Dijkstra's algorithm
Prim's algorithm

The (weighted) shortest path problem

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

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

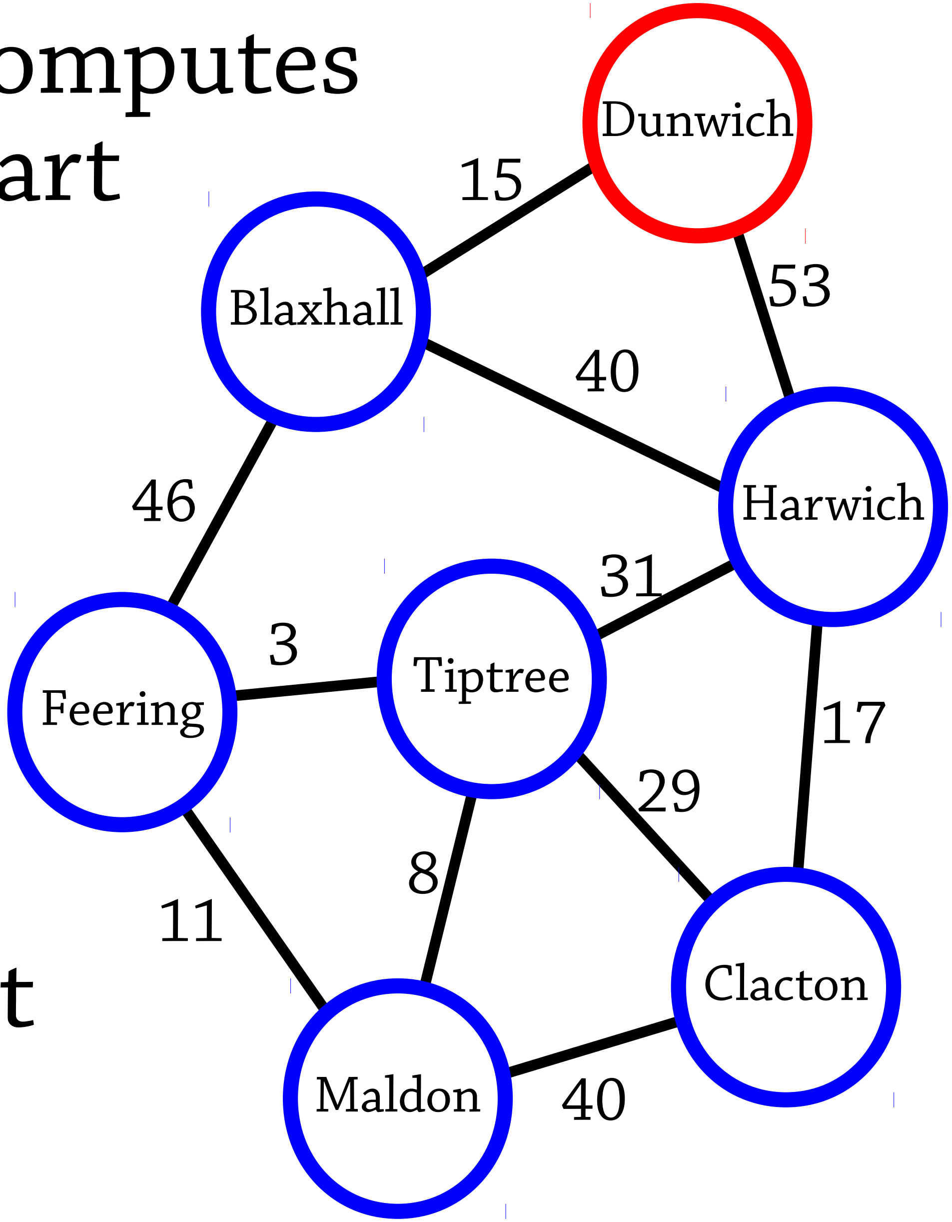
Most common approach: *Dijkstra's algorithm*, which works when all edges have positive weight

Dijkstra's algorithm

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

Idea: maintain a set S of nodes whose distances we know, and their distances

Initially, S only contains the start node, with distance 0



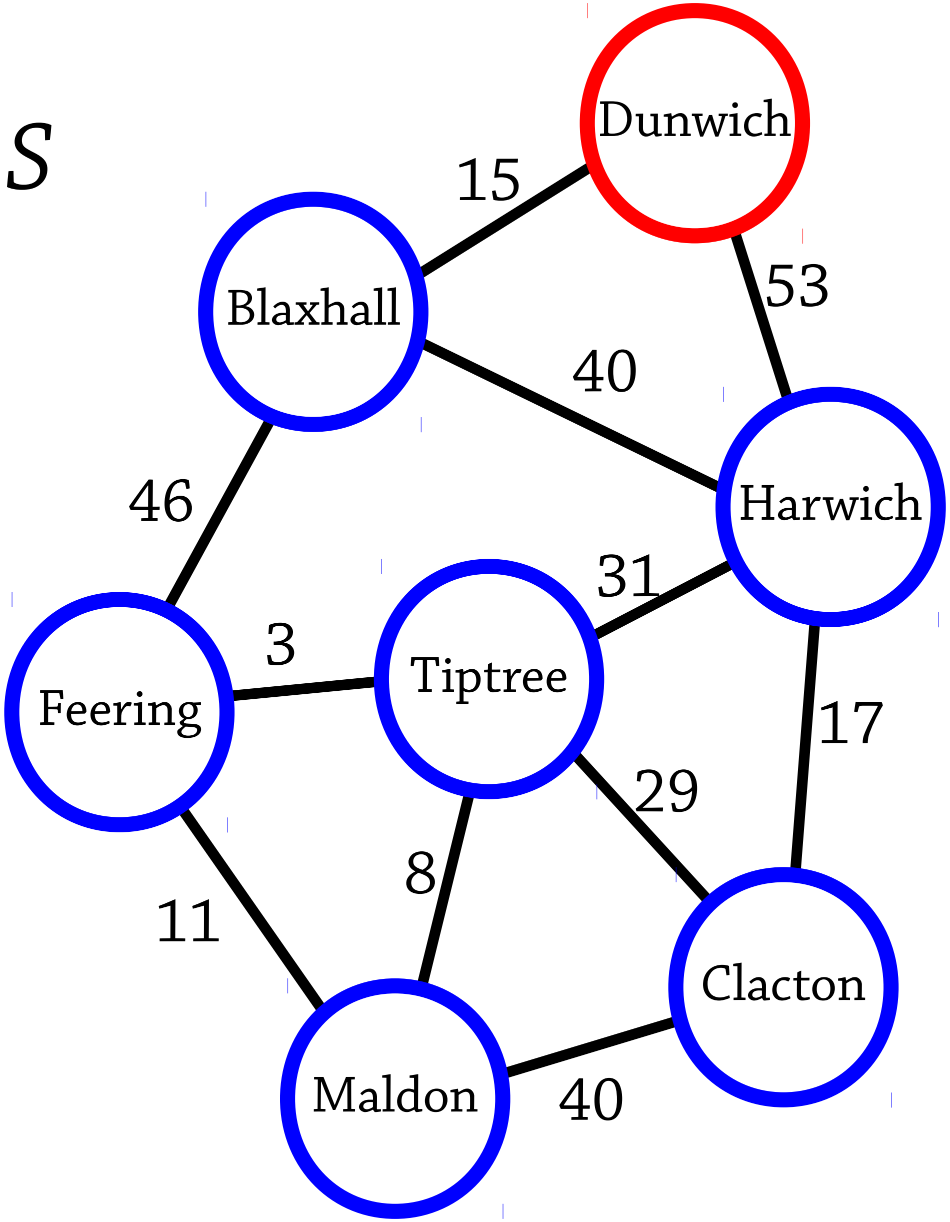
Dijkstra's algorithm

At each step: find the *closest node that's not in S*

This node must be adjacent to a node in S (why?)

Hence the path to that node must consist of:

- Taking the shortest path to some node in S, then
- taking a single edge to get to the new node



Dijkstra's algorithm

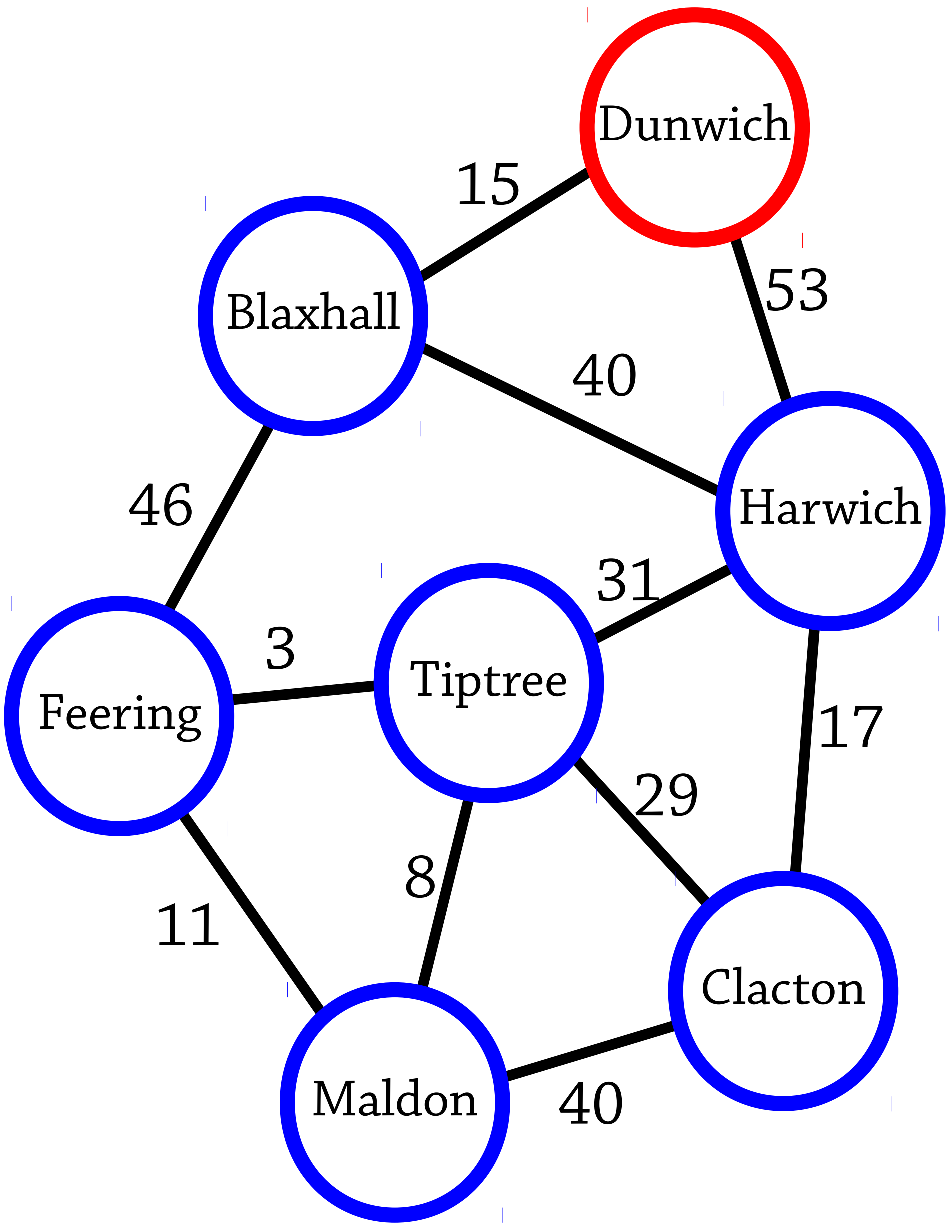
For each node *x* in S, and each neighbour *y* of *x*:

- Add the distance to *x* and the distance from *x* to *y*

Whichever node *y* has the shortest distance, add it to S!

- This is the closest node not in S (what is the path to this node?)

Repeat until all nodes are in S

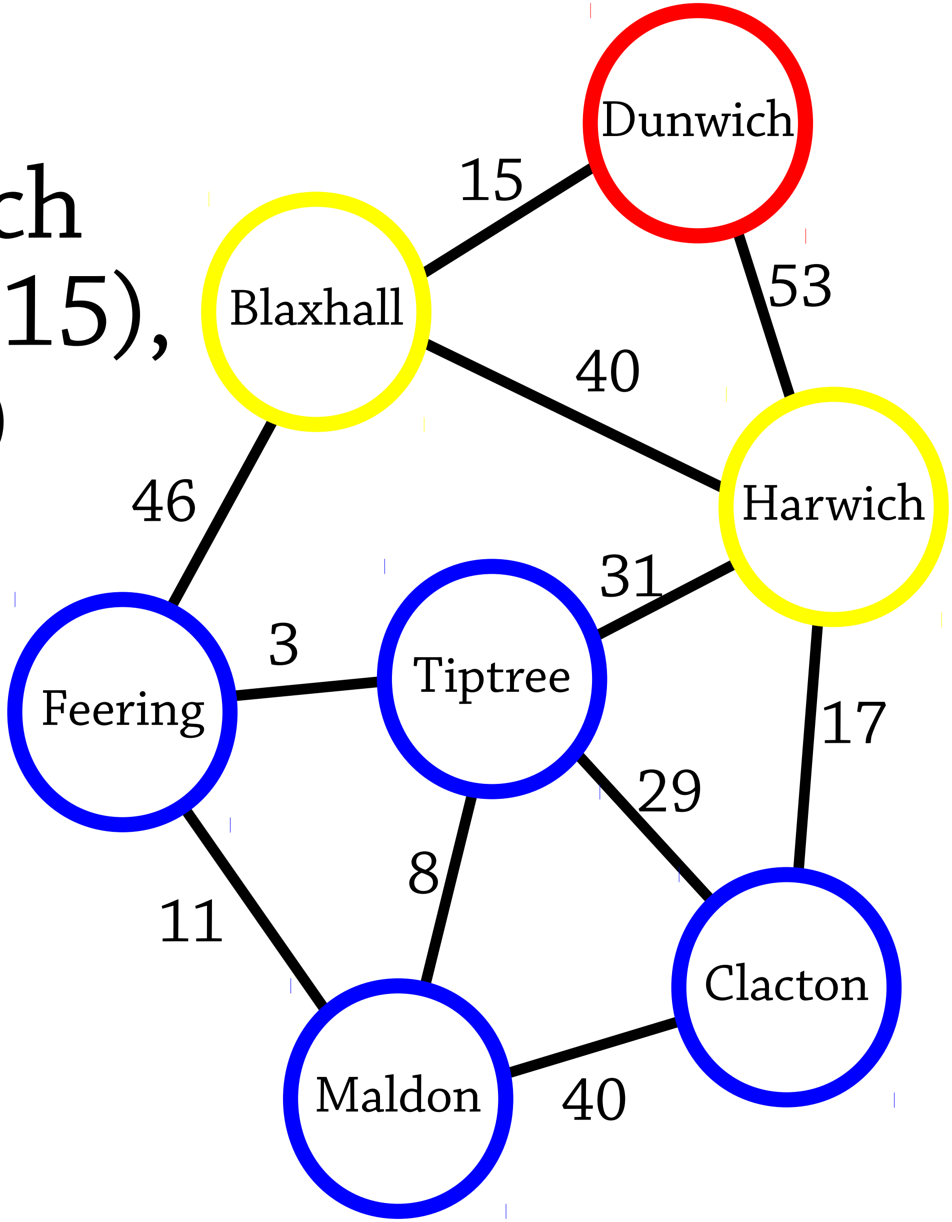


Dijkstra's algorithm

S = {Dunwich → 0}

Neighbours of Dunwich are Blaxhall (distance 15), Harwich (distance 53)

So add Blaxhall → 15 to S



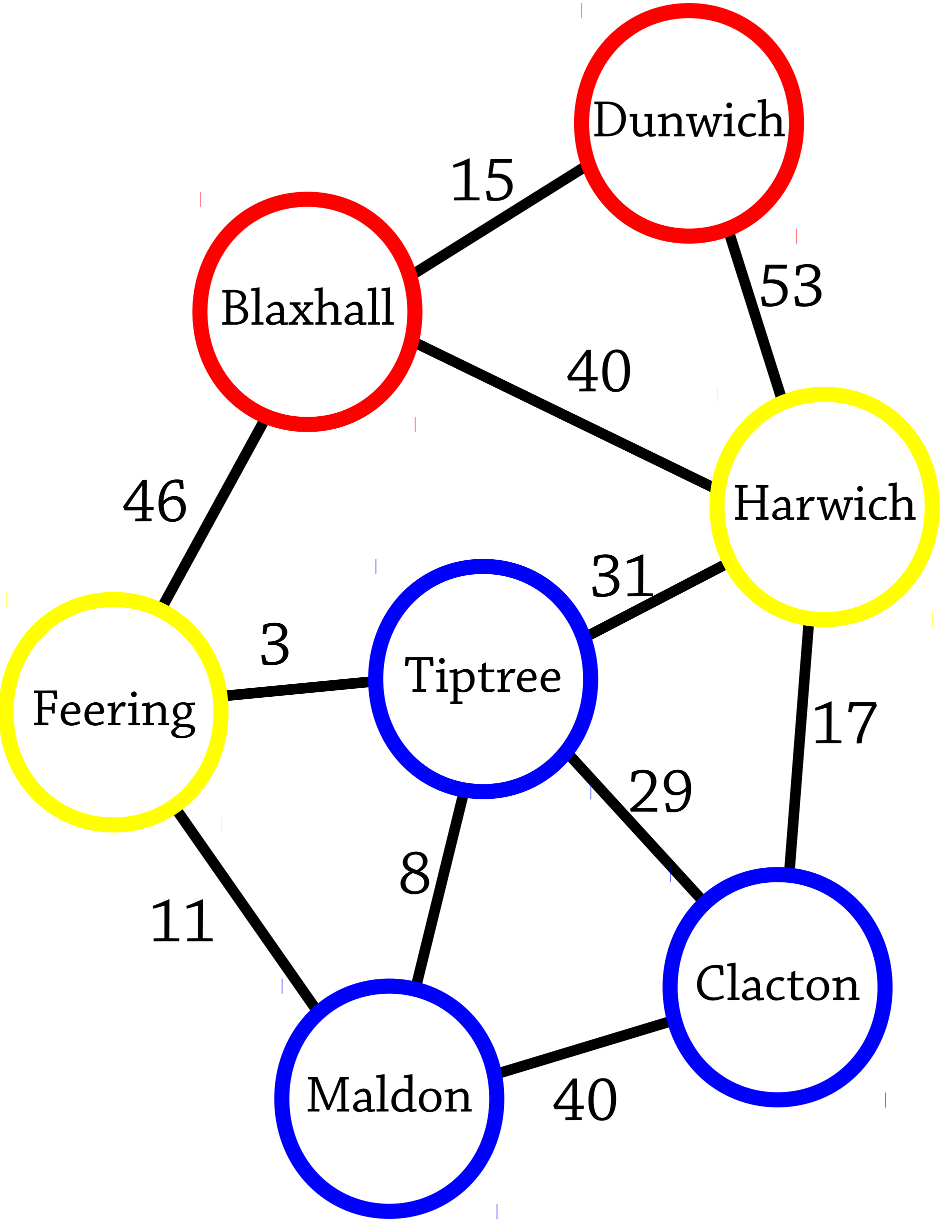
Dijkstra's algorithm

S = {Dunwich → 0,
Blaxhall → 15}

Neighbours of S are:

