Dijkstra's algorithm, Prim's algorithm

The 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

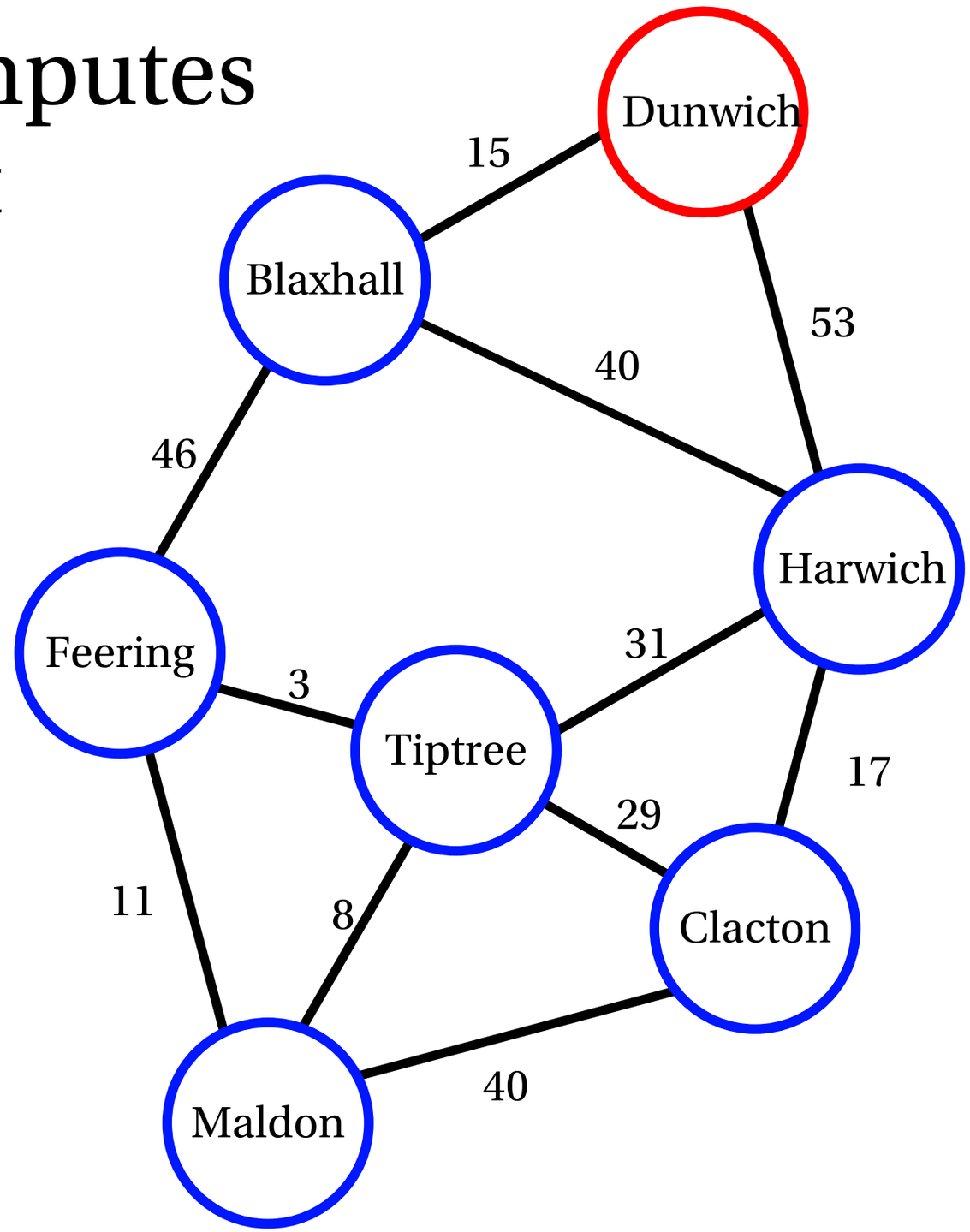
Most common approach:
Dijkstra's algorithm,
which works when all
edges have positive weight

Dijkstra's algorithm

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

Idea: maintain a set S (actually a map) of nodes whose distances we know

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



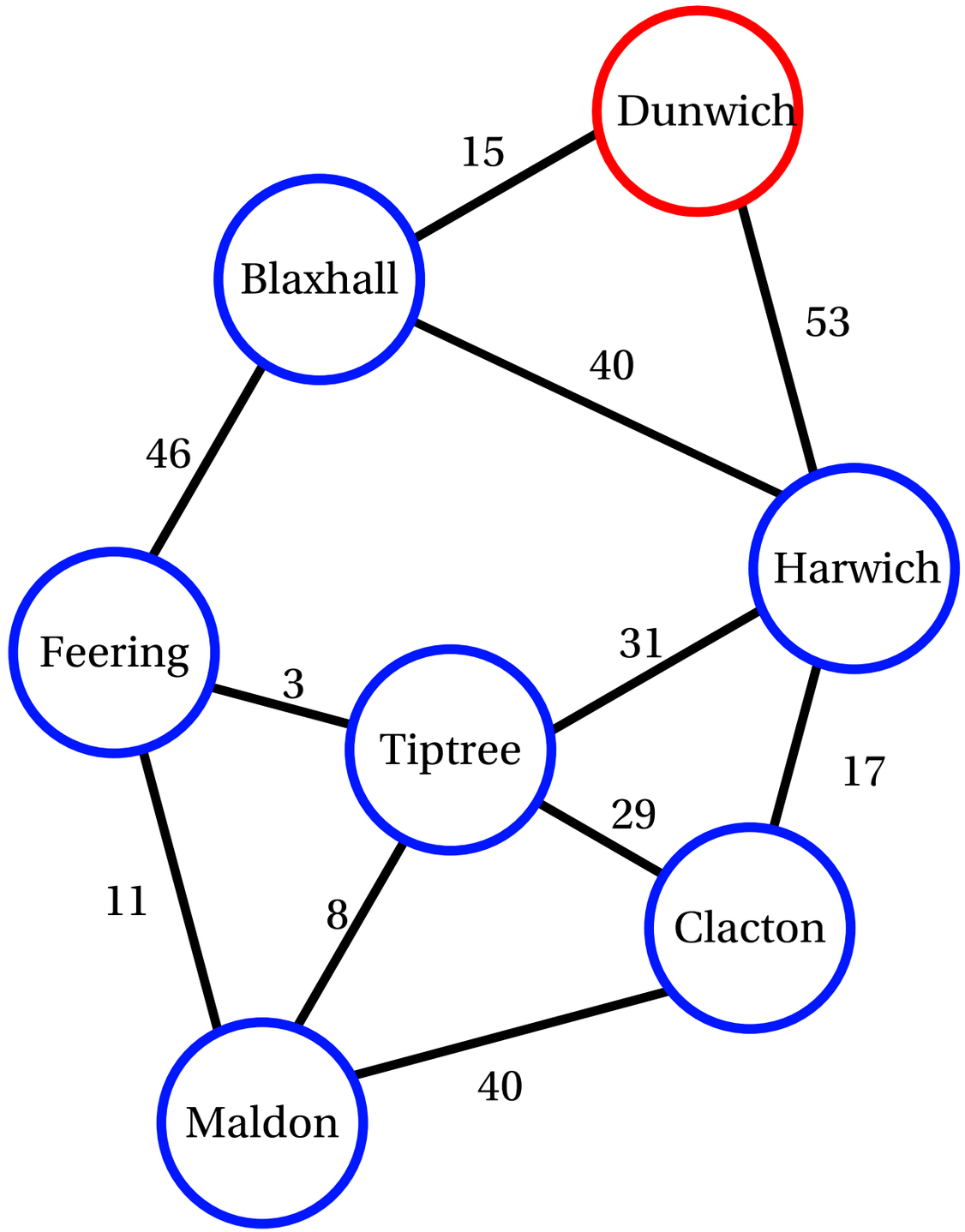
Dijkstra's algorithm

The goal: to find the shortest path that *does not lead to a node in S*

This path must be:

- The shortest path to some node in S, then
- a single edge to get to the node

(Why?)



Dijkstra's algorithm

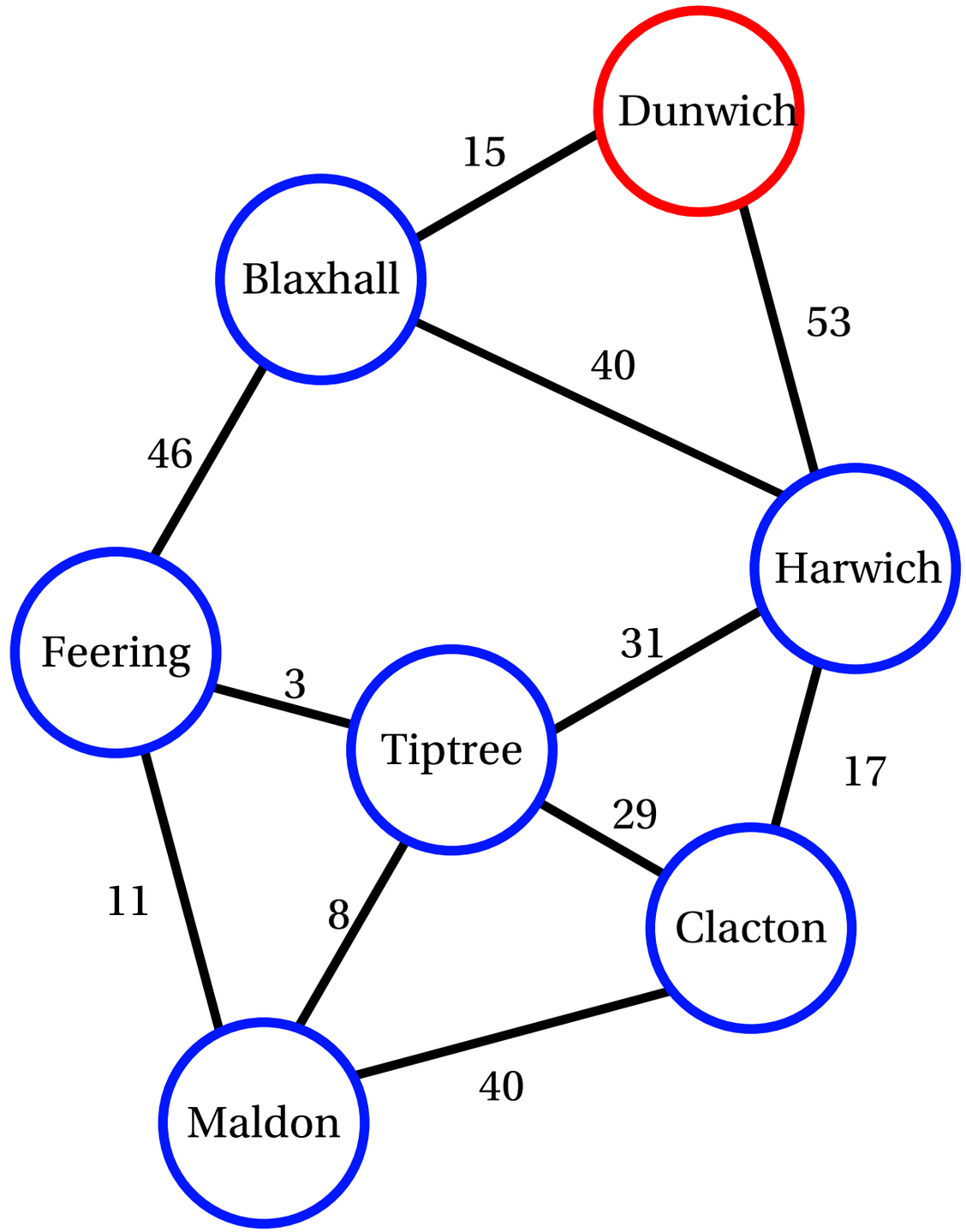
For each node x in S, and each neighbour y of x:

