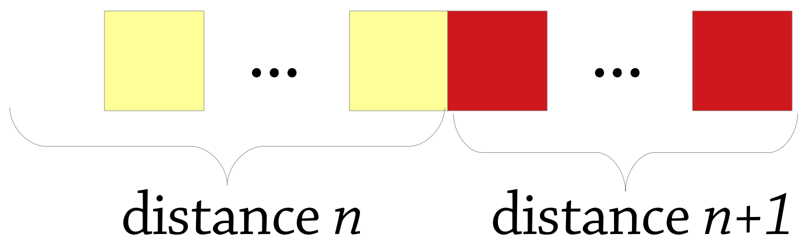


Why does using a queue work?

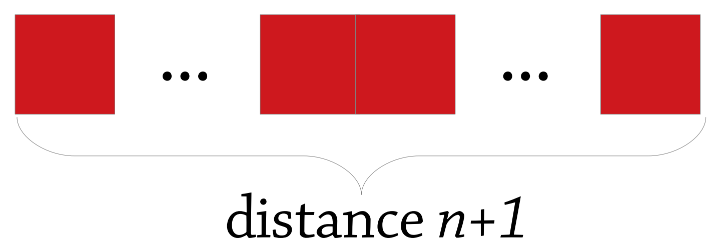
Suppose the queue contains all nodes that are distance *n* from the starting node:



We remove the first node and add its neighbours, which are at a distance of *n+1*:



Since queues are FIFO, we then visit all the other distance *n* nodes, adding each node's neighbours to the queue. The queue now consists only of distance *n+1* nodes!

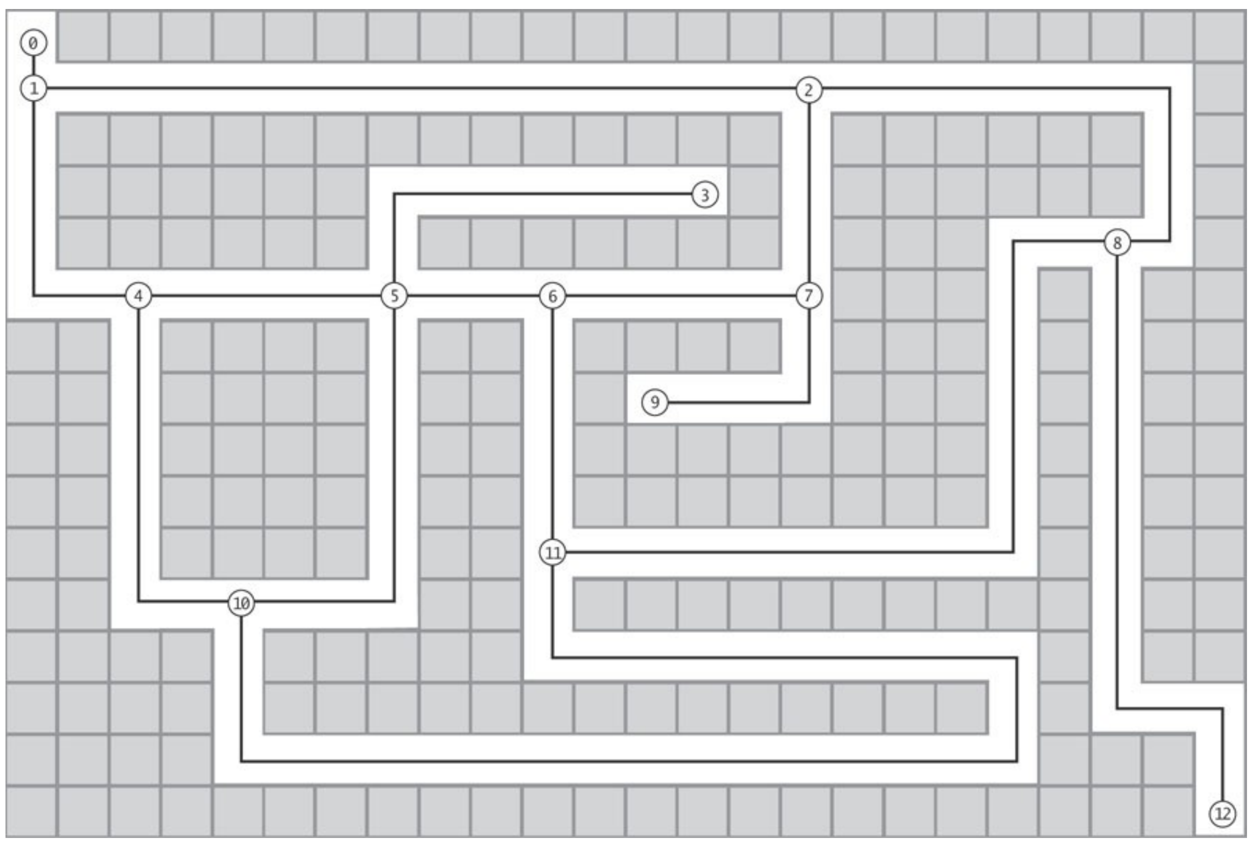


So we explore all nodes of distance *n* before getting to nodes of distance *n+1*.

Side note: if we use a stack instead of a queue, we get depth-first search!

Application: unweighted shortest path

We can represent a maze as a graph – nodes are junctions, edges are paths. We want to find the simplest way (fewest choices) to get from entrance to exit. This is the *shortest path*

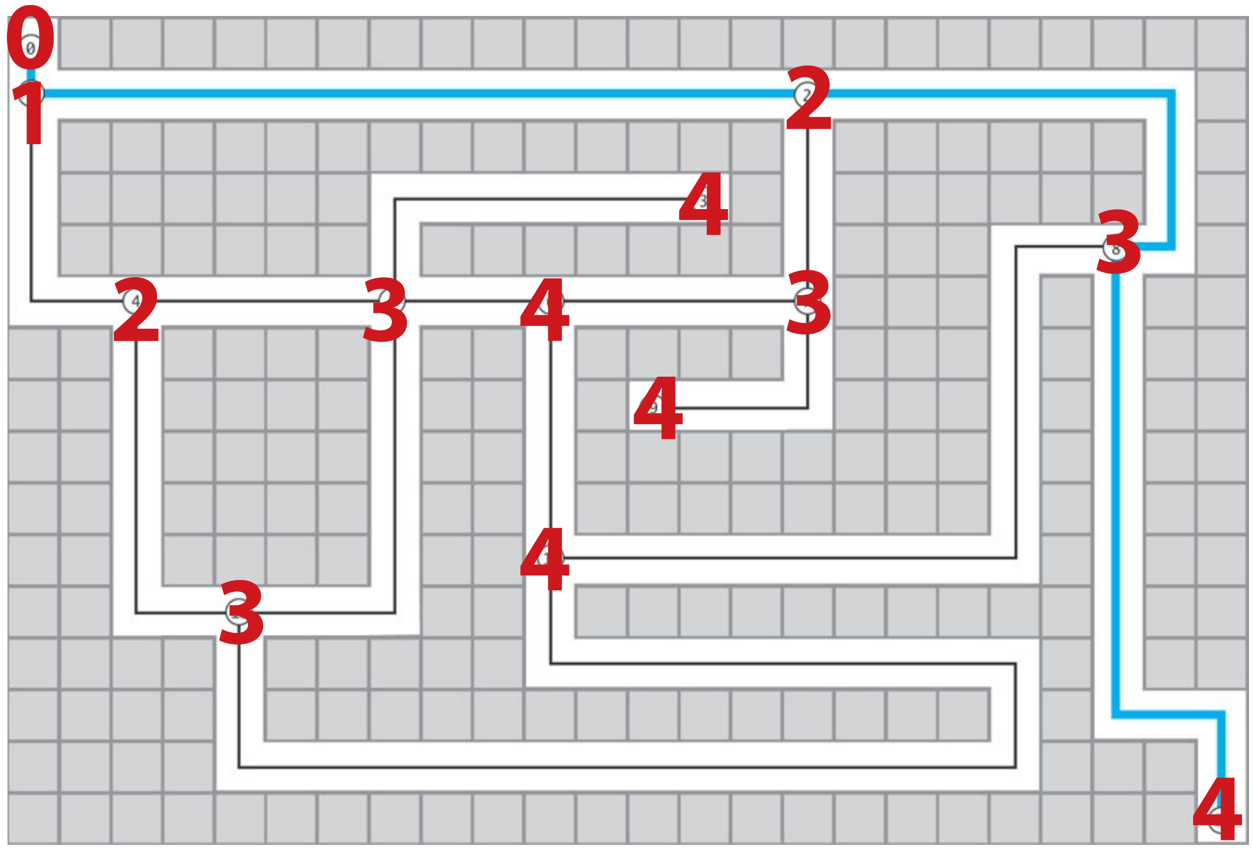


Application: unweighted shortest path

We do a breadth-first search from the entrance and remember the *distance* from the entrance to each node

- Distance to a node = distance to "parent node" + 1

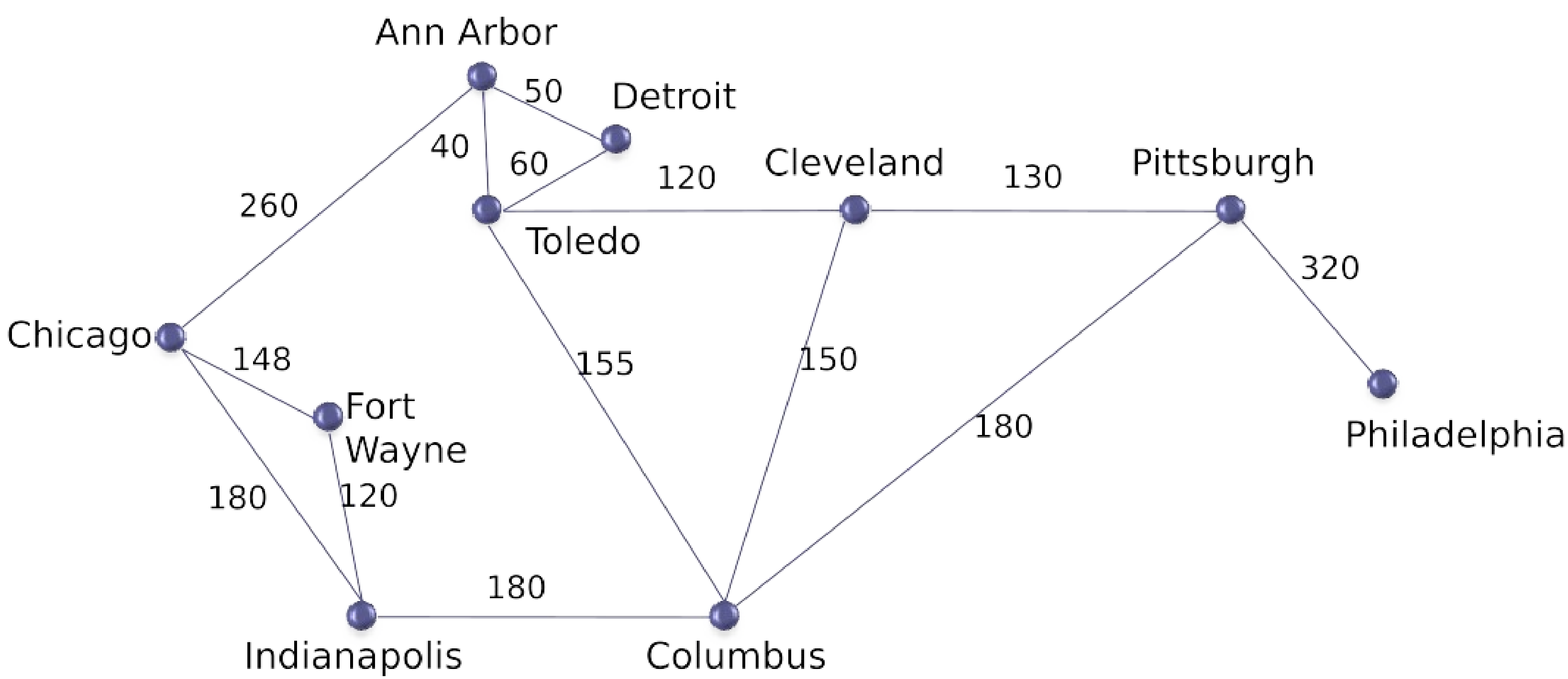
Using these distances, we can trace back from the exit to the entrance!



Dijkstra'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 non-negative weight

Dijkstra's algorithm

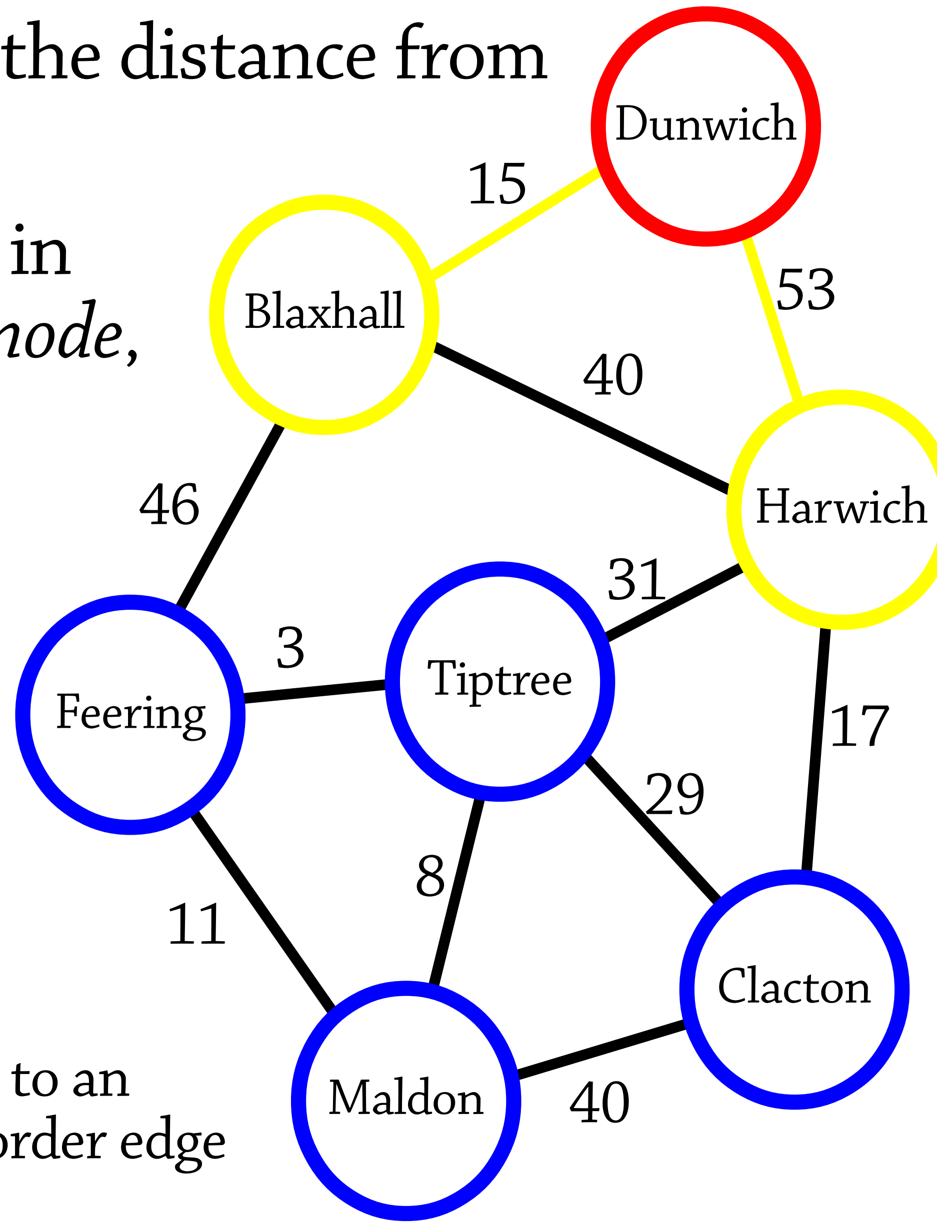
Dijkstra's algorithm computes the distance from a start node to *all other nodes*

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

We first visit the start node, which has a distance of 0

We are going to use the idea of a *border edge*, which is an edge from a visited node to an unvisited node (yellow here)

- If you want to get from the start node to an unvisited node, you have to go via a border edge



Dijkstra's algorithm

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

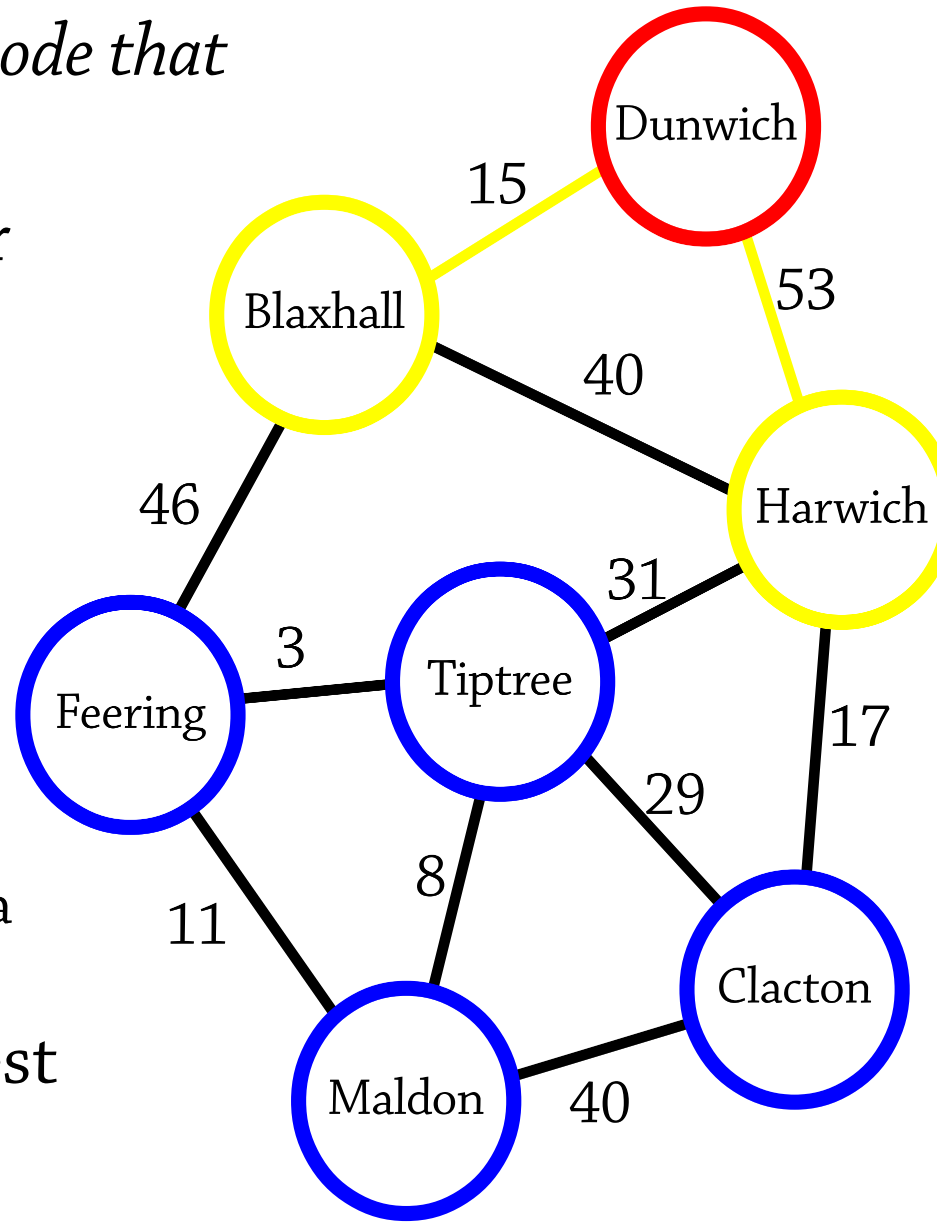
This node must be the neighbour of a visited node (why?)