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

So add Harwich → 53 to S



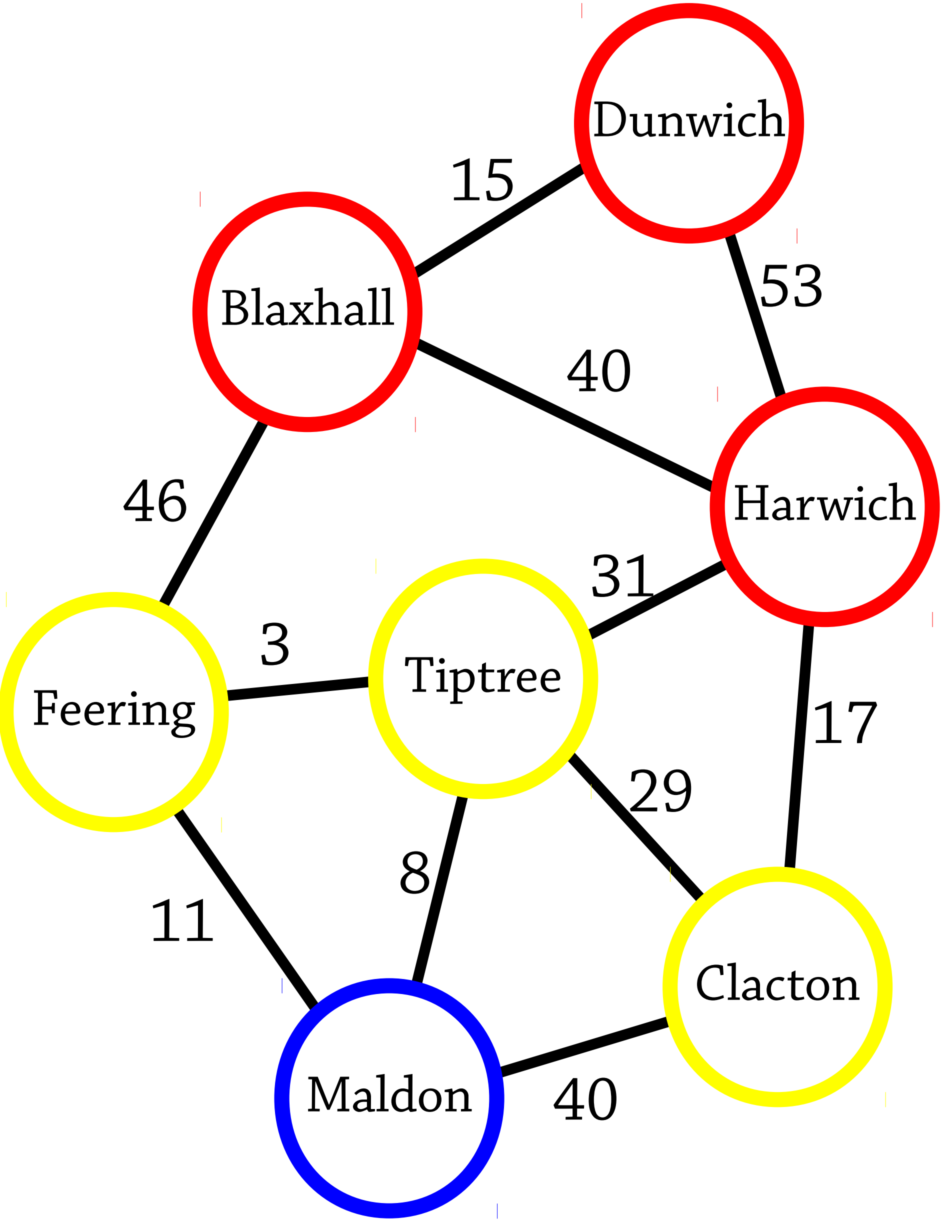
Dijkstra's algorithm

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

Neighbours of S are:

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

So add Feering → 61 to S



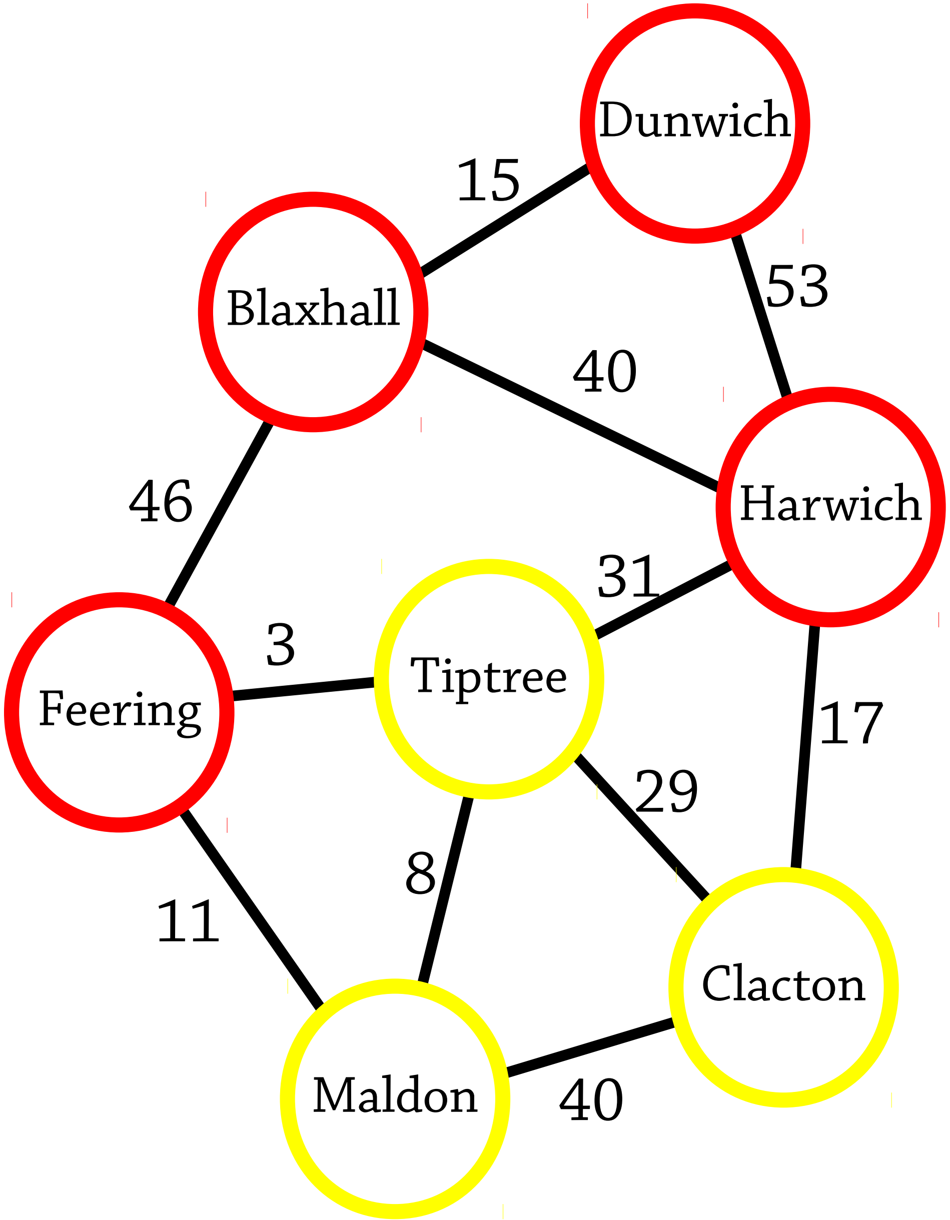
Dijkstra's algorithm

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

Neighbours of S are:

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

So add Tiptree → 64 to S



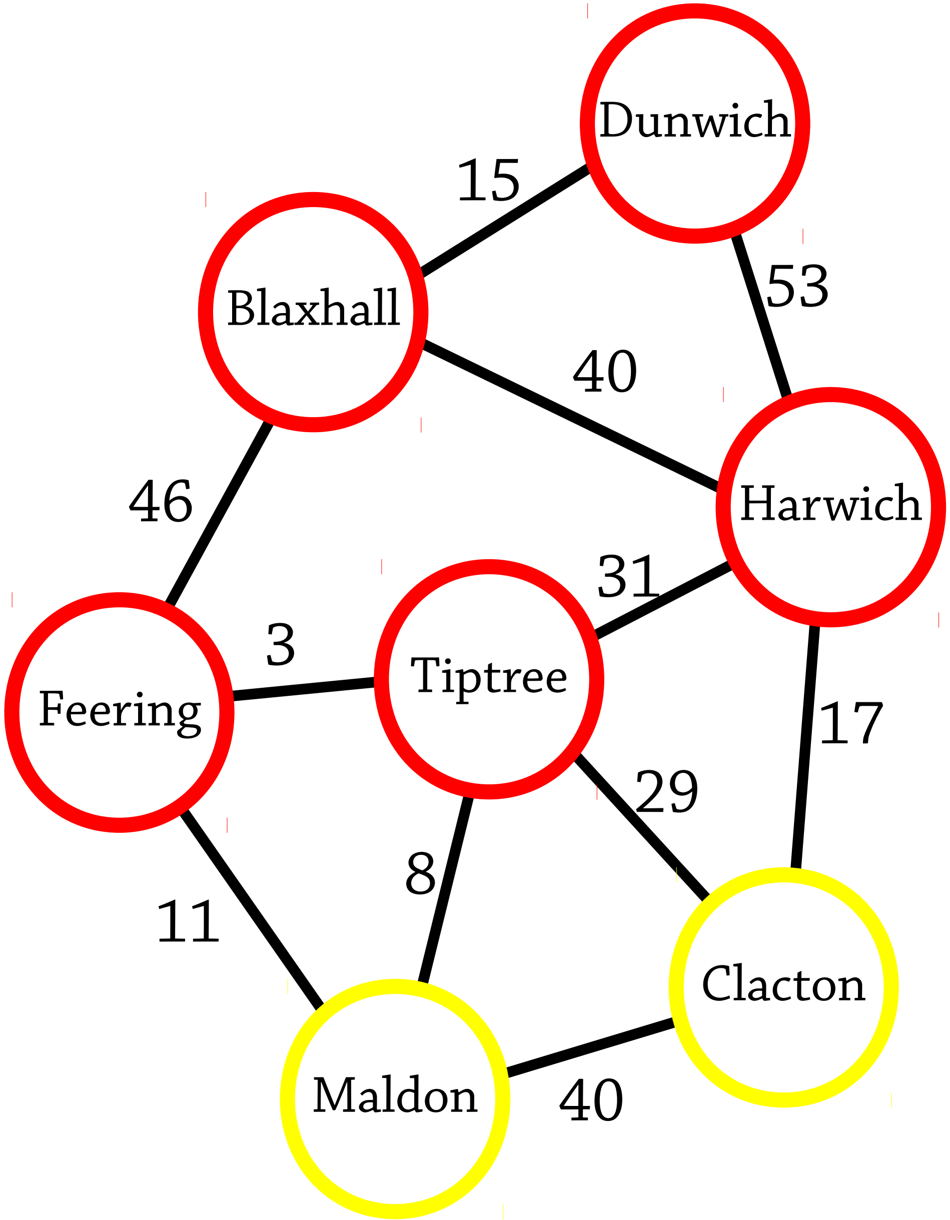
Dijkstra's algorithm

S = {Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64}

Neighbours of S are:

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

So add Clacton → 70 to S



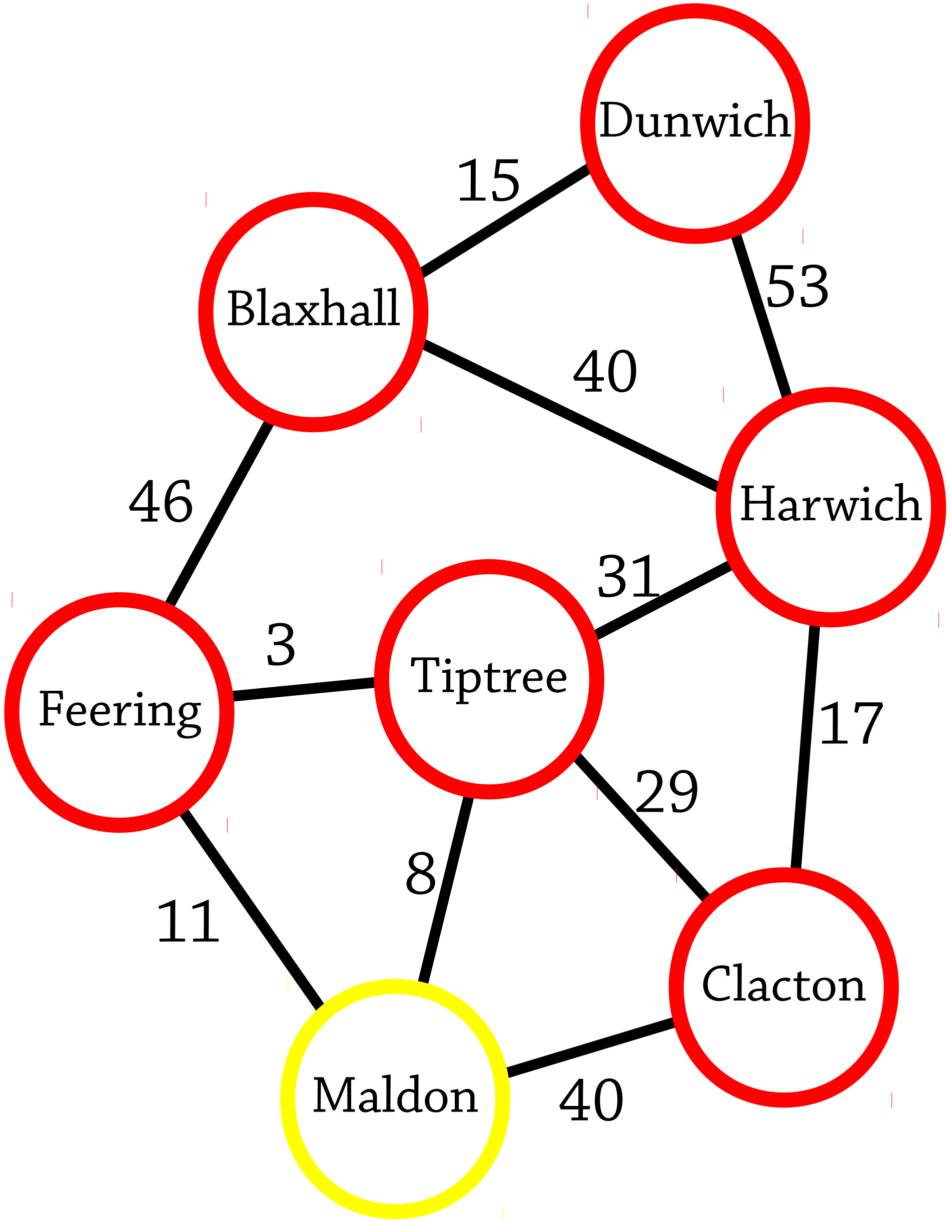
Dijkstra's algorithm

S = {Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70}

Neighbours of S are:

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

So add Maldon → 72 to S

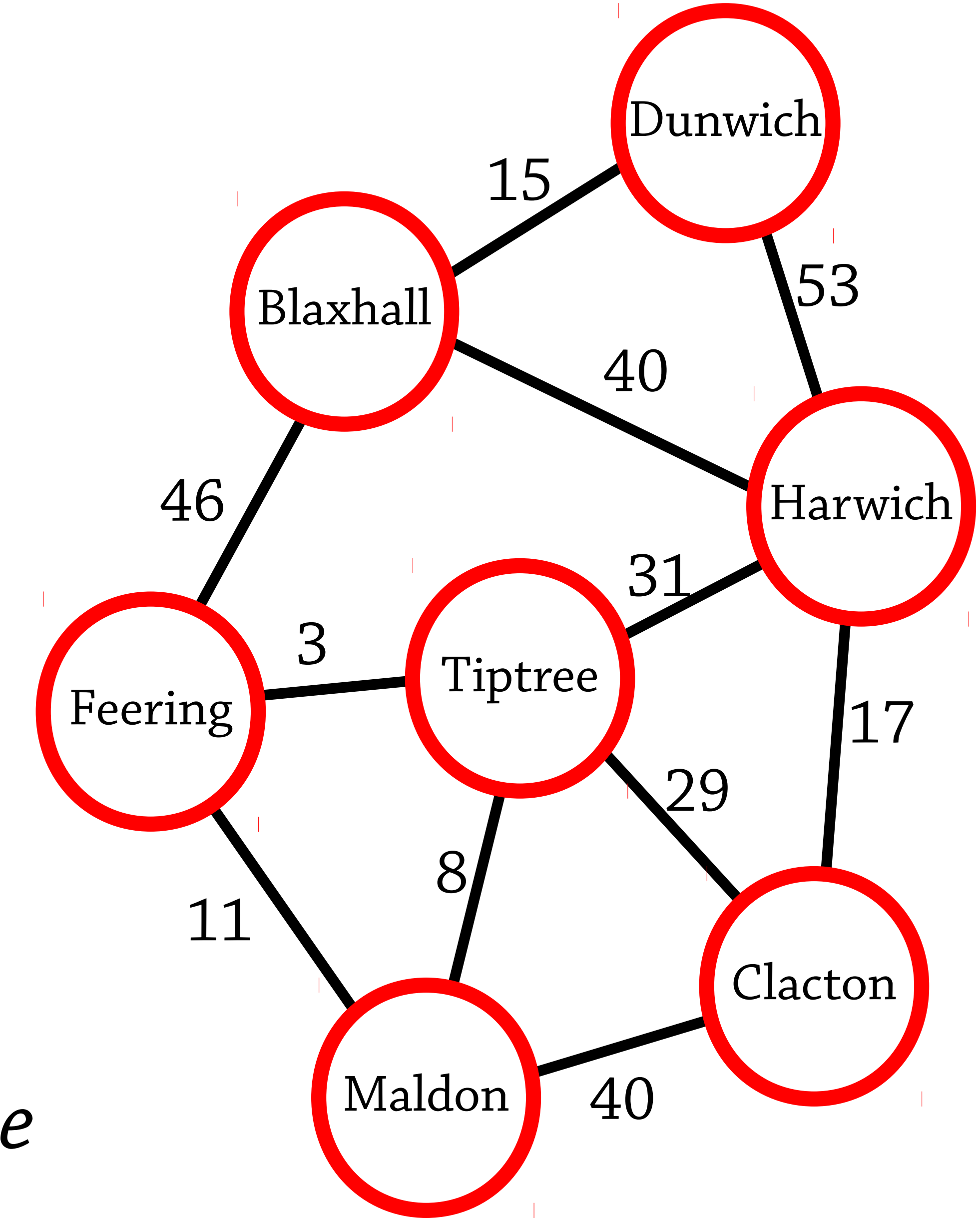


Dijkstra's algorithm

S = {Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72}

Finished!