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

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

- The shortest path to y is: the shortest path to x, plus the edge from x to y

Repeat until all nodes are in S

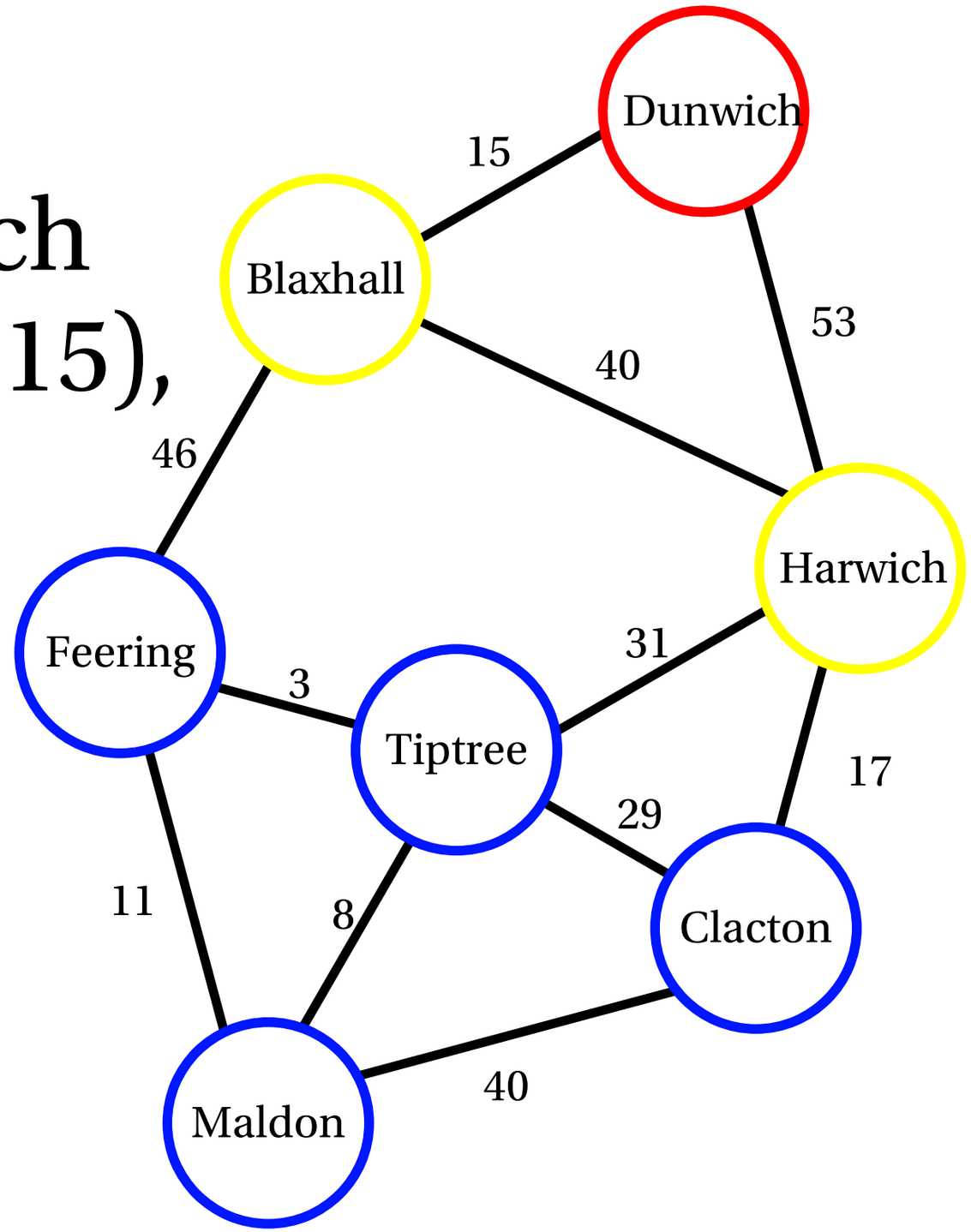


Dijkstra's algorithm

S = {Dunwich → 0}

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

So add Blaxhall → 15 to S



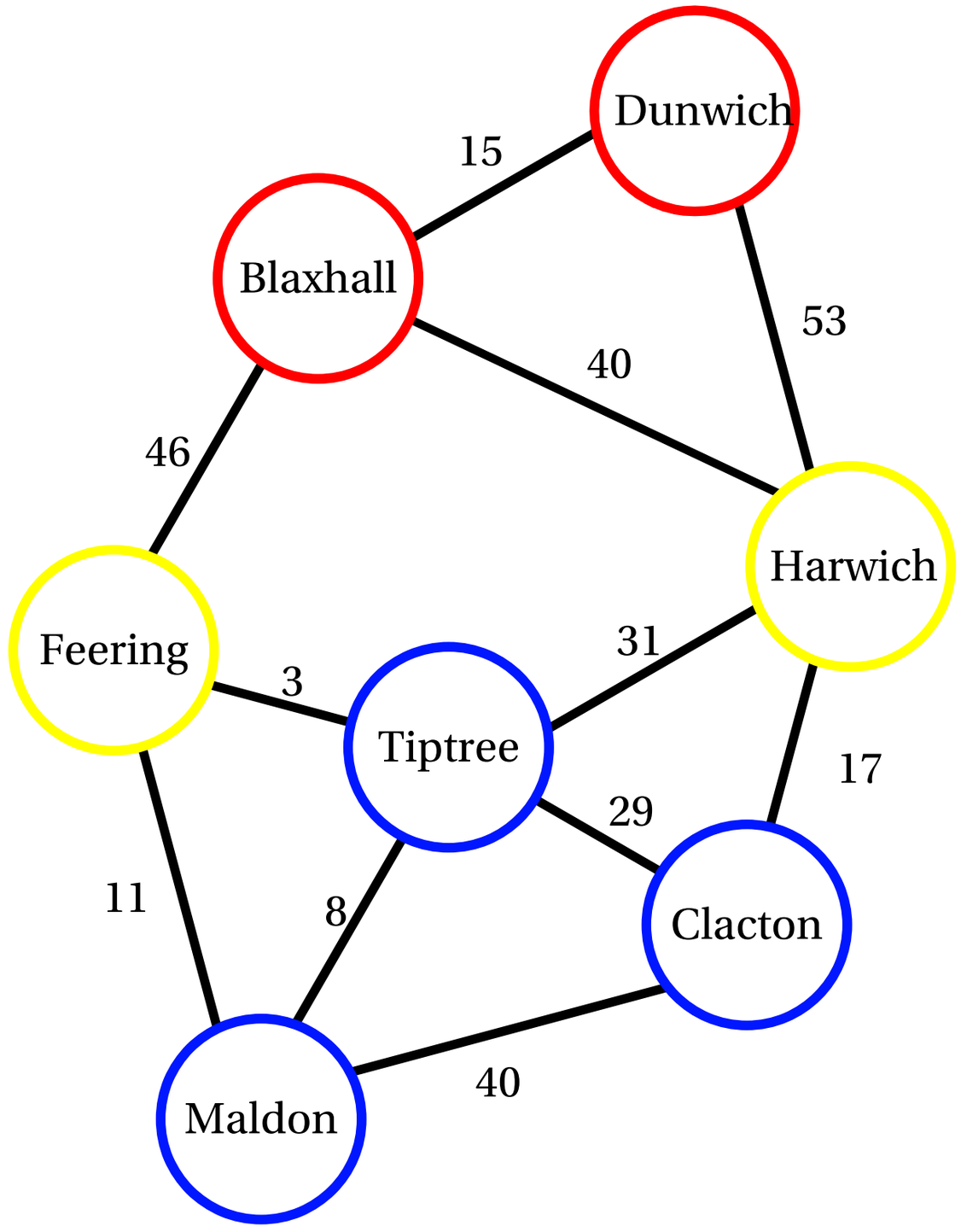
Dijkstra's algorithm

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



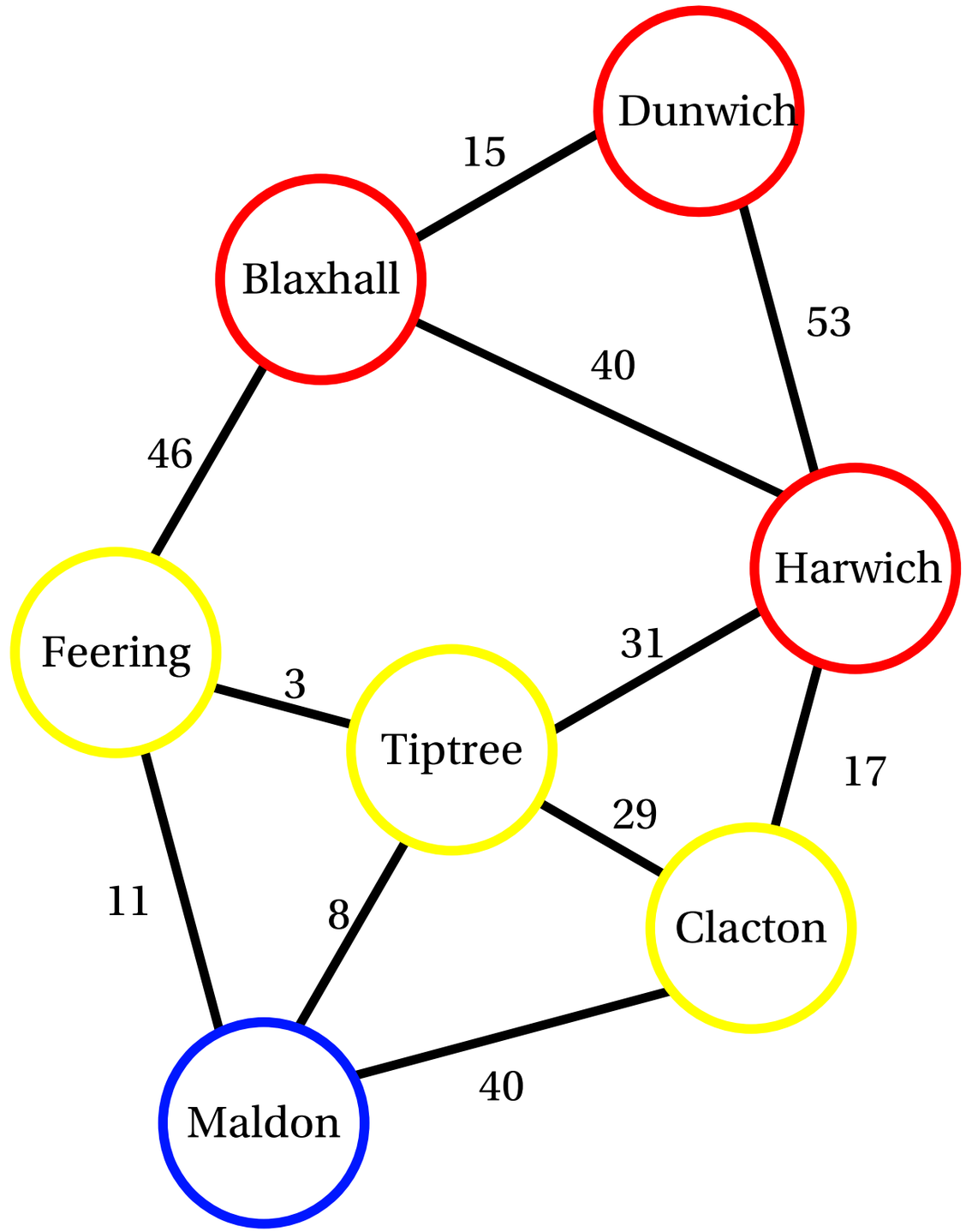
Dijkstra's algorithm

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



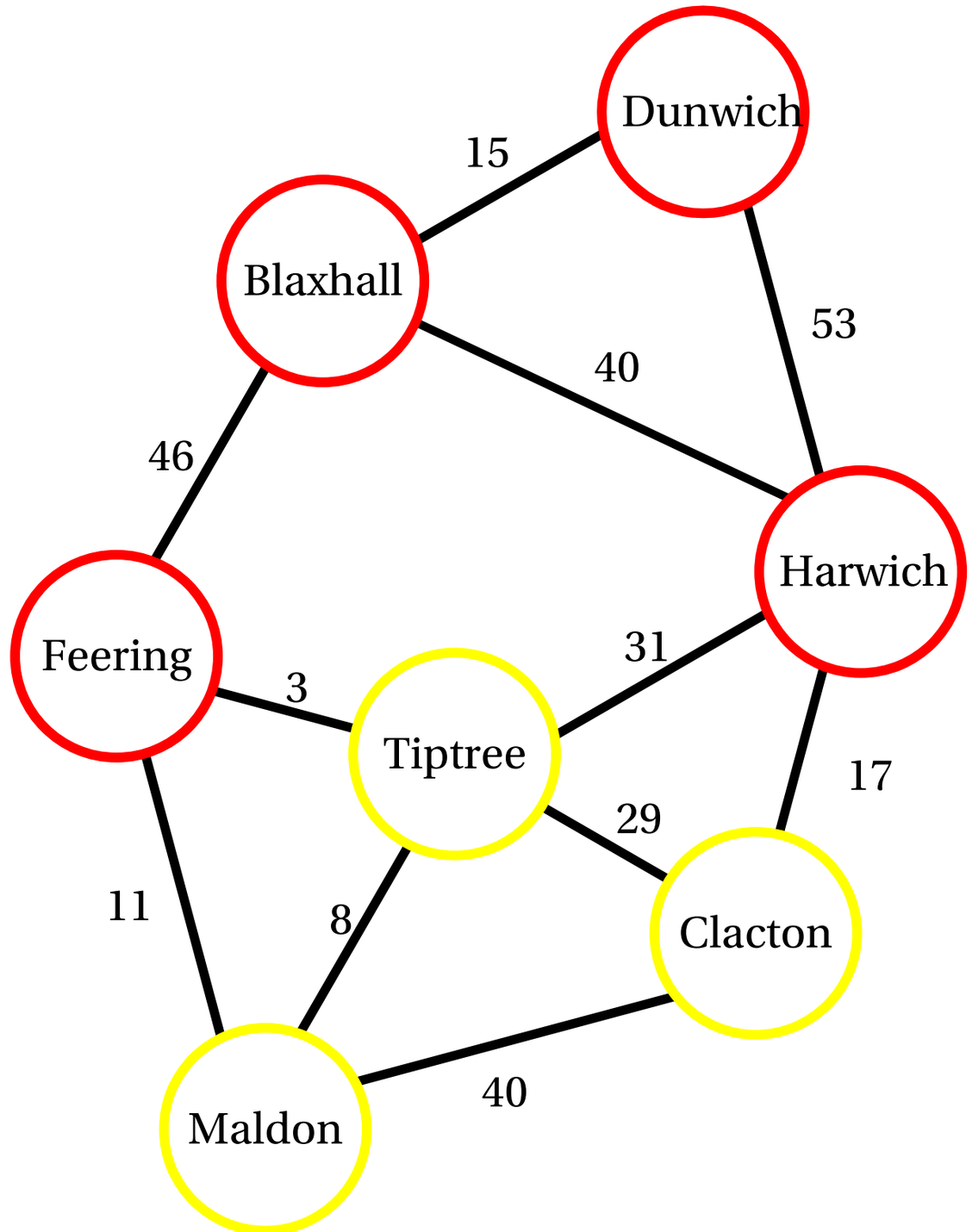
Dijkstra's algorithm

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)
- Maldon (distance 61 + 11 = 72)

