Dijkstra's algorithm: summary

- Finds the shortest path between two specified vertices, or between one source vertex and every other.

Dijkstra's algorithm: summary

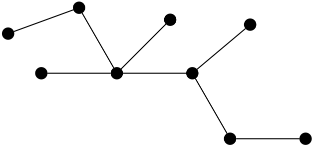
- Finds the shortest path between two specified vertices, or between one source vertex and every other.
- Does so by exploring vertices in order of their distance from the source.

Dijkstra's algorithm: summary

- Finds the shortest path between two specified vertices, or between one source vertex and every other.
- Does so by exploring vertices in order of their distance from the source.
- Running time is $O((n + e)\log n)$ for a graph with n vertices and e edges.

Trees as graphs

- We have seen trees as data structures.
- A tree is also a graph: a connected graph with no cycles:

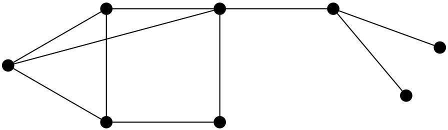


- Every tree with n vertices has $n-1$ edges (prove by induction).

Subgraphs

A **subgraph** of a graph G is a graph that can be made from G by deleting vertices and/or edges.

A **spanning subgraph** is one made by edge deletions only.

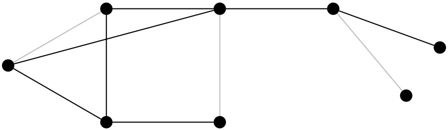


A graph

Subgraphs

A **subgraph** of a graph G is a graph that can be made from G by deleting vertices and/or edges.

A **spanning subgraph** is one made by edge deletions only.

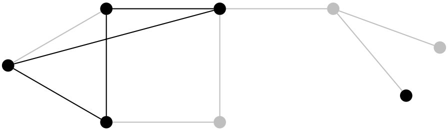


A spanning subgraph

Subgraphs

A **subgraph** of a graph G is a graph that can be made from G by deleting vertices and/or edges.

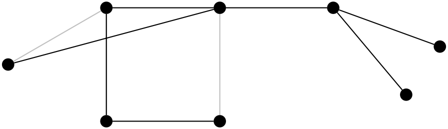
A **spanning subgraph** is one made by edge deletions only.



A non-spanning subgraph

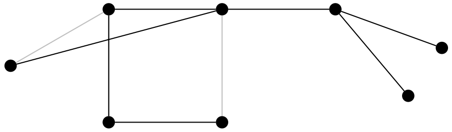
Spanning trees

A **spanning tree** is a spanning subgraph that is a tree.



Spanning trees

A **spanning tree** is a spanning subgraph that is a tree.



Spanning trees connect all the vertices using the least possible number of edges.

Every connected graph has at least one spanning tree.

Minimum spanning trees

The **minimum spanning tree** (MST) of a weighted graph is the spanning tree that has the least total edge weight.

Minimum spanning trees

The **minimum spanning tree** (MST) of a weighted graph is the spanning tree that has the least total edge weight.

E.g., you're building a wind farm and you want to connect your turbines using the least amount of cable.

(Assuming cables must run from turbine to turbine with no junctions in the middle.)

Minimum spanning trees

The **minimum spanning tree** (MST) of a weighted graph is the spanning tree that has the least total edge weight.

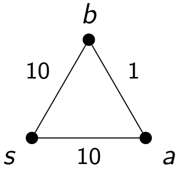
E.g., you're building a wind farm and you want to connect your turbines using the least amount of cable.

(Assuming cables must run from turbine to turbine with no junctions in the middle.)

Strictly, we should say "**An** MST is **a** spanning tree that has least possible weight" – a graph may have more than one MST.

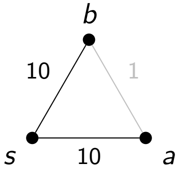
Dijkstra doesn't compute MSTs

Dijkstra's algorithm computes a spanning tree, but not necessarily a minimal one.



Dijkstra doesn't compute MSTs

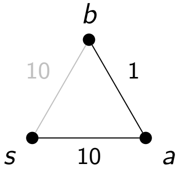
Dijkstra's algorithm computes a spanning tree, but not necessarily a minimal one.



Spanning tree computed by Dijkstra from s

Dijkstra doesn't compute MSTs

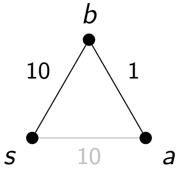
Dijkstra's algorithm computes a spanning tree, but not necessarily a minimal one.