Dijkstra's algorithm enumerates nodes in order of *how far away they are from the start node*



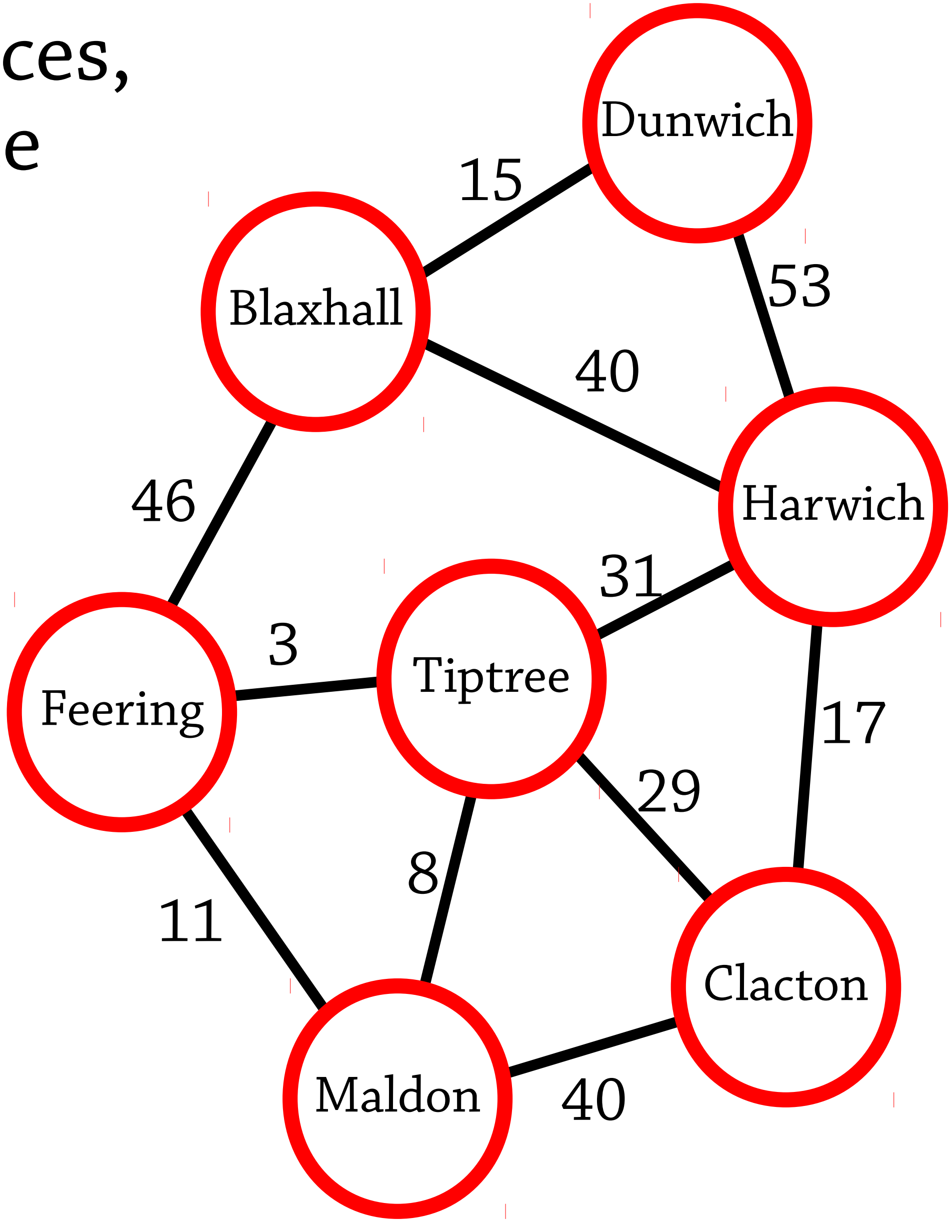
Dijkstra's algorithm

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

e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



hm

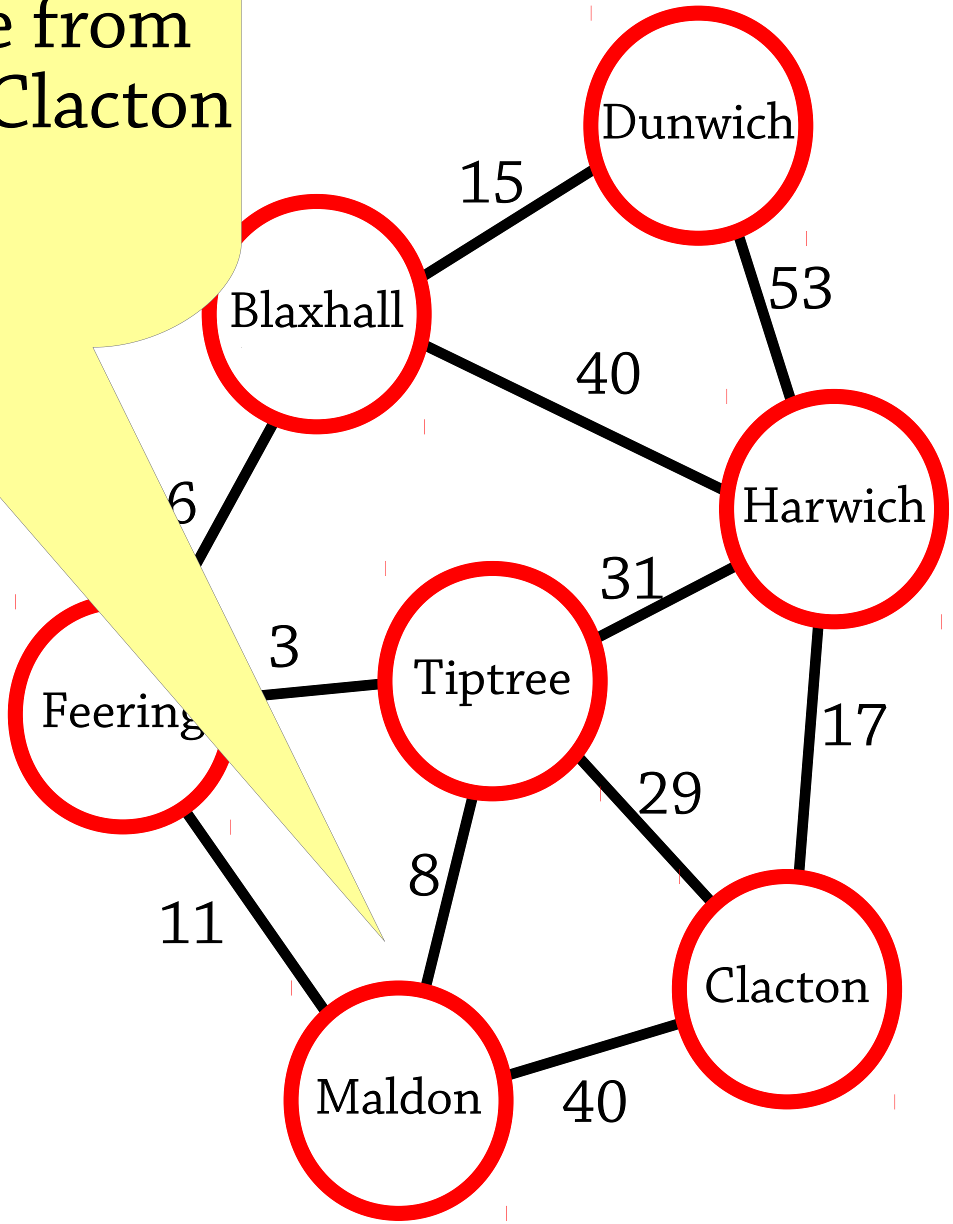
Once we
we can u
shortest

e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72

To arrive at Maldon, we must take the edge from Feering, Tiptree or Clacton



Dunwich → Clacton: **70**
Clacton → Maldon edge: **40**

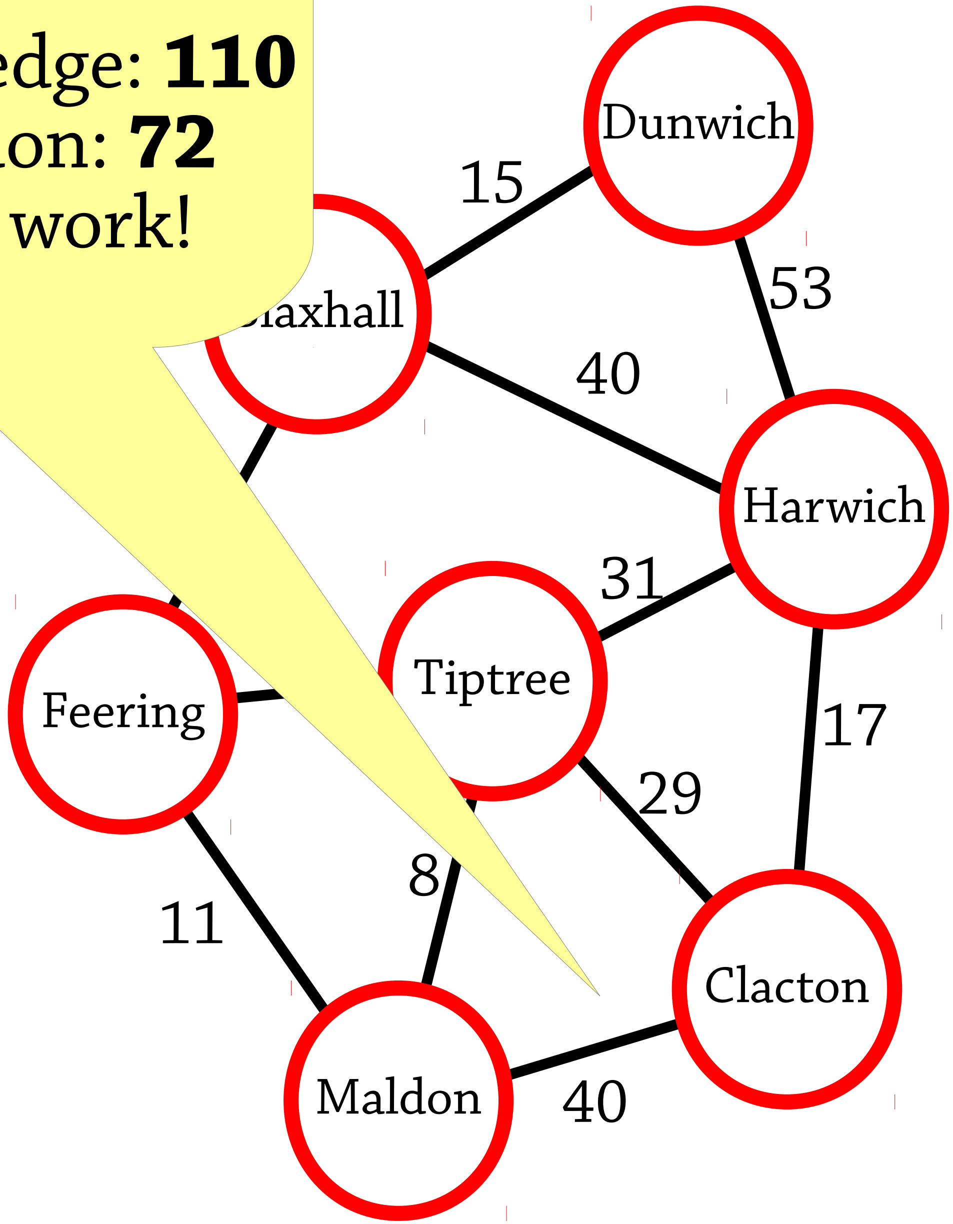
So coming via this edge: **110**
Dunwich → Maldon: **72**
This route won't work!

Once we
we can u
shortest

e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



m

Once we
we can u
shortest

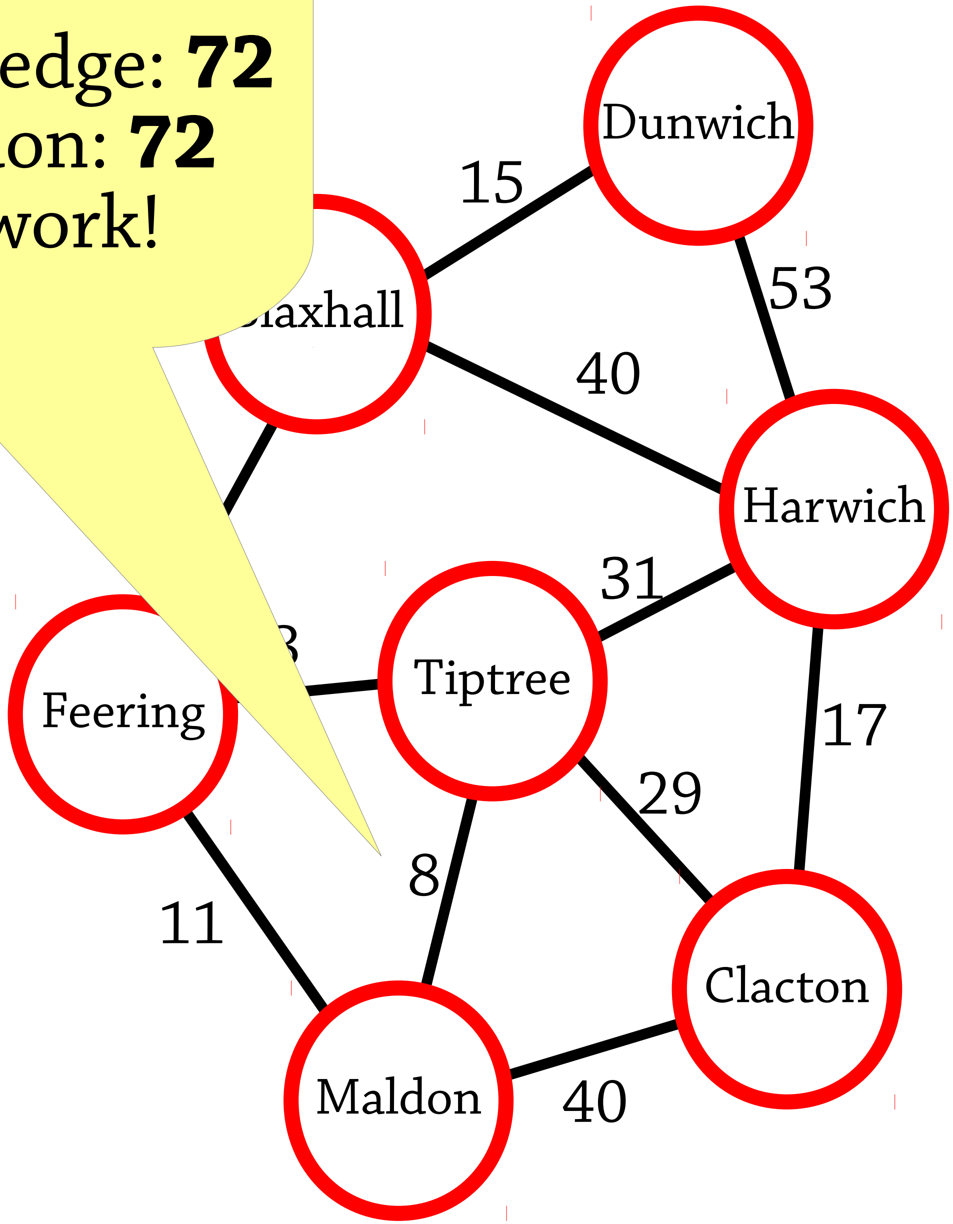
e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72

Dunwich → Tiptree: **64**
Tiptree → Maldon edge: **8**

So coming via this edge: **72**
Dunwich → Maldon: **72**
This route will work!



m

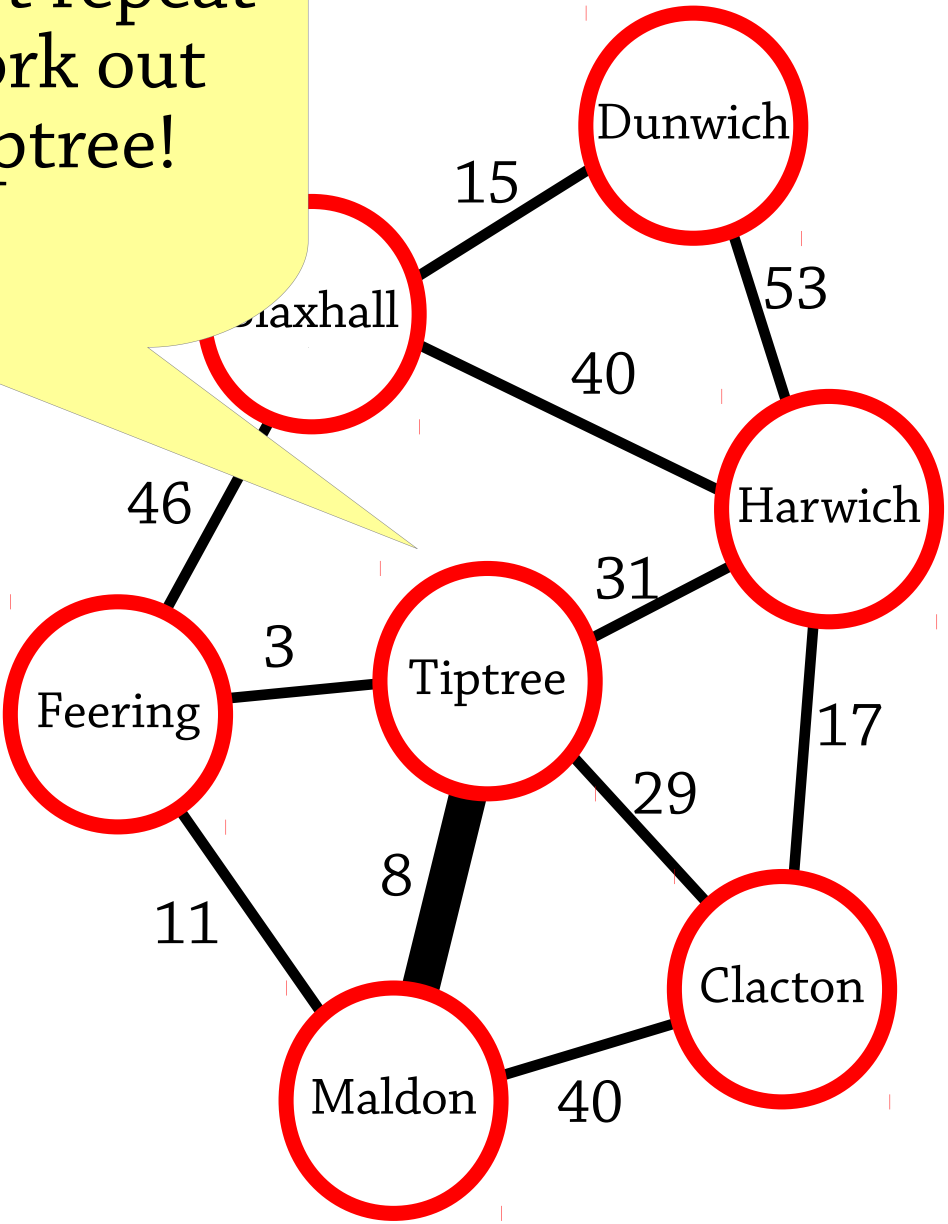
Now we know we can come via Tiptree – so just repeat the process to work out how to get to Tiptree!

Once we
we can u
shortest

e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



m

Dunwich → Harwich: **53**
Harwich→ Tiptree edge: **31**

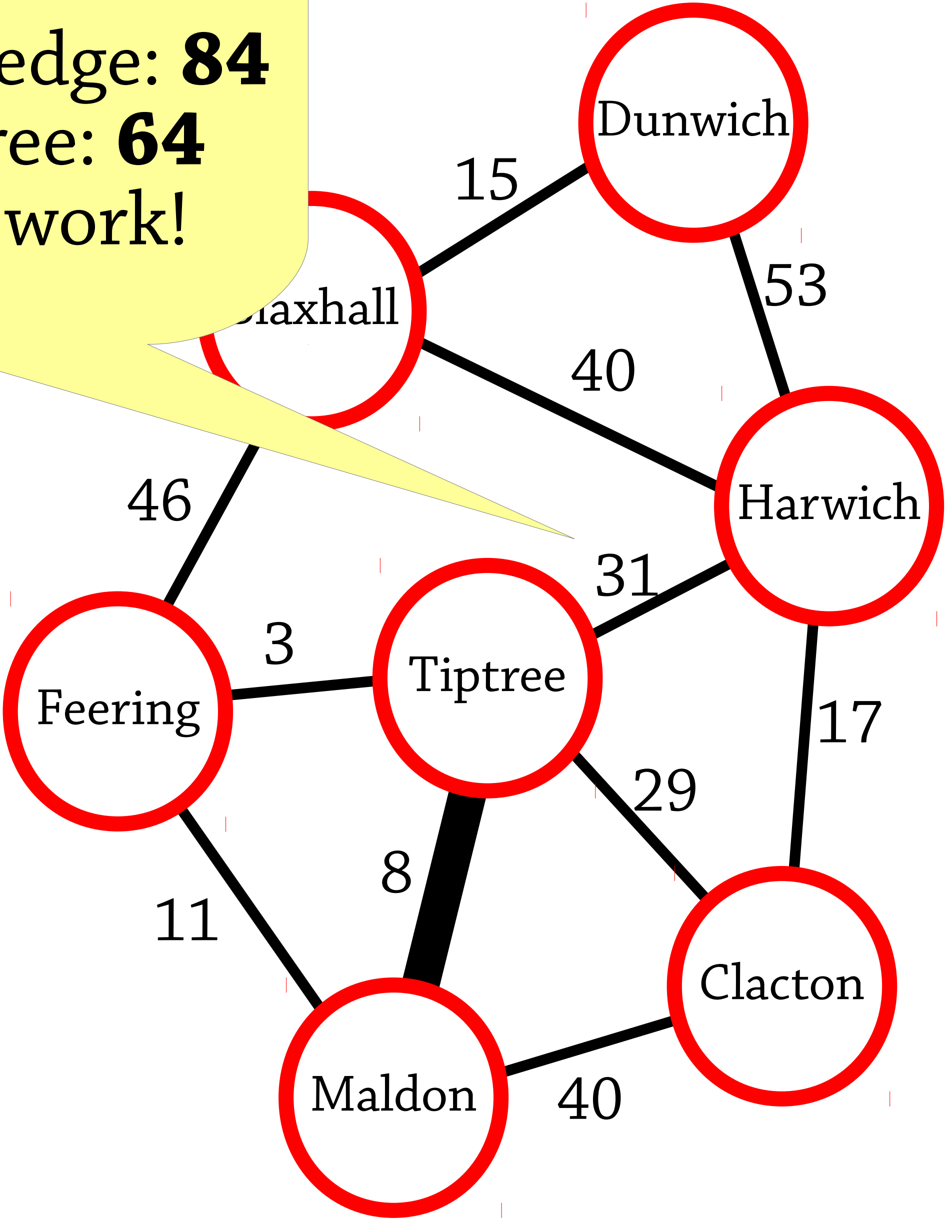
So coming via this edge: **84**
Dunwich → Tiptree: **64**
This route won't work!