So add Tiptree → 64 to S



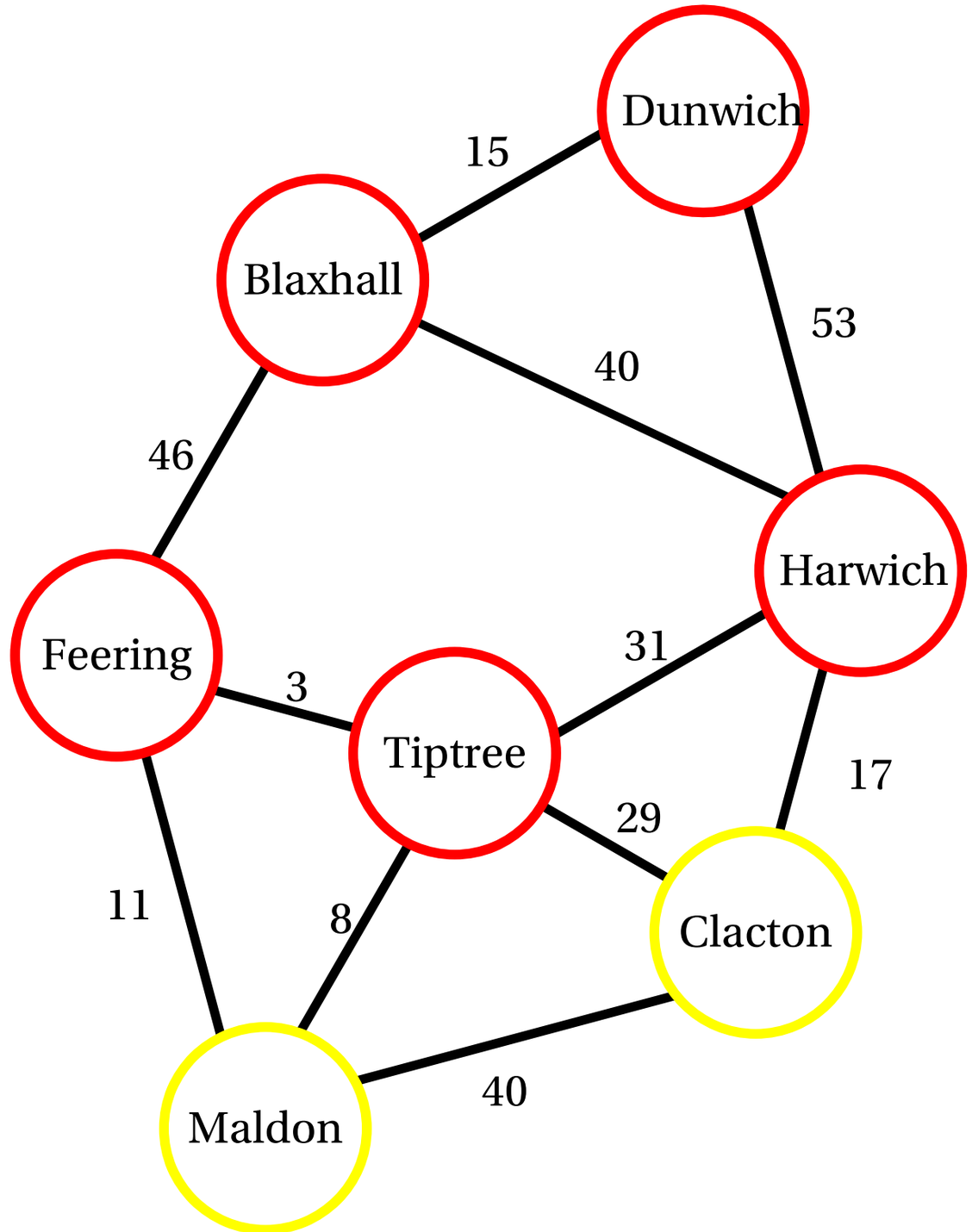
Dijkstra's algorithm

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



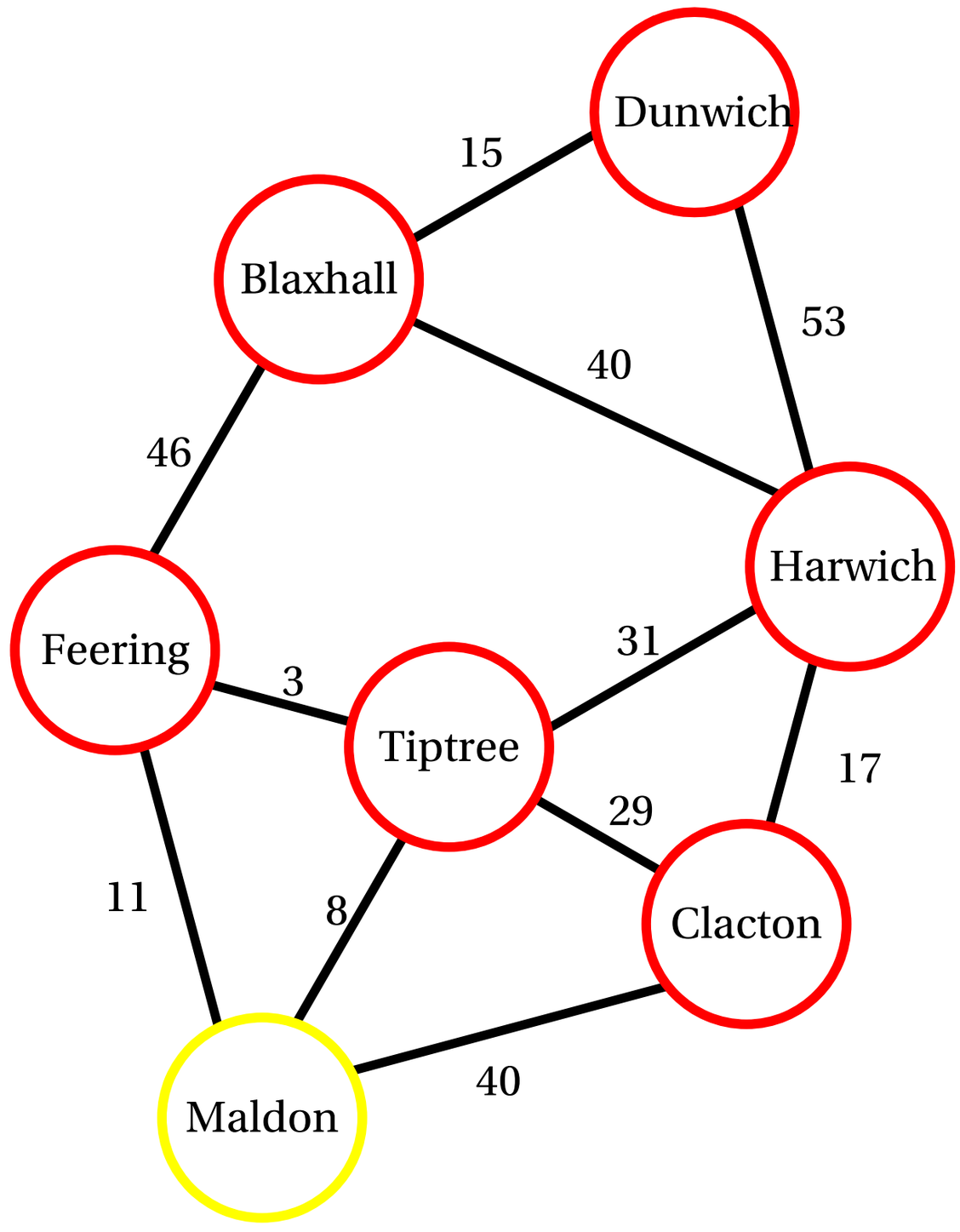
Dijkstra's algorithm

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 + =

So add Maldon → 72 to S

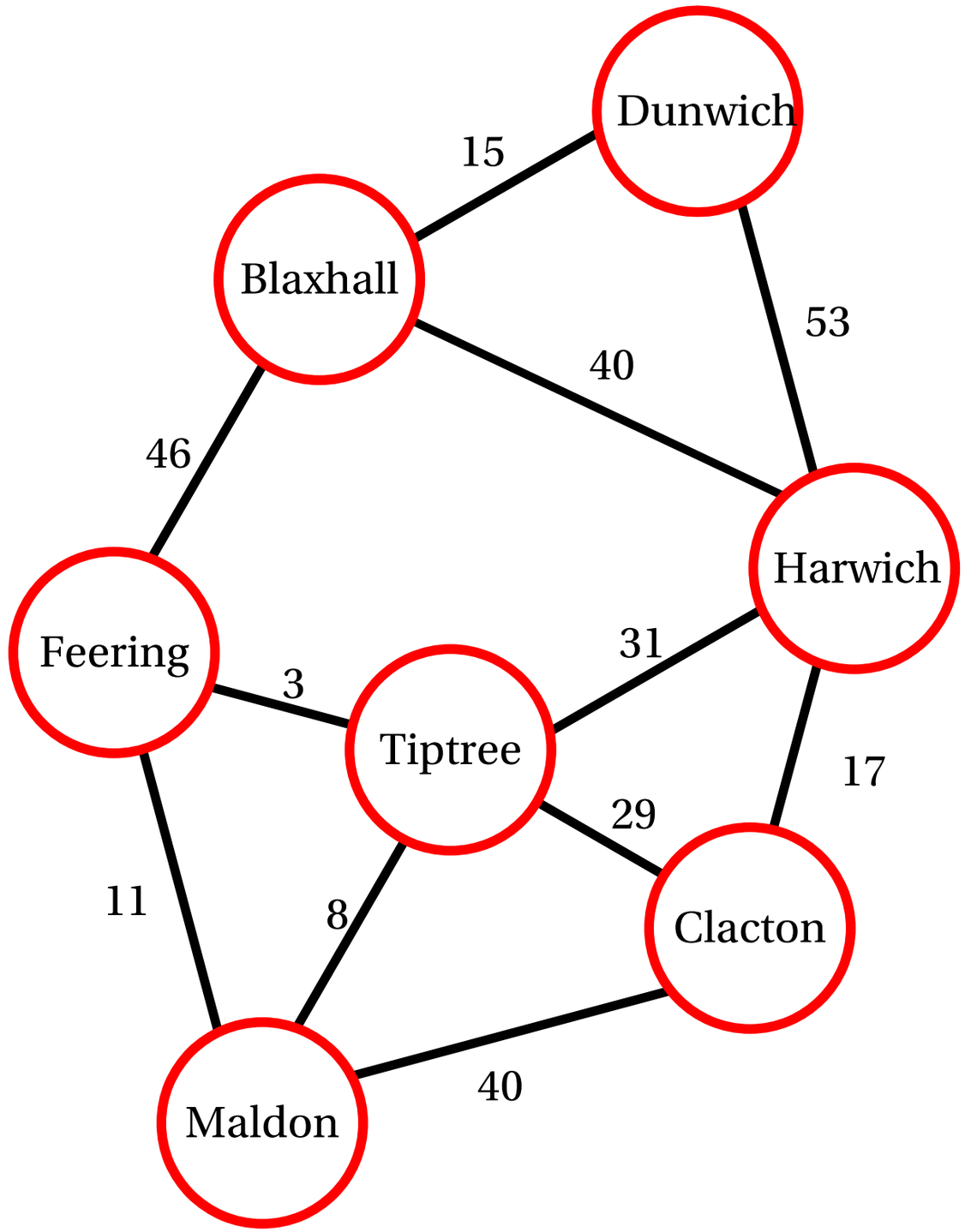


Dijkstra's algorithm

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

Let S = {start node → 0}

While not all nodes are in S,

- For each node $x \rightarrow d$ 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 $y \rightarrow d'$ to S

What is the efficiency of this algorithm?

Dijkstra's algorithm, refined

This version of the algorithm has two problems:

- It is not very efficient ($O(n^2)$, where n is the number of nodes)
- It only computes distances, not paths

The second one is easy to fix: also maintain a map *P* that maps a node to its *predecessor* in the shortest path

Dijkstra's algorithm, with paths

Let S = {start node → 0} and P = {}

While not all nodes are in S,

- For each node $x \rightarrow d$ 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 ***x* and** y and add $y \rightarrow d'$ to S**, and *y* → *x* to P**

You can find the path from the start node to any node by tracing backwards through P

Dijkstra's algorithm with paths, an excerpt

S = {Dunwich → 0,
Blaxhall → 15}

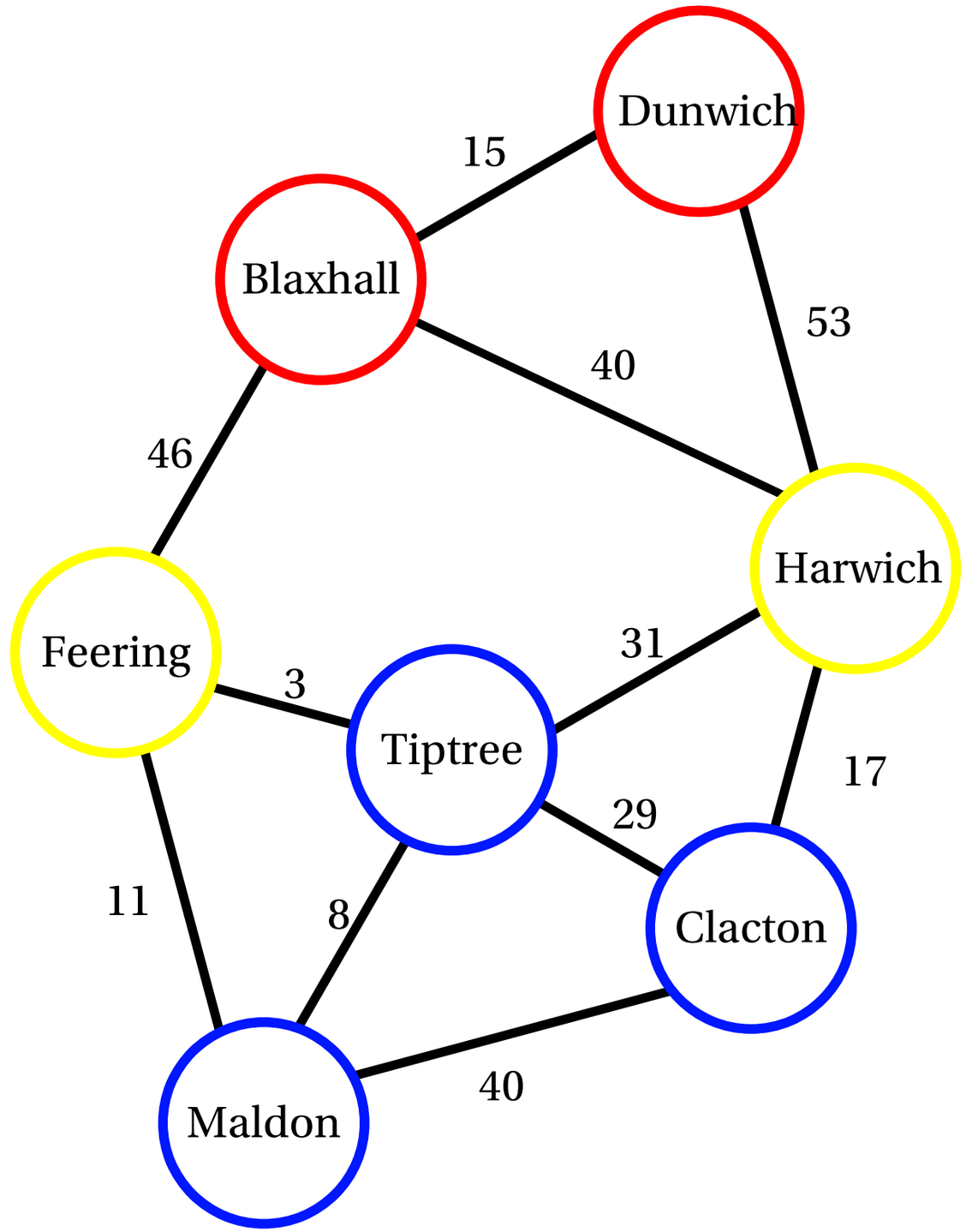
P = {Blaxhall → Dunwich}

Neighbours of S are:

- Feering (distance 61 via Blaxhall)
- Harwich (distance 53 via Dunwich)

So add Harwich → 53 to S

Also add
Harwich → Dunwich to P



Dijkstra's algorithm with paths, an excerpt

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

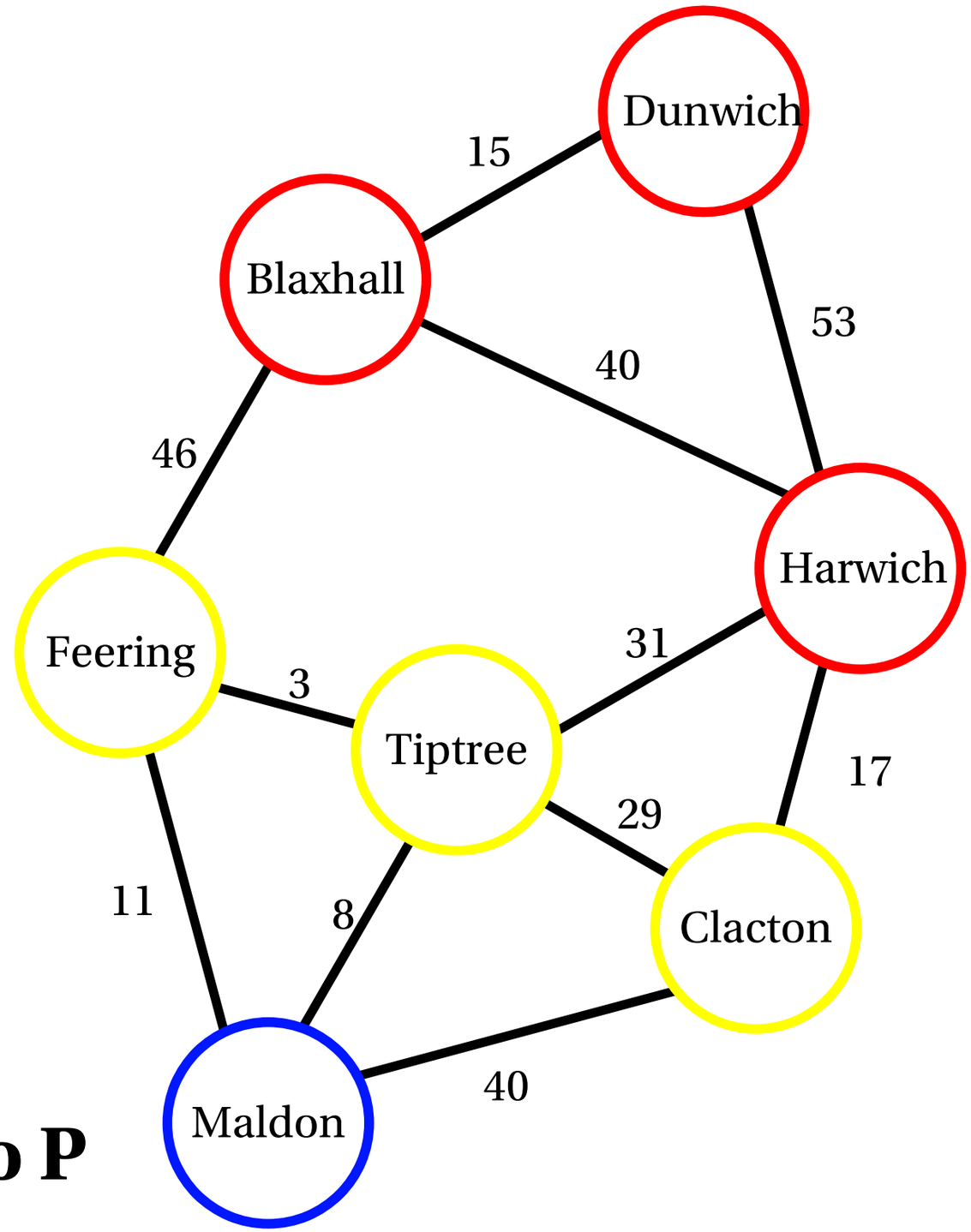
P = {Blaxhall → Dunwich,
Harwich → Dunwich}

Neighbours of S are:

- Feering (distance 61 via Blaxhall)
- Tiptree (distance 84 via Harwich)
- Clacton (distance 70 via Harwich)

So add Feering → 61 to S

Also add Feering → Blaxhall to P



Dijkstra's algorithm with paths, an excerpt

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

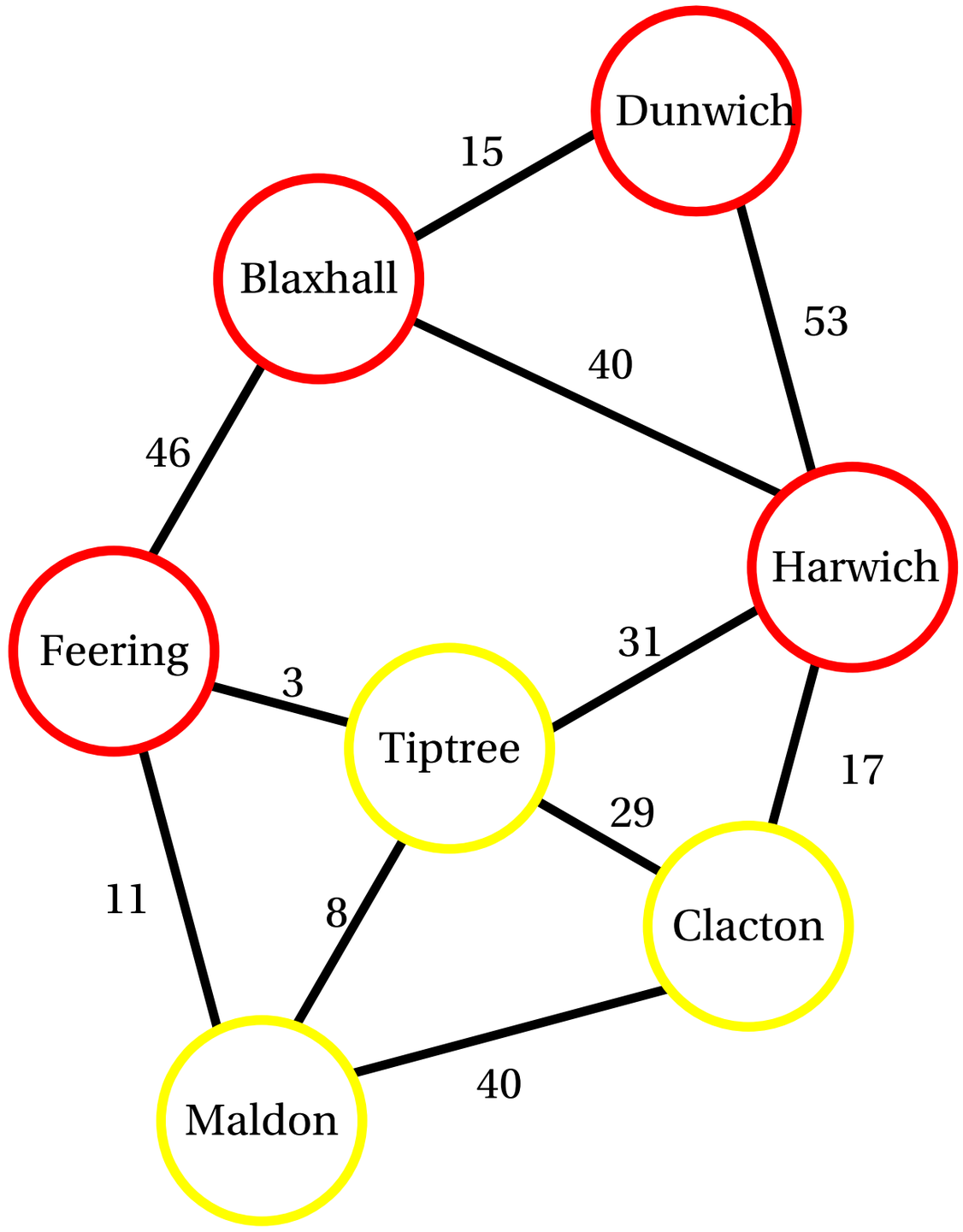
P = {Blaxhall → Dunwich,
Harwich → Dunwich,
Feering → Blaxhall}

Neighbours of S are:

- Tiptree (distance 64 via Feering)
- Clacton (distance 53 via Harwich)
- Maldon (distance 61 via Feering) 61 + 11 = 72)

So add Tiptree → 64 to S

Also add Tiptree → Feering to S



Dijkstra's algorithm with paths, an excerpt

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

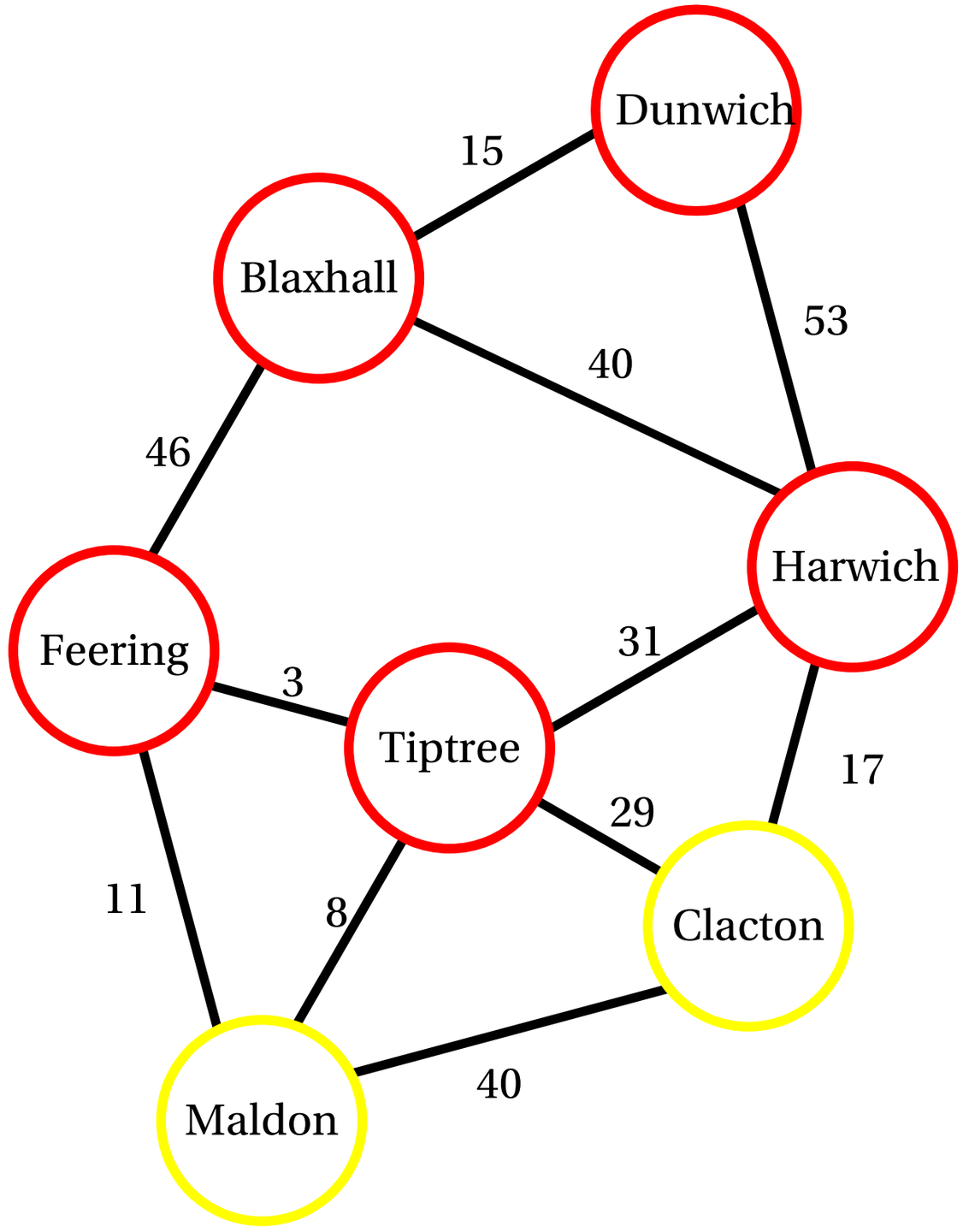
P = {Blaxhall → Dunwich,
Harwich → Dunwich,
Feering → Blaxhall,
Tiptree → Feering}

We know Tiptree is 64 away from Dunwich

If we want to know the path, look in P:

- Tiptree → Feering
- Feering → Blaxhall
- Blaxhall → Dunwich

So the shortest path is
Dunwich → Blaxhall → Feering → Tiptree



Dijkstra's algorithm, made efficient

The other problem with Dijkstra's algorithm so far was that it's $O(n^2)$

This is because this step:

- For each node $x \to d$ in S, and each neighbour y of x, calculate $d' = d + \textit{cost of edge from } x \textit{ to } y$

takes O(n) time, and we execute it for every node we add to S

How can we make this faster?