- Here either Blaxhall or Harwich
- That means it must be the target of a border edge

For each border edge x → y:

- Add the distance to *x* and the weight of the edge *x* → *y*
- This is the total distance to y, going via that border edge

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

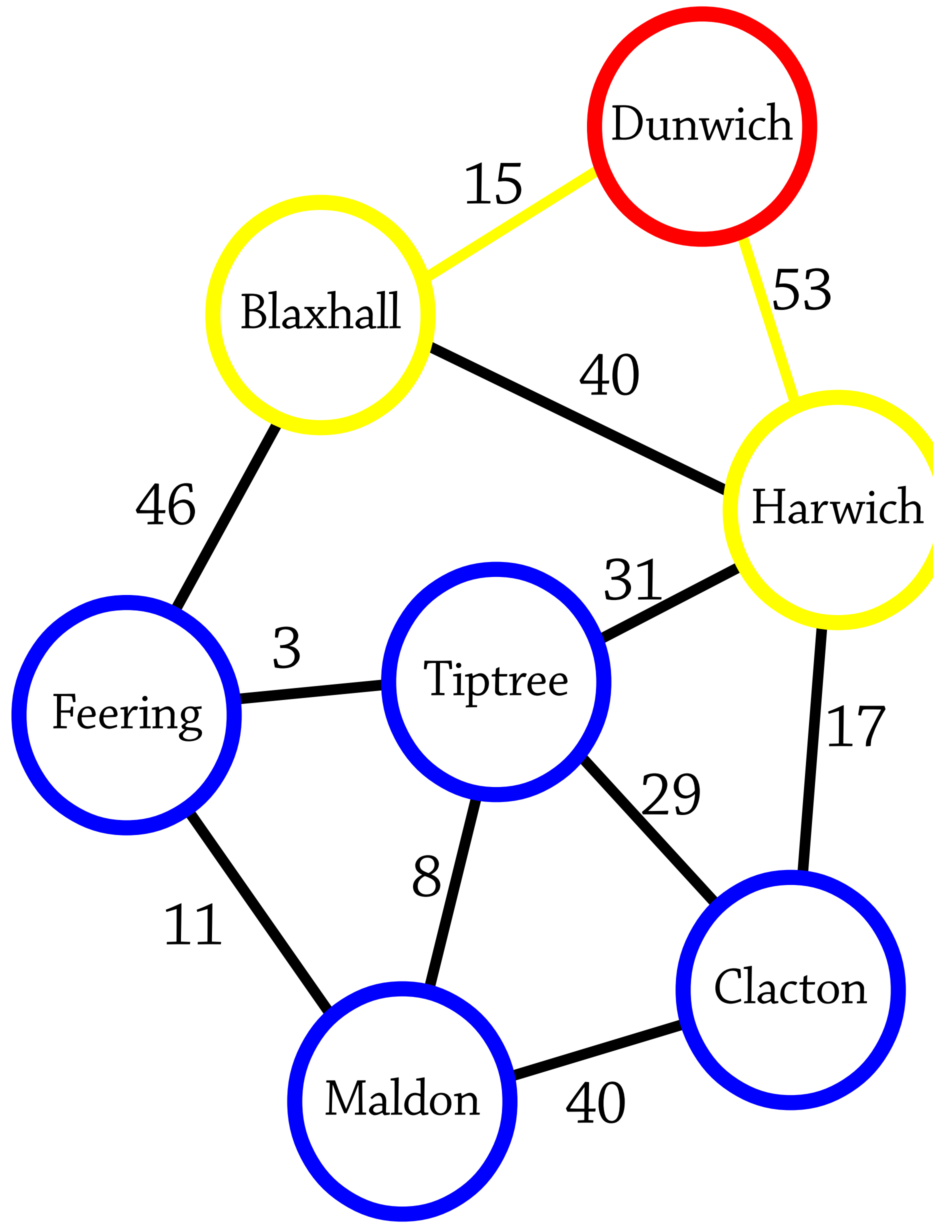


Dijkstra's algorithm

Visited nodes (red):
Dunwich distance 0

Border edges lead to:
Blaxhall (distance 15),
Harwich (distance 53)

So visit Blaxhall
(distance 15)



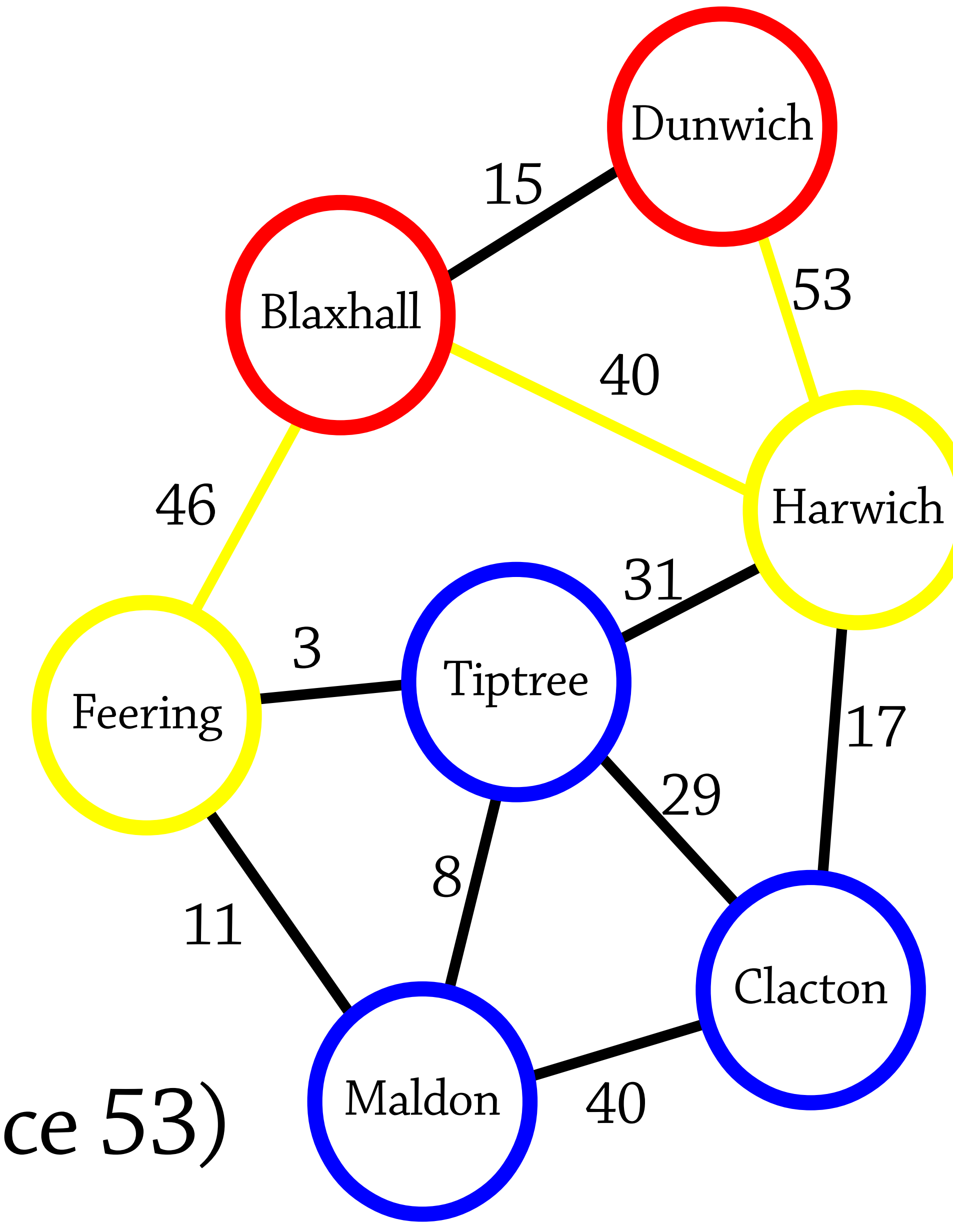
Dijkstra's algorithm

Visited nodes:
Dunwich distance 0
Blaxhall distance 15

Border edges lead to:

- Feering (distance 15 + 46 = 61)
- Harwich (via Dunwich, distance 53)
- Harwich (via Blaxhall, distance 15 + 40 = 55)

So visit Harwich (distance 53)



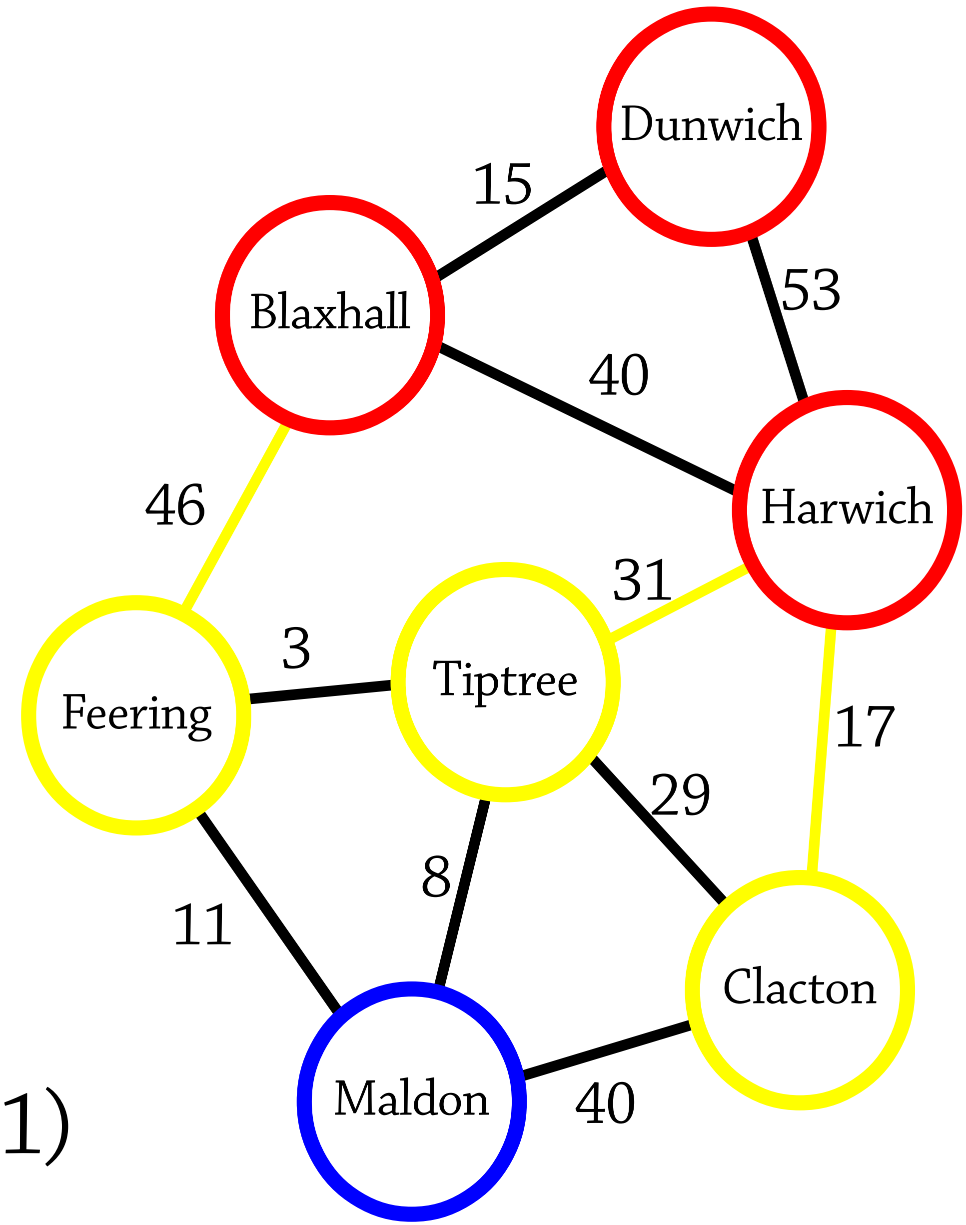
Dijkstra's algorithm

Visited nodes:
Dunwich distance 0
Blaxhall distance 15
Harwich distance 53

Neighbours (yellow) are:

- Feering (distance 15 + 46 = 61)
- Tiptree (distance 53 + 31 = 84)
- Clacton (distance 53 + 17 = 70)

So visit Feering (distance 61)



Dijkstra's algorithm

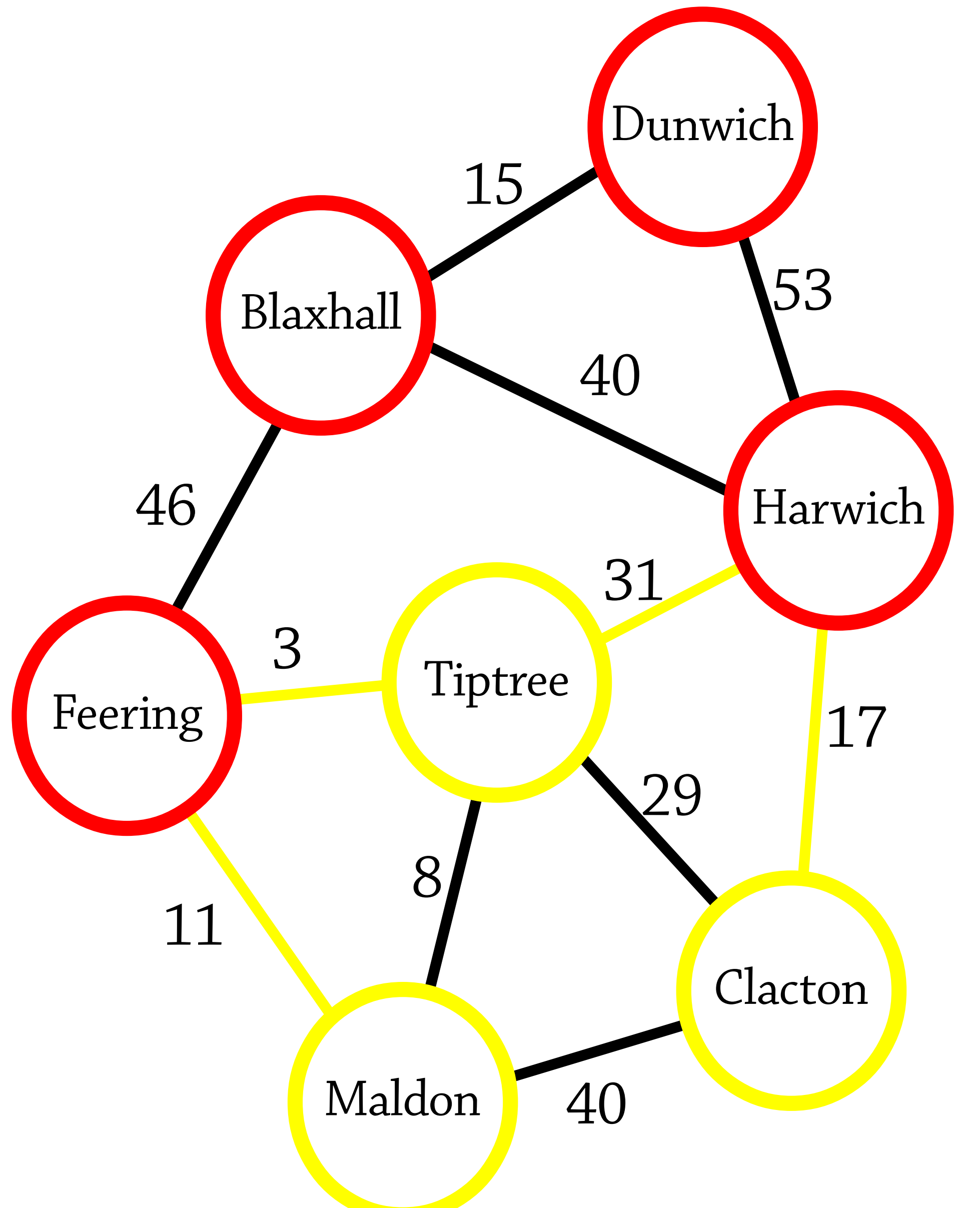
Visited nodes:

Dunwich distance 0
Blaxhall distance 15
Harwich distance 53
Feering distance 61

Neighbours are:

- Tiptree via Feering (distance 61 + 3 = 64)
- Tiptree via Harwich (distance 55 + 29 = 84)
- Clacton (distance 53 + 17 = 70)
- Malden (distance 61 + 11 = 72)

So visit Tiptree (distance 64)



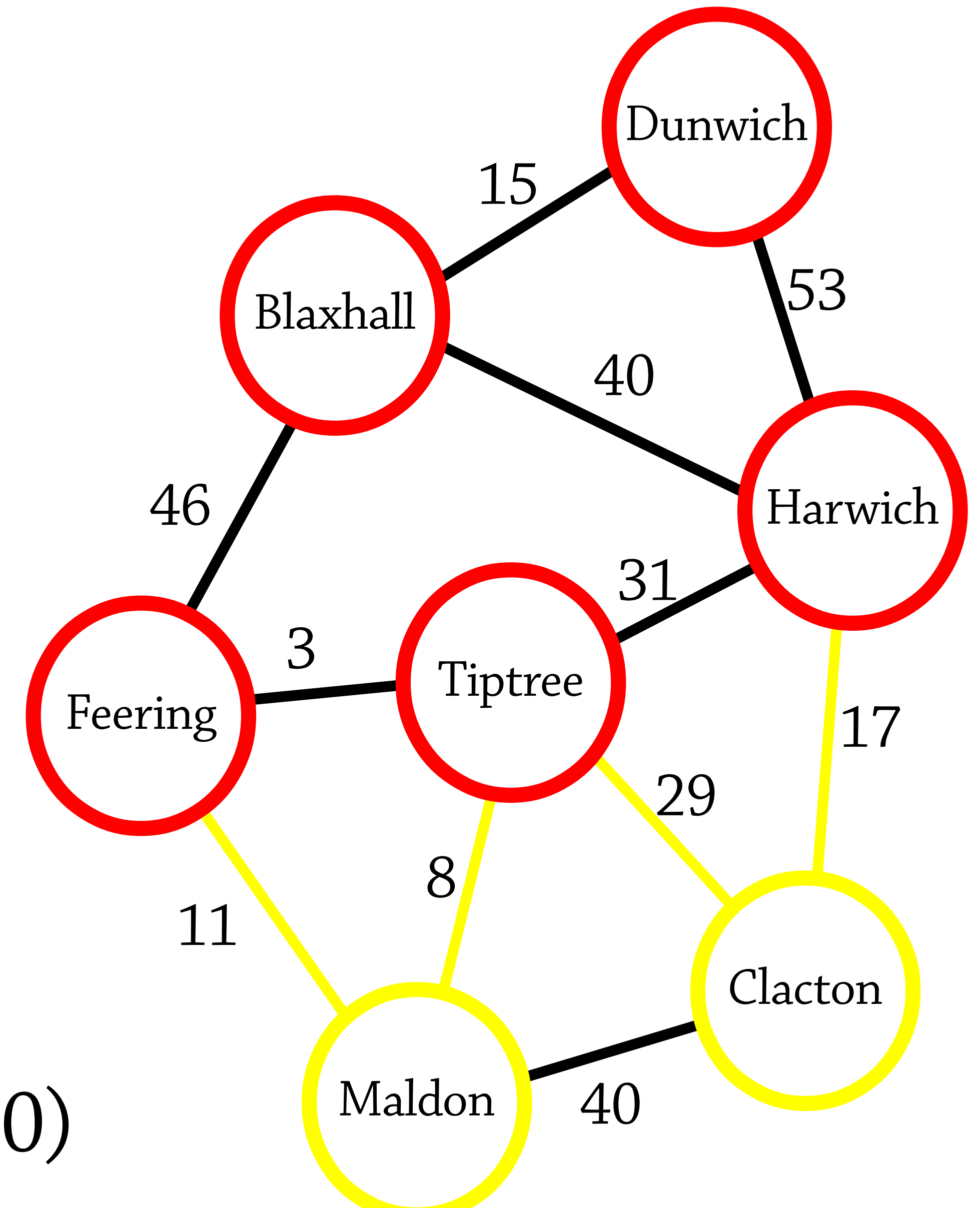
Dijkstra's algorithm

Visited nodes:
Dunwich distance 0
Blaxhall distance 15
Harwich distance 53
Feering distance 61
Tiptree distance 64

Neighbours are:

- Clacton (distance 53 + 17 = 70, also via Tiptree 64 + 29 = 93)
- Maldon (distance 61 + 11 = 72, also via Tiptree 64 + 8 = 72)

So visit Clacton (distance 70)



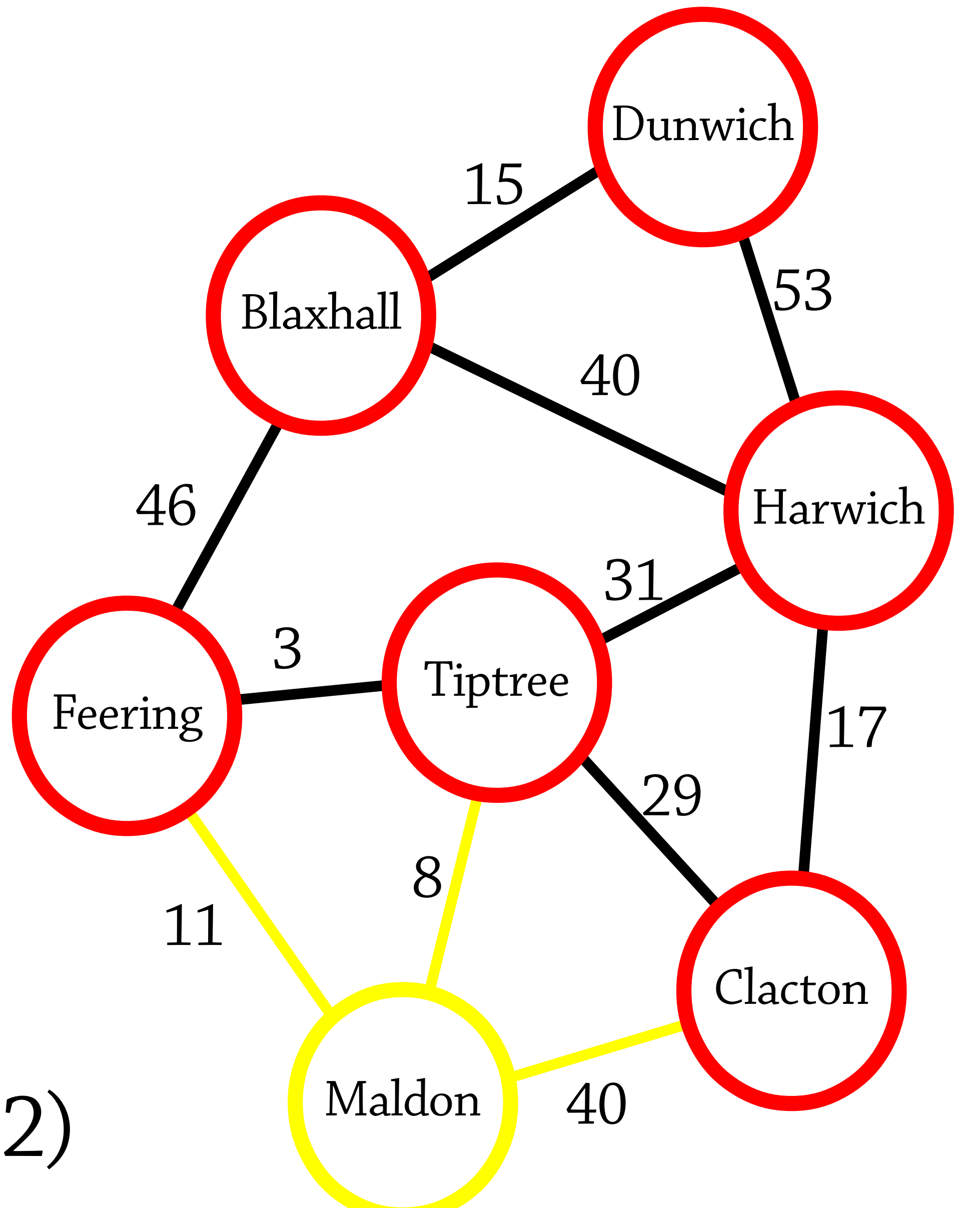
Dijkstra's algorithm

Visited nodes:
Dunwich distance 0
Blaxhall distance 15
Harwich distance 53
Feering distance 61
Tiptree distance 64
Clacton distance 70

Neighbours are:

- Maldon (distance 61 + 11 = 72, also via Tiptree 64 + 8 = 72, also via Clacton 70 + 40 = 110)

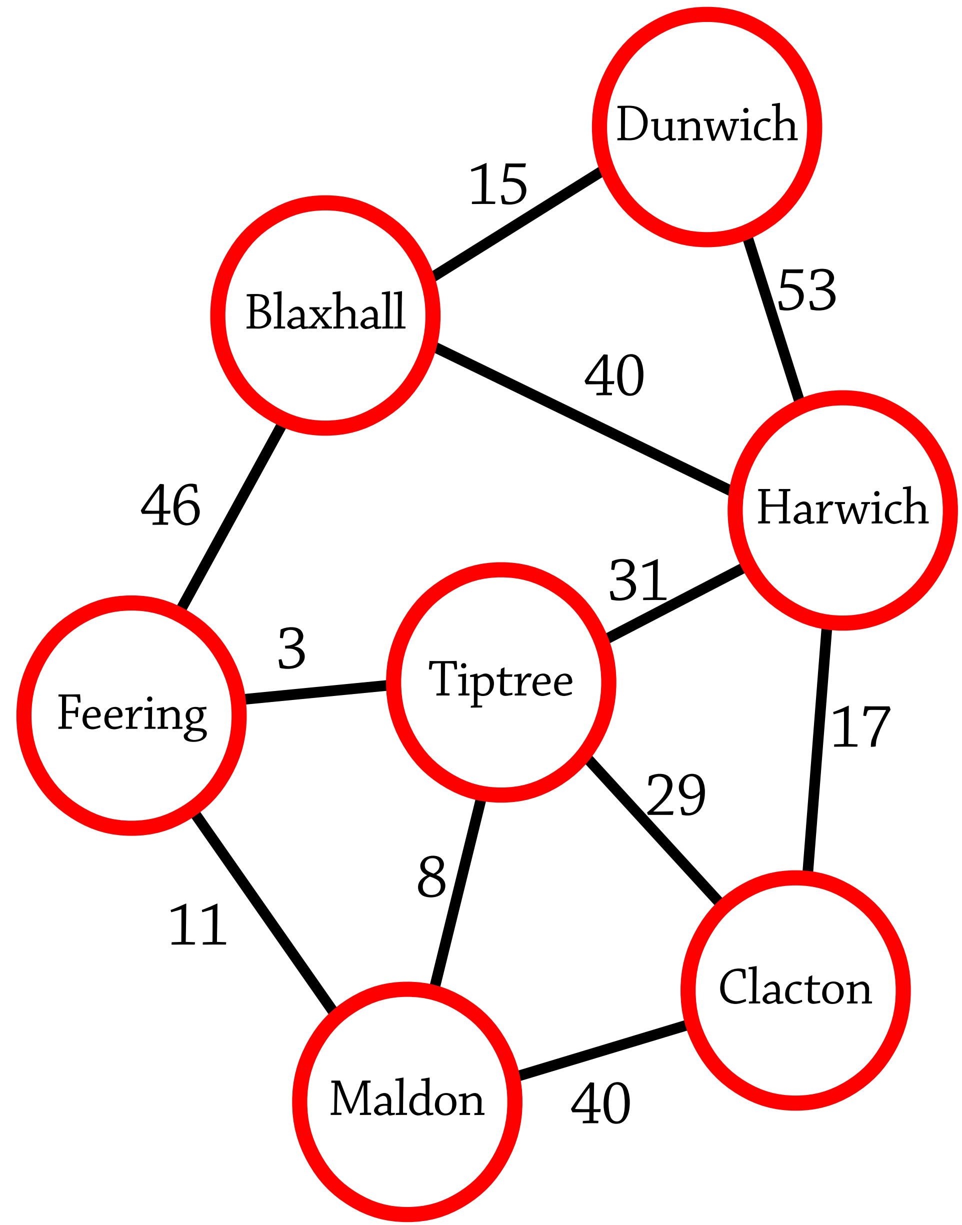
So visit Maldon (distance 72)



Dijkstra's algorithm

Visited nodes:
Dunwich distance 0
Blaxhall distance 15
Harwich distance 53
Feering distance 61
Tiptree distance 64
Clacton distance 70
Maldon distance 72

Finished!



Two problems

1. How to implement this efficiently?
   - Naive implementation takes $O(|E| \times |V|)$ time, where $|E|$ = number of edges, $|V|$ = number of nodes
   - This is because, in order to choose the next node to visit, we have to go through all border edges to find the best one
   - We can solve this by storing the border edges in a priority queue!
2. How to find not only the distance to each node, but the shortest path?
   - One possibility: use the same trick as we did for breadth-first search – work backwards from the target node, only following edges that reduce the total distance sufficiently
   - A simpler approach: when we visit a node, remember which edge we came from to get to the node

Dijkstra's algorithm, made efficient

To find the closest unvisited node, we store the targets of all border edges in a priority queue

- The priority is the *total distance* to the node via that edge
- To make it easier to find paths, we also record the source of the border edge
- To determine which node to visit next, we just take the node with the smallest priority from the priority queue
- The node might already have been visited, in which case we ignore it

Whenever we visit a node, we will add the target of all of its outgoing edges to the priority queue

When the priority queue is empty, we are done!

Dijkstra's algorithm

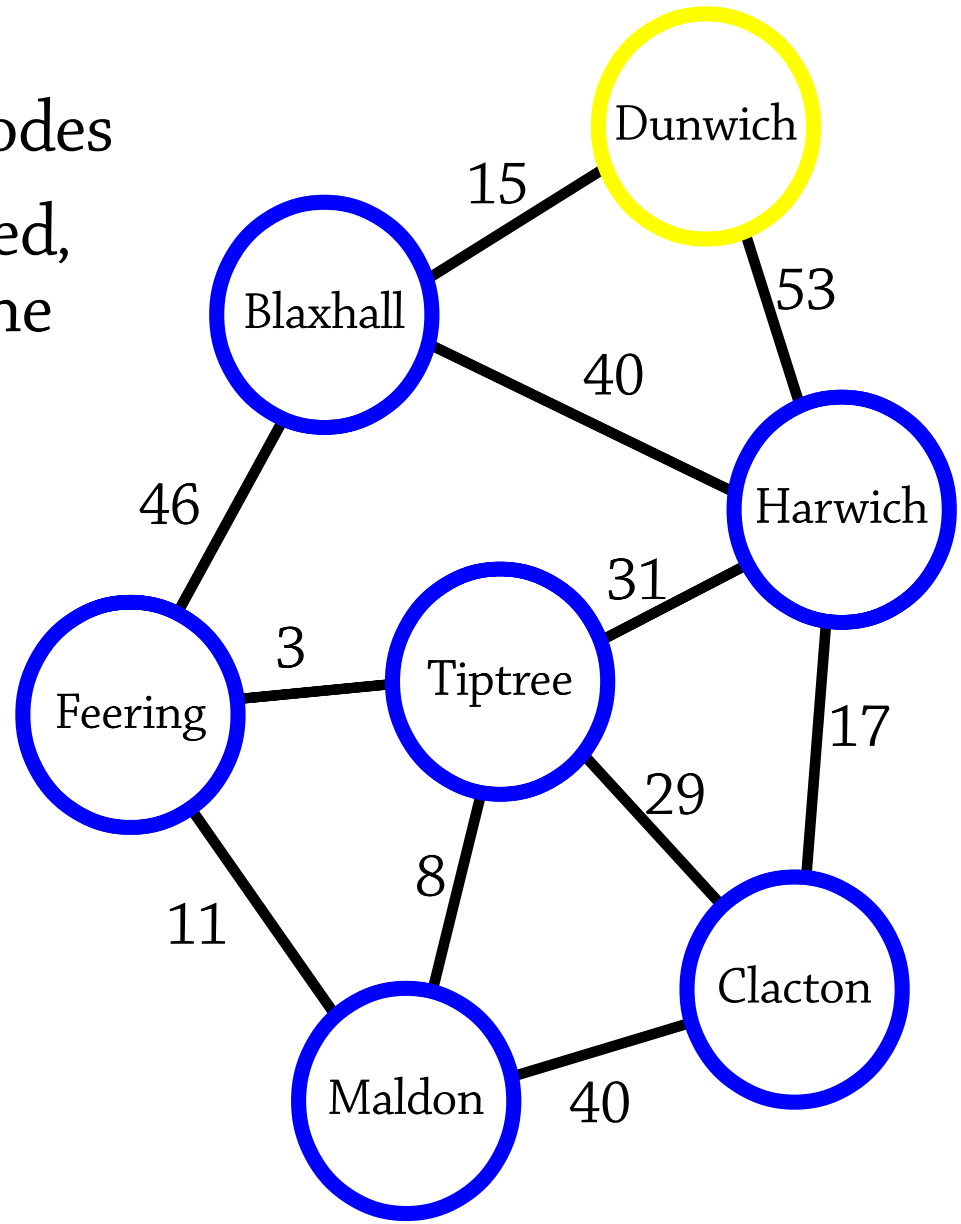
S is the visited set and Q is the priority queue of neighbouring nodes