Once we
we can u
shortest

e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



m

Once we
we can u
shortest

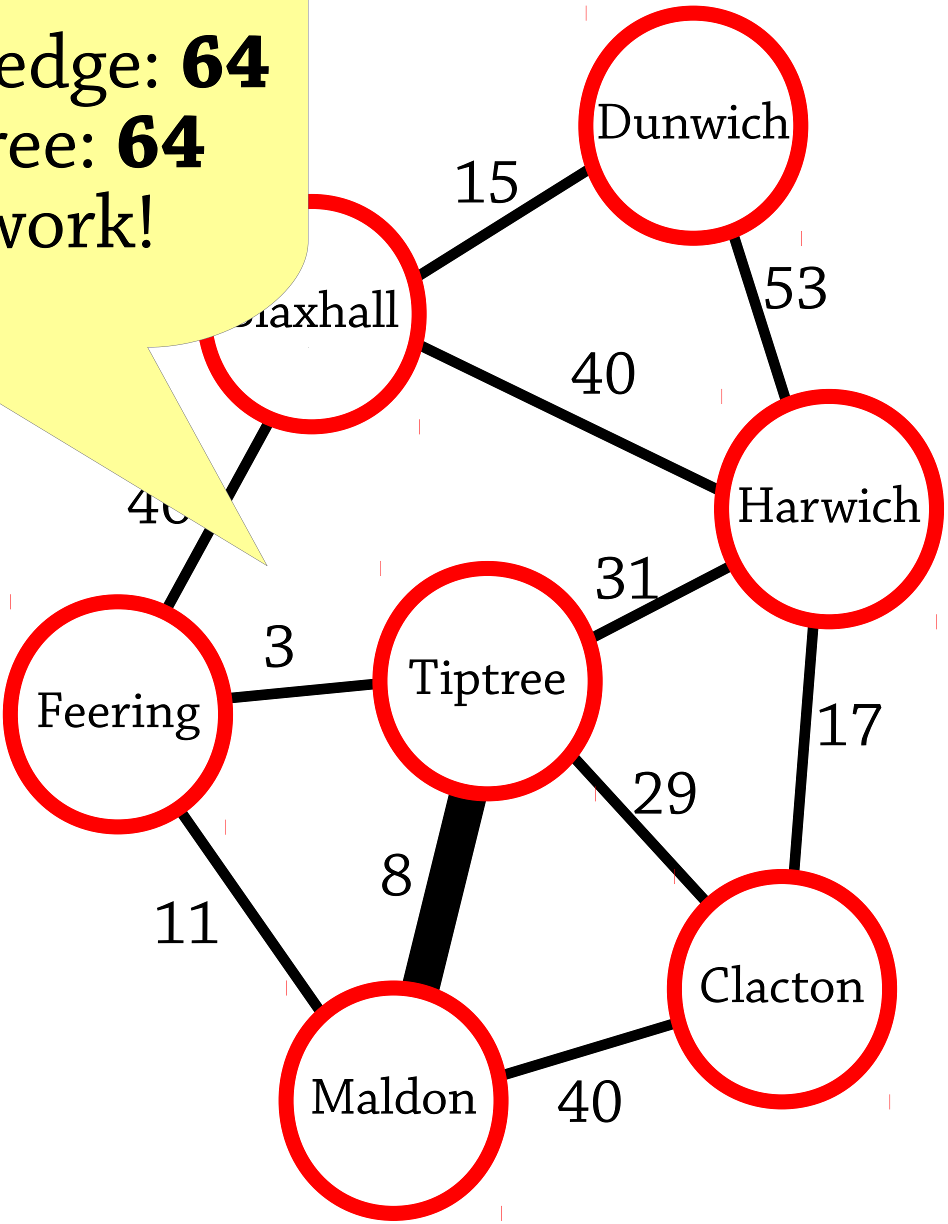
e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72

Dunwich → Feering: **61**
Feering → Tiptree edge: **3**

So coming via this edge: **64**
Dunwich → Tiptree: **64**
This route will work!



m

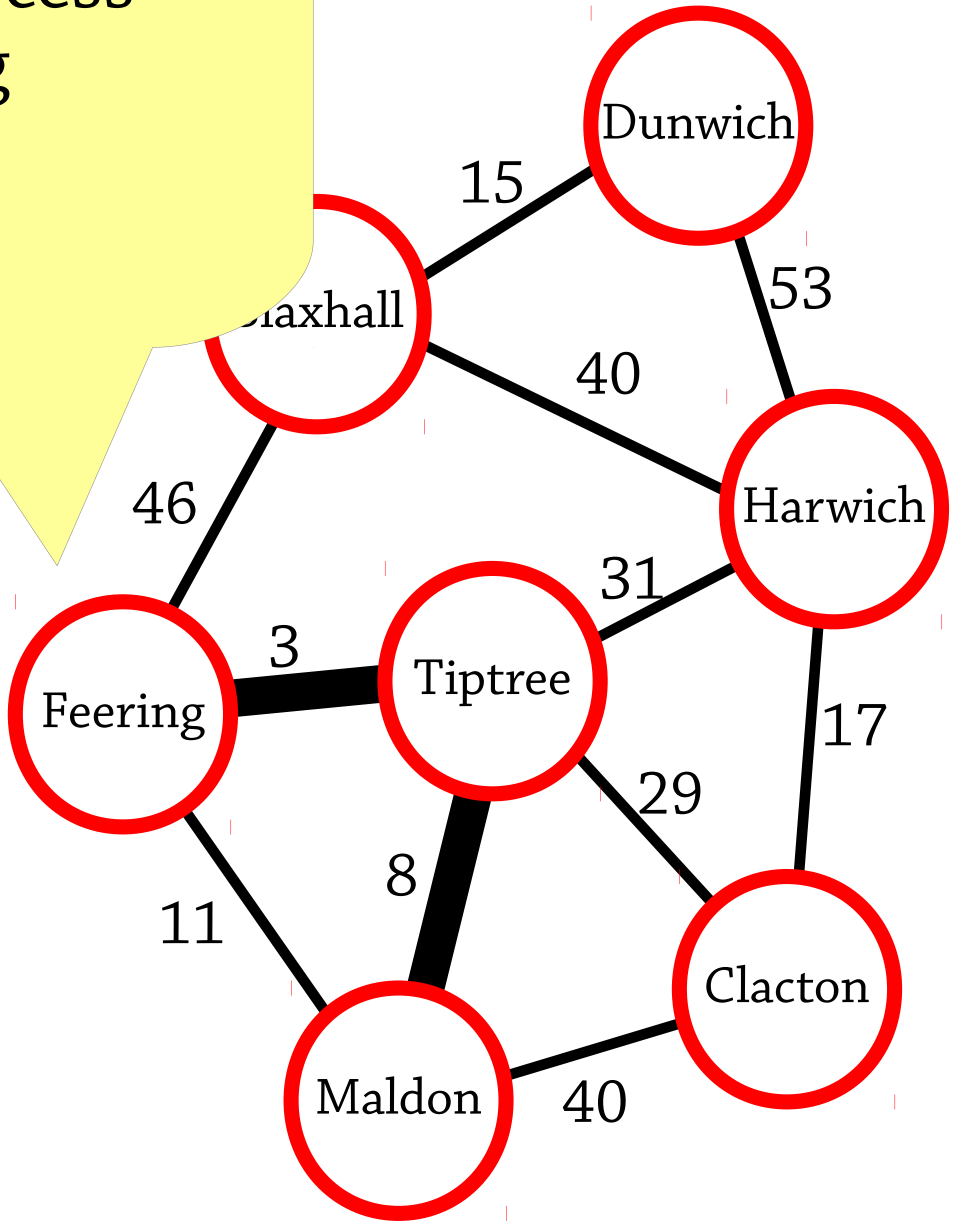
Repeat the process
for Feering

Once we
we can u
shortest

e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



m

Dunwich → Blaxhall: **15**
Blaxhall → Feering edge: **46**

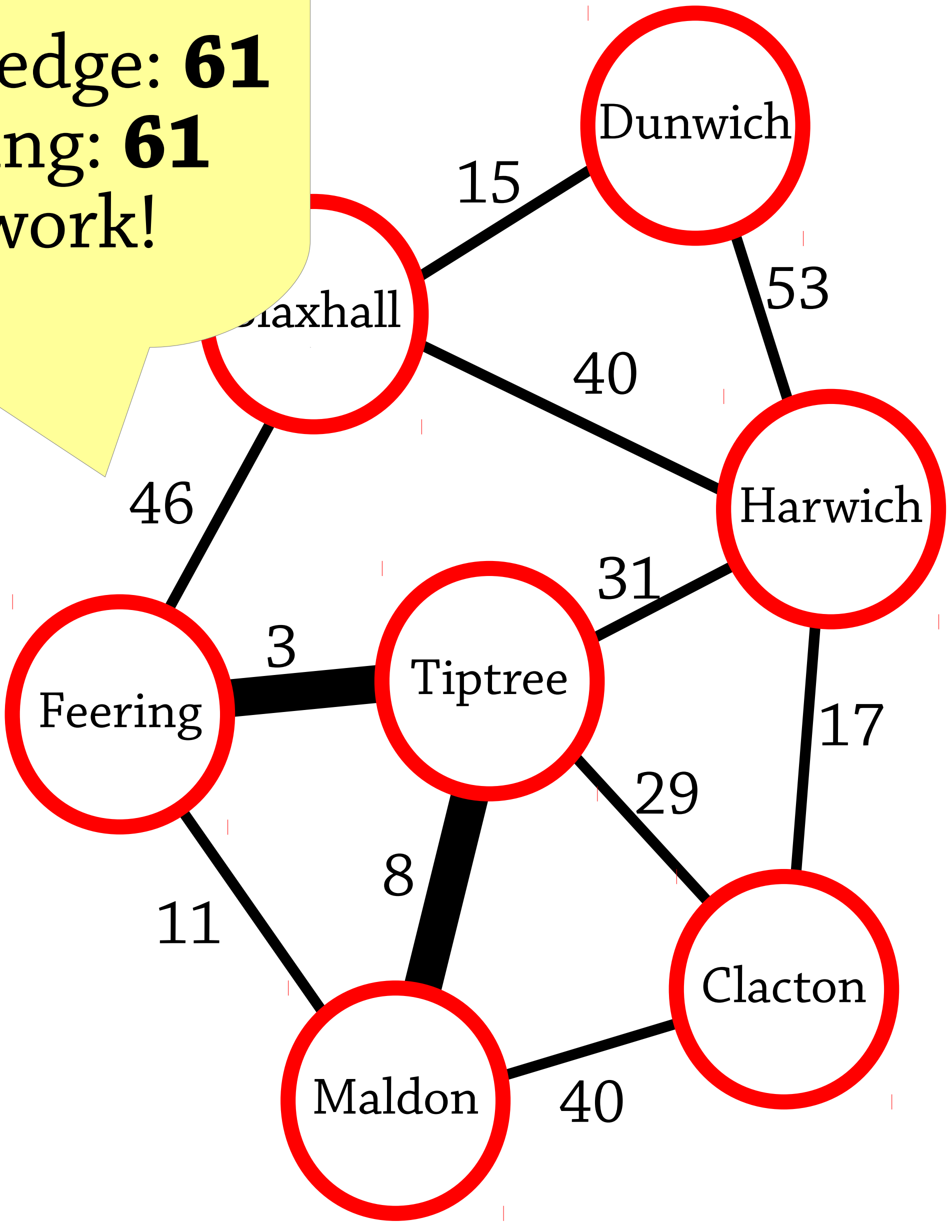
So coming via this edge: **61**
Dunwich → Feering: **61**
This route will work!

Once we
we can u
shortest

e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



algorithm

Repeat the process
for Blaxhall

ances,

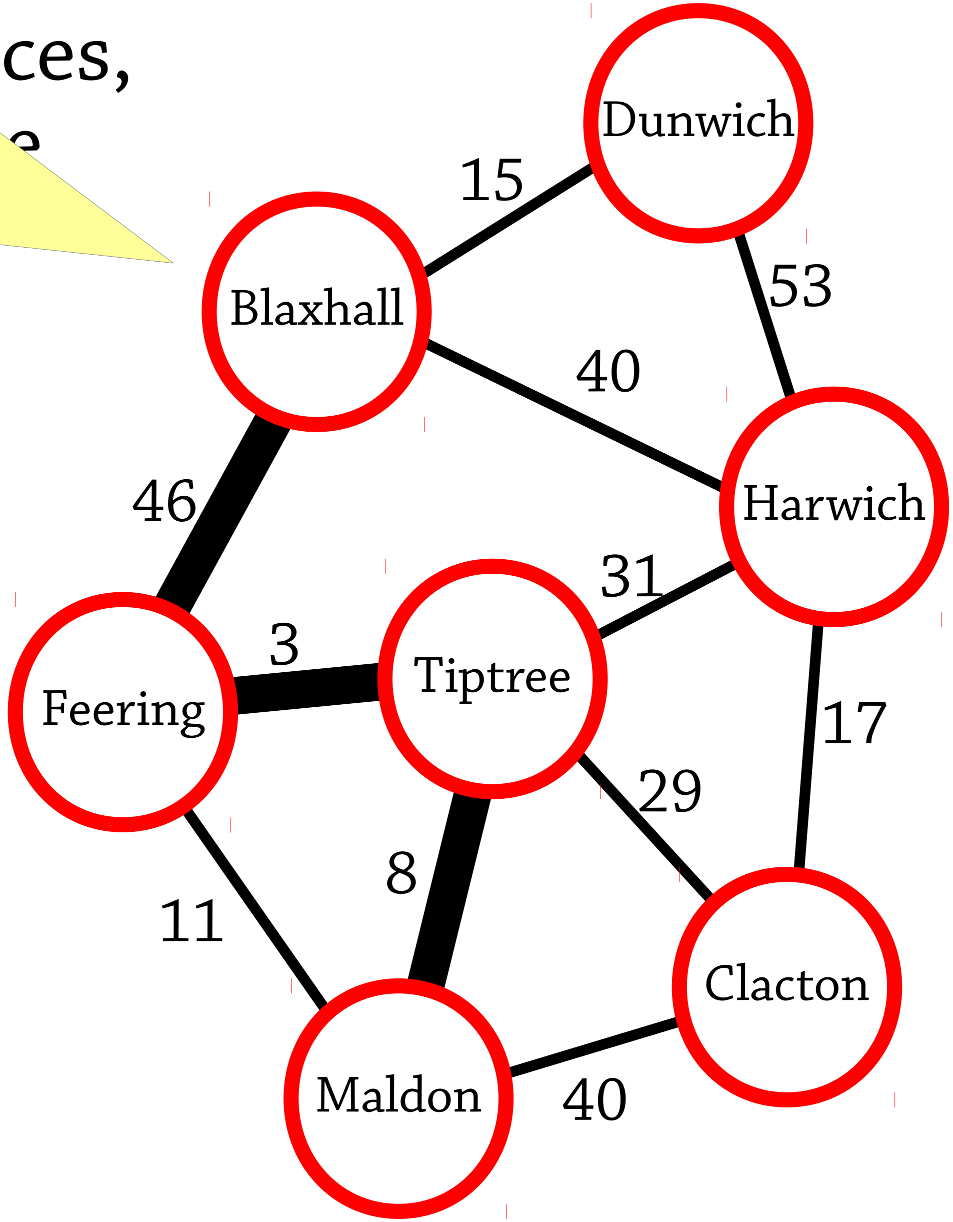
e

le:

e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



algorithm

Dunwich → Harwich: **53**
Harwich → Blaxhall edge: **40**

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

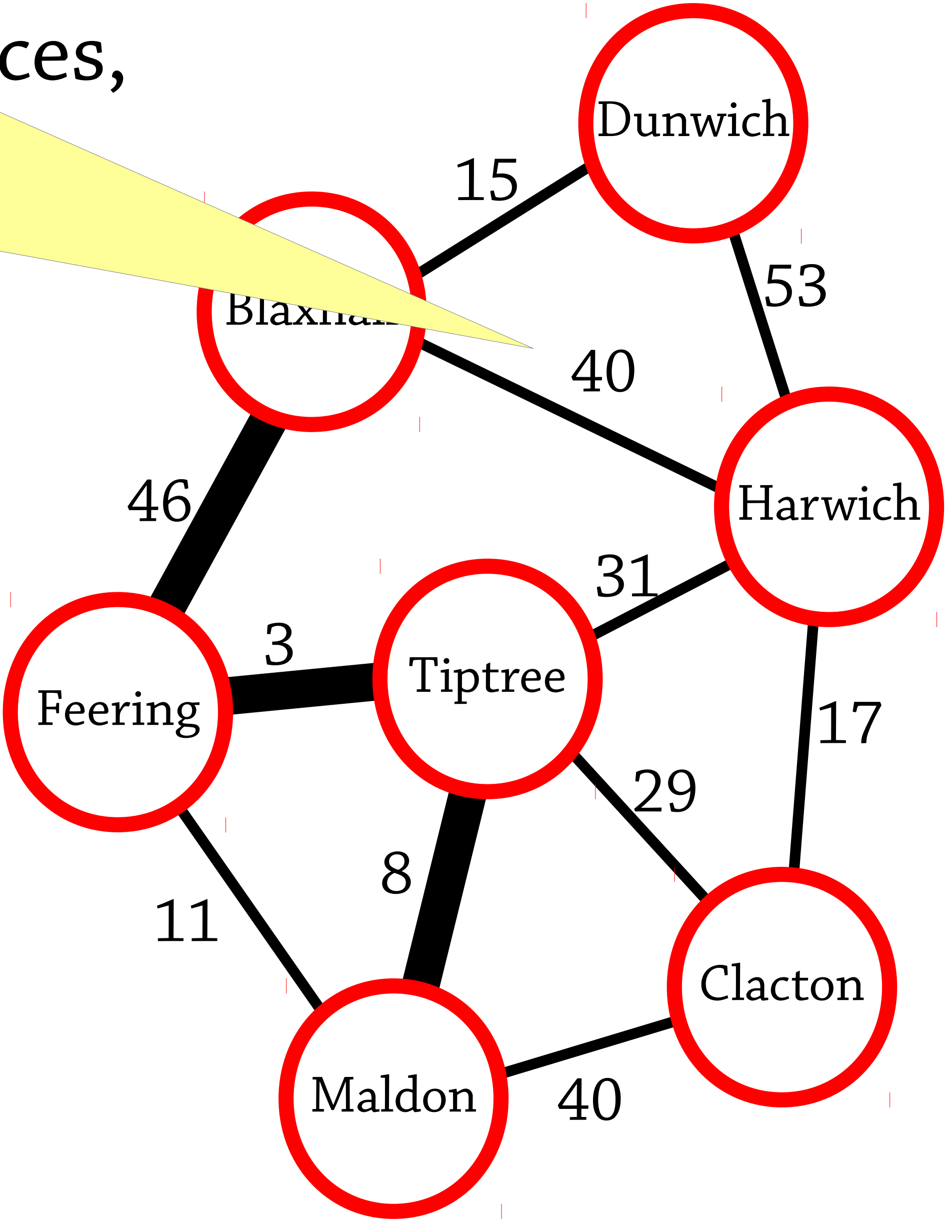
ances,

e:

e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



algorithm

Dunwich → Dunwich: **0**
Dunwich → Blaxhall edge: **15**

So coming via this edge: **15**
Dunwich → Blaxhall: **15**
This route will work!