Dijkstra's algorithm, made efficient

Answer: use a priority queue!

The queue will store, for each neighbour of a node in S, a record containing:

- The node
- The node's predecessor in the path
- The distance to the node following that path

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

Instead of searching for the nearest node not in S, we will ask the priority queue for the record with the smallest distance

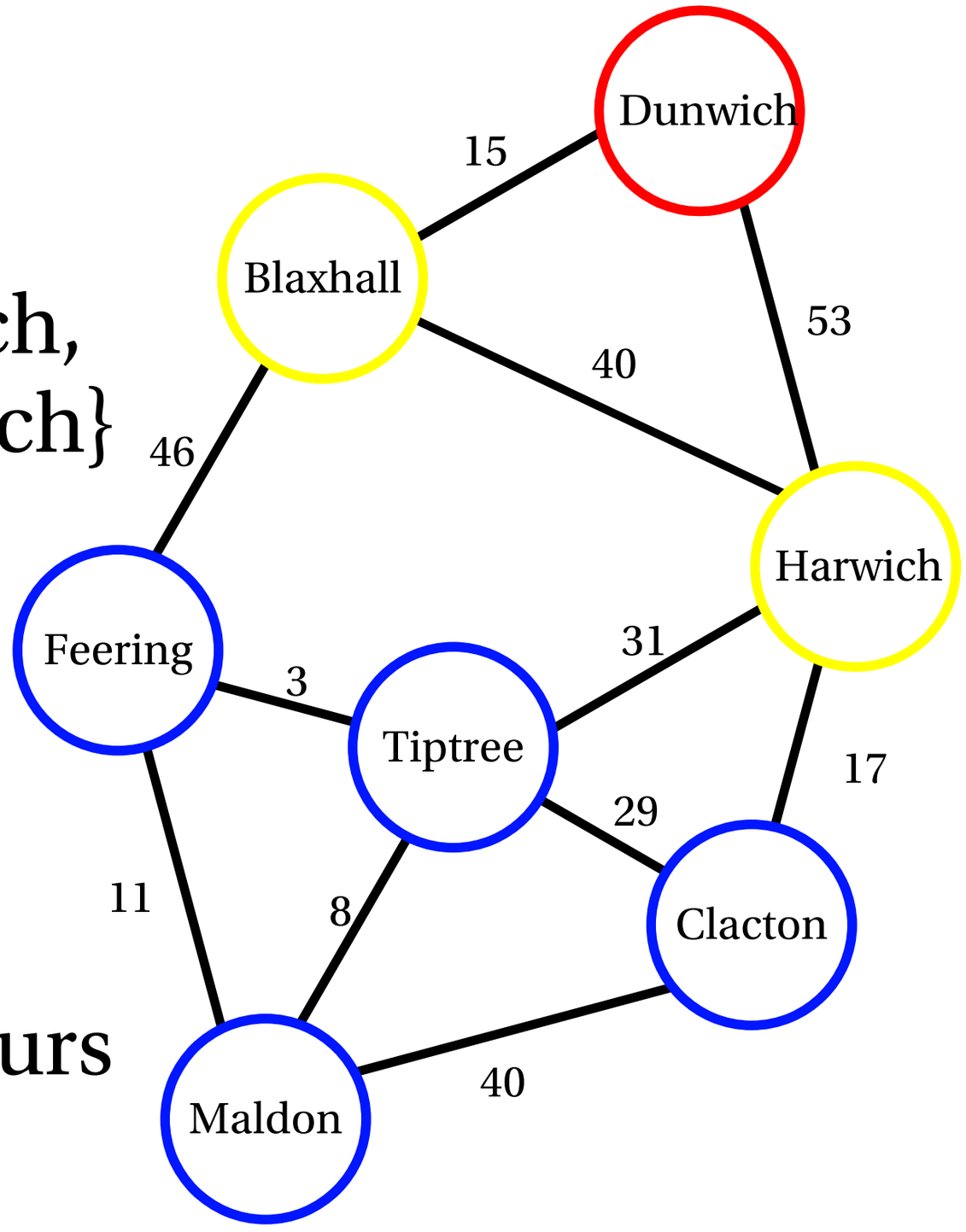
Dijkstra's algorithm

S = {Dunwich → 0}

P = {}

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

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



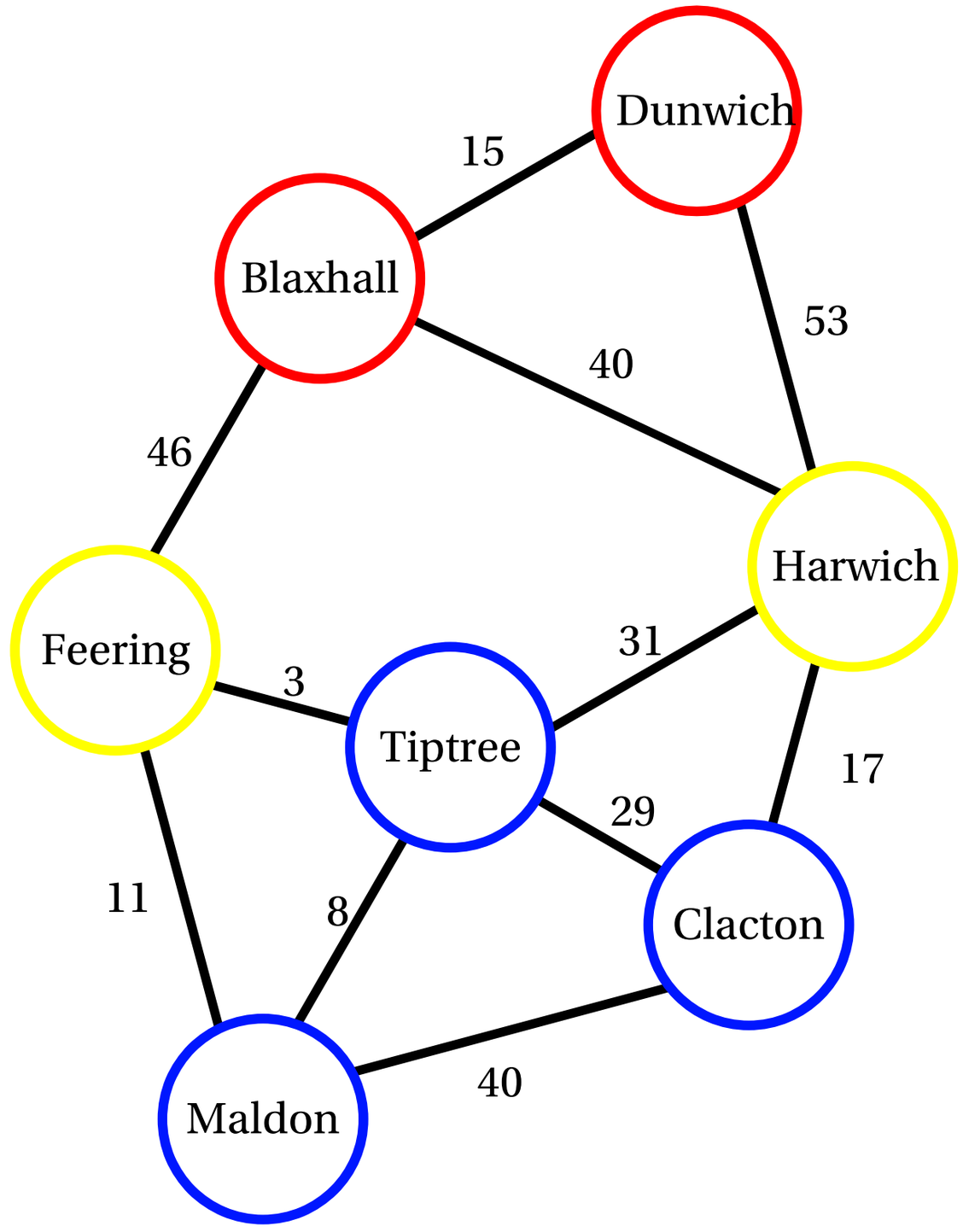
Dijkstra's algorithm

S = {Dunwich → 0,
Blaxhall → 15}

P = {Blaxhall → Dunwich}

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

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



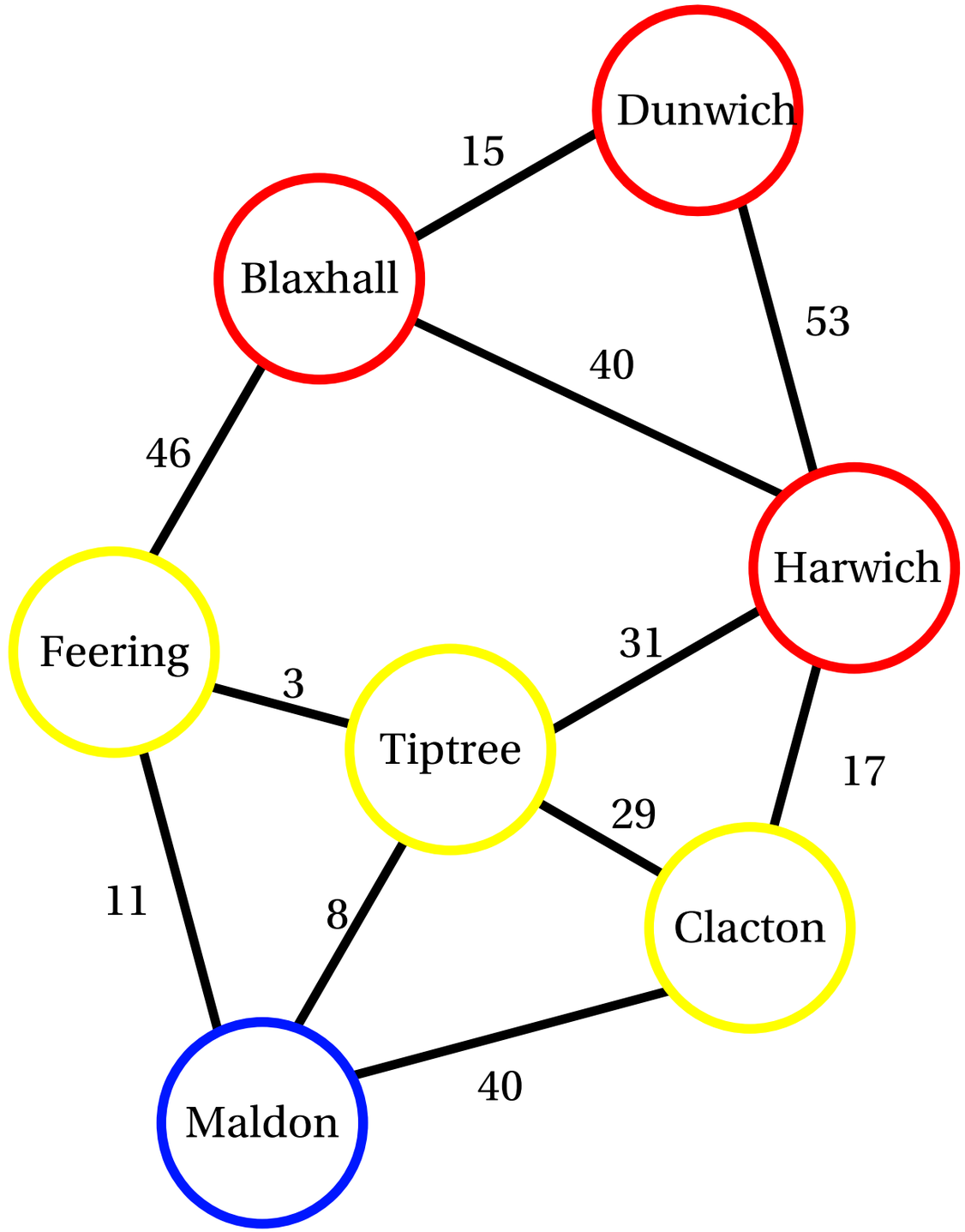
Dijkstra's algorithm

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

P = {Blaxhall → Dunwich,
Harwich → Dunwich}

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

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



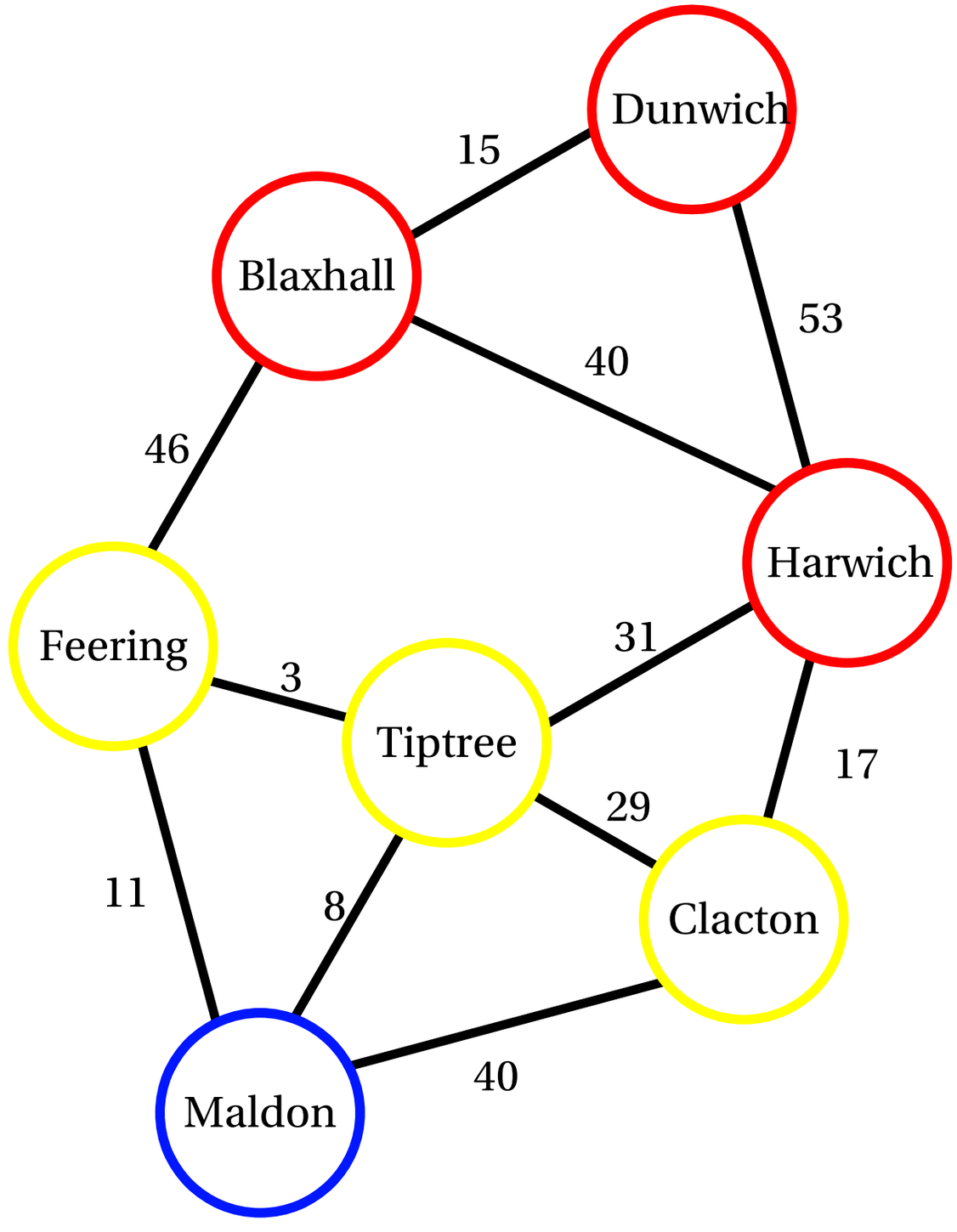
Dijkstra's algorithm

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

P = {Blaxhall → Dunwich,
Harwich → Dunwich}

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

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

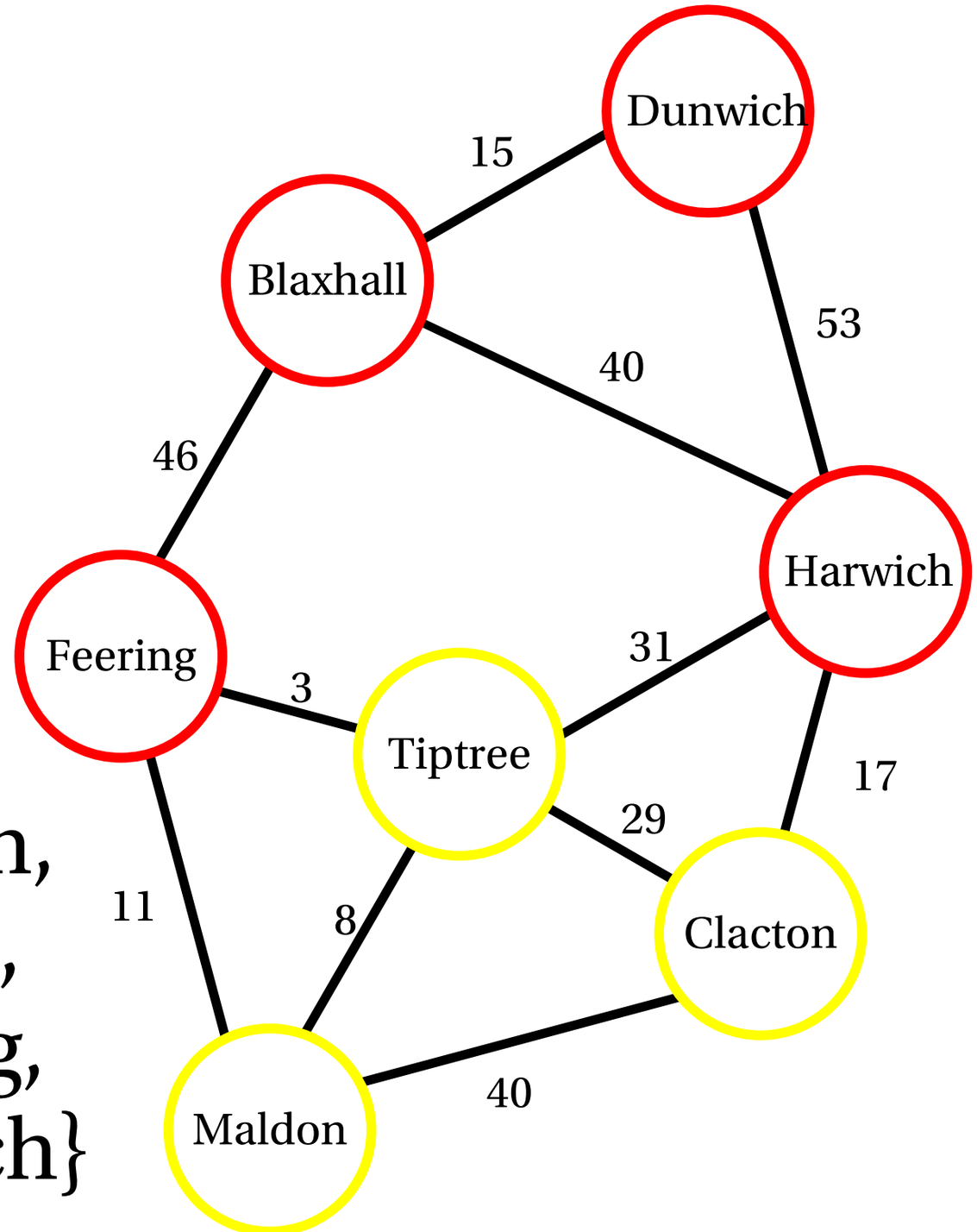


Dijkstra's algorithm

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

P = {Blaxhall → Dunwich,
Harwich → Dunwich,
Feering → Blaxhall}