Initially, no nodes have been visited, and the priority queue contains the start node:

S = {}
Q = {Dunwich 0}

The smallest element of Q is "Dunwich 0":

- Remove it from Q
- Add "Dunwich 0" to S
- Add Dunwich's outgoing edges to Q



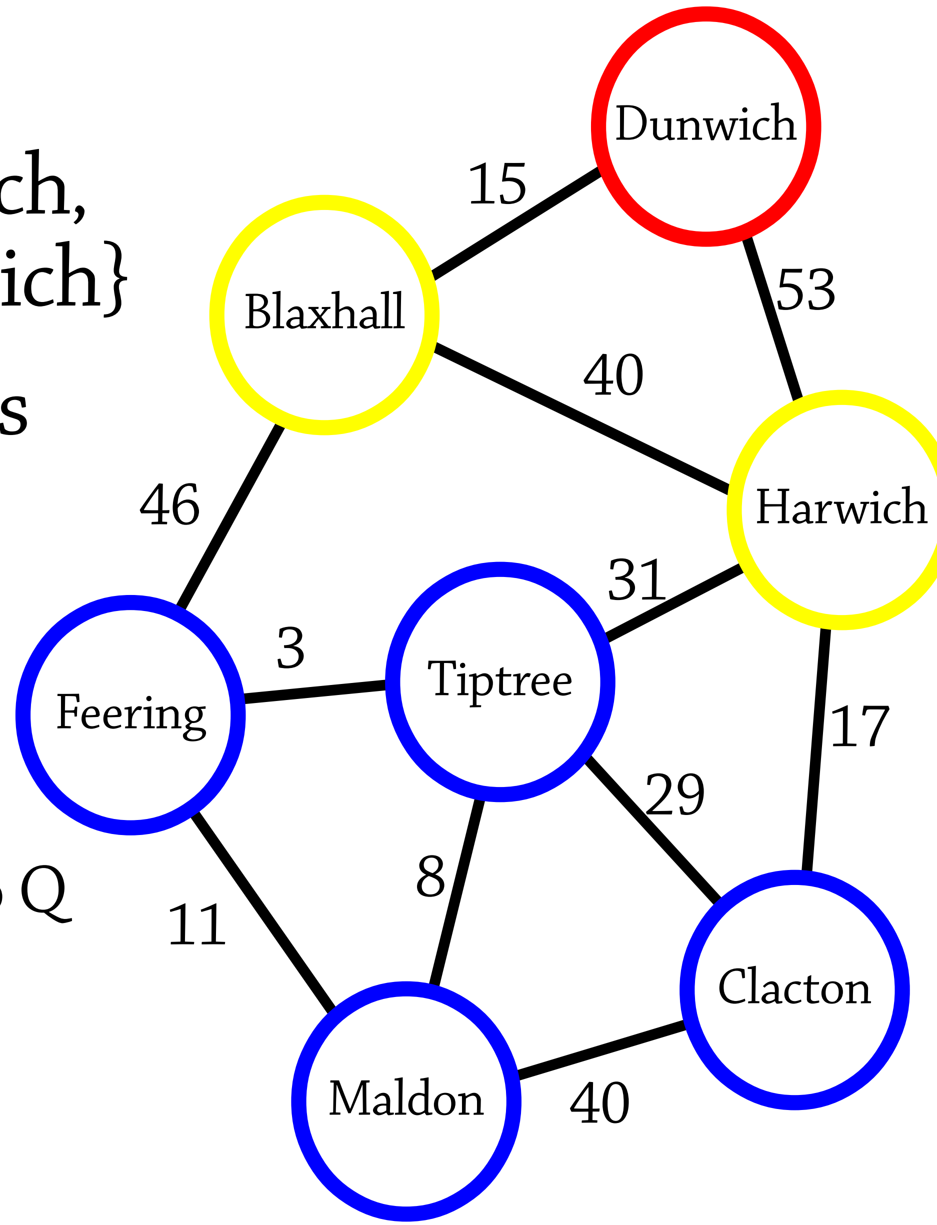
Dijkstra's algorithm

S = {Dunwich 0}

Q = {Blaxhall 15 via Dunwich,
Harwich 53 via Dunwich}

The smallest element of Q is "Blaxhall 15 via Dunwich":

- Remove it from Q
- Add "Blaxhall 15 via Dunwich" to S
- Add Blaxhall's outgoing edges to Q



Dijkstra's algorithm

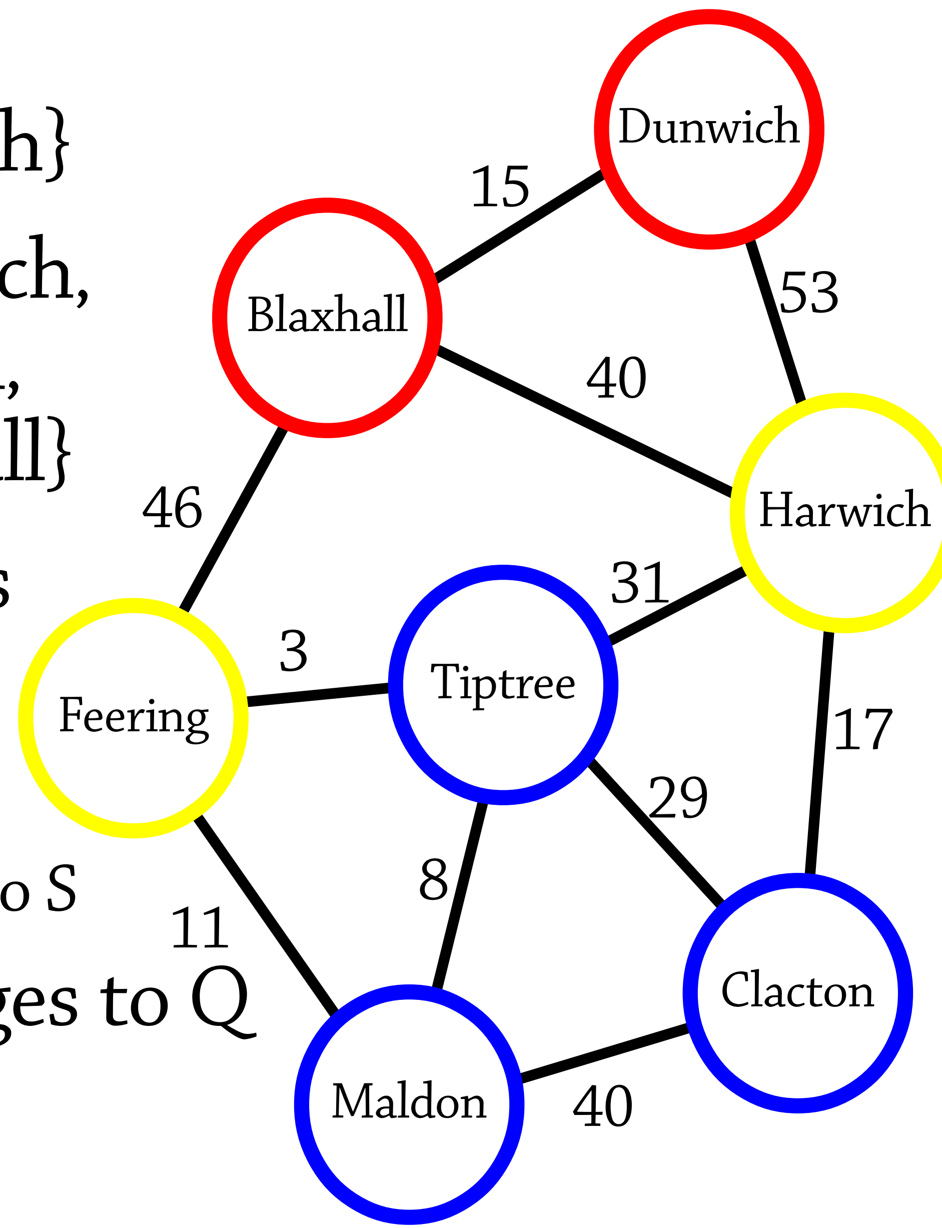
S = {Dunwich 0,
Blaxhall 15 via Dunwich}

Q = {Harwich 53 via Dunwich,
Feering 61 via Blaxhall,
Harwich 55 via Blaxhall}

The smallest element of Q is "Harwich 53 via Dunwich":

- Remove it from Q
- Add "Harwich 53 via Dunwich" to S

Add Harwich's outgoing edges to Q

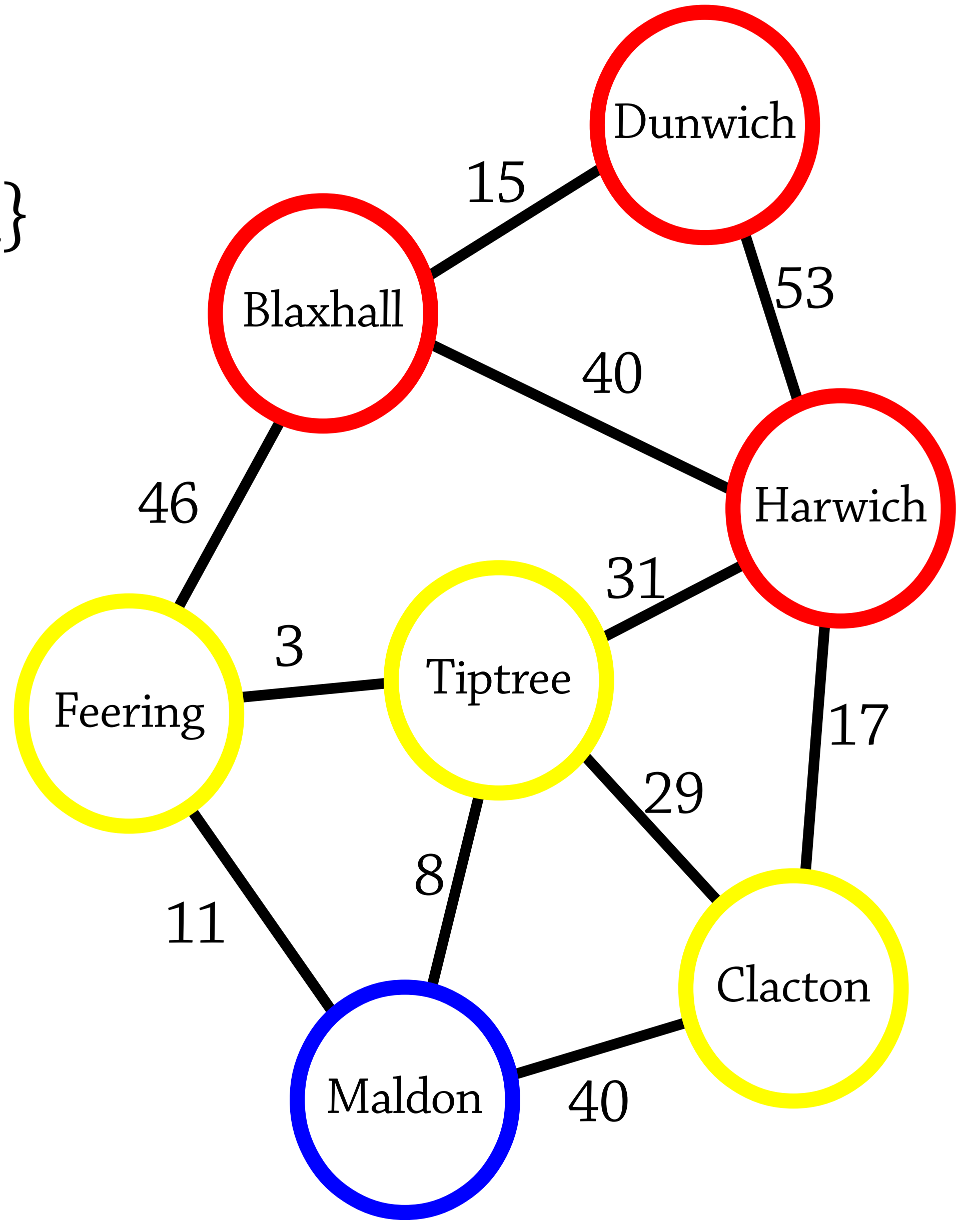


Dijkstra's algorithm

S = {Dunwich 0,
Blaxhall 15 via Dunwich,
Harwich 53 via Dunwich}

Q = {Feering 61 via Blaxhall,
Harwich 55 via Blaxhall,
Tiptree 84 via Harwich,
Clacton 70 via Harwich}

The smallest element of Q is "Harwich 55 via Blaxhall". But Harwich is already in S! So just ignore it.



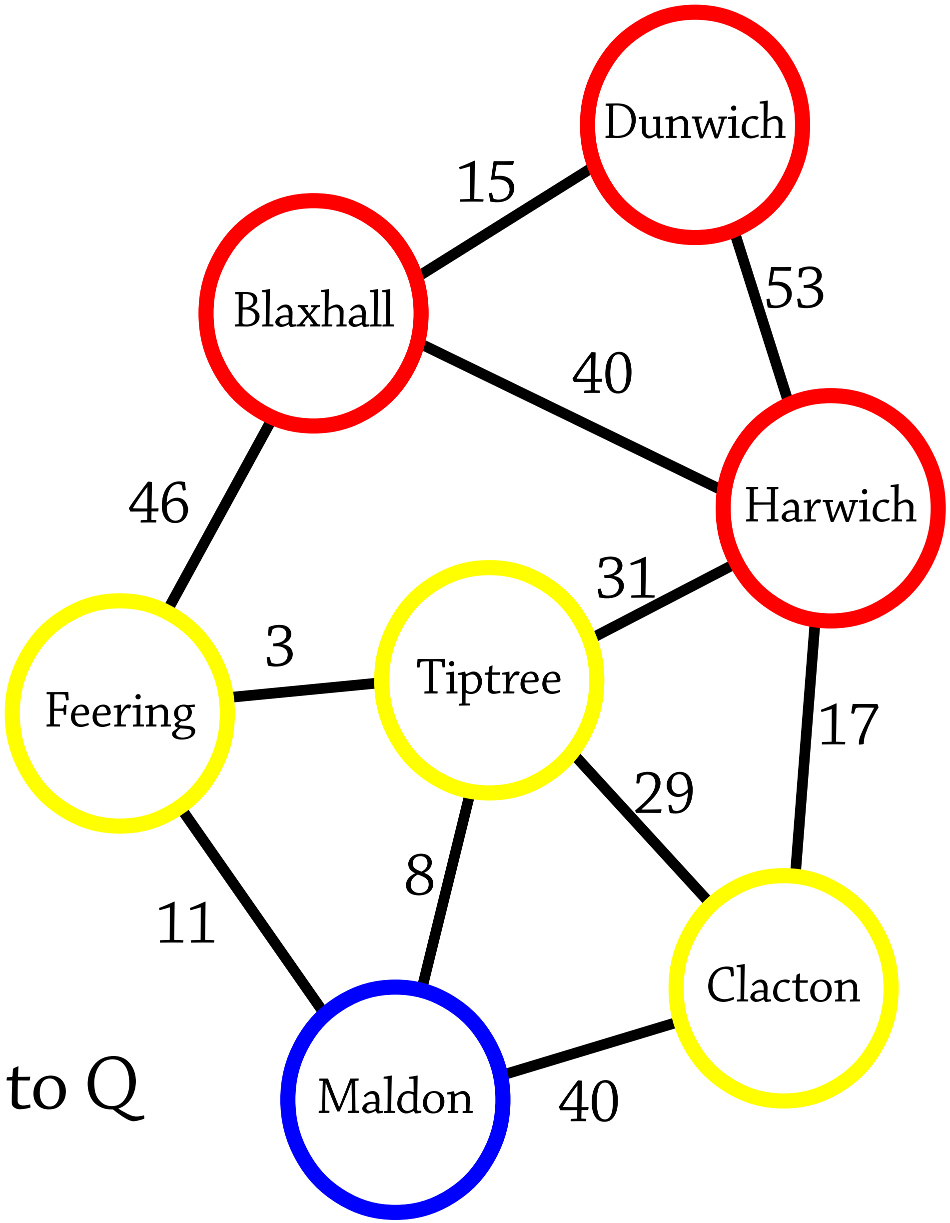
Dijkstra's algorithm

S = {Dunwich 0,
Blaxhall 15 via Dunwich,
Harwich 53 via Dunwich}

Q = {Feering 61 via Blaxhall,
Tiptree 84 via Harwich,
Clacton 70 via Harwich}

The smallest element of Q is "Feering 61 via Blaxhall":