A minimum spanning tree

Dijkstra doesn't compute MSTs

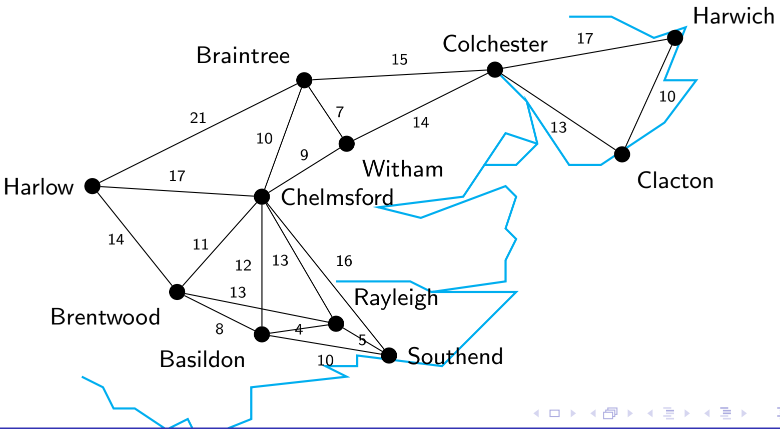
Dijkstra's algorithm computes a spanning tree, but not necessarily a minimal one.



The other minimum spanning tree

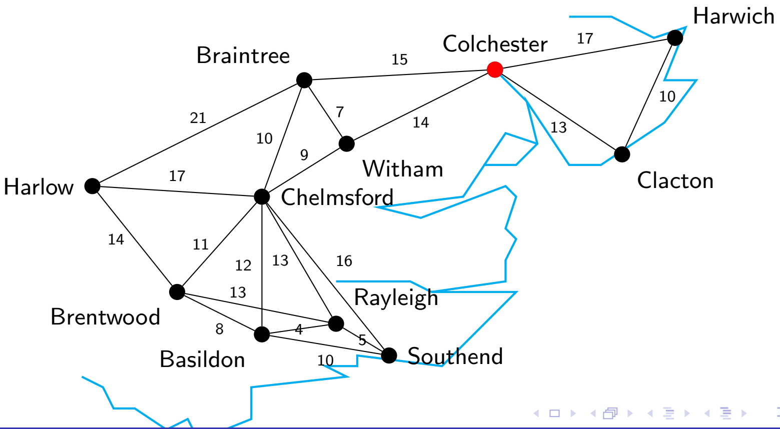
Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



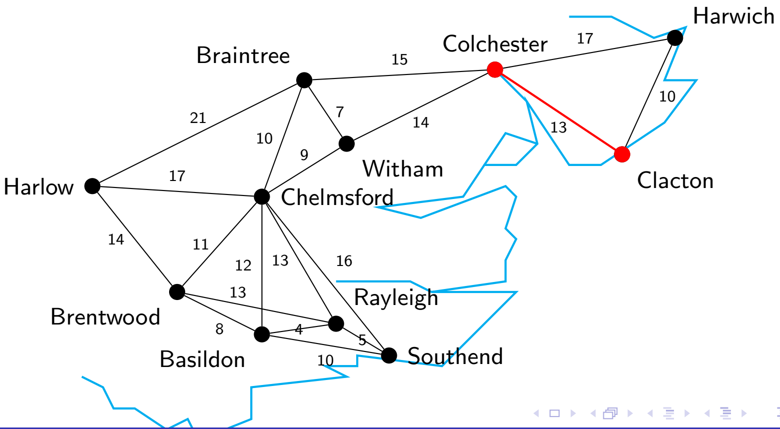
Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



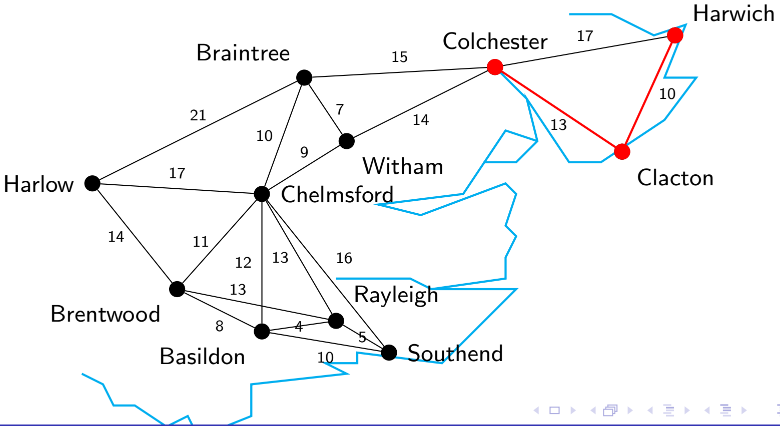
Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