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



Dijkstra's algorithm, efficiently

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

Add a record *"x* distance *d* via start node" for all of the start node's neighbours *x* (where the edge has weight *d*)

While not all nodes are in S,

- Remove the smallest element "y distance d via x" of Q (the record that has the smallest distance)
- If *y* is in S, do nothing
- Otherwise, add y → d to S, y → x to P, and for all of *y*'s neighbours *z* add "*z* distance *(d + weight of edge from y to z)* via *y*" to Q

Dijkstra's algorithm, efficiently

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

Add a record *"x* distanc r all of the start node's neigh edge has weight *d*)

While not all nodes are

- Remove the smallest element *y* distance *d* via P of Q (the record that has the smallest distance)
- If *y* is in S, do nothing
- Otherwise, add y → d to S, y → x to P, and for all of *y*'s neighbours *z* add "*z* distance *(d + weight of edge from y to z)* via *y*" to Q

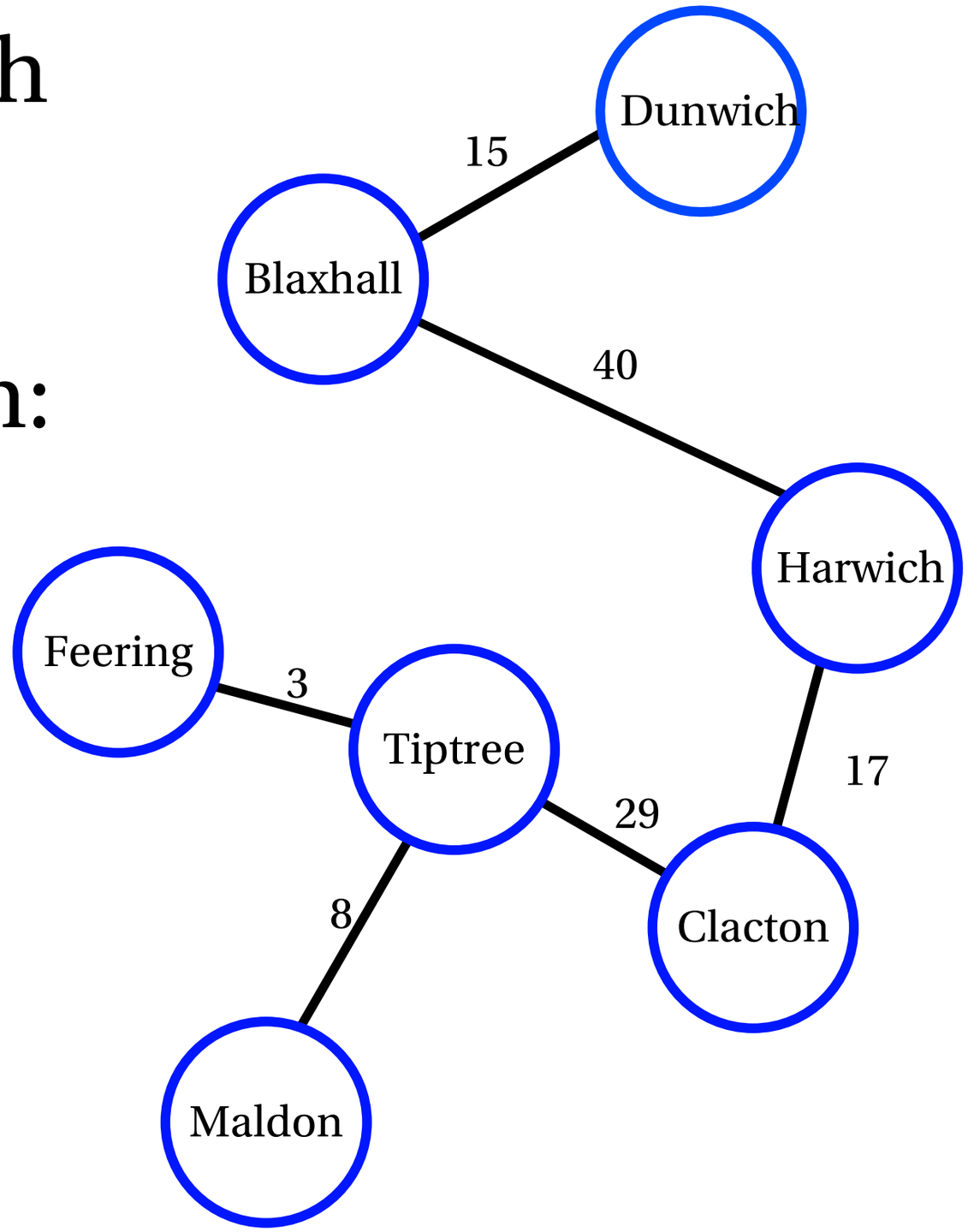
Running time:
O(V log E)
where V = number of nodes
E = number of edges

Minimum spanning trees

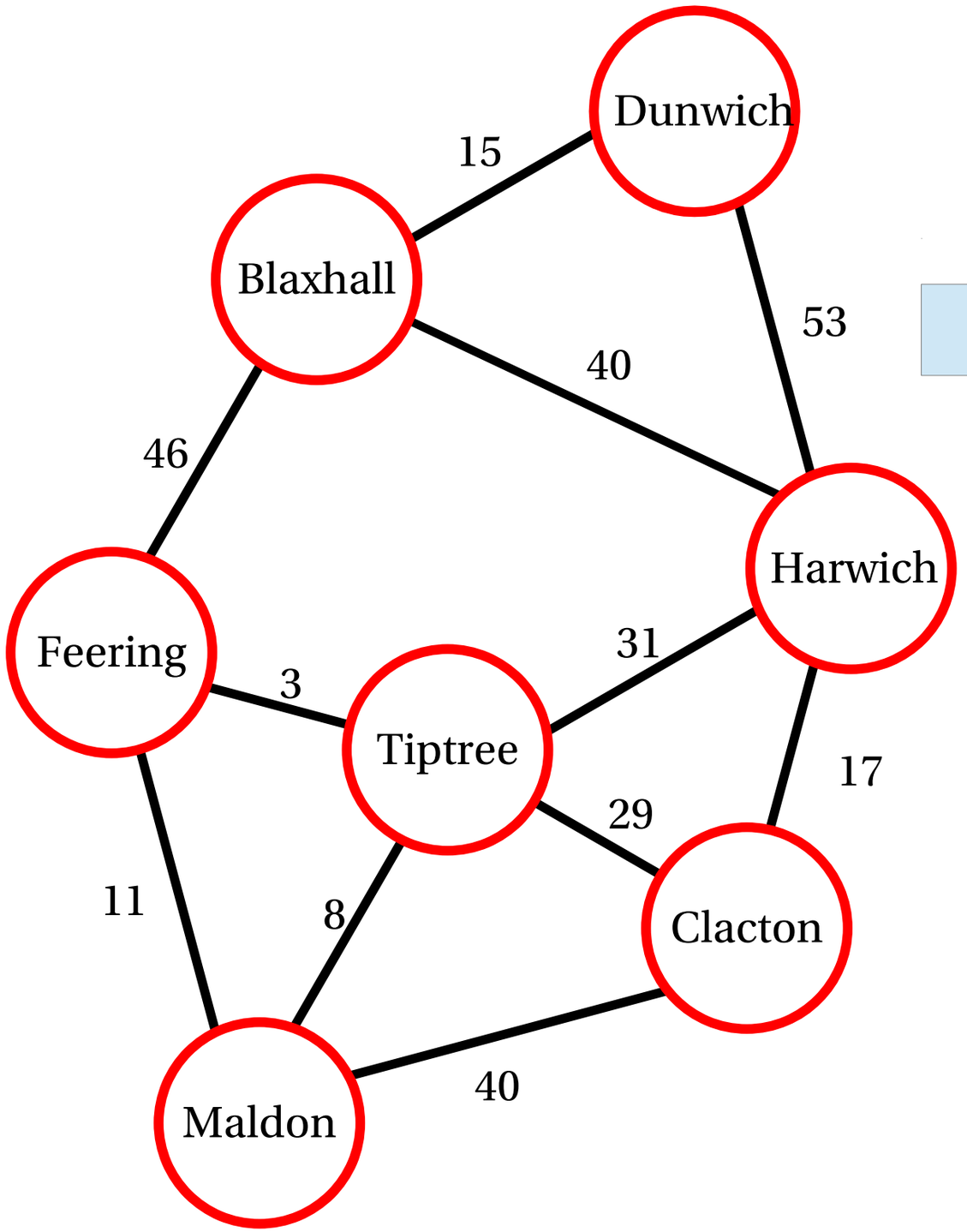
A *spanning tree* of a graph is a subgraph (a graph obtained by deleting some of the edges) which:

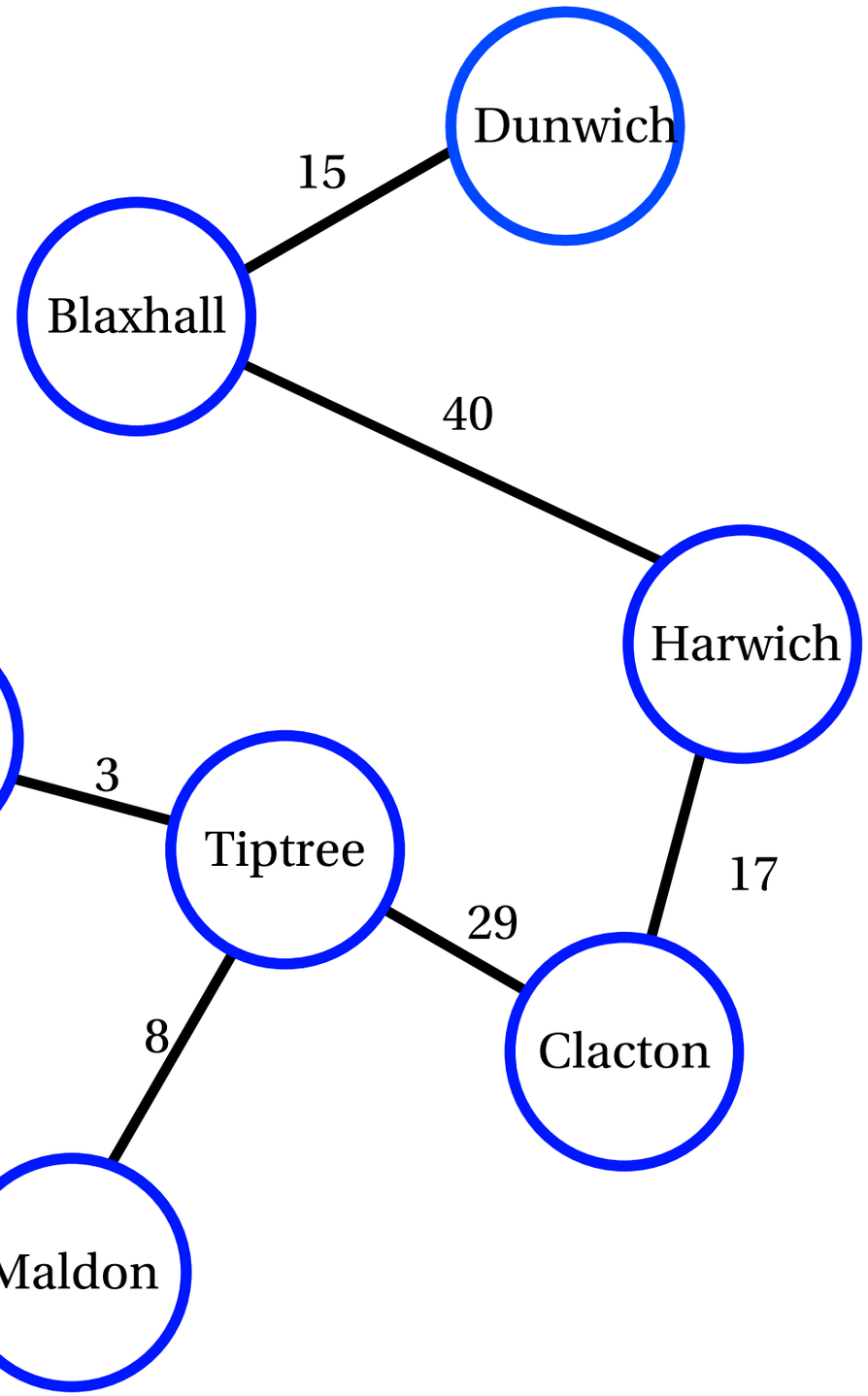
- is acyclic
- is connected

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

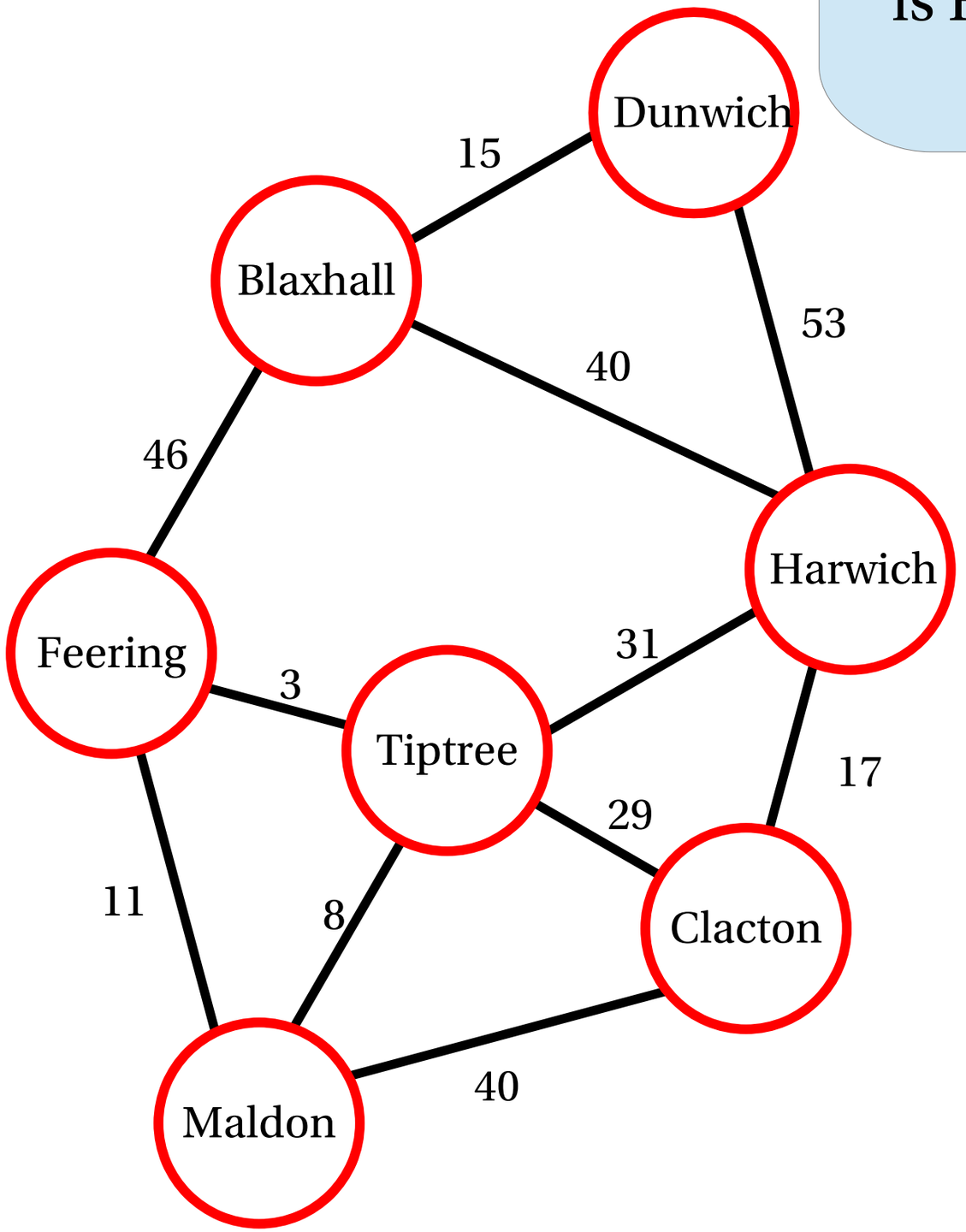
Keep a set S of all the nodes that are in the tree so far, initially containing one arbitrary node