- Remove it from Q
- Add "Feering 61 via Blaxhall" to S
- Add Feering's outgoing edges to Q



Dijkstra's algorithm

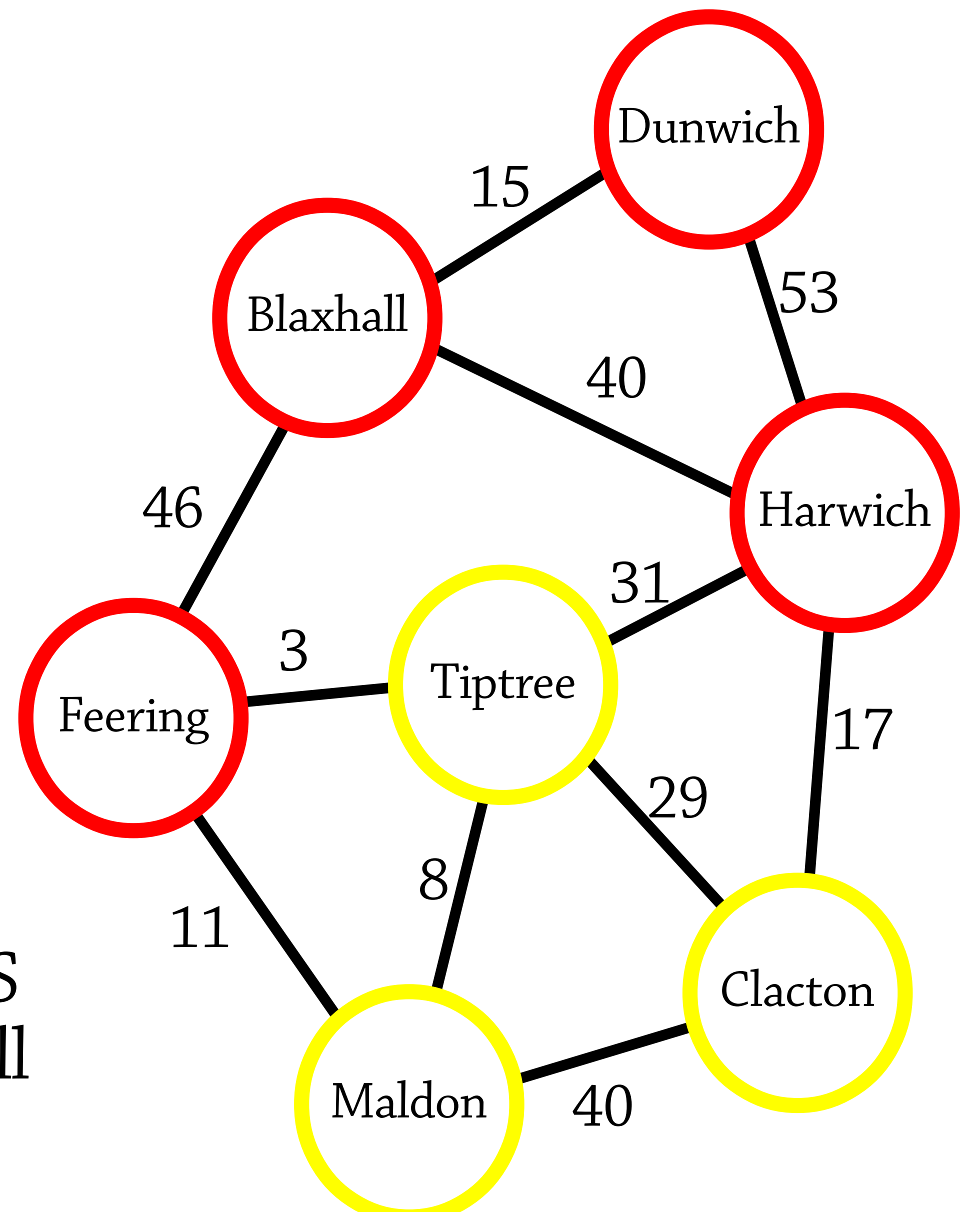
S = {Dunwich 0,
Blaxhall 15 via Dunwich,
Harwich 53 via Dunwich,
Feering 61 via Blaxhall}

Q = {Tiptree 84 via Harwich,
Tiptree 64 via Feering,
Maldon 72 via Feering,
Clacton 70 via Harwich}

Note: the shortest path to Feering is:

Dunwich → Blaxhall → Feering

and we can tell this by looking at S since we get to Feering via Blaxhall and to Blaxhall via Dunwich.



Dijkstra's algorithm, efficiently

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

While Q is not empty:

- Remove the node *x* from Q that has the smallest priority (distance), and let that distance be *d*
- If *x* is in S, do nothing
- Otherwise, add *x* to S with distance *d,* and for each outgoing edge $x \rightarrow y$, add y to Q with priority d + (*weight of edge* $x \rightarrow y$)

Implementation notes:

- Each entry in Q and S should also record “via” information, in order to easily find paths
- *S* can be implemented via a map, or by adding extra fields to the node class

Each edge in the graph is processed once, and added to Q at most once, so complexity is O(n log n) where n = number of edges in graph. Good!

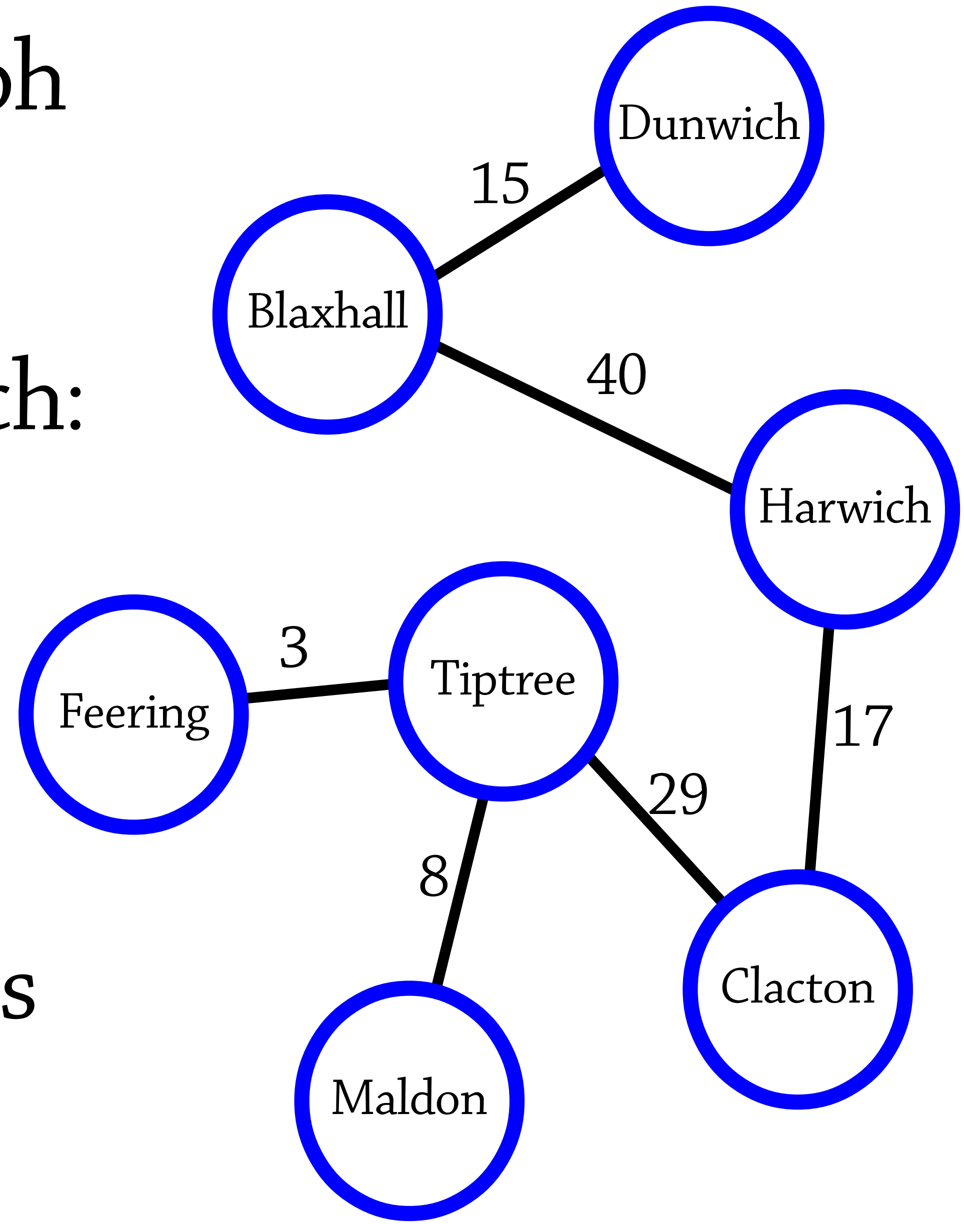
Prim's algorithm

Minimum spanning trees

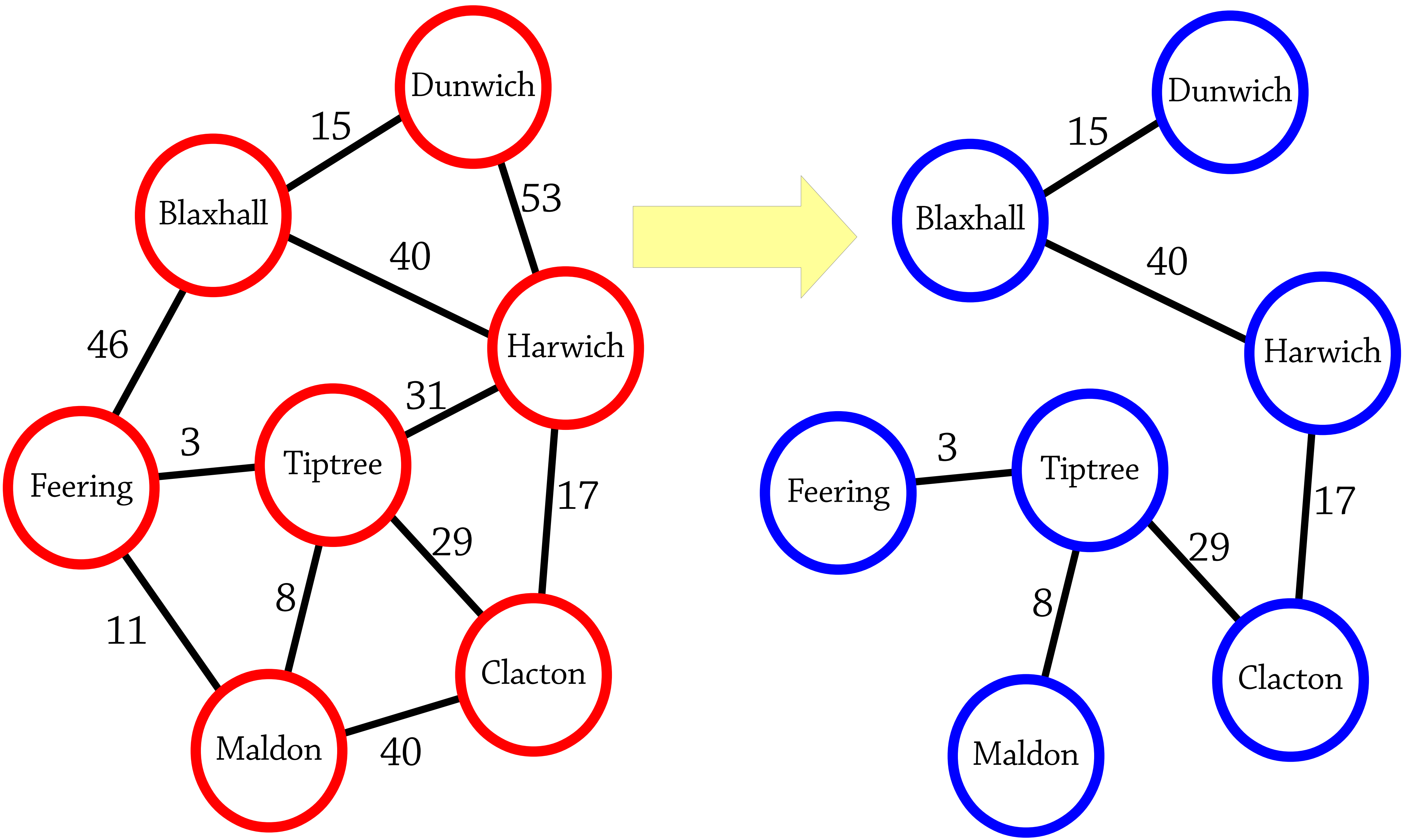
A *spanning tree* of a graph is a subgraph (a graph obtained by deleting some of the edges) which:

- 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

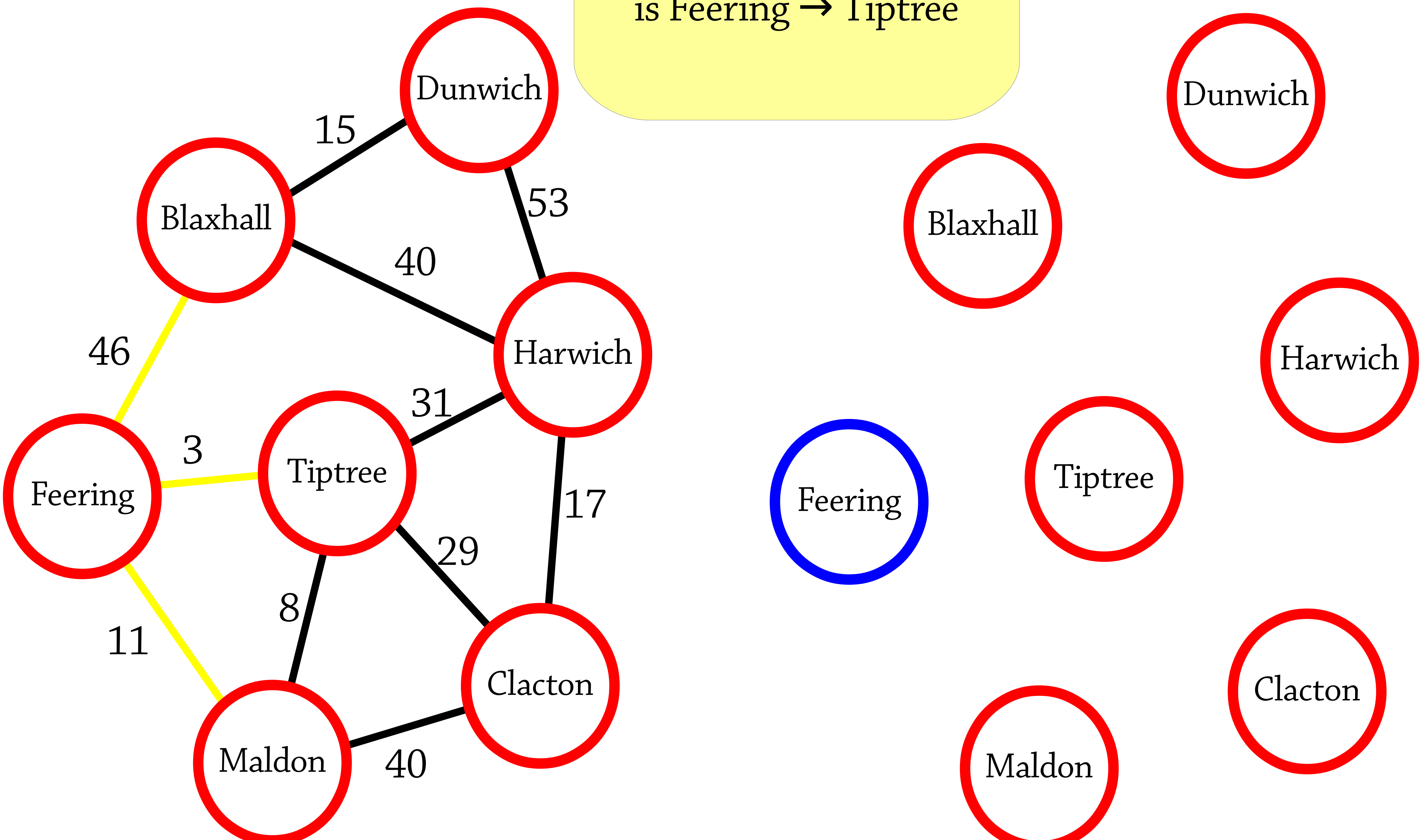
We call an edge a *border edge* if it connects a node in S to a node not in S

While there is a node not in S:

- Pick the *lowest-weight* border edge
- Add that edge to the spanning tree, and add the newly-connected node to S