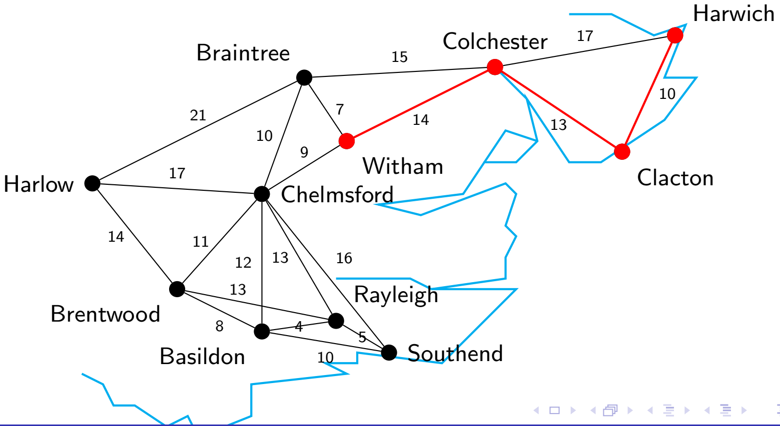
Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



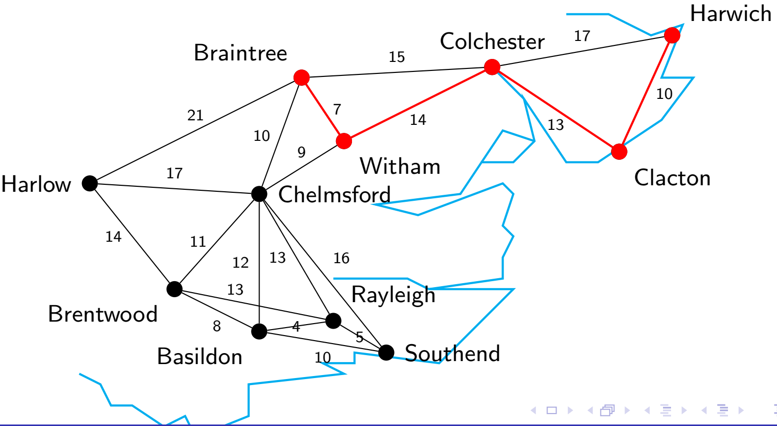
Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



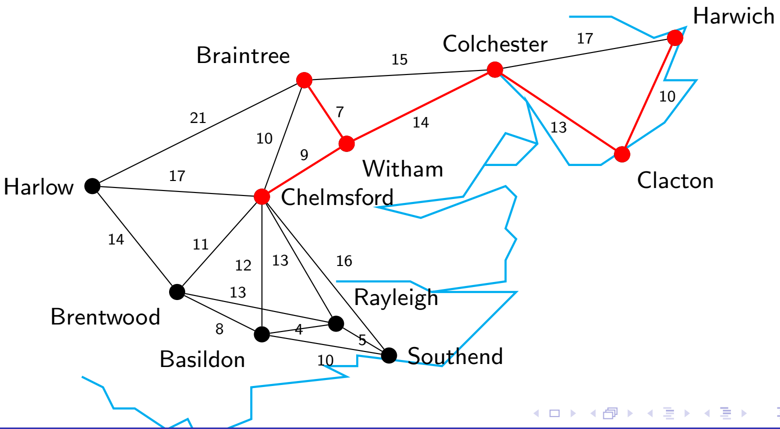
Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