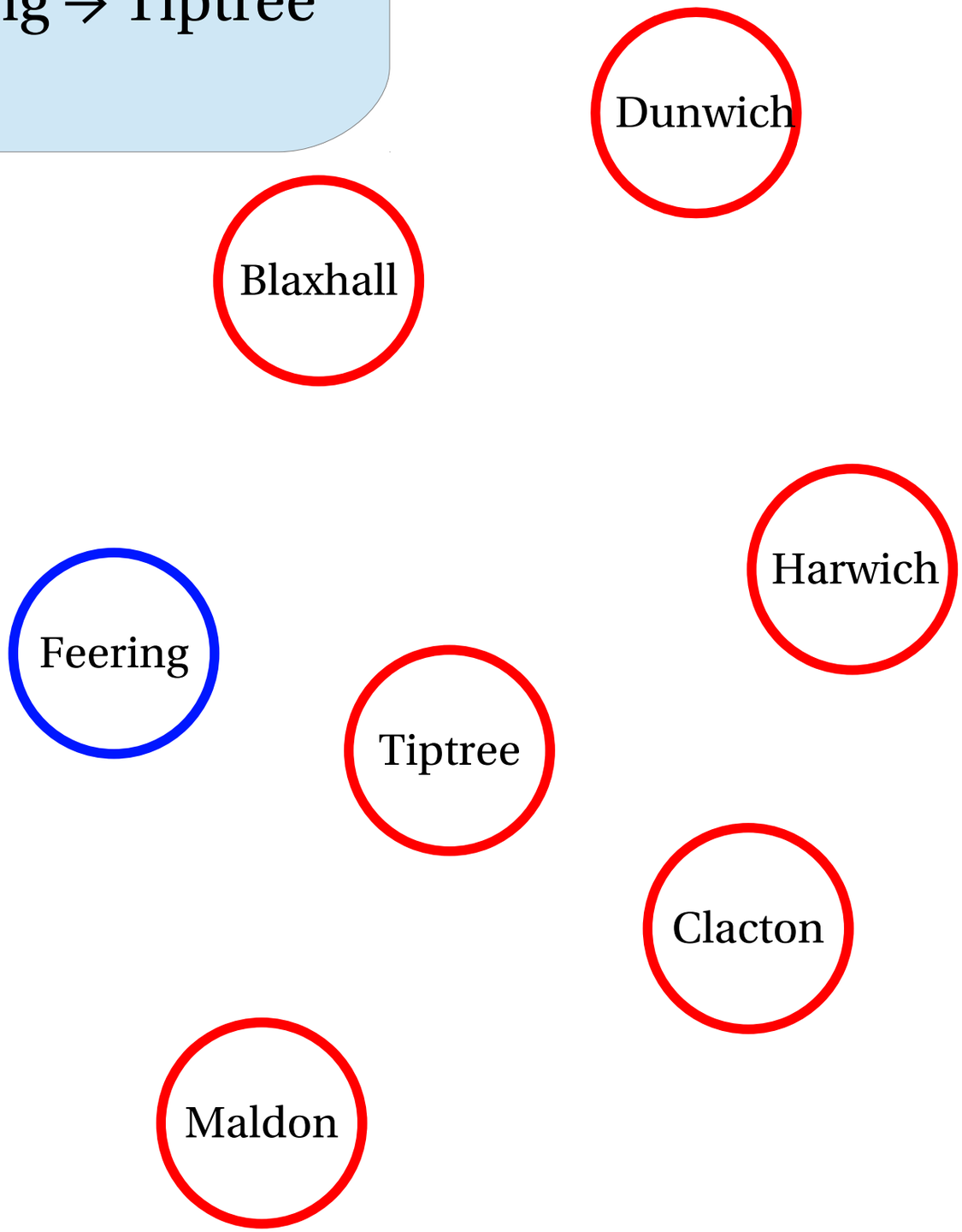
While there is a node not in S:

- 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

Minimum[illegible]ees

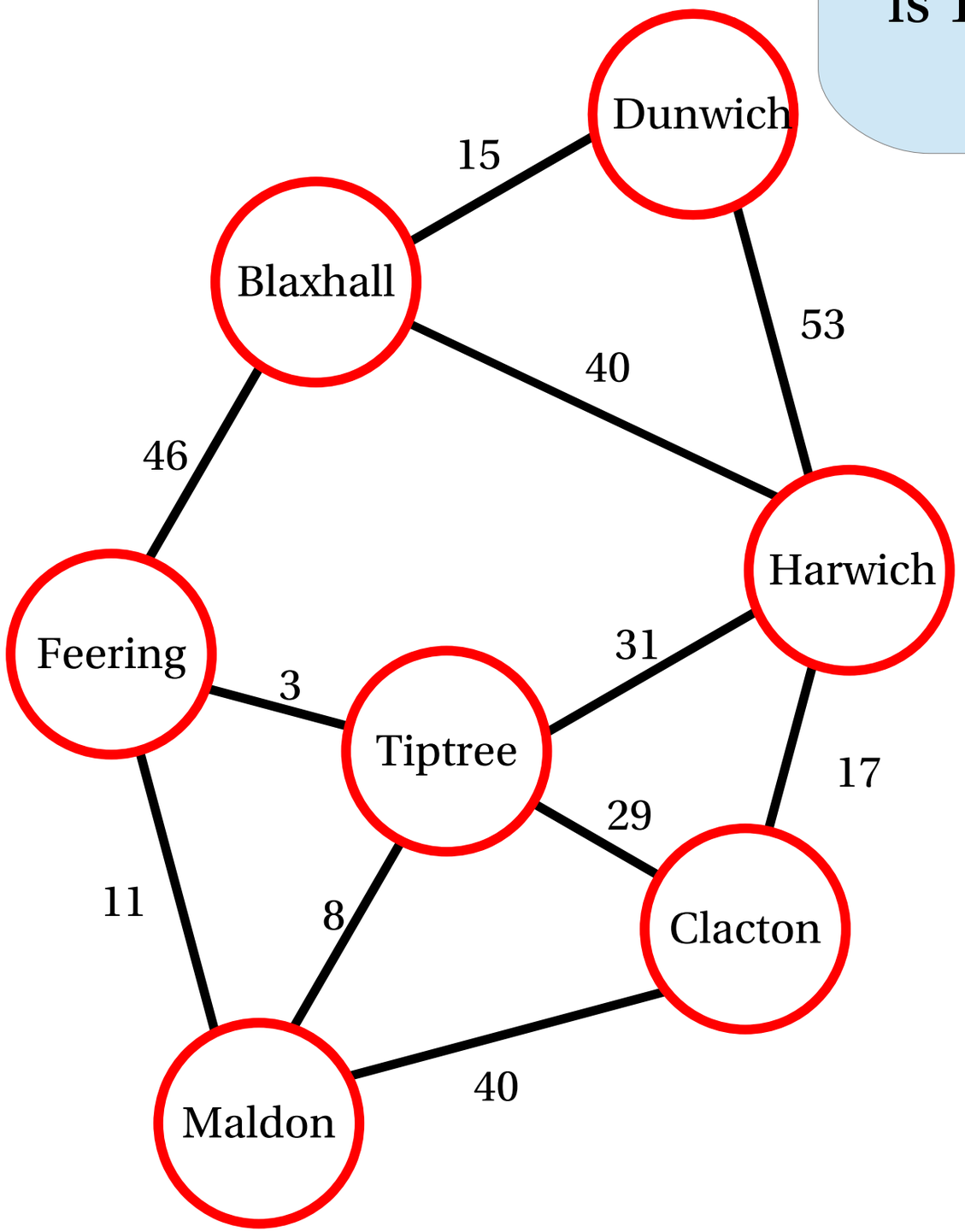
S = {Feering}
Lowest-weight edge
from S to not-S
is Feering → Tiptree

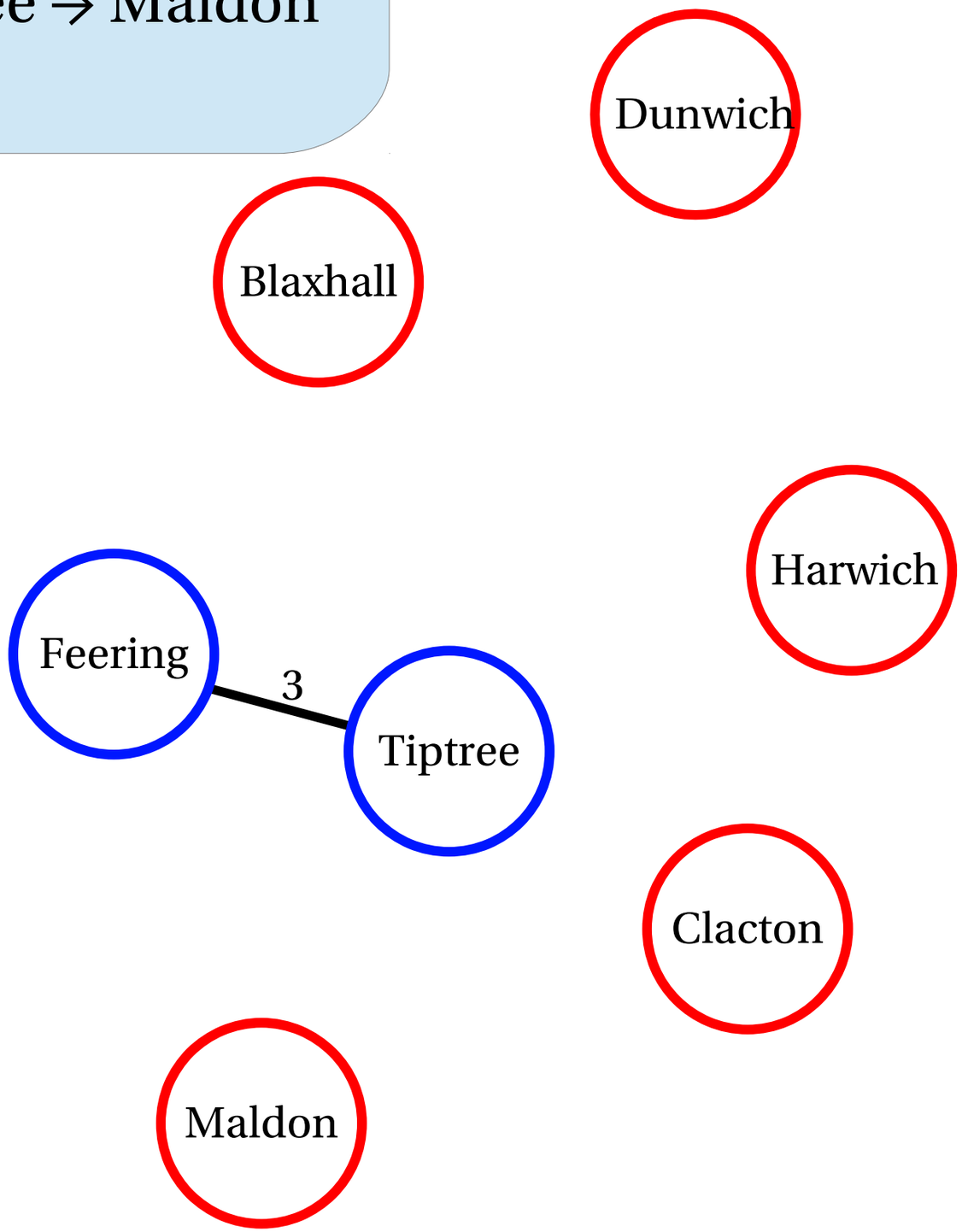




Minimum[illegible]ees

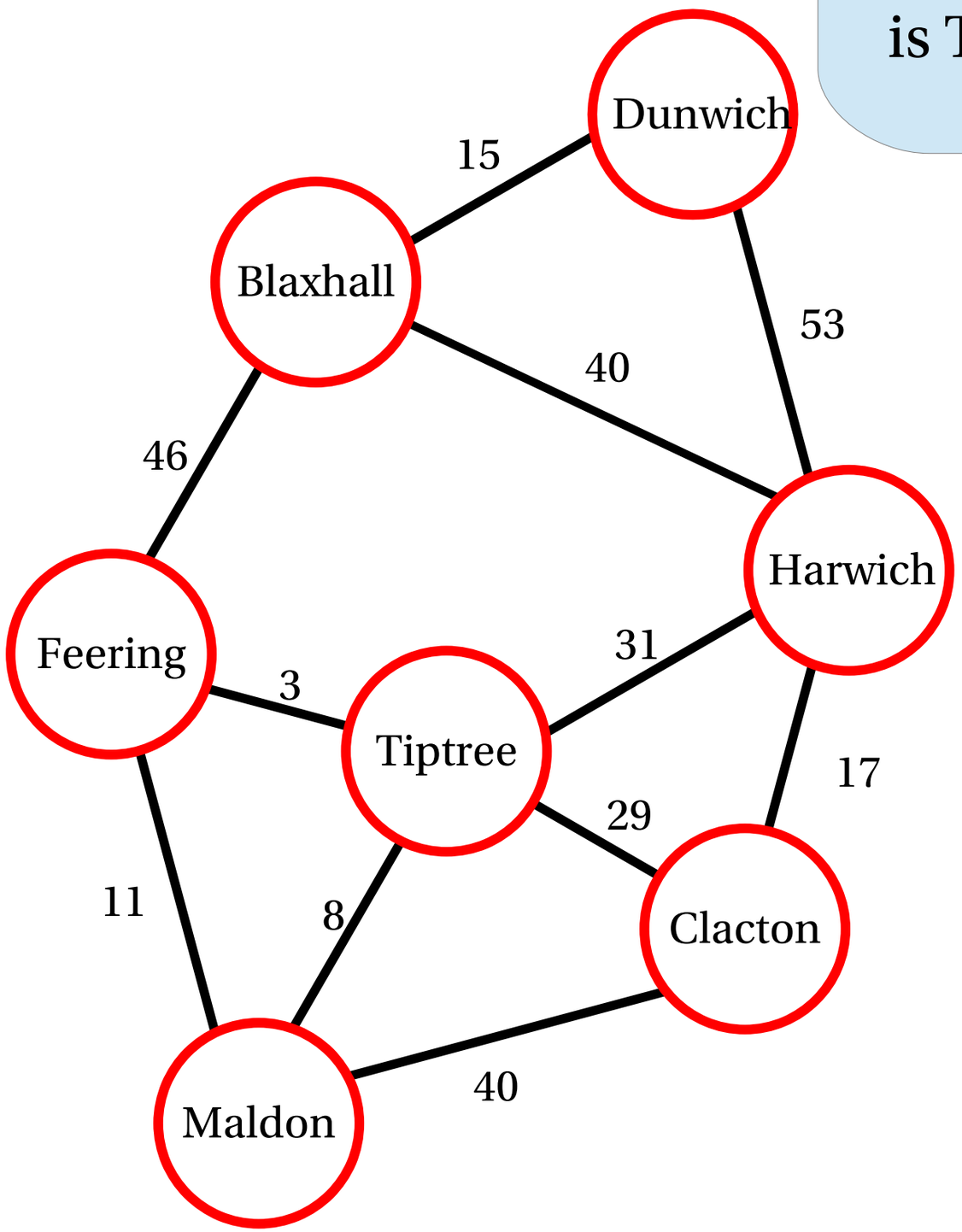
S = {Feering, Tiptree}
Lowest-weight edge
from S to not-S
is Tiptree → Maldon