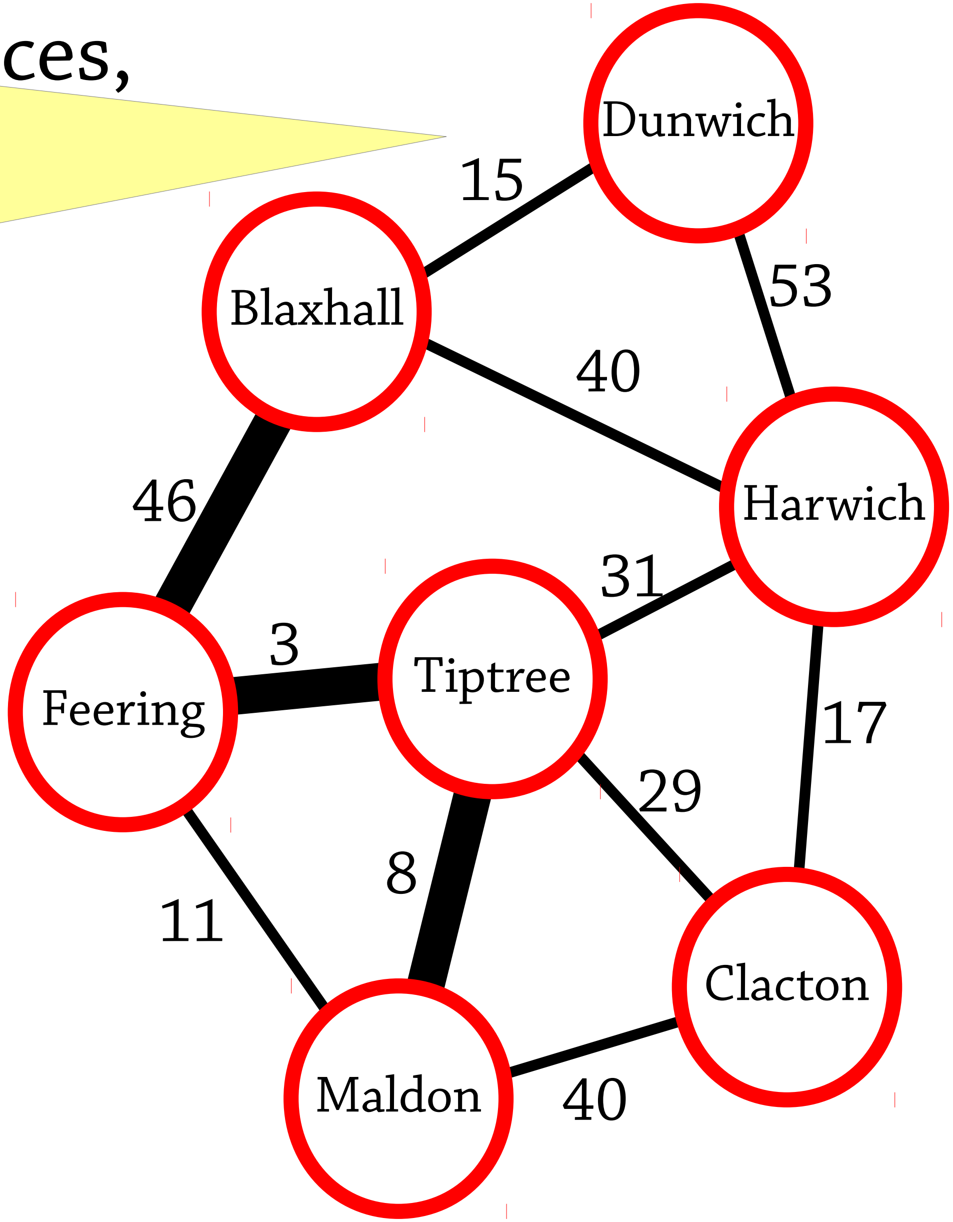
ances,

e!

e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



algorithm

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

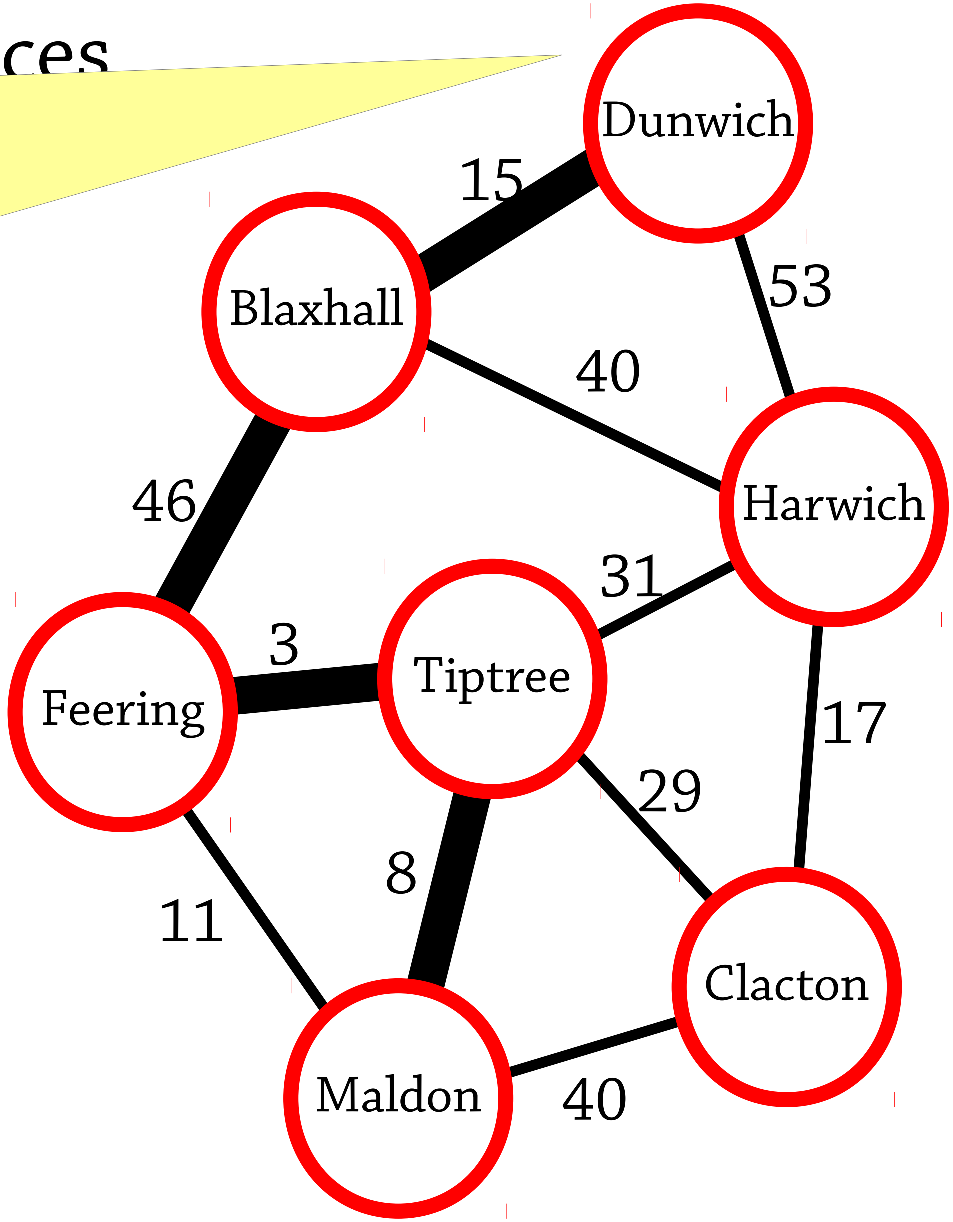
ances

e!

e.g. take Maldon

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

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



Dijkstra's algorithm

Let S = {start node → 0}

While not all nodes are in S,

- For each node $x \to d$ in S, and each neighbour y of x, calculate $d' = d + \textit{cost of edge from } x \textit{ to } y$
- Take the smallest d' calculated and its y and add $y \to d'$ 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?

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

ra's algori

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

de → 0}

While not all nodes are in S

- For each node *x* → *d* in S and each neighbour *y* of *x*, calculate *d'* = *d* + *cost of edge from x to y*
- Take the smallest *d'* calculated and its *y* and add *y* → *d'* to S

This computes the sh nce to each node, from w construct the shortest path

What is the efficie s algorithm?

Total: O(|VE|)!

Dijkstra's algorithm, made efficient

The algorithm so far is $O(|V|^2)$

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 $O(|V|)$ 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!

Our priority queue will contain:

- all neighbours of nodes in *S* (the yellow nodes from our diagram)
- together with their distances

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

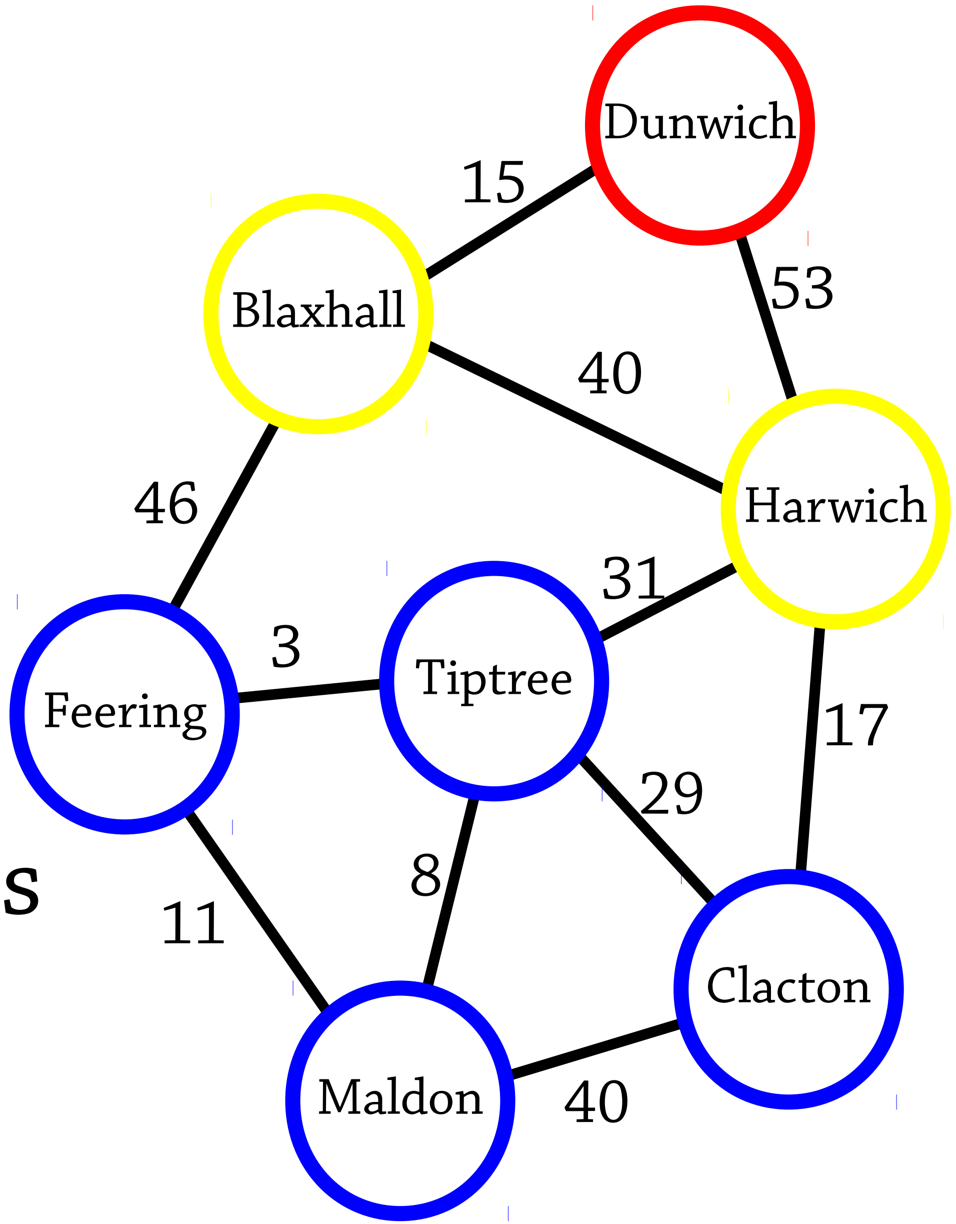
Whenever we add a node to S, we will add each of its neighbours that are not in S to the priority queue

Dijkstra's algorithm

S = {Dunwich → 0}

Q = {Blaxhall 15,
Harwich 53}

Remove the smallest element of Q, “Blaxhall 15”. Add Blaxhall → 15 to S, and add Blaxhall's neighbours to Q.

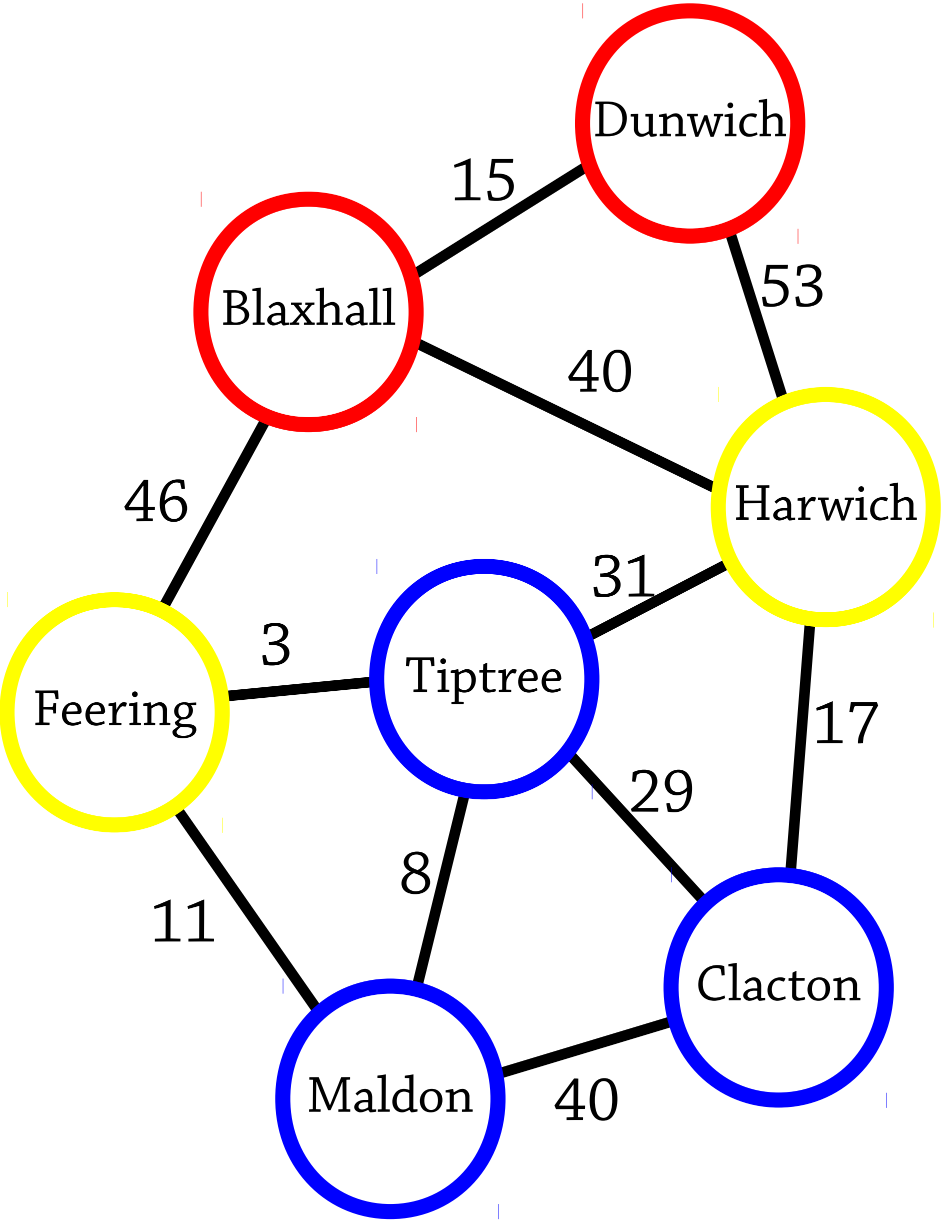


Dijkstra's algorithm

S = {Dunwich → 0,
Blaxhall → 15}

Q = {Harwich 53,
Feering 61,
Harwich 55}

Remove the smallest element of Q, “Harwich 53”.
Add Harwich → 53 to S, and add Harwich's neighbours to Q.