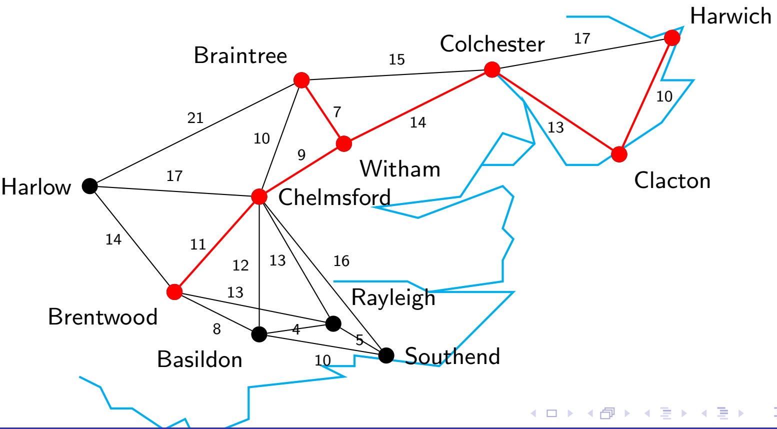
Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



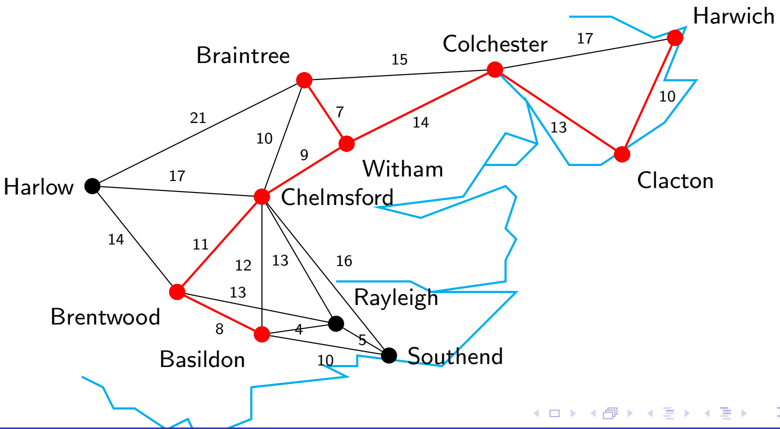
Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



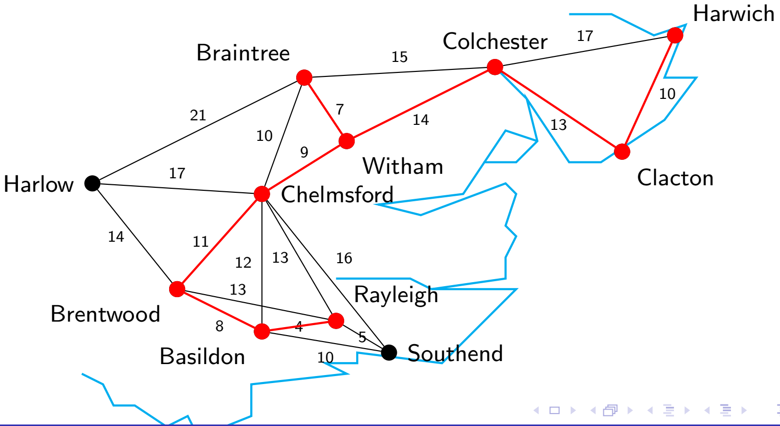
Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



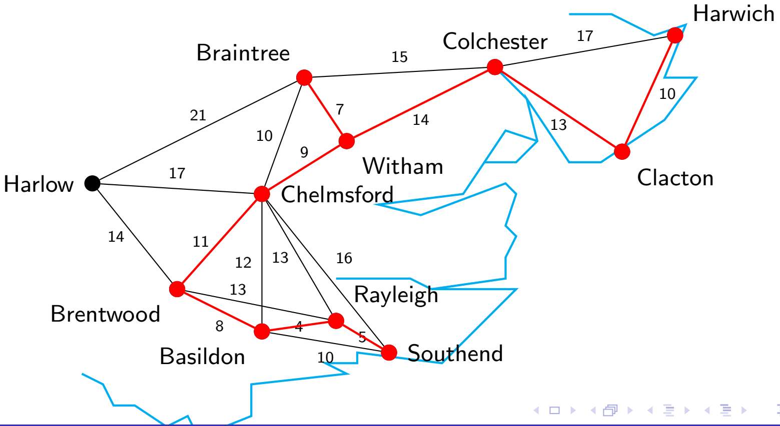
Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