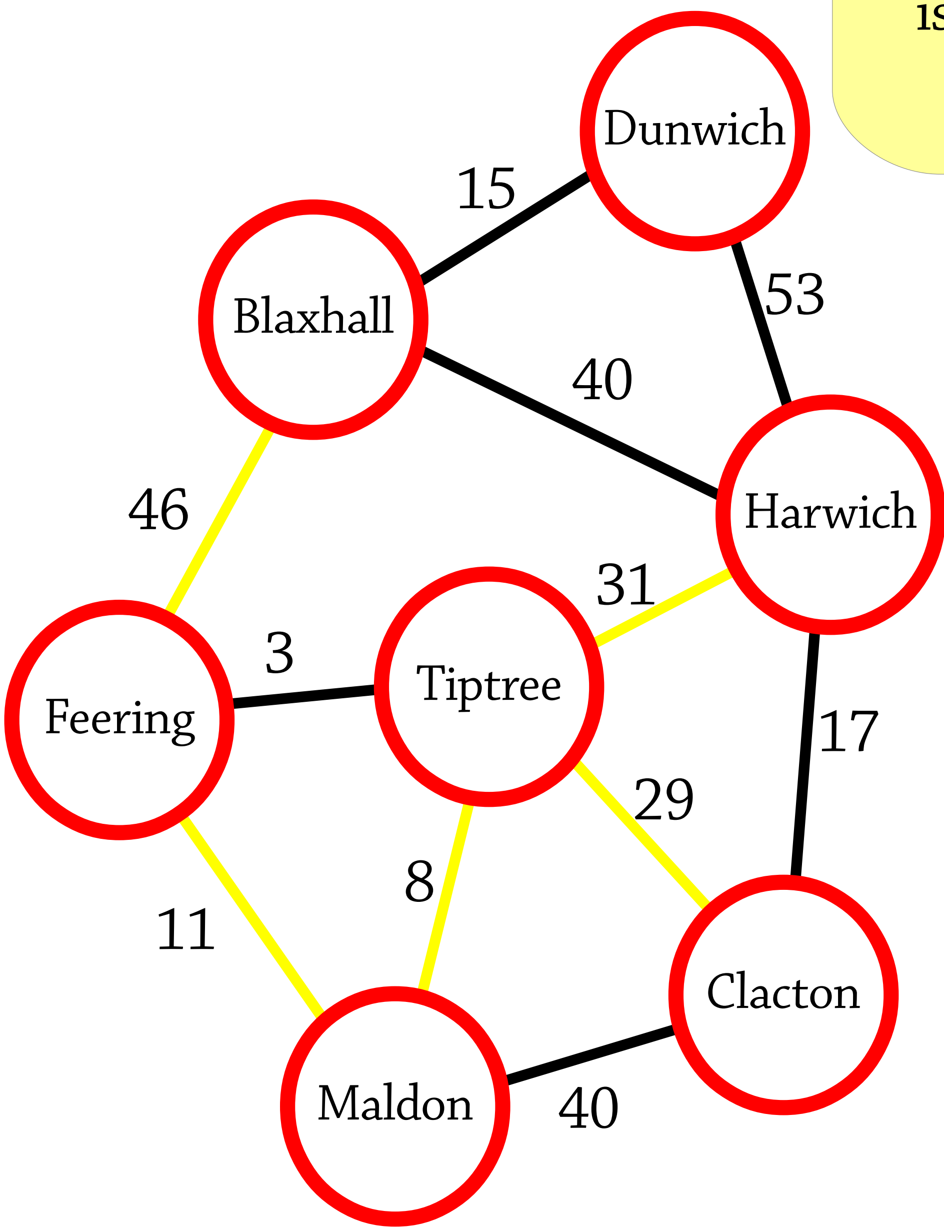
Minimun ees

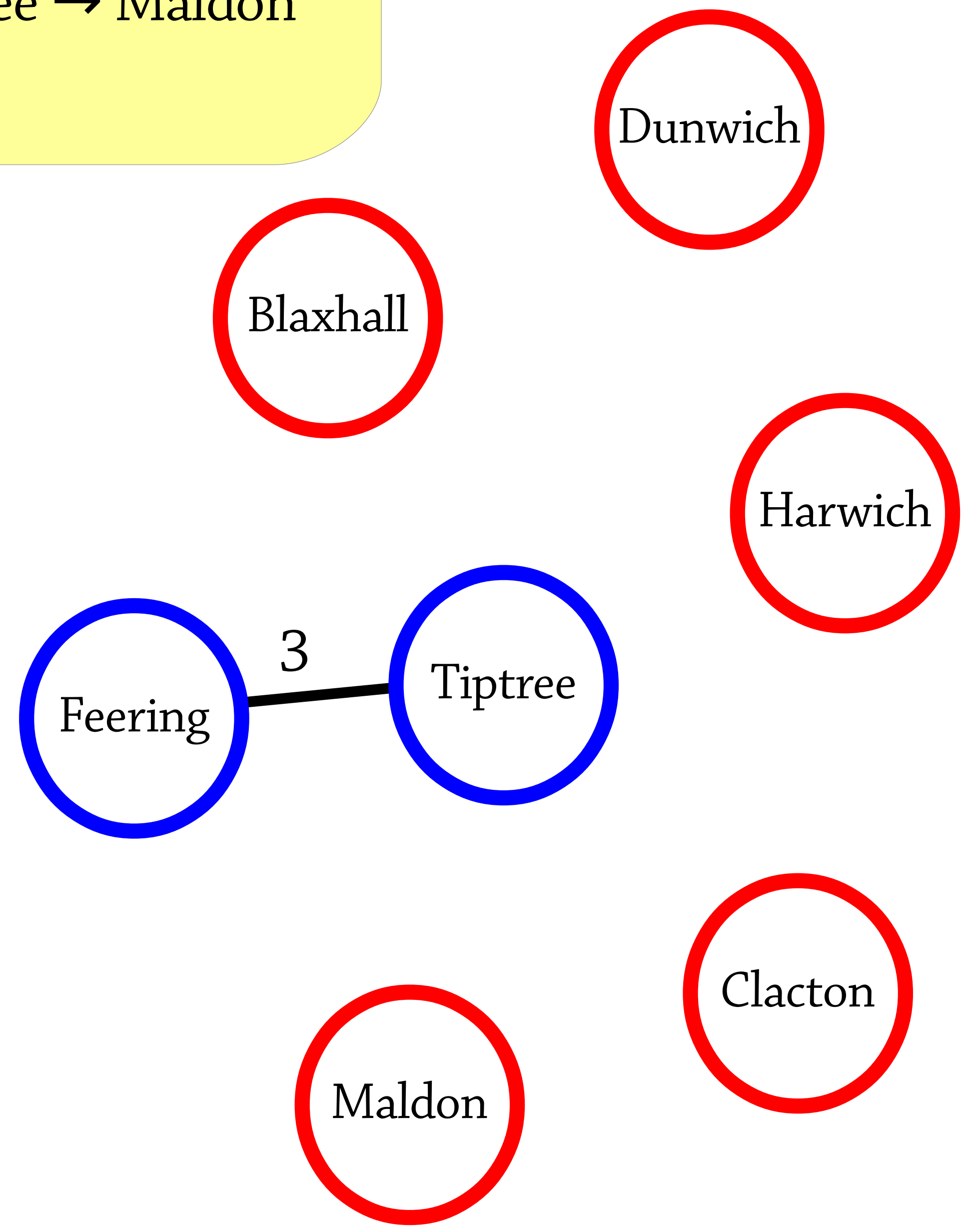
S = {Feering}
Lowest-weight
border edge
is Feering → Tiptree



Minimum … ees

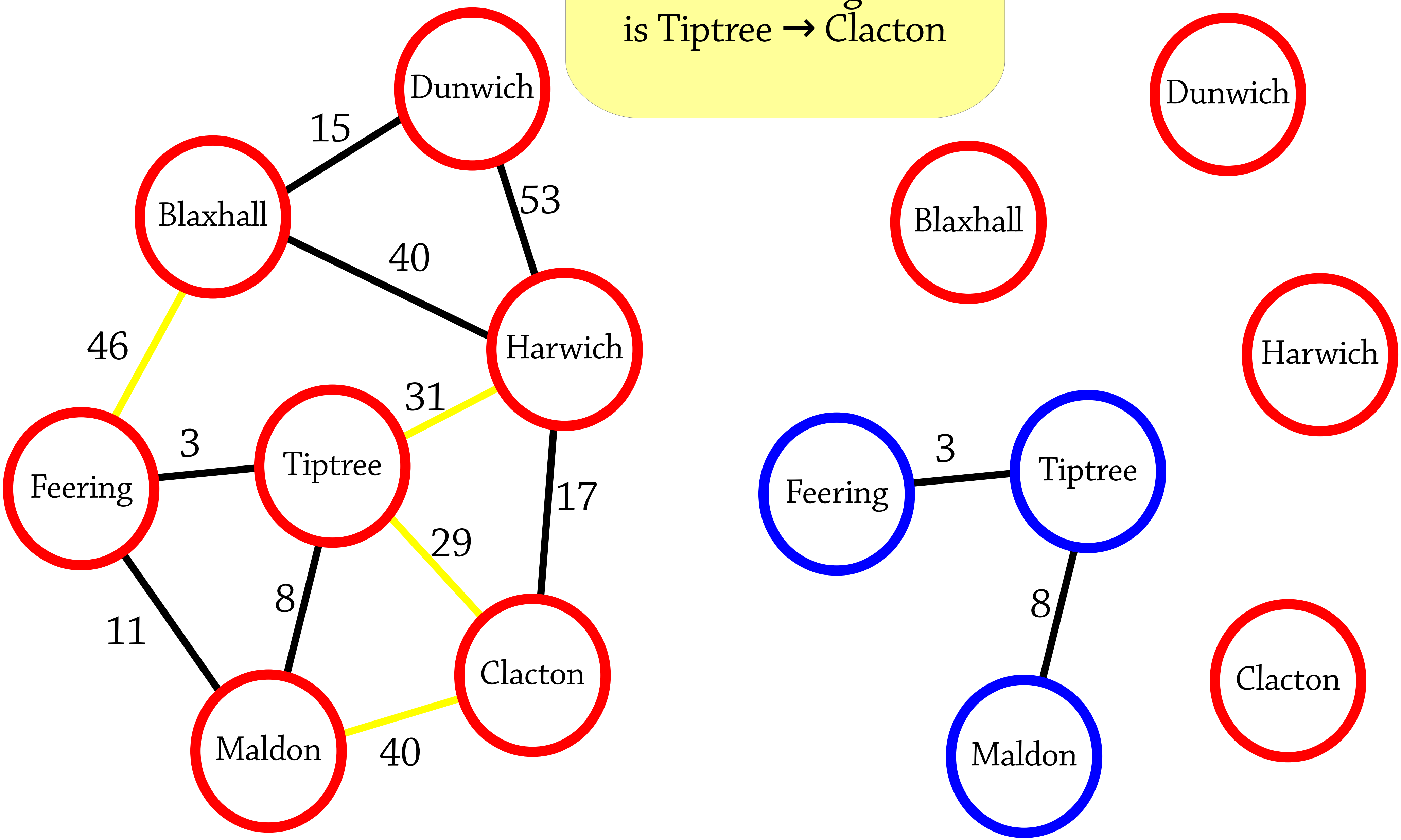
S = {Feering, Tiptree}
Lowest-weight
border edge
is Tiptree → Maldon





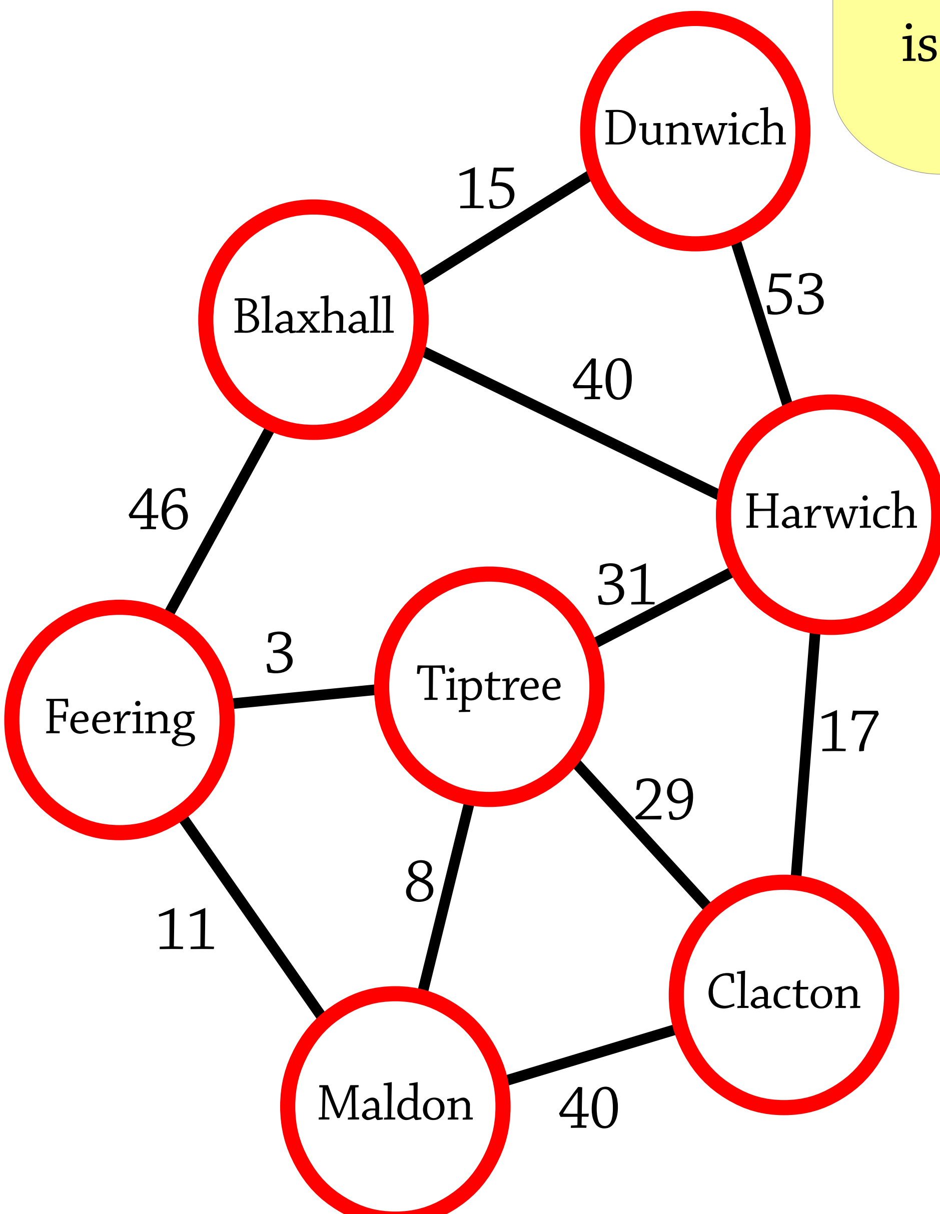
Minimun ees

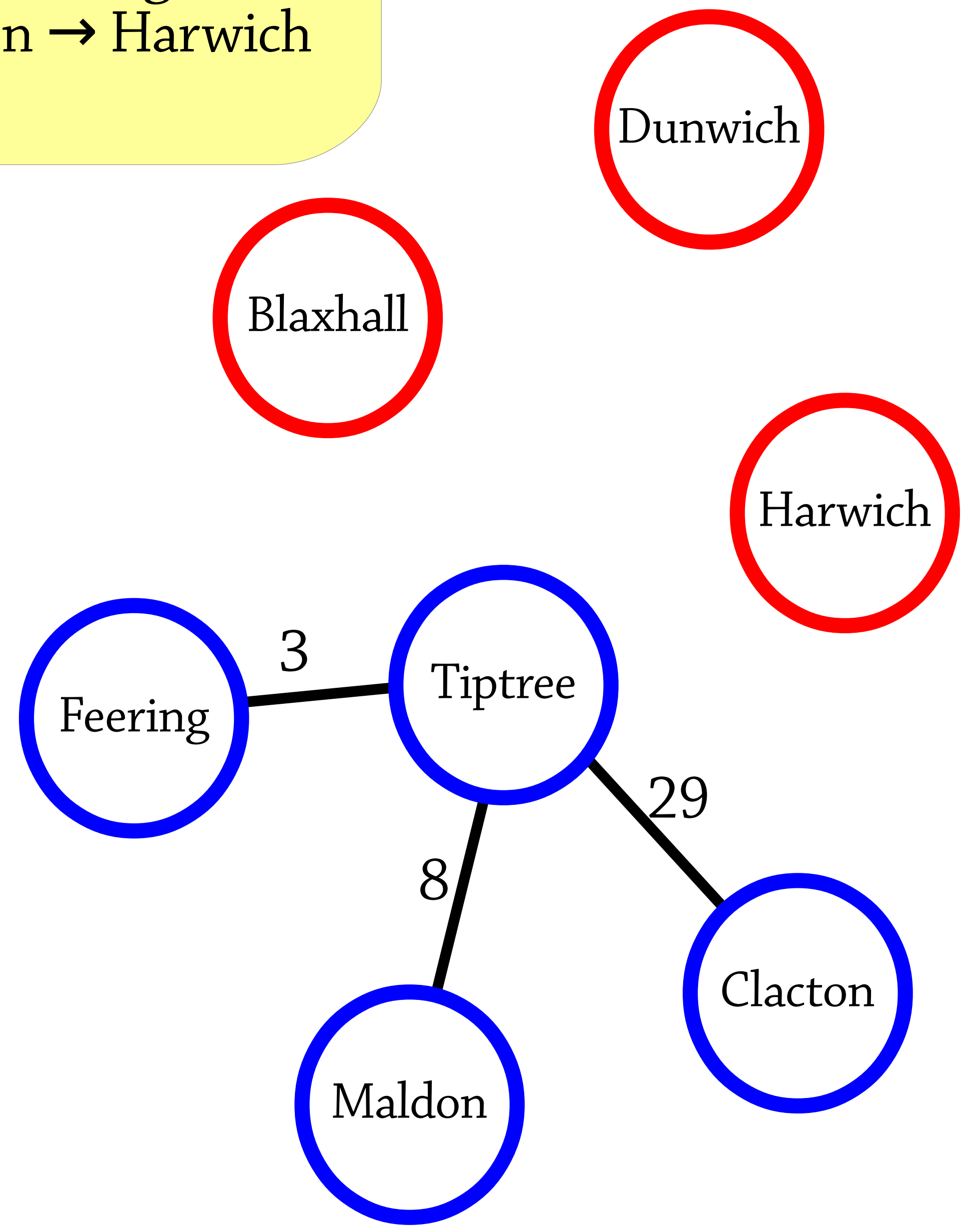
S = {Feering, Tiptree, Maldon}
Lowest-weight border edge
is Tiptree → Clacton