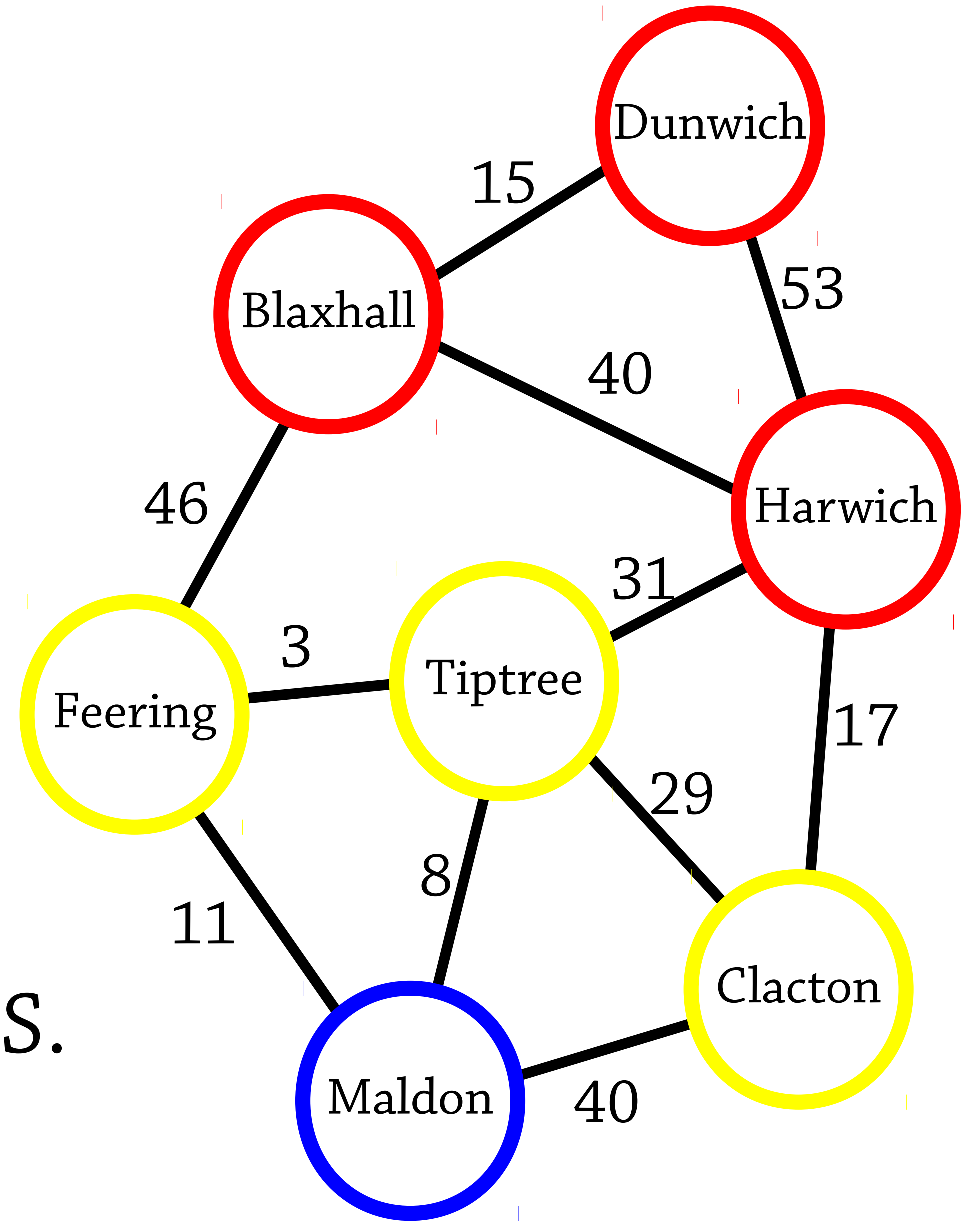


Dijkstra's algorithm

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

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

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

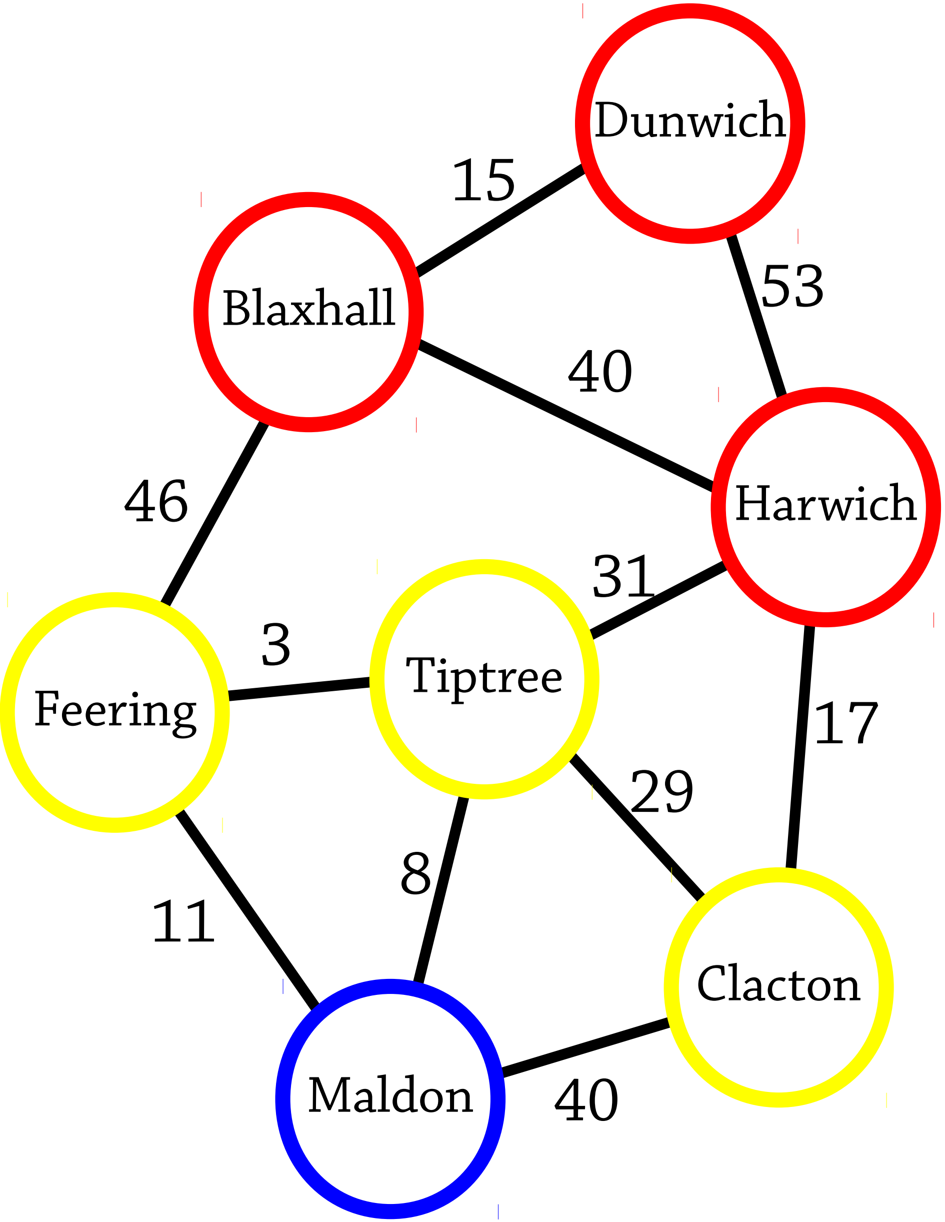


Dijkstra's algorithm

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

Q = {Feering 61,
Tiptree 84,
Clacton 70}

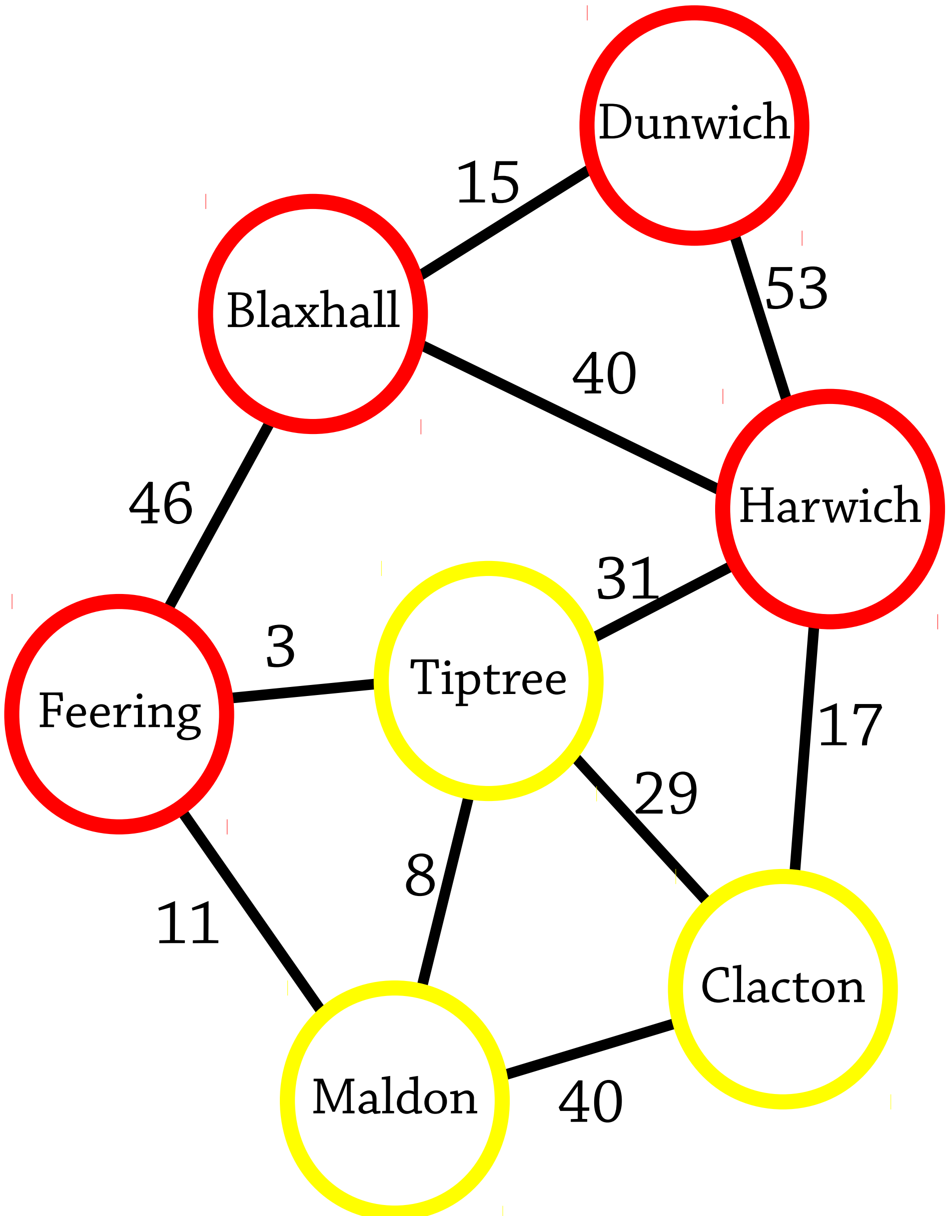
Remove the smallest element of Q, “Feering 61”. Add Feering → 61 to S, and add Feering's neighbours to Q.



Dijkstra's algorithm

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

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



Dijkstra's algorithm, efficiently

Let S = {start node → 0} and Q = {}

For each of the start node's neighbours *x*, where the edge has weight *d*, add *x* to Q with priority *d*

While not all nodes are in S,

- Remove the node *y* from Q that has the smallest priority (distance)
- If *y* is in S, do nothing
- Otherwise, add y → d to S and for all of *y*'s neighbours *z* add *z to Q* with priority *“d + weight of edge from y to z”*

Dij… …iently

Maximum size of Q is |E|,
total of O(|V| + |E|)
priority queue operations,
so total time:
O((|V| + |E|) log |E|)
or
O(n log n) where n = |V| + |E|

Let S =

For ea… …ours *x*,
where … to Q with
priority *d*

While not all nodes are in S,

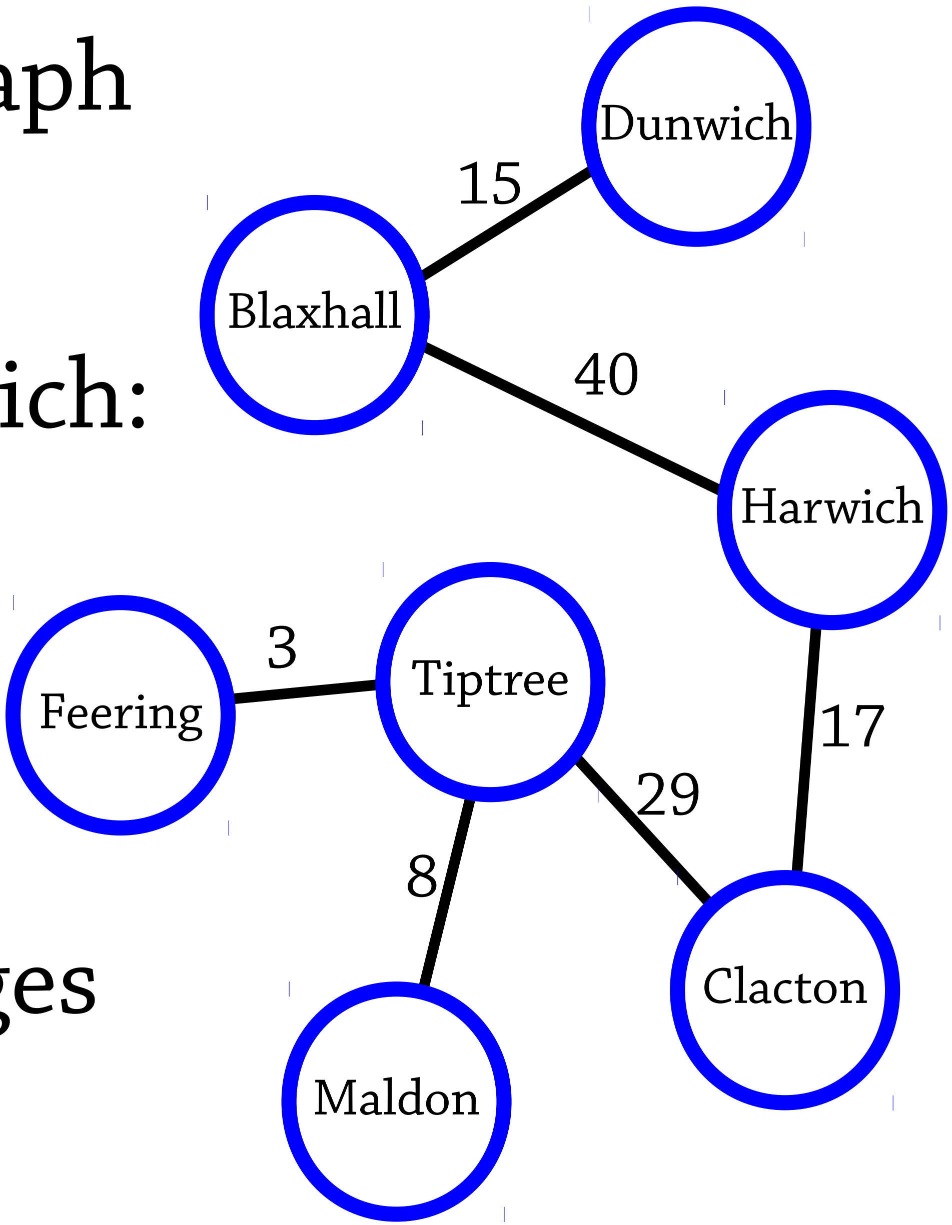
- Remove the node *y* from Q that has the smallest priority (distance)
- If *y* is in S, do nothing
- Otherwise, add y → d to S and for all of *y*'s neighbours *z* add *z to Q* with priority *“d + weight of edge from y to z”*

Minimum spanning trees

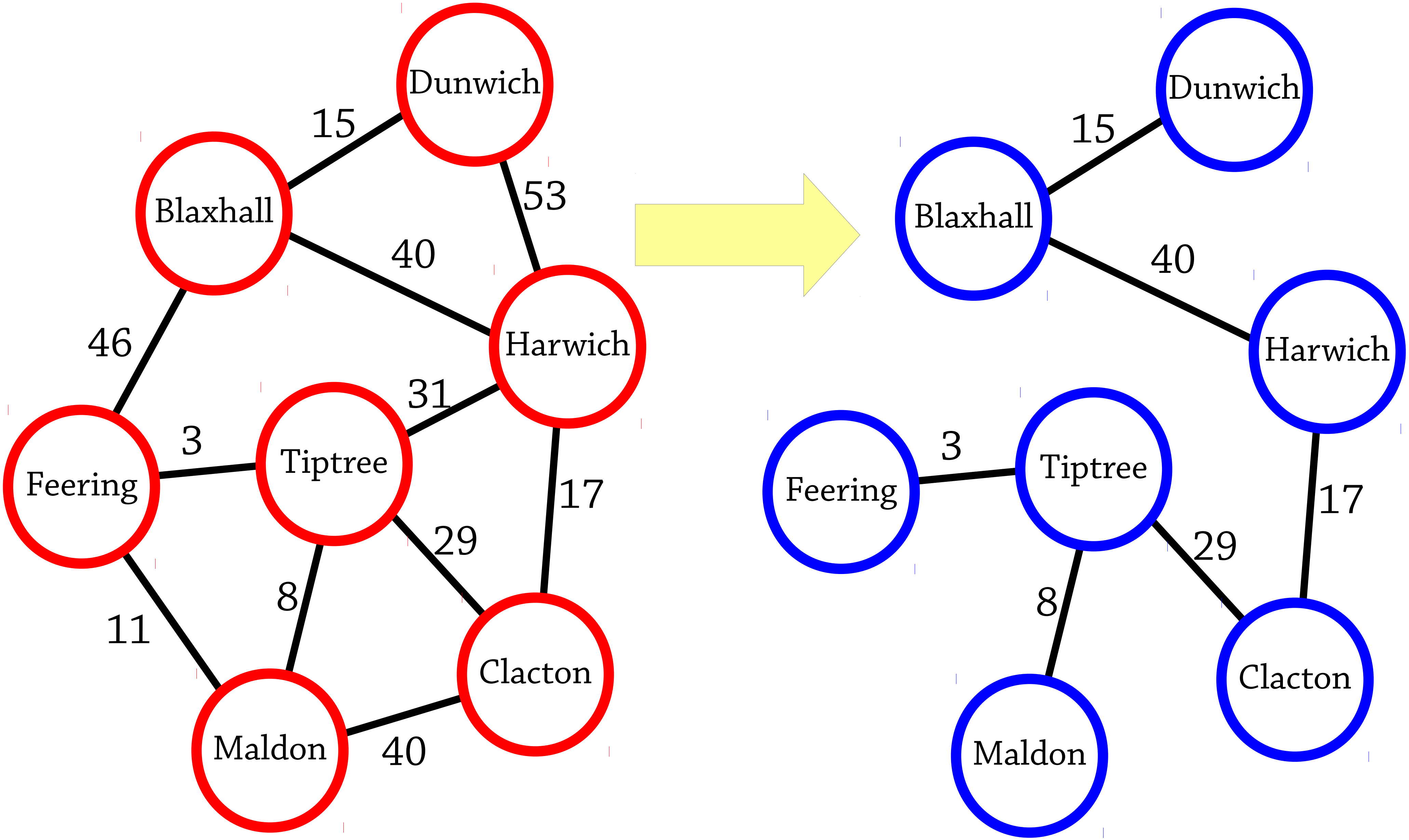
A *spanning tree* of a graph is a subgraph (a graph obtained by deleting some of the edges) which:

- 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

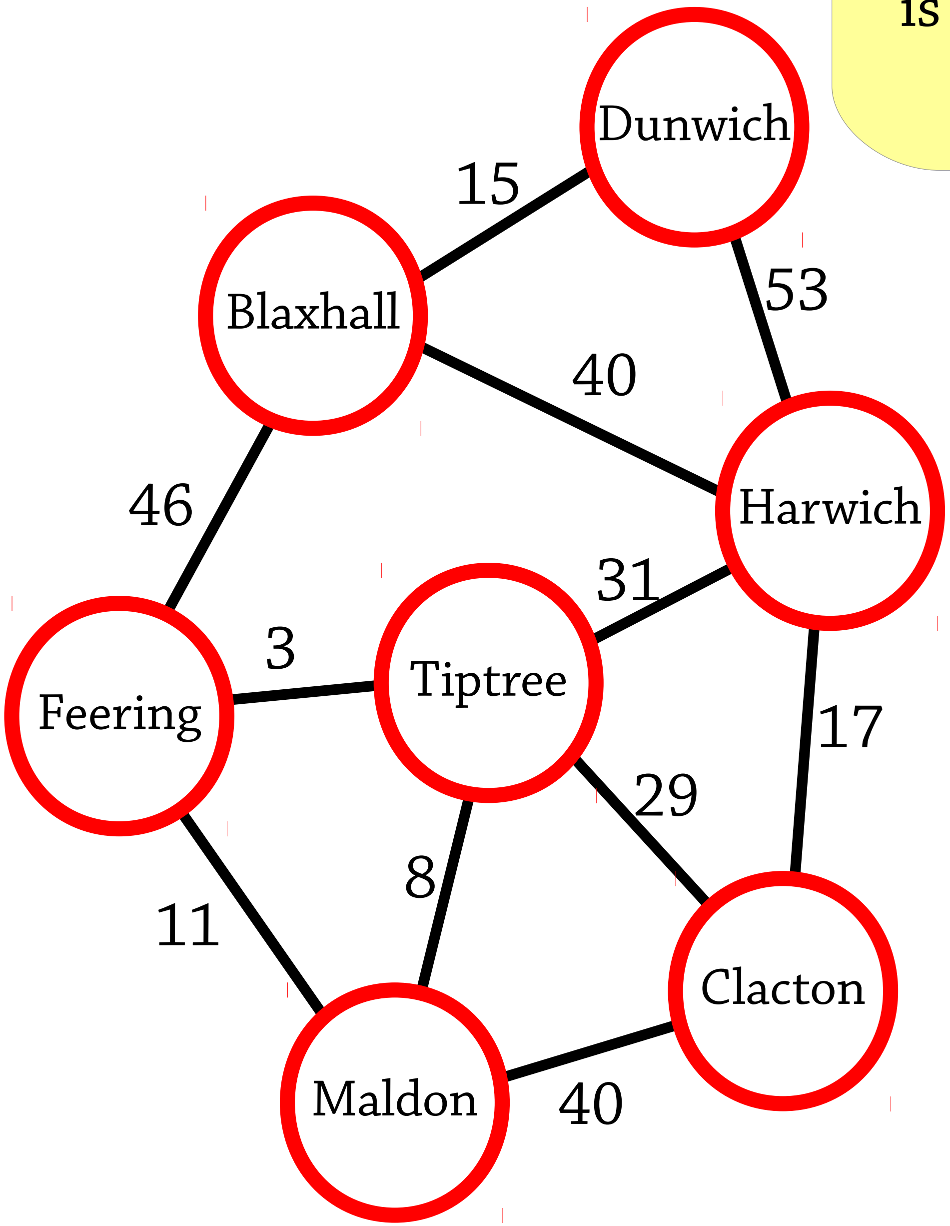
Keep a set S of all the nodes that are in the tree so far, initially containing one arbitrary node