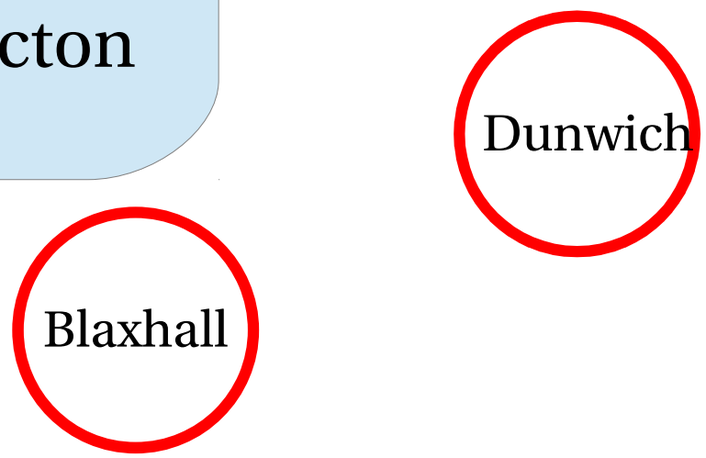


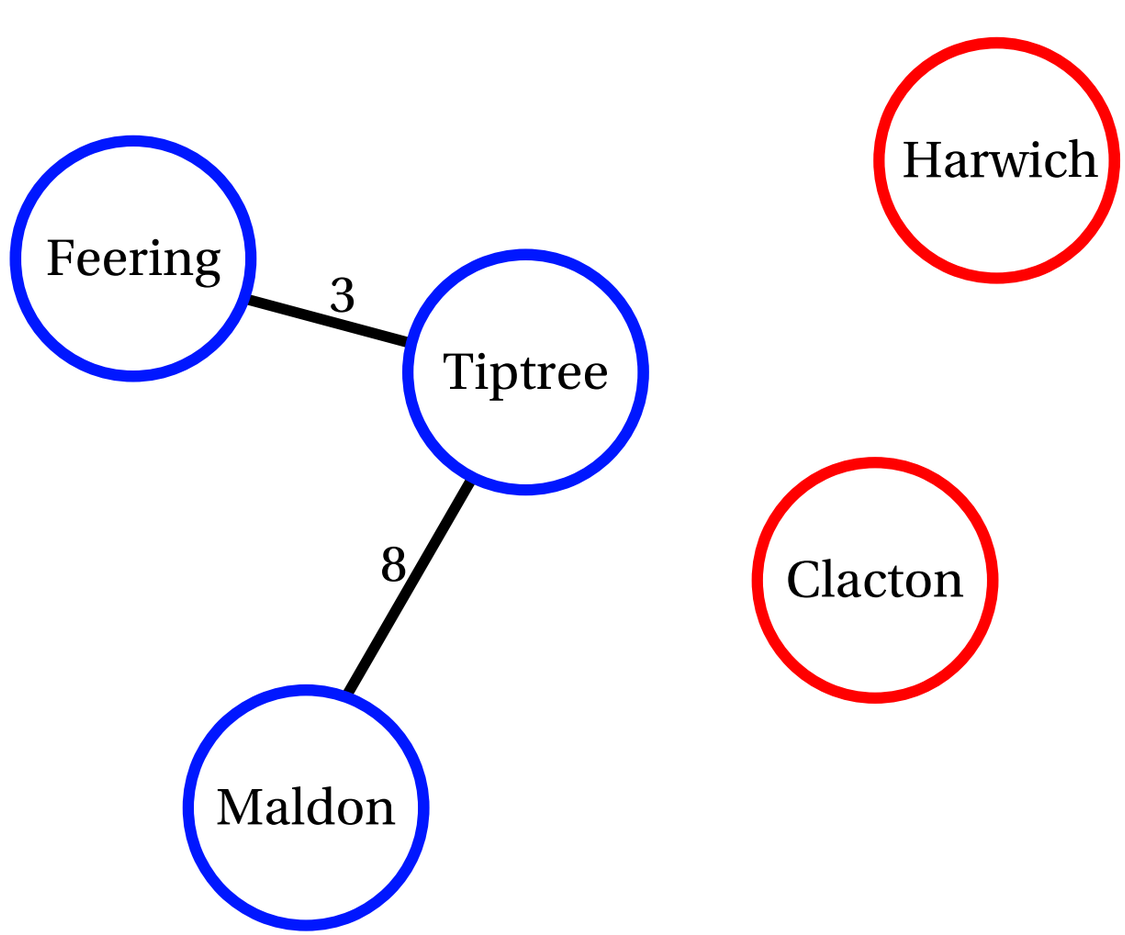


Minimum ees

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

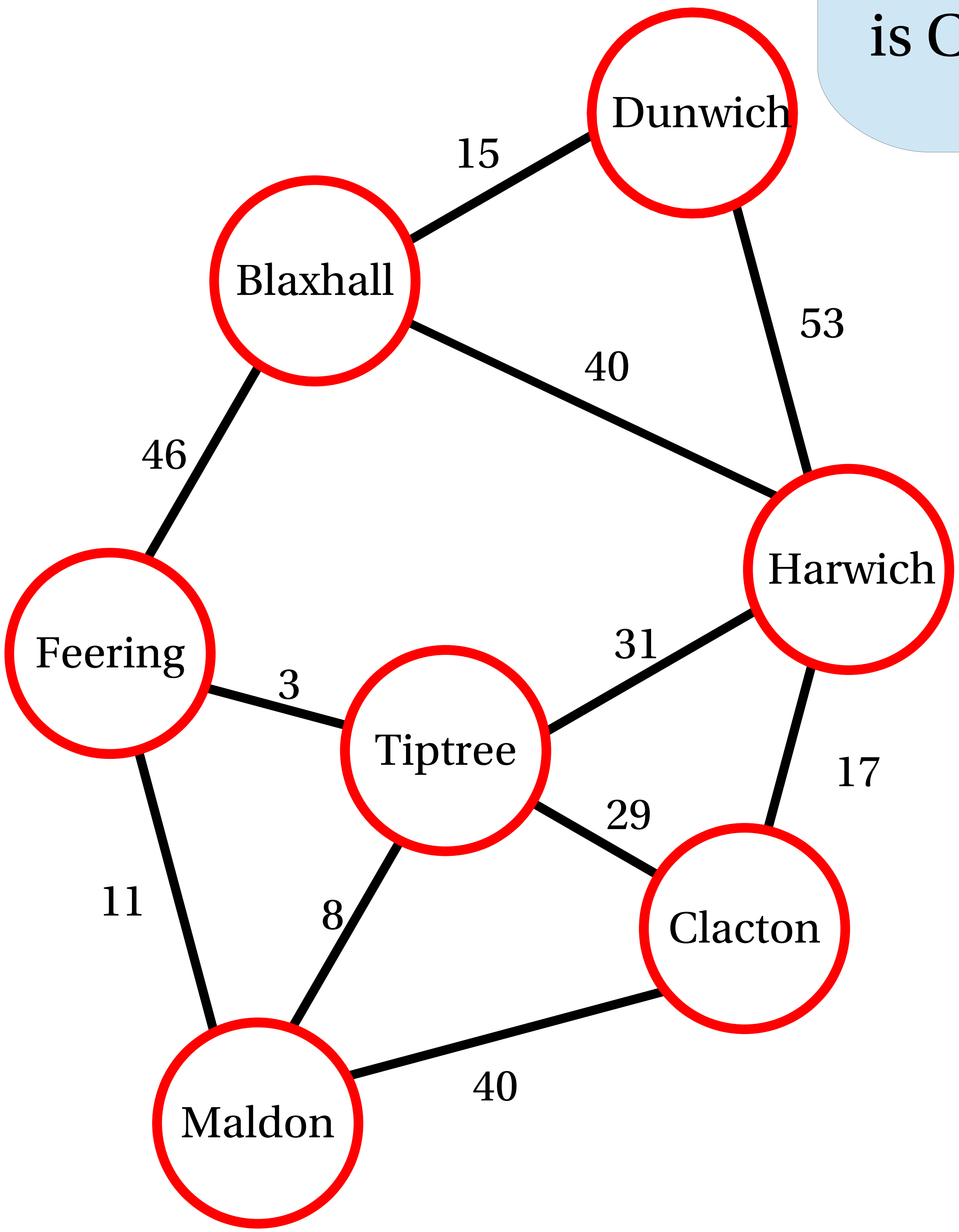


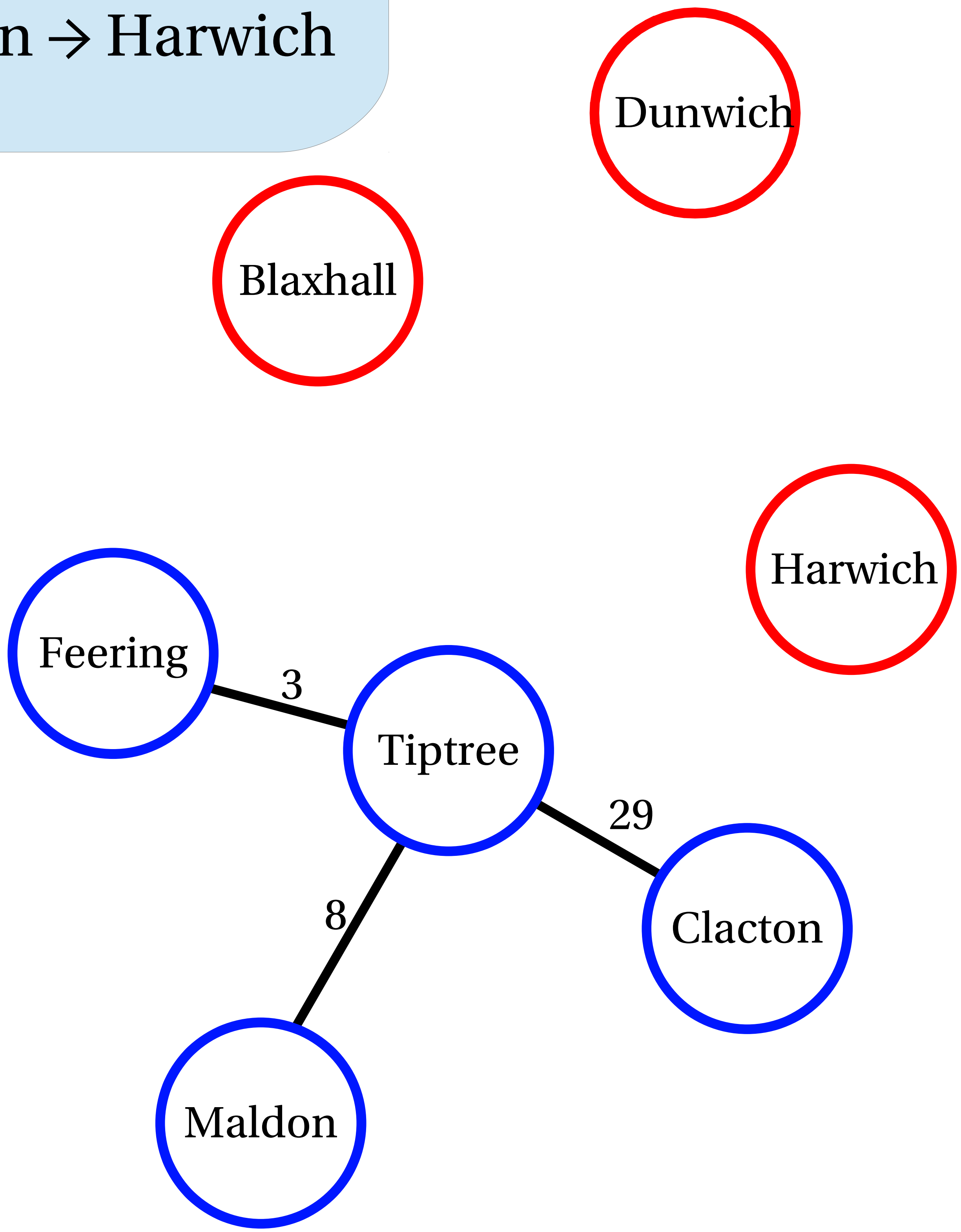




Minimu… ees

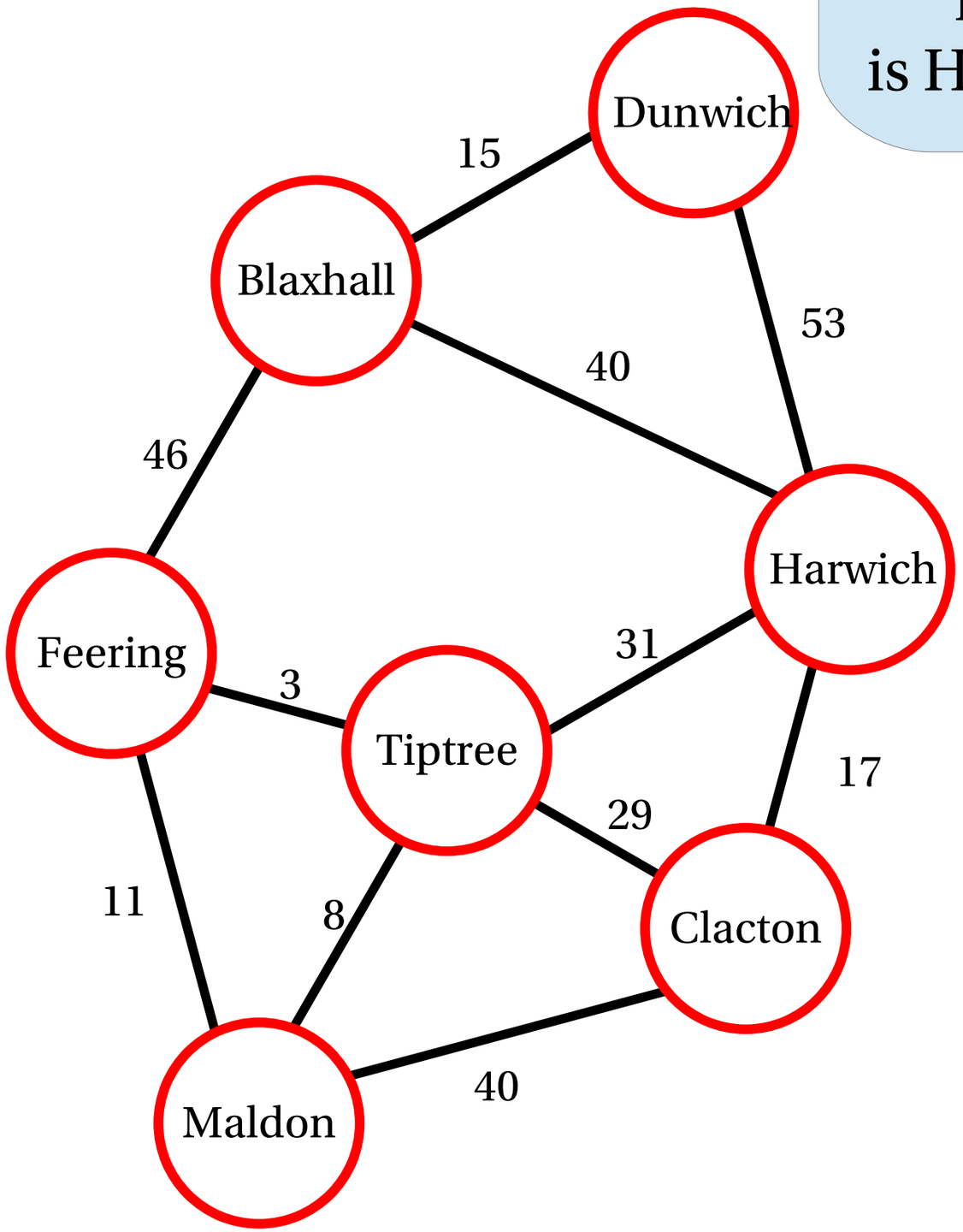
S = {Feering, Tiptree, Maldon, Clacton}
Lowest-weight edge from S to not-S
is Clacton → Harwich