Minimun ees

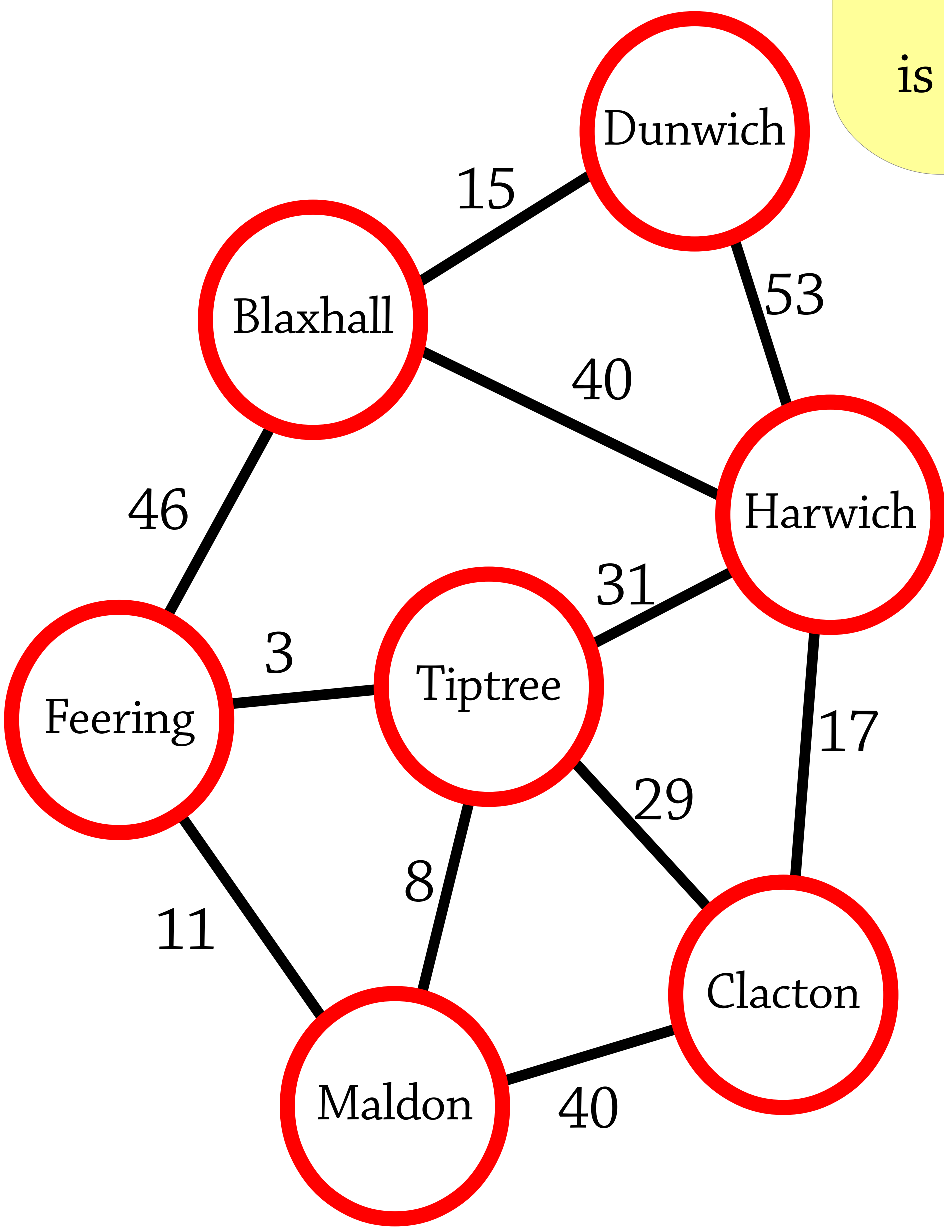
S = {Feering, Tiptree, Maldon, Clacton}
Lowest-weight
border edge
is Clacton → Harwich

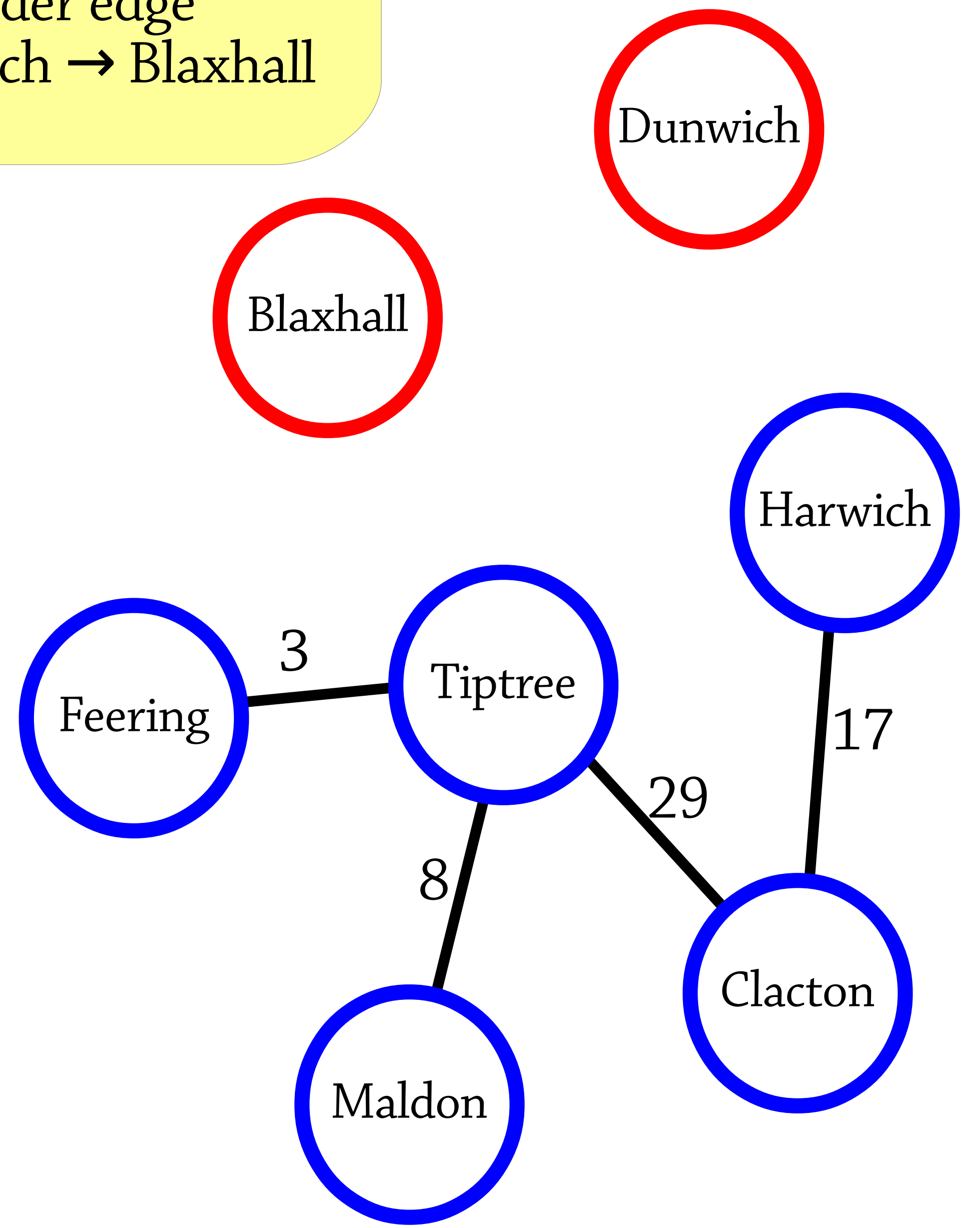




Minimum ees

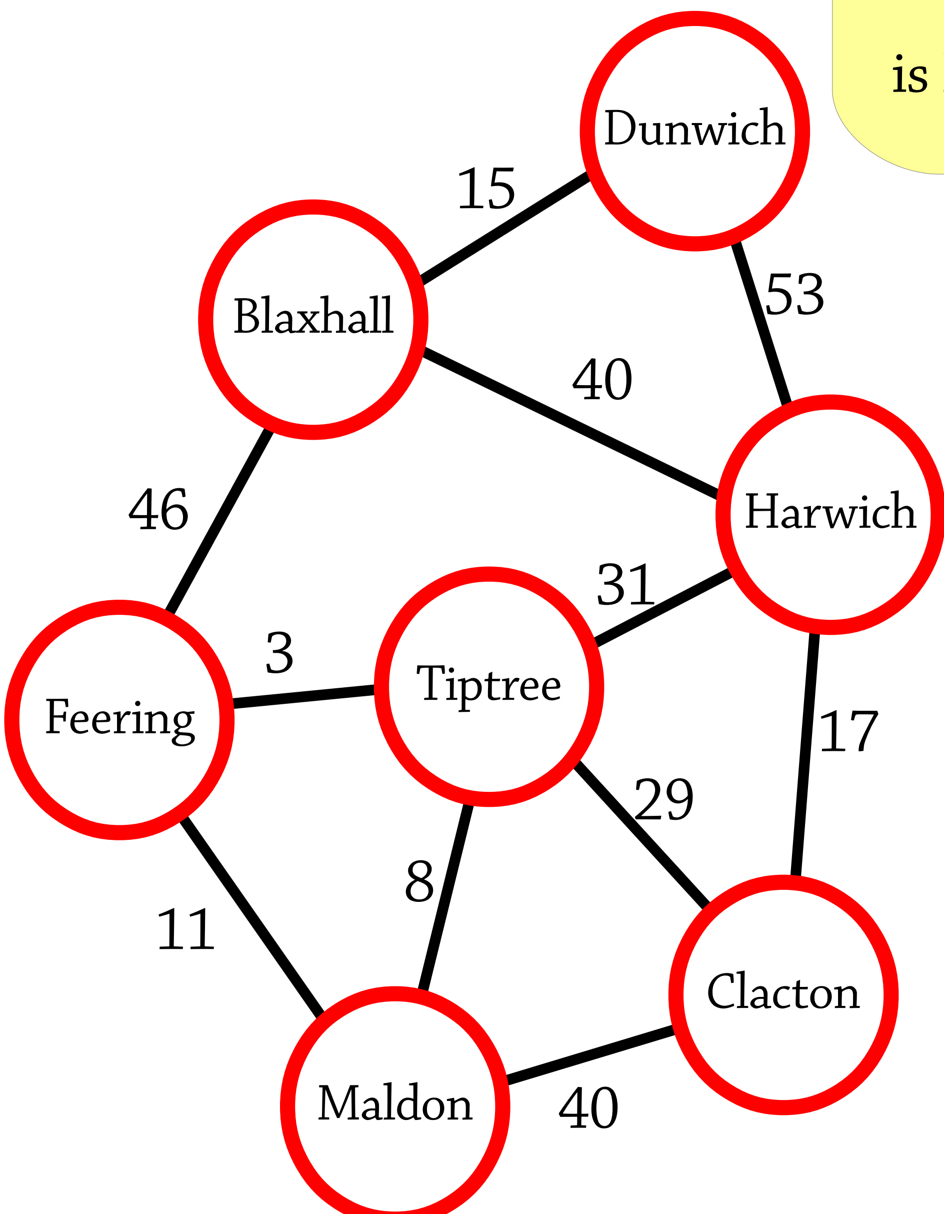
S = {Feering, Tiptree, Maldon, Clacton, Harwich}
Lowest-weight border edge is Harwich → Blaxhall

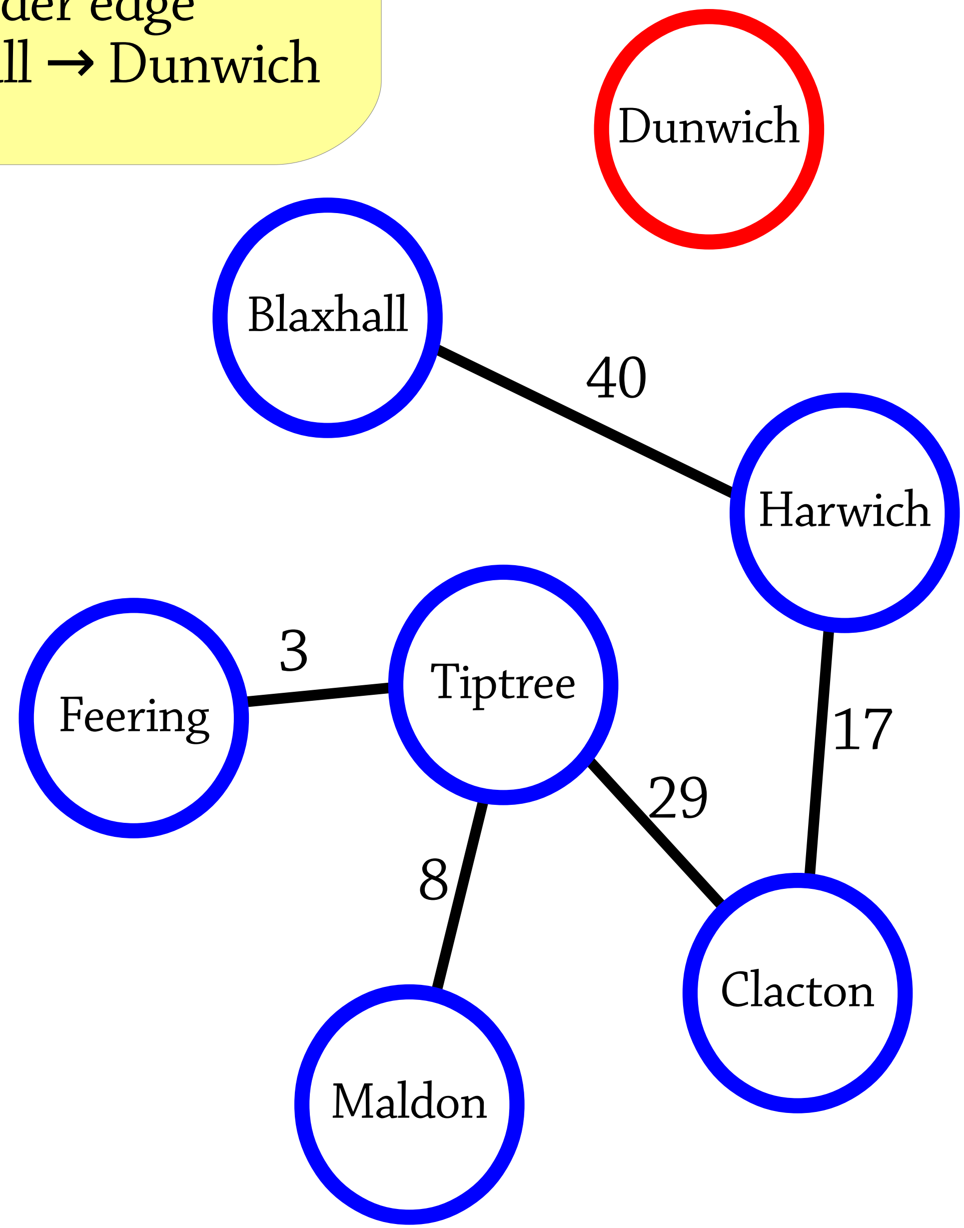




Minimum … ees

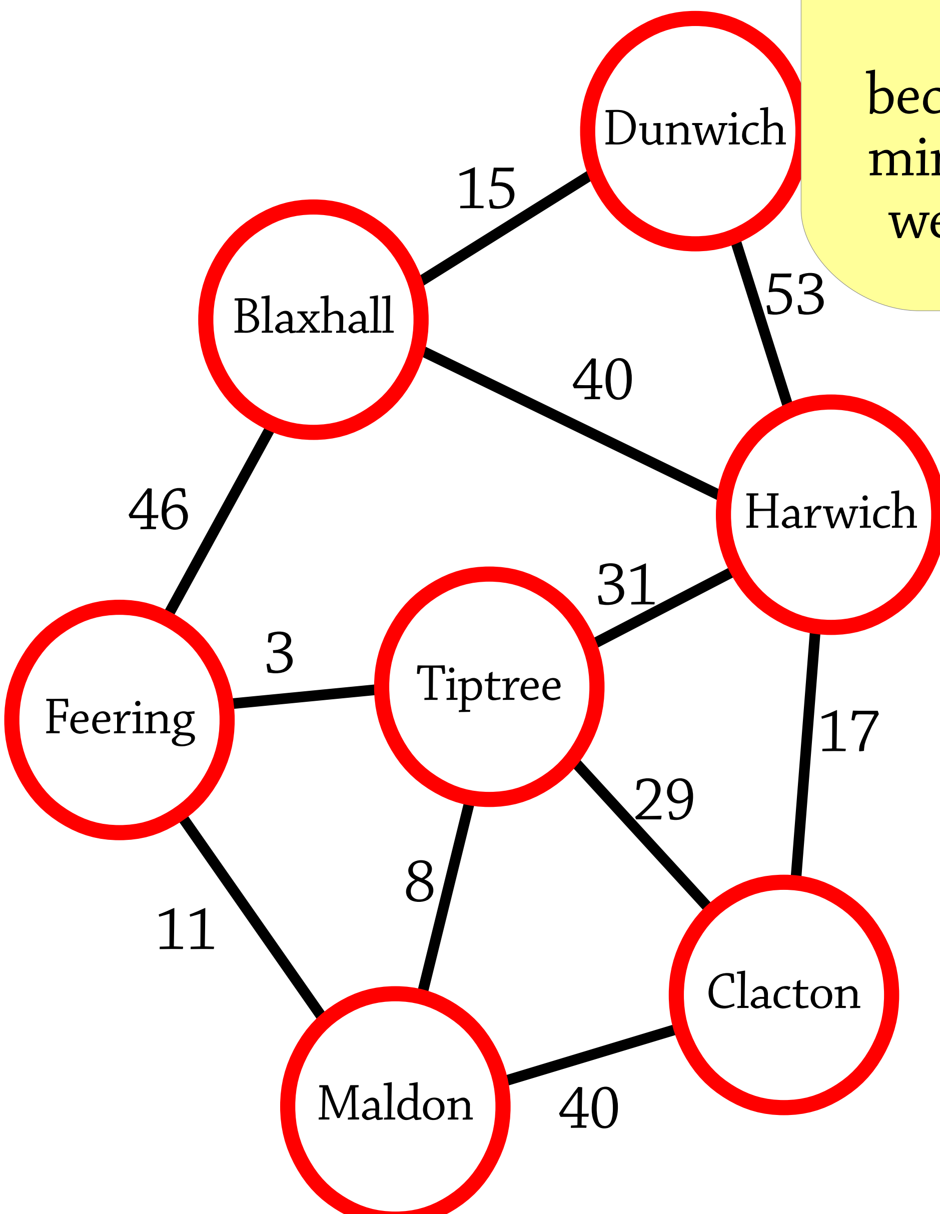
S = {Feering, Tiptree, Maldon, Clacton, Harwich, Blaxhall}
Lowest-weight border edge
is Blaxhall → Dunwich