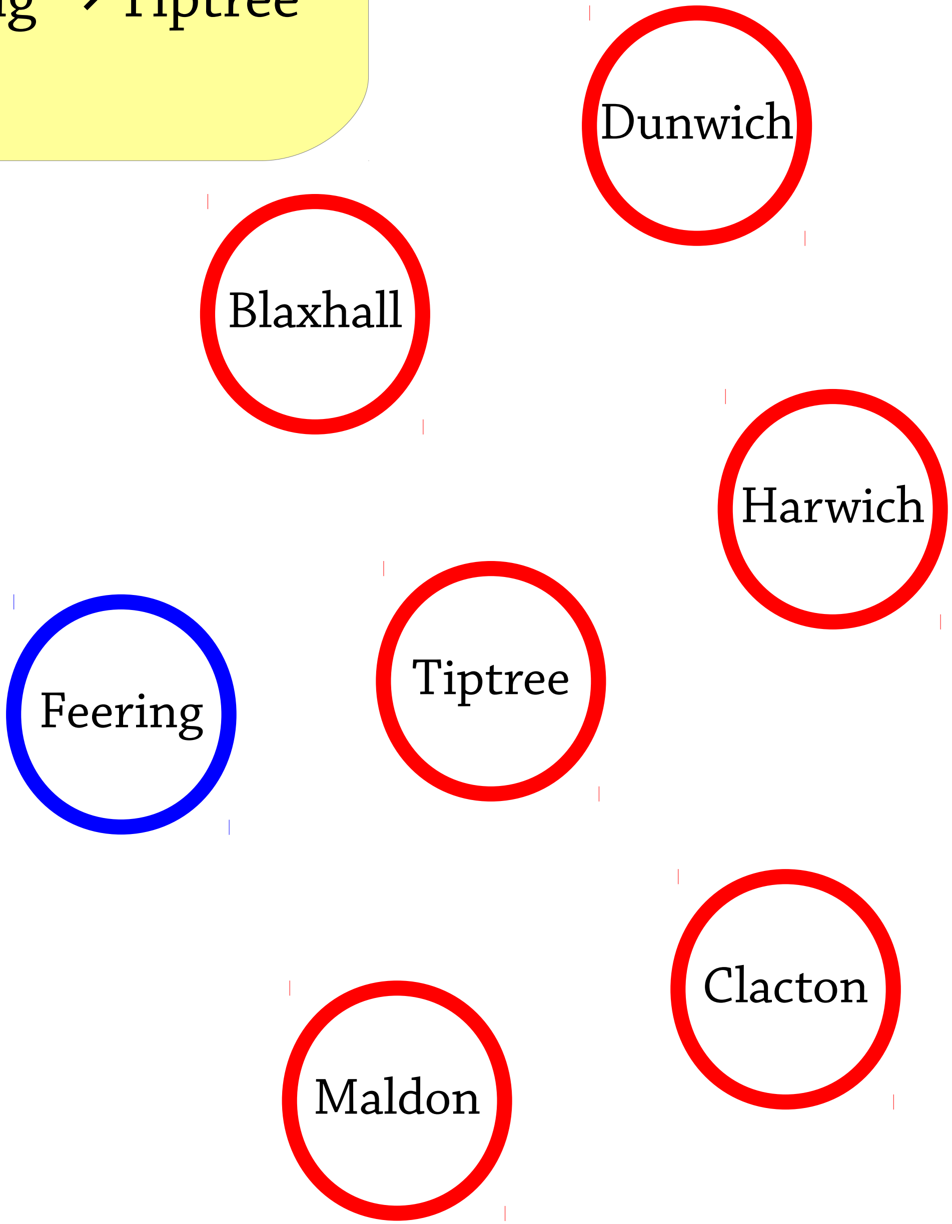
While there is a node not in S:

- 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

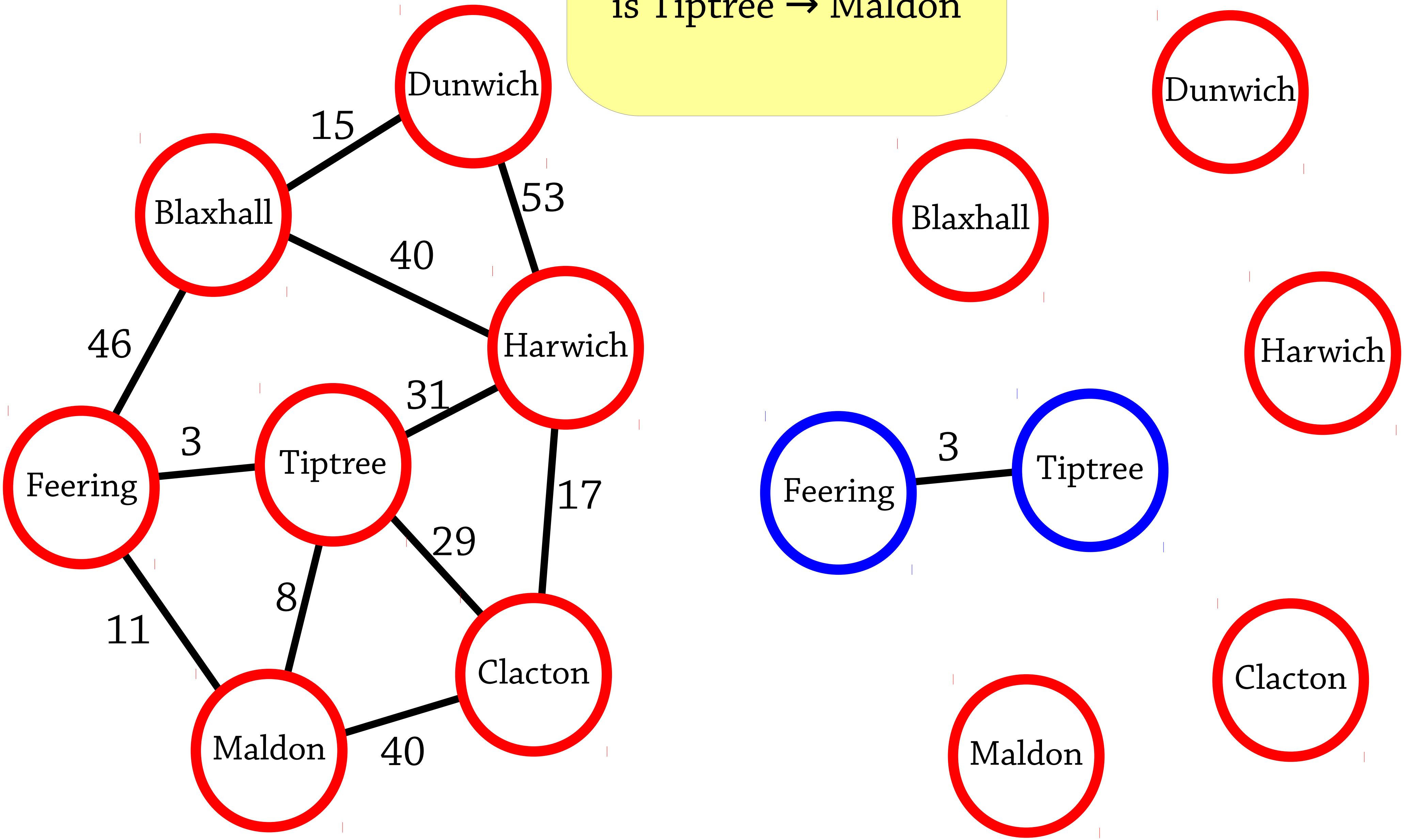
S = {Feering}
Lowest-weight edge
from S to not-S
is Feering → Tiptree





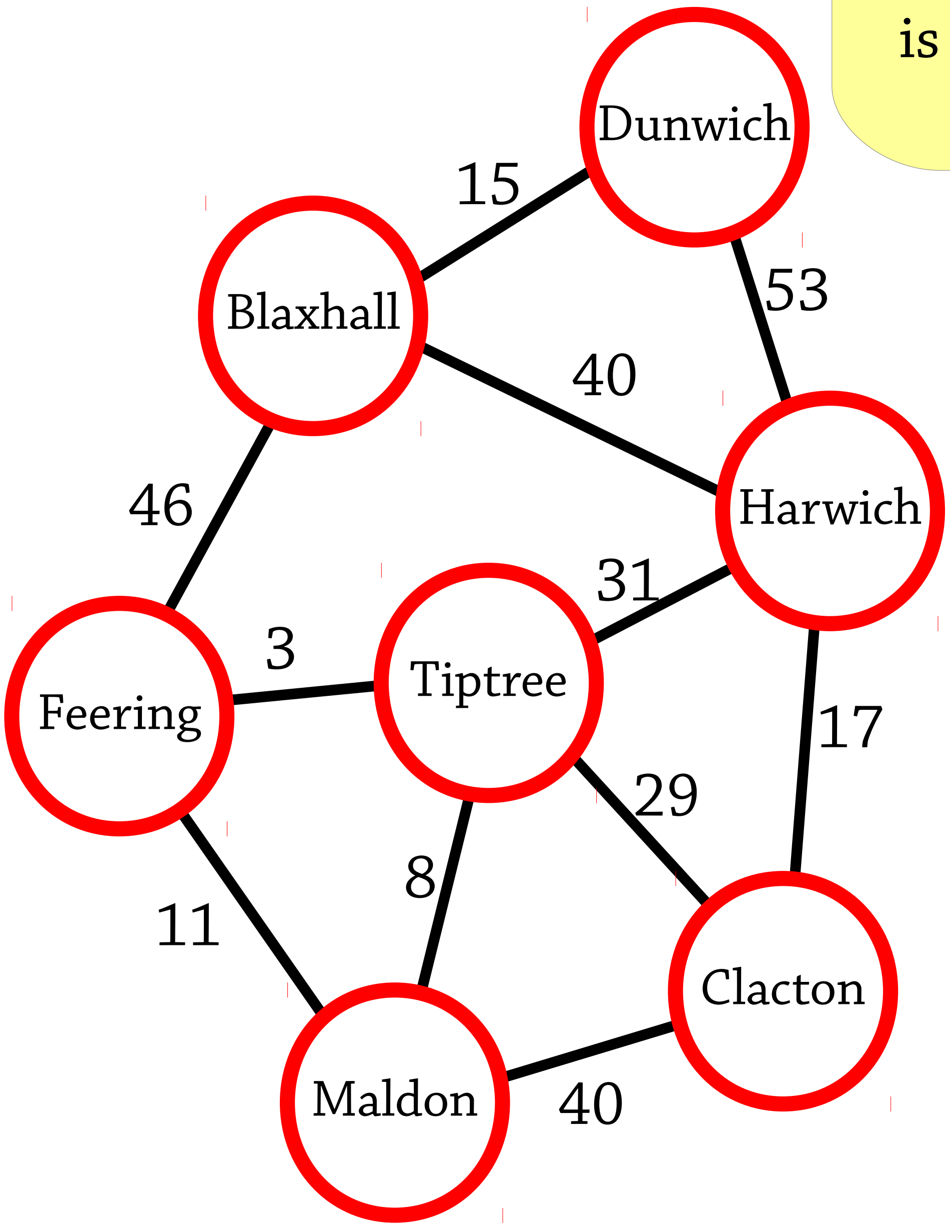
Minimun ees

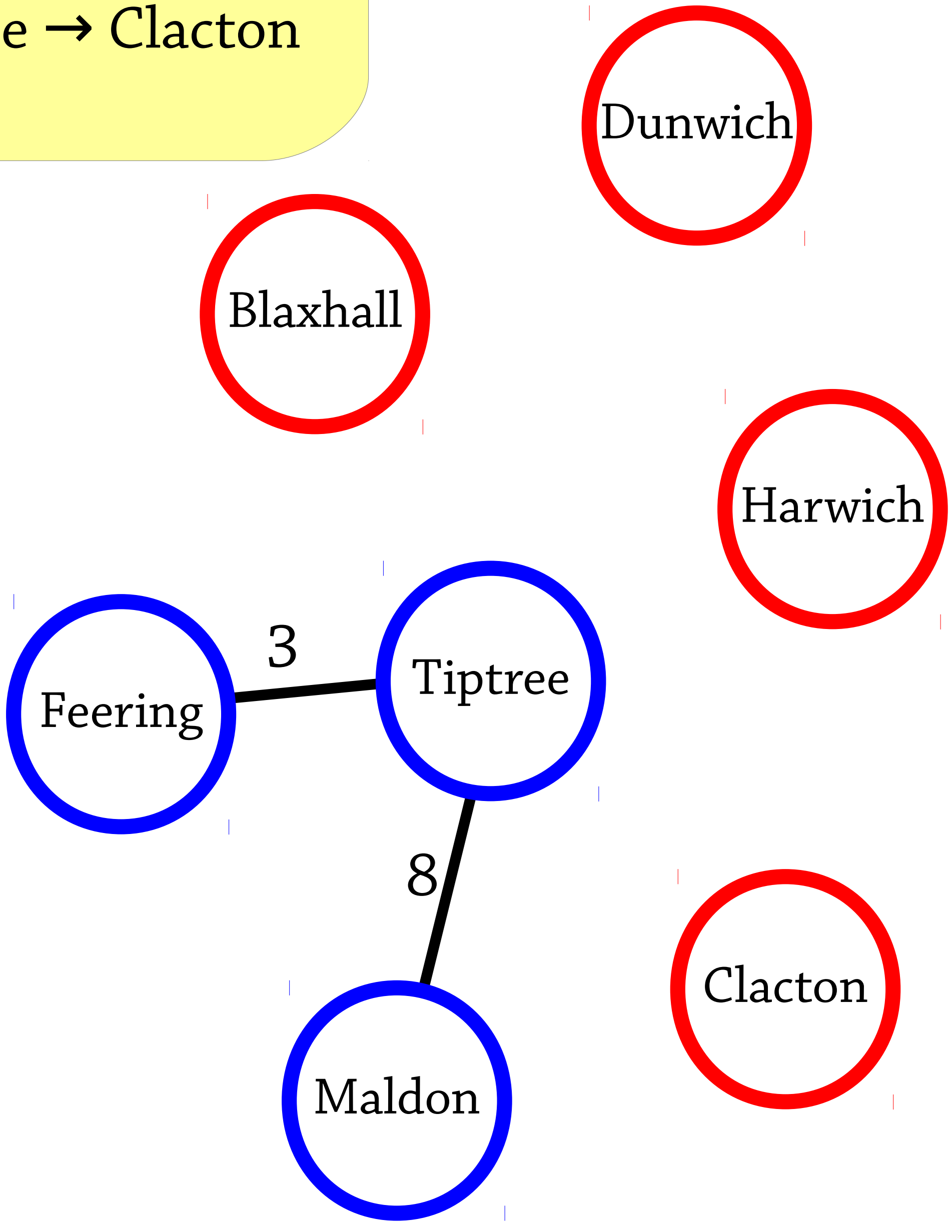
S = {Feering, Tiptree}
Lowest-weight edge
from S to not-S
is Tiptree → Maldon



Minimun ees

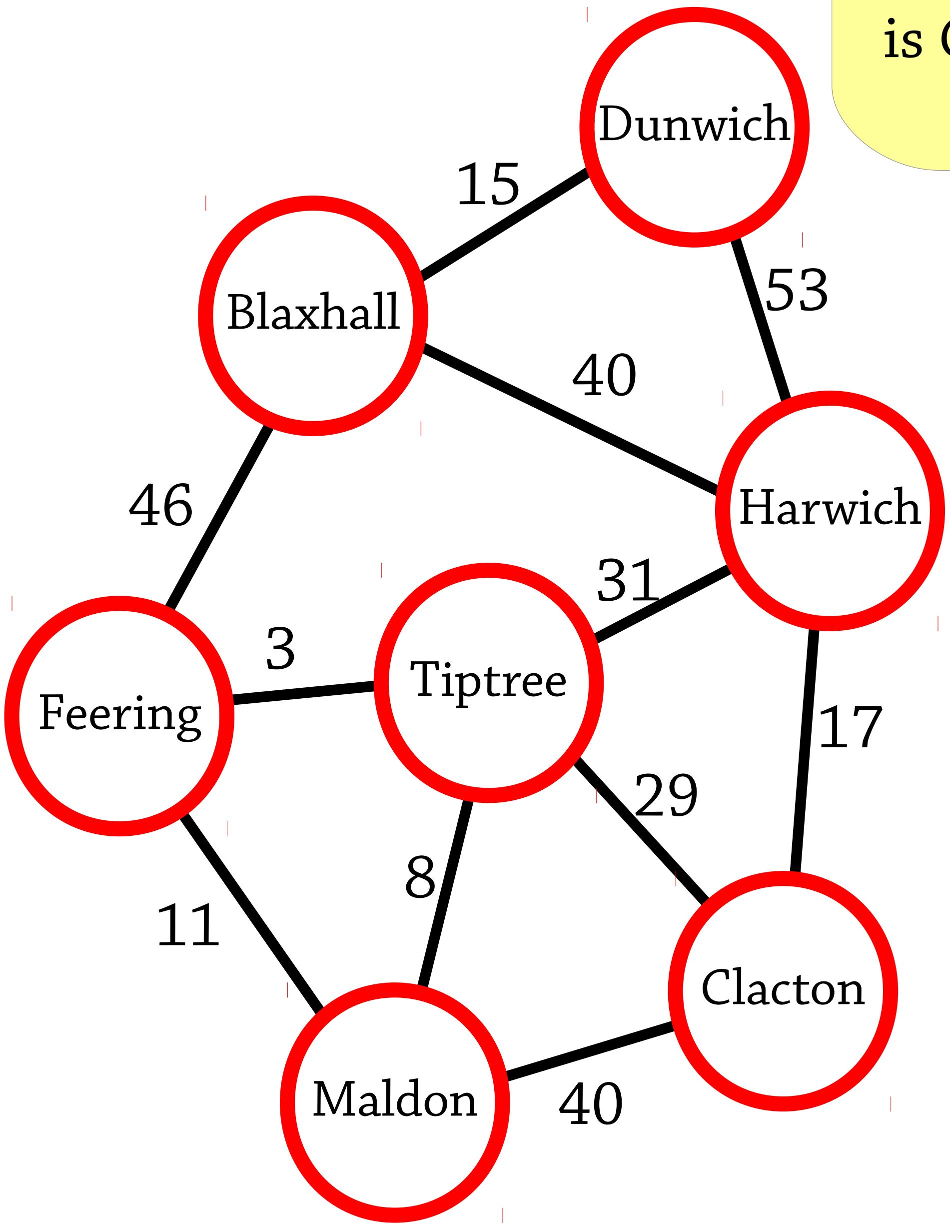
S = {Feering, Tiptree, Maldon}
Lowest-weight edge from S to not-S
is Tiptree → Clacton