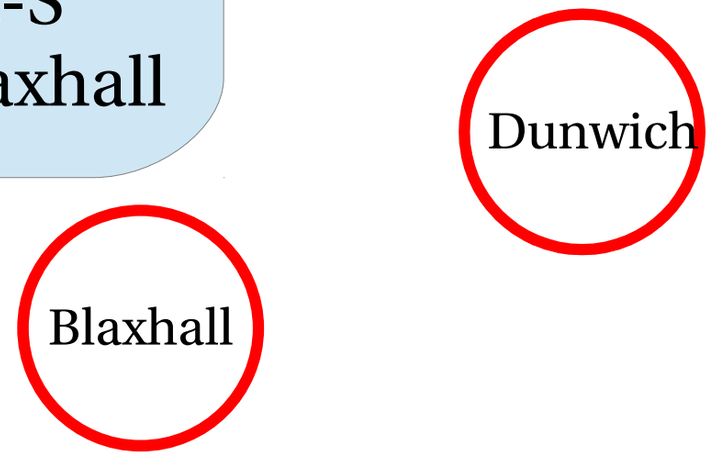


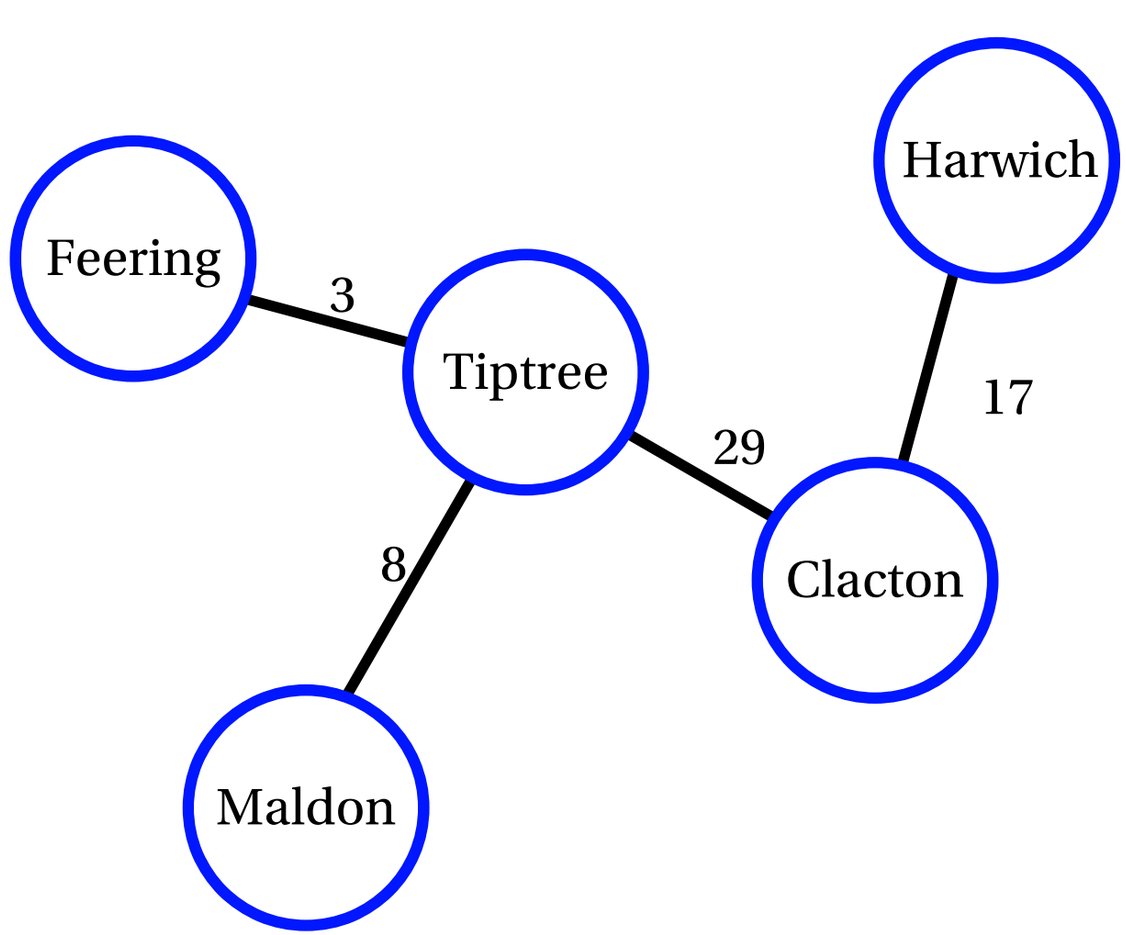


Minimum[illegible]ees

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

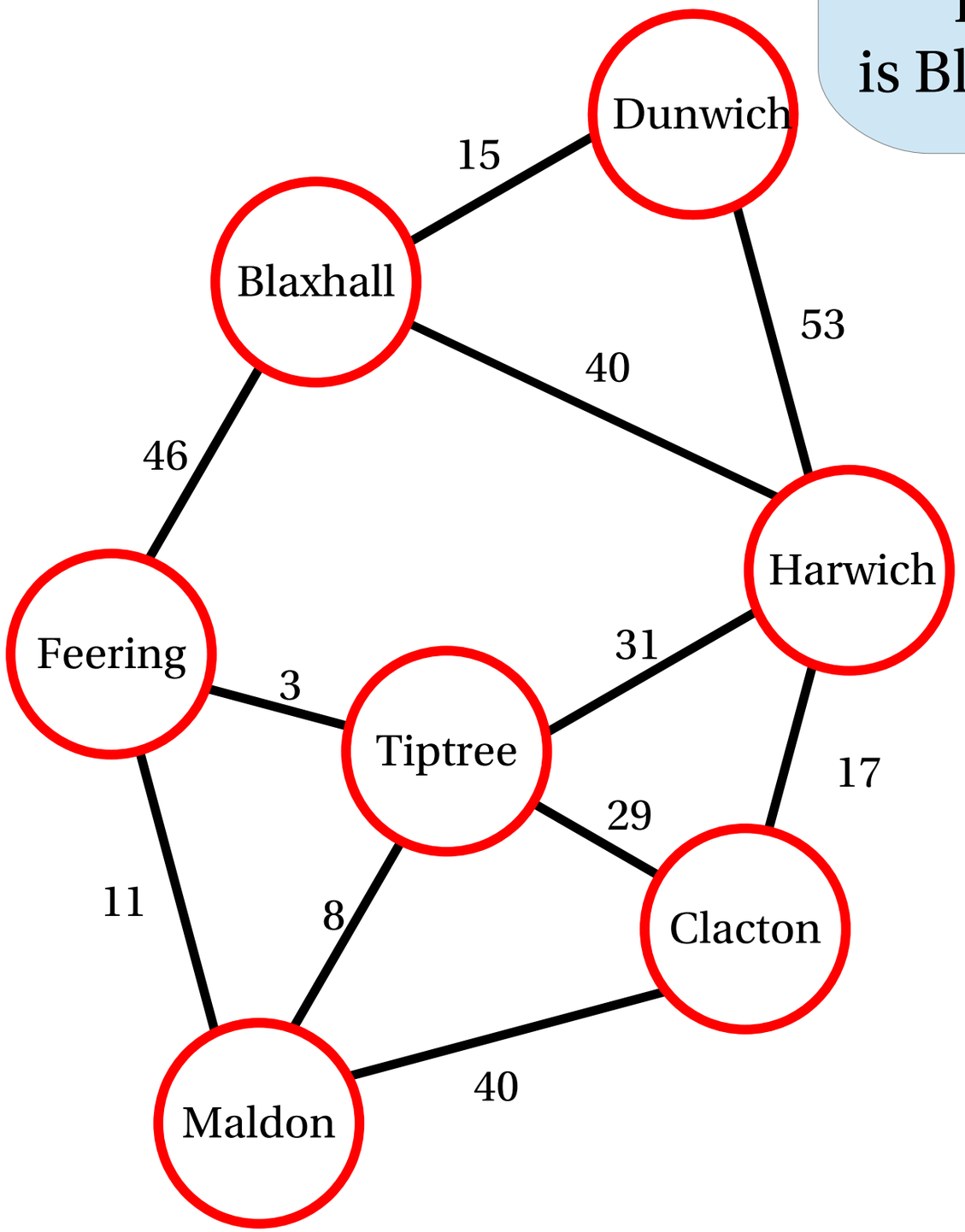


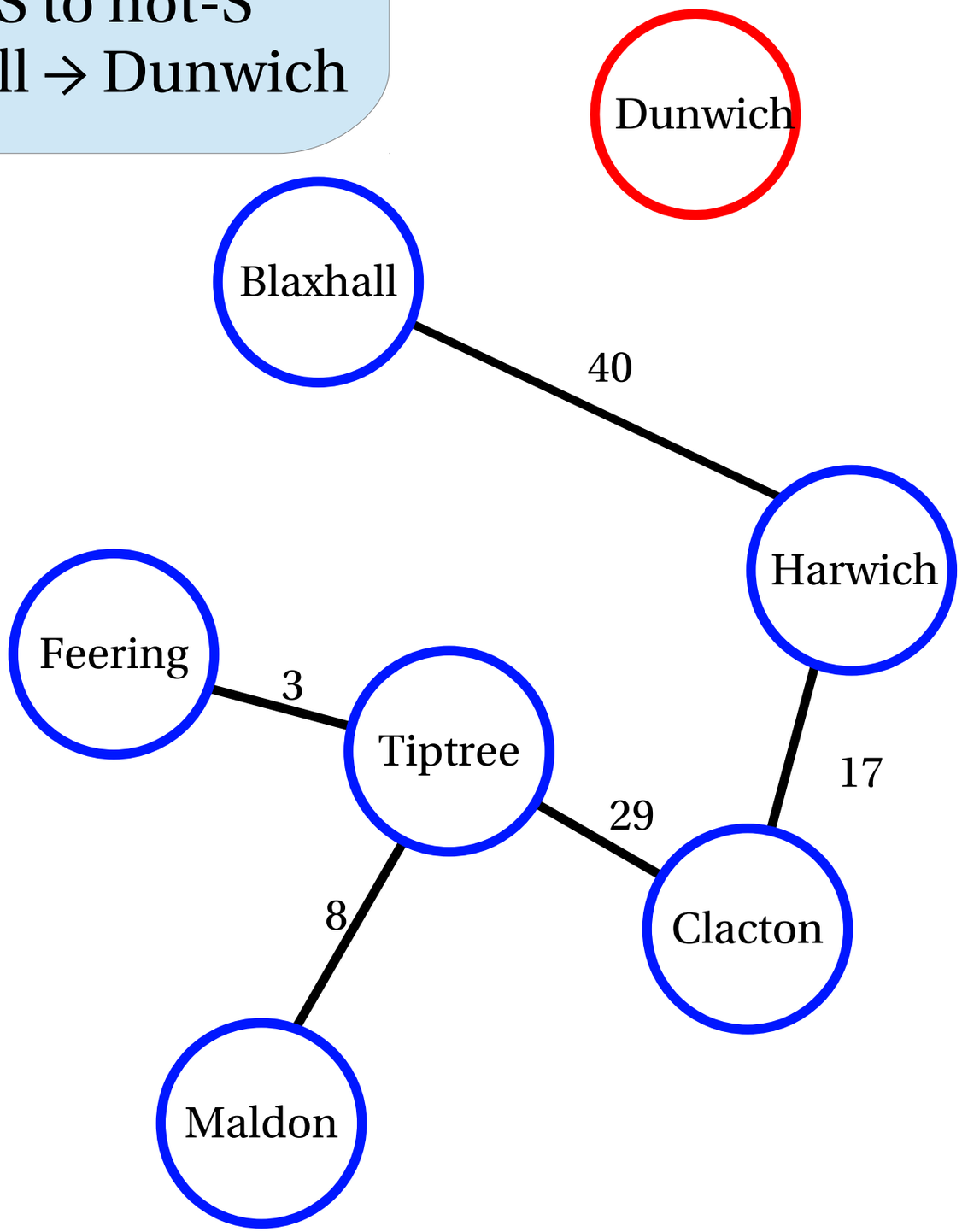




Minimum[illegible]ees

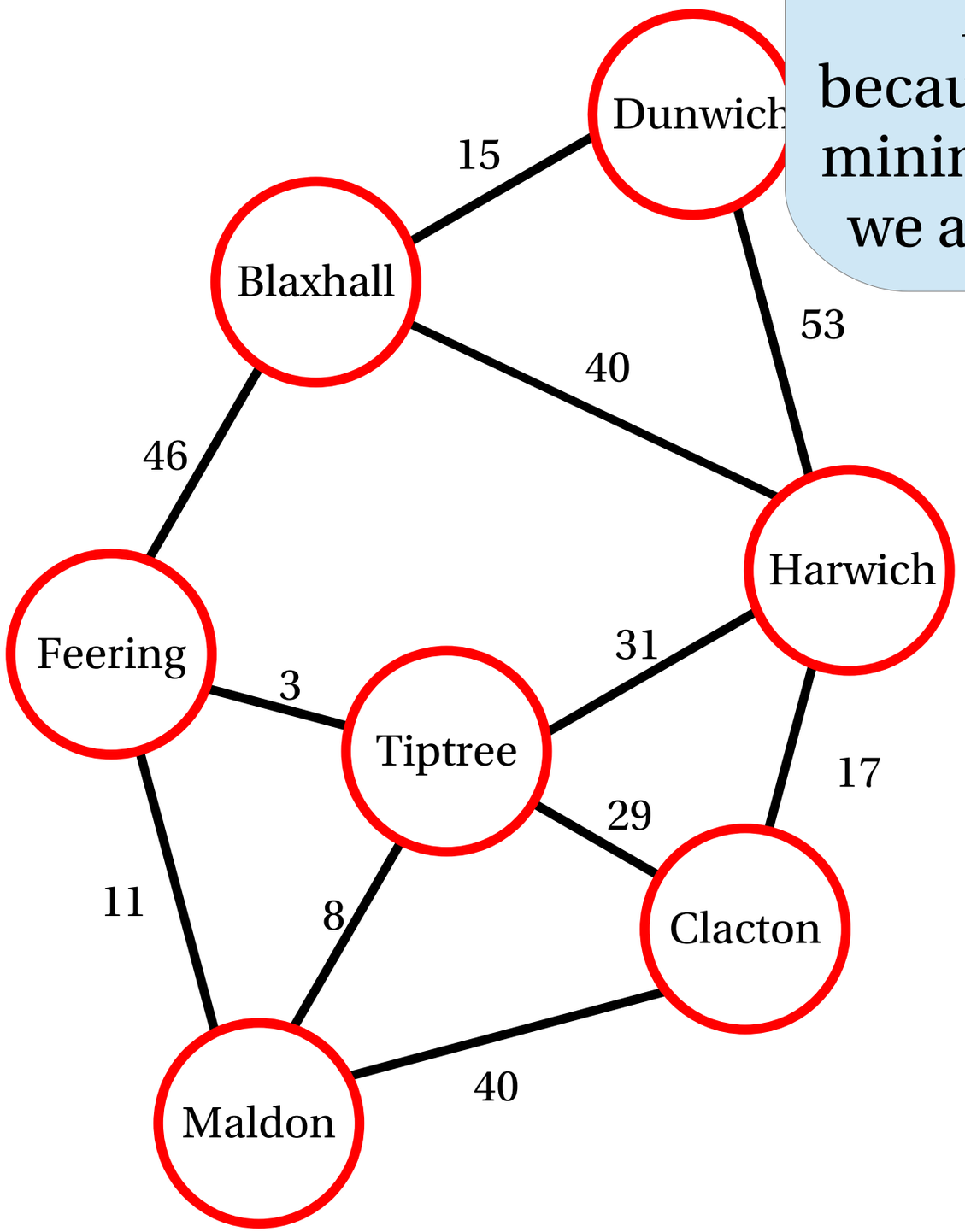
S = {Feering, Tiptree, Maldon, Clacton, Harwich, Blaxhall}
Lowest-weight edge from S to not-S is Blaxhall → Dunwich

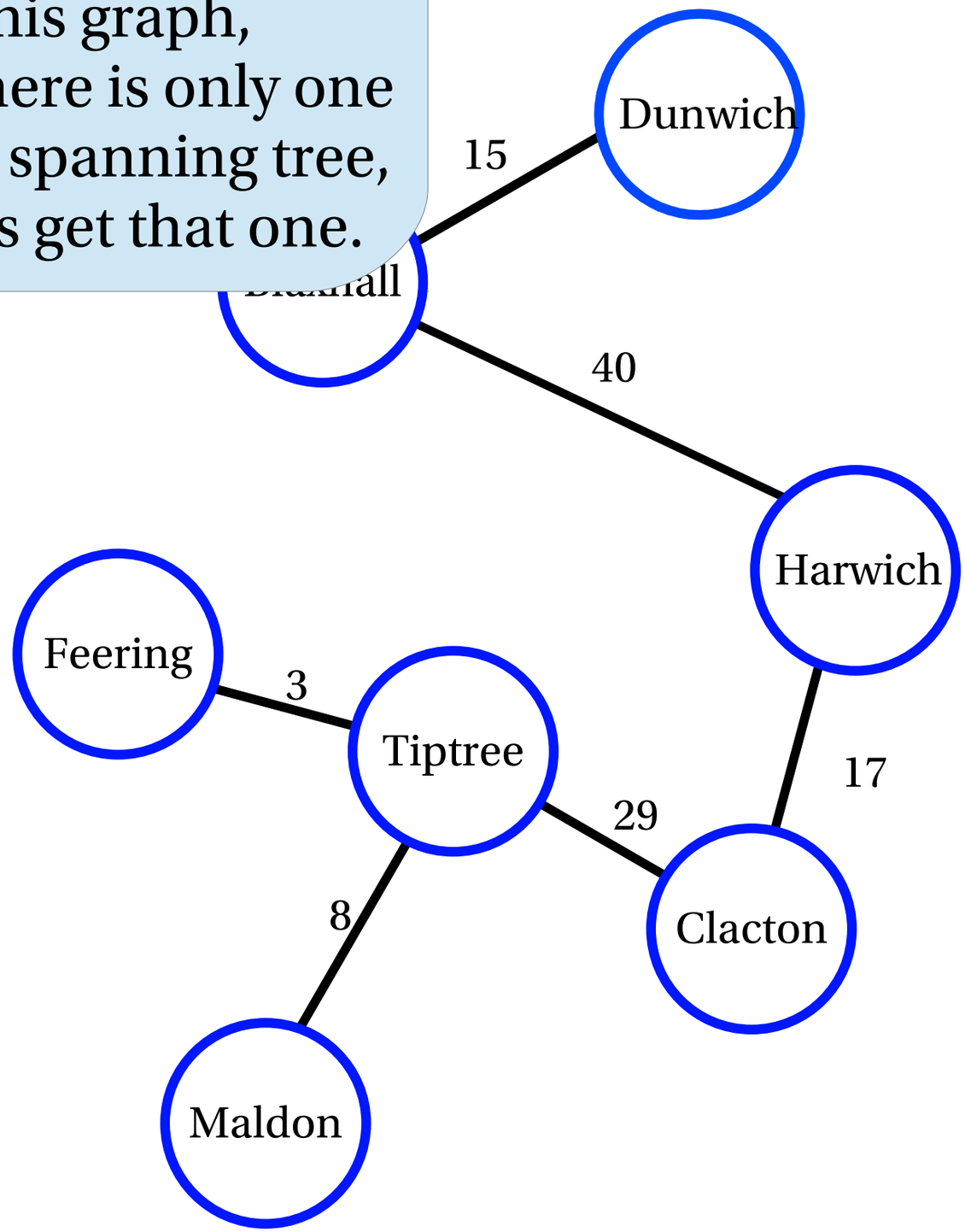




Minimu…ees

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

- Whenever you add a node to S, add all of its edges to 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

- 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