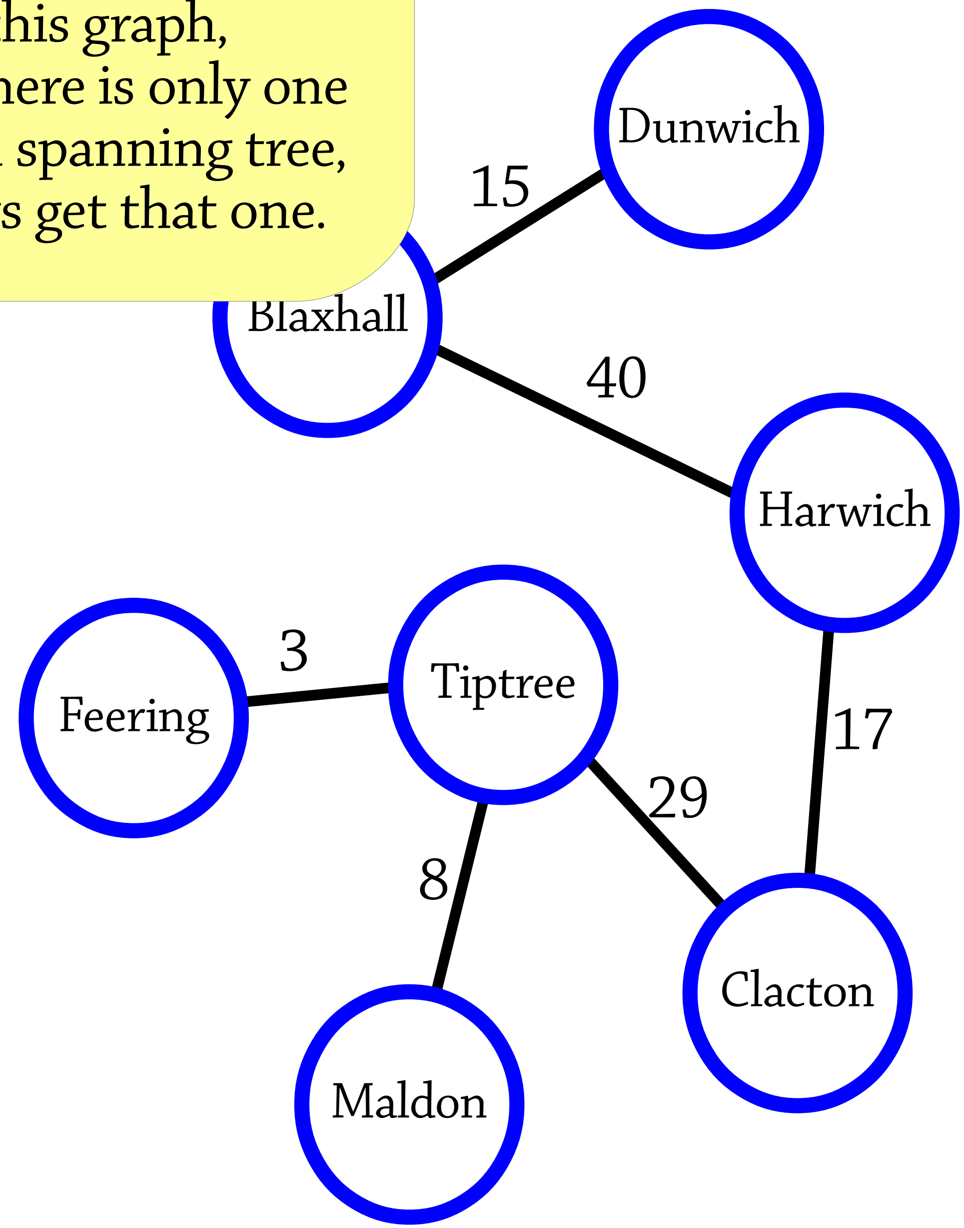




Minimu… …ees

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(n) time if we're not careful! Then Prim's algorithm will be $O(n^2)$

To implement Prim's algorithm, use a priority queue containing all border edges

- Whenever you add a node to S, add all of its edges (that are not to nodes in 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(n log n) :)

Why does it work? **(not on exam)**

Proof sketch (drawing a diagram helps):

Suppose that Prim's algorithm gives a non-minimal spanning tree, and imagine that we are at the earliest point in the algorithm where it goes wrong:

- We have a minimum spanning tree T for S; the smallest border edge *e* goes to node *x* (not in S)
- T can be extended to a minimum spanning tree T' for the whole graph, but T plus *e* cannot

We will show that T plus *e* can be extended to a minimal spanning tree, which is a contradiction:

- Observation: in a tree, there is exactly one path between every pair of nodes.
- Therefore, in T', there is exactly one path from an arbitrary node in S to *x*
- This path must go through a border edge of S. Remove this border edge; now *S* is disconnected from x. Add the edge *e*; this results in a spanning tree. This new spanning tree is minimal, since T' is minimal and *e* had minimum weight among all border edges.

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 (Dijkstra allows weights to be zero but not 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

Dijkstra's and Prim's algorithms are based on the idea of choosing the "best" border edge

- This is called a *greedy algorithms* – it repeatedly finds the "best" next element
- Common style of algorithm design when trying to find the "best" solution to a problem; finds at least a locally optimal solution – but for the algorithms today is globally optimal

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

A* search
(not on exam)

A problem with Dijkstra's algorithm

We can use Dijkstra's algorithm to find the shortest route from A to B

But it explores *all* nodes in the graph that are closer than B!

A person planning a route would try to move *towards* B

Gothenburg to Stockholm?

The A* algorithm

Often we have a notion of *distance* in a graph

- e.g., Gothenburg to Stockholm is 400km as the crow flies
- No possible route can be shorter than this!

A* uses distance to guide the search towards the destination

- Try to pick edges that reduce the distance to the destination, avoid edges that increase the distance
- But still guaranteeing to find the shortest path!

The A* algorithm

We assume there is a function *h(x)* (the *heuristic*)

- In our example, h(x) is the distance from x to Stockholm as the crow flies

When we take an edge x → y, we are interested not only in the weight but also in how *h* changes:

- If h(y) > h(x), we moved *away* from the target (bad); if h(y) < h(x), we moved *towards* the target (good)

Idea: give a bonus to edges that reduce the value of h!

- If we have an edge from *x* to *y*, we increase its weight by h(y)-h(x) – so "good" edges get cheaper and "bad" edges get more expensive

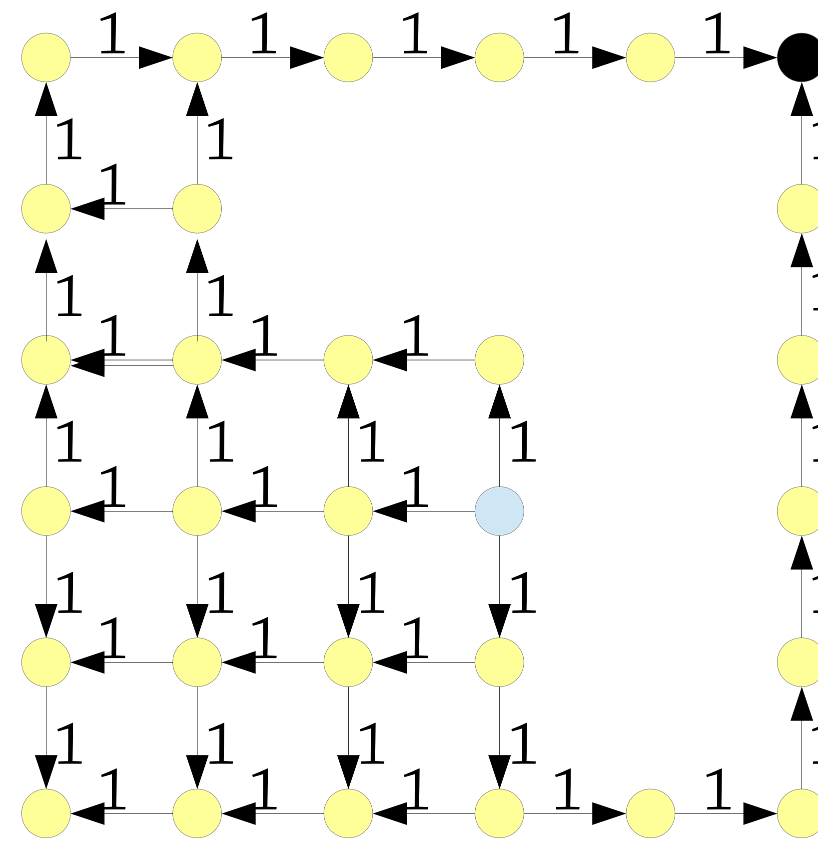
Then we run Dijkstra's algorithm on this new graph!

A* – an example

A* was originally invented for robot motion planning! Here is a floor with an obstacle in. (Edges given directions for simplicity.)

The robot wants to get from the blue node to the black node.

The shortest path has weight 9 – Dijkstra's algorithm will explore the whole graph!



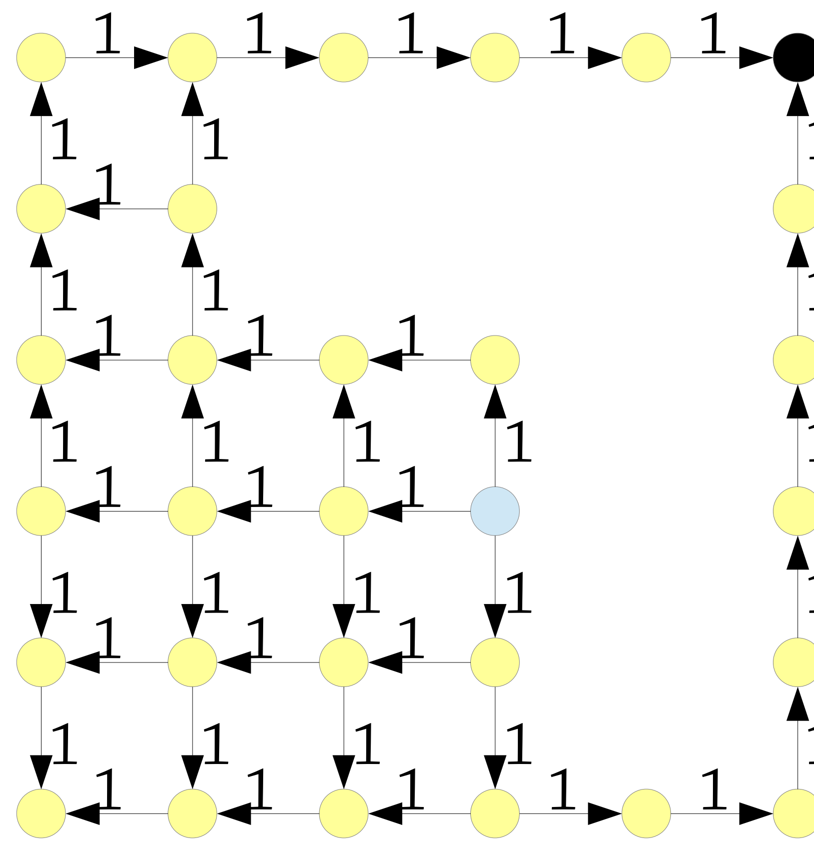
A* – an example

Now let's use the heuristic h(x) = "Manhattan distance" (x coordinate + y coordinate) from x to black node

e.g., h(blue node) = 5, because black node is 2 right and 3 up from black node

If there is an edge from x to y, we add h(y)-h(x), so for this graph:

- If the edge goes up or right, we decrease its weight by 1
- If it goes down or left, we increase its weight by 1

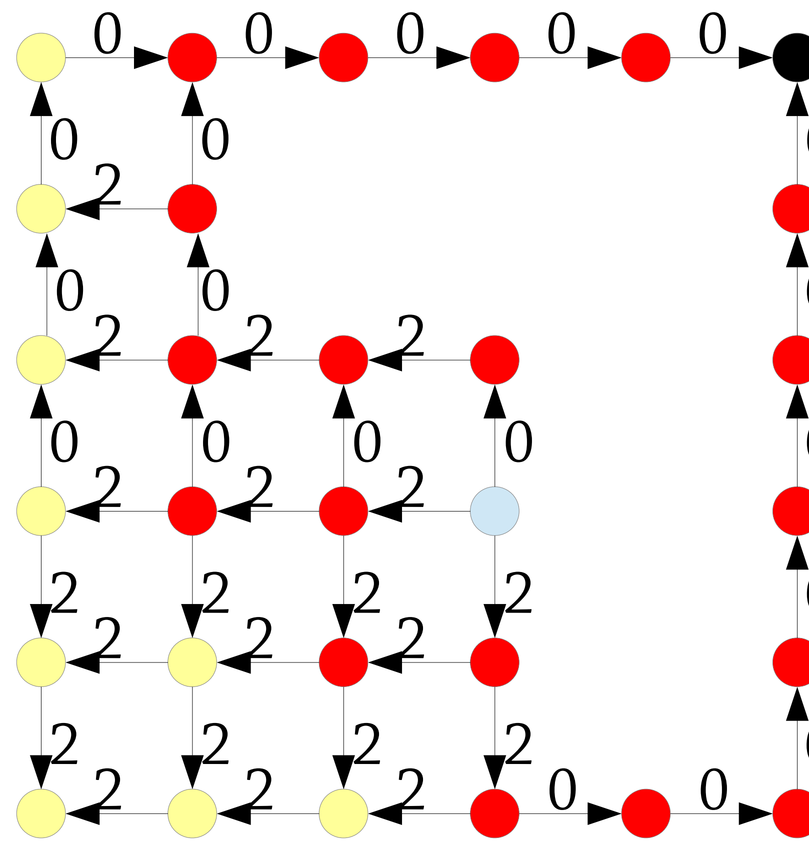


A* – an example

In the new graph, the up and right edges have weight 0, and the left and down edges have weight 2

The shortest path has weight 4 – you have to go left twice

The area the algorithm explores is highlighted in red



Gothenburg to Stockholm

A* – why does it work?

In A^*, we change the weights of all the edges – are we still going to get the shortest path for the original graph? Yes!

Let's look at a path $a \rightarrow b \rightarrow c$:

- Assume the weights of the two edges are w_{ab} and w_{bc}
- A^* modifies the weights to $w_{ab} + h(b) – h(a)$ and $w_{bc} + h(c) – h(b)$
- The weight of the path becomes $w_{ab} + h(b) - h(a) + w_{bc} + h(c) - h(b) =$ $w_{ab} + w_{bc} + h(c) – h(a)$
- In other words, the weight of the path increases by $h(c) – h(a)$. In fact, the same thing happens for paths of any length!

So the total weight of each path from *source* to *target* is increased by h(target) - h(source) – a constant

The weight of each path changes, but by the same amount – so the shortest path is still the shortest path!

Some technicalities

Dijkstra's algorithm doesn't work if there is an edge with a negative weight

So we'd better be sure that modifying the weights never makes them negative

If we have an edge from x to y of weight w, the new weight is w+h(y)-h(x), so this is fine as long as:

- $h(x) \leq w + h(y)$

That is, by following an edge you can't reduce the distance to the target by more than the weight of that edge – this is true e.g. of distance in maps

A* – summary

An extension of Dijkstra's algorithm that uses distance information to move *towards* the destination instead of exploring in all directions

- Still guaranteed to find the shortest path

Works very well in practice!

If we multiply the heuristic function by a constant, we can direct the search less or more aggressively

- But if we're too aggressive and the heuristic function returns too large values, the edge weights will become negative
- In this case we can't use Dijkstra's algorithm, but there is a more complex version of A* we can use instead
- But this aggressive version of A* can find suboptimal paths