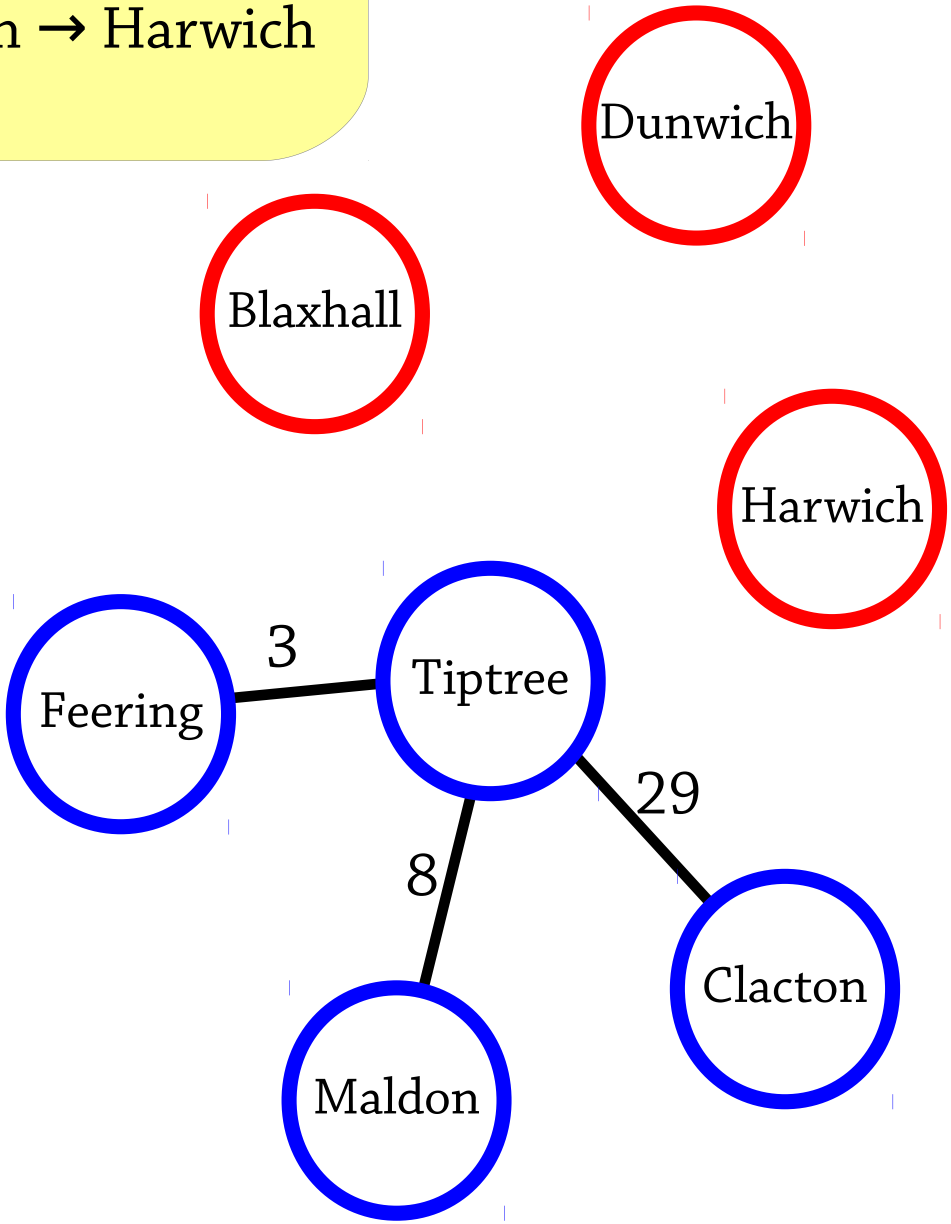




Minimun ees

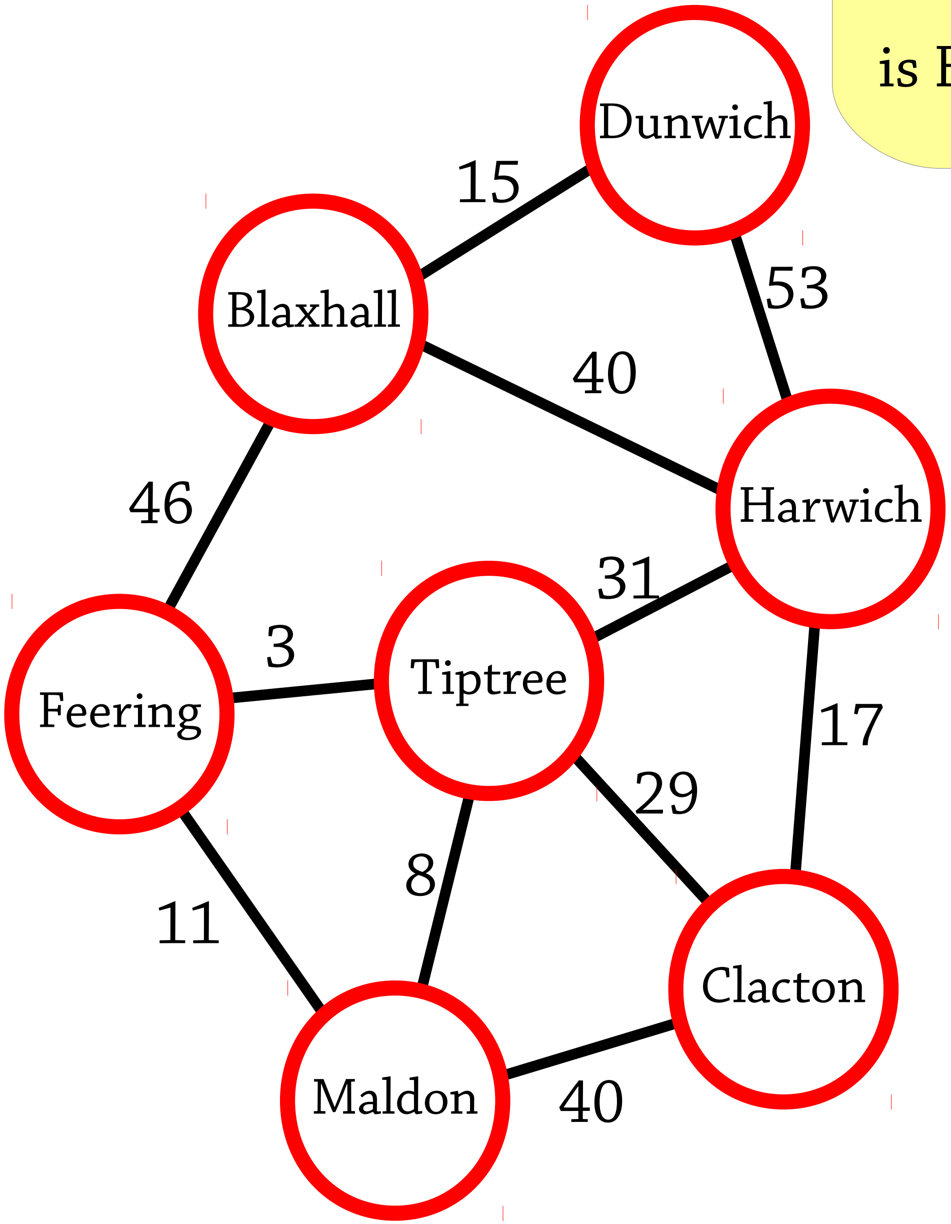
S = {Feering, Tiptree,
Maldon, Clacton}
Lowest-weight edge
from S to not-S
is Clacton → Harwich

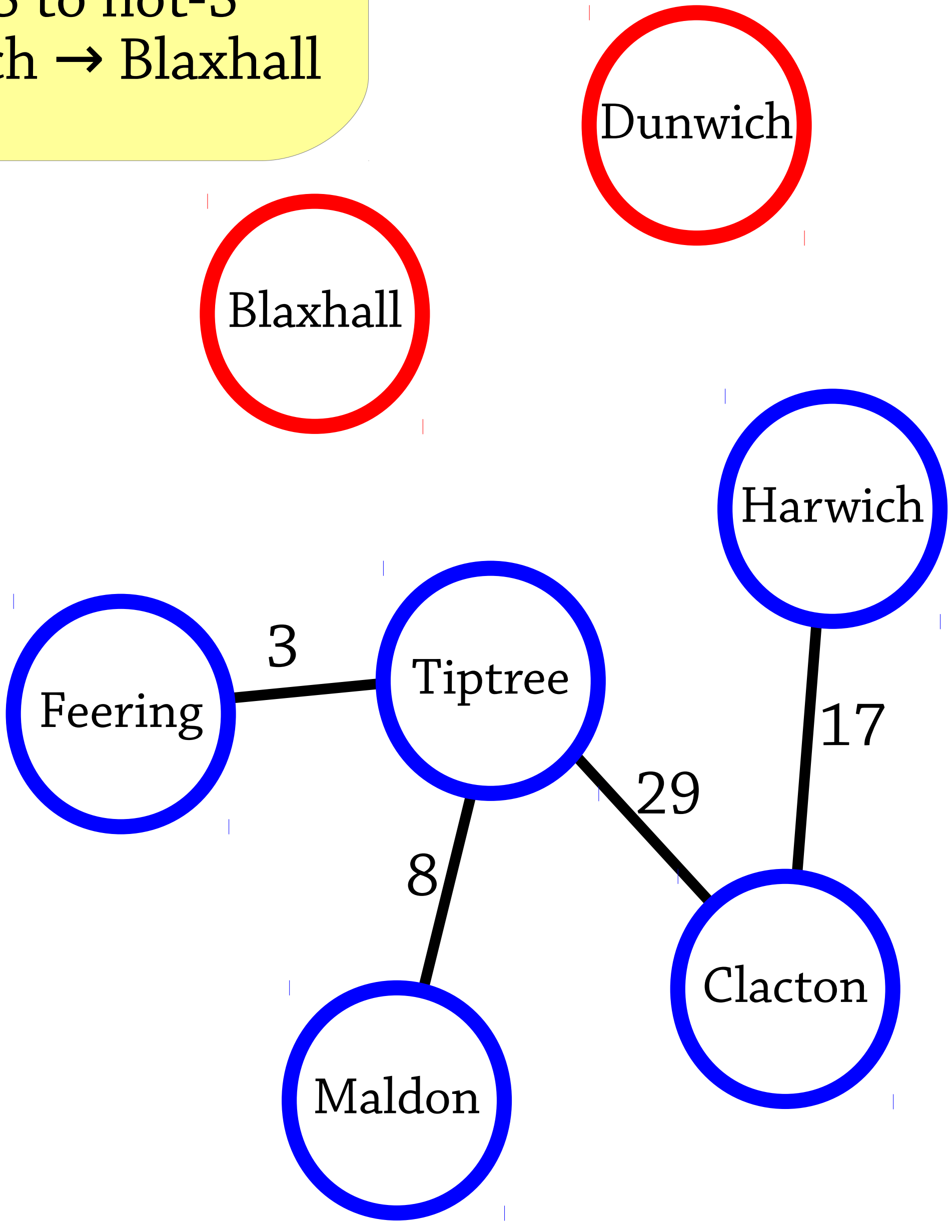




Minimun ees

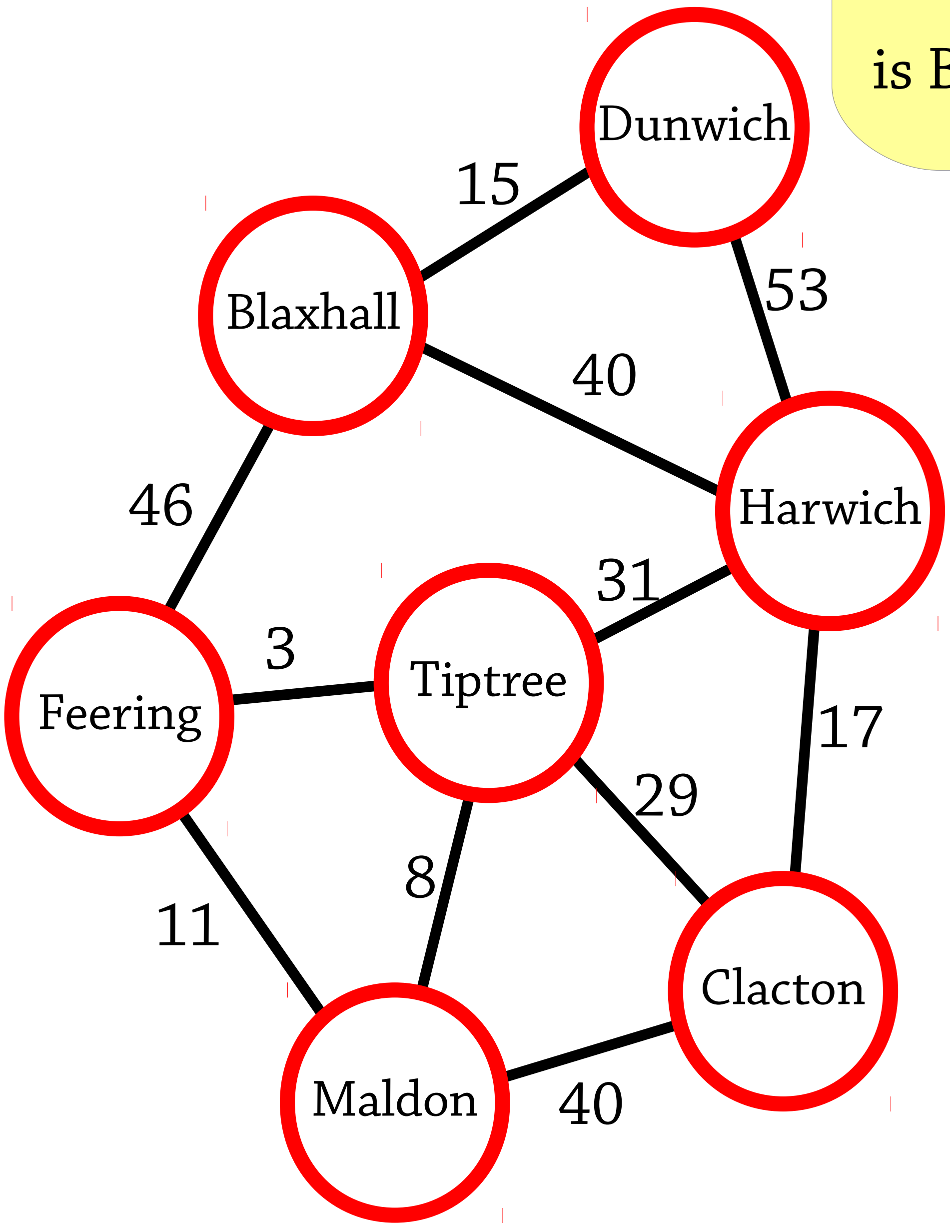
S = {Feering, Tiptree, Maldon, Clacton, Harwich}
Lowest-weight edge from S to not-S
is Harwich → Blaxhall

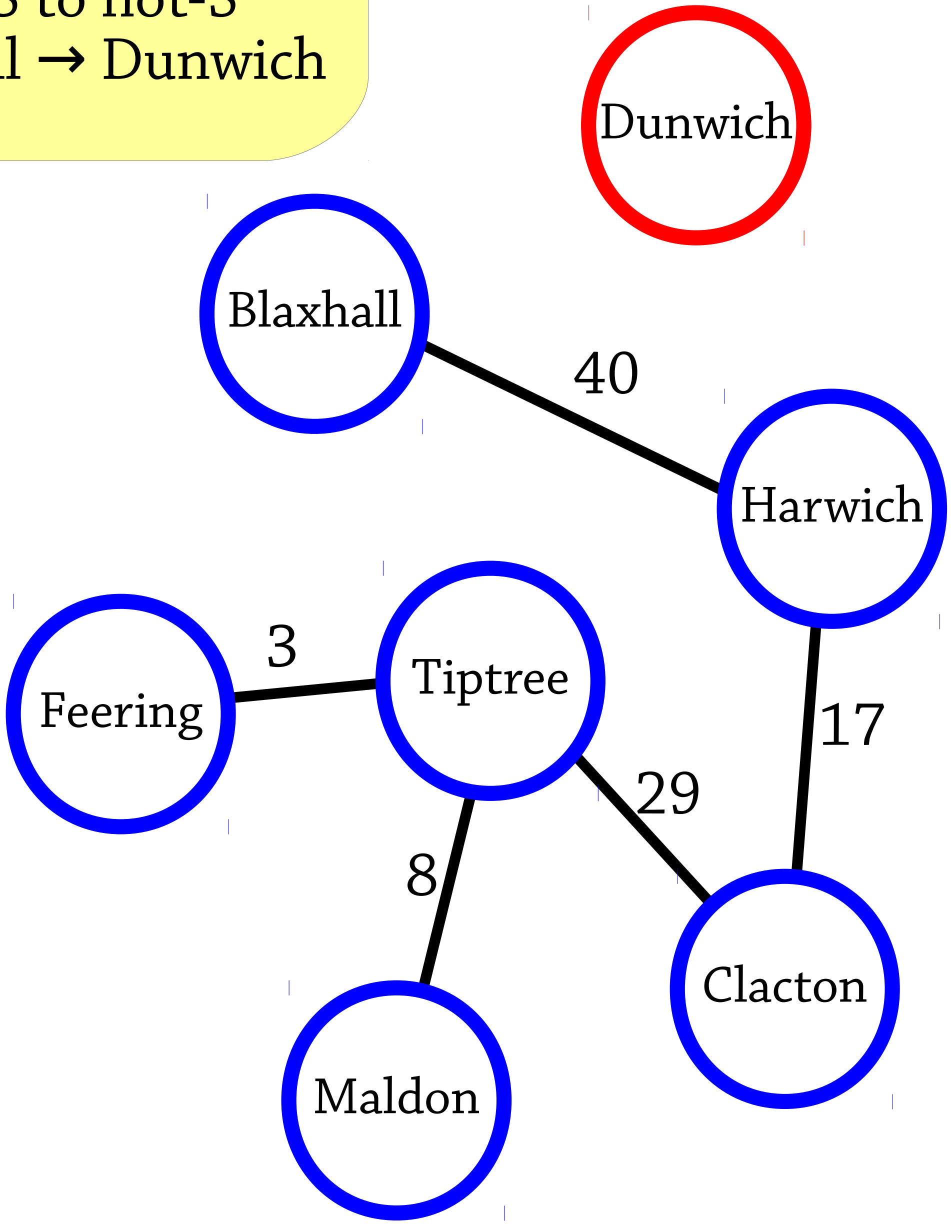




Minimun ees

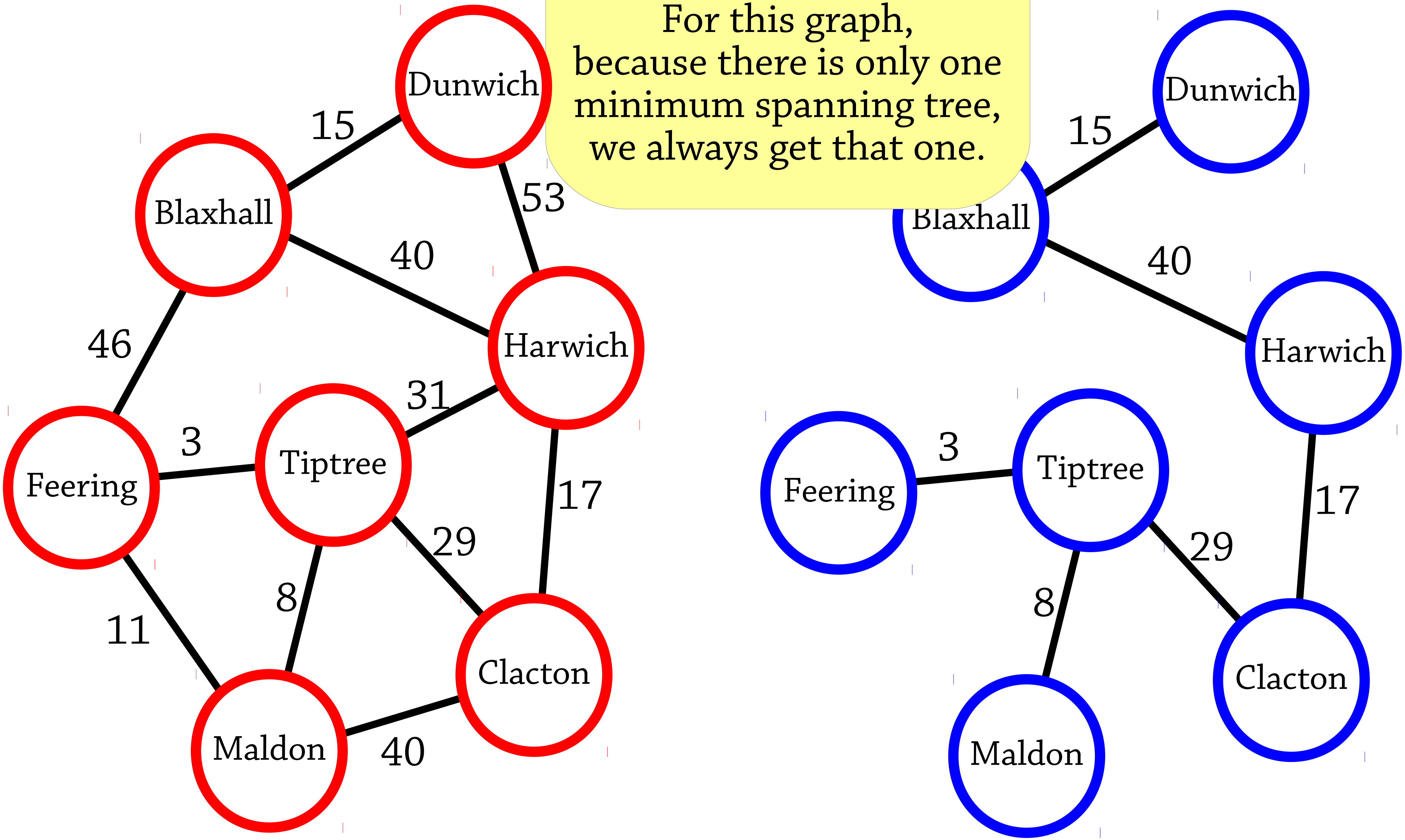
S = {Feering, Tiptree, Maldon, Clacton, Harwich, Blaxhall}
Lowest-weight edge from S to not-S
is Blaxhall → Dunwich





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 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(n log n) :)

Summary

Dijkstra's algorithm – finding shortest paths in weighted graphs – some extensions (not in course):

- Bellman-Ford: works when weights are negative
- A* - faster but assumes the *triangle inequality*

Prim's algorithm – finding minimum spanning trees

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)

Many many many more graph algorithms

- Unfortunately the book doesn't mention many – see http://en.wikipedia.org/wiki/List_of_algorithms#Graph_algorithms for a long list