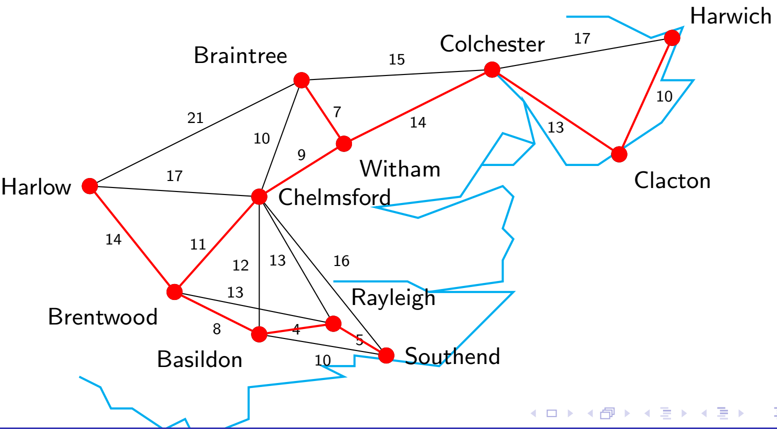
Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



Prim's algorithm

1. Add an arbitrary vertex to the tree.
2. Add to the tree the least-weight edge in G that connects a vertex in the tree to one not yet in it.
3. Repeat 2. until the tree is spanning.



Prim's algorithm: implementation

To implement Prim's algorithm, we use the following data structures:

- `boolean[] inTree` tells us which vertices have been added to the tree so far.

Prim's algorithm: implementation

To implement Prim's algorithm, we use the following data structures:

- `boolean[] inTree` tells us which vertices have been added to the tree so far.
- A priority queue of vertices that are adjacent to the tree built so far, with the weight of the shortest connecting edge as priority.

Prim's algorithm: implementation

To implement Prim's algorithm, we use the following data structures:

- `boolean[] inTree` tells us which vertices have been added to the tree so far.
- A priority queue of vertices that are adjacent to the tree built so far, with the weight of the shortest connecting edge as priority.
- `int[] treeNbr` stores, for each vertex `x` in the queue, its nearest neighbour in the tree built so far.

Prim's algorithm: pseudocode (1)

Prim's algorithm for a graph with n vertices.

```
// Initialization
inTree = {true, false, false, ..., false};
treeNbr = {0, ..., 0};
Q = new empty priority queue;
mst = new graph with n vertices;

for each nedge (0,y) in G
    Q.insert (weight (0,y), y);

...
```

Prim's algorithm: pseudocode (2)

```
// Main loop
while (Q not empty)
    // Add next vertex to tree
    x = Q.next();
    mst.addEdge (x, treeNbr[x], weight (x, treeNbr[x]));
    inTree[x] = true;

    // Process neighbours
    for each edge (x,y) in G
        if (!inTree[y])
            if (!Q.contains(y))
                Q.insert (weight(x, y), y);
                treeNbr[y] = x;
            else if (weight(x,y) < Q.priorityOf(y))
                Q.setPriority (weight(x,y), y);
                treeNbr[y] = x;
return mst;
```

A trick with priority queues

- Graph algorithms (Dijkstra, Prim, etc.) often need to find or update the priority of a specific item in a priority queue.
- A standard priority queue takes time $\Theta(n)$ to do this: you just have to search the array.

A trick with priority queues

- Graph algorithms (Dijkstra, Prim, etc.) often need to find or update the priority of a specific item in a priority queue.
- A standard priority queue takes time $\Theta(n)$ to do this: you just have to search the array.
- The items in the queue are usually vertices: these come from a known set.
- We can find items in time $O(1)$ by maintaining an array that tells us the index of each vertex in the priority queue's internal array.

A trick with priority queues

- Graph algorithms (Dijkstra, Prim, etc.) often need to find or update the priority of a specific item in a priority queue.
- A standard priority queue takes time $\Theta(n)$ to do this: you just have to search the array.
- The items in the queue are usually vertices: these come from a known set.
- We can find items in time $O(1)$ by maintaining an array that tells us the index of each vertex in the priority queue's internal array.
- Allows find priority in time $O(1)$ (just look it up) and update priority in time $O(\log n)$ (because of bubble up/down).

Prim's algorithm: running time

For an input graph with n vertices and e edges, using adjacency lists.

- The main loop runs n times (once per vertex) and calls `next()` each time. Total time for `next()` is $O(n \log n)$.

Prim's algorithm: running time

For an input graph with n vertices and e edges, using adjacency lists.

- The main loop runs n times (once per vertex) and calls `next()` each time. Total time for `next()` is $O(n \log n)$.
- `for each edge...` processes each edge once in total.
- For each edge, it can call `insert()` or `setPriority()`: each call takes time $O(\log n)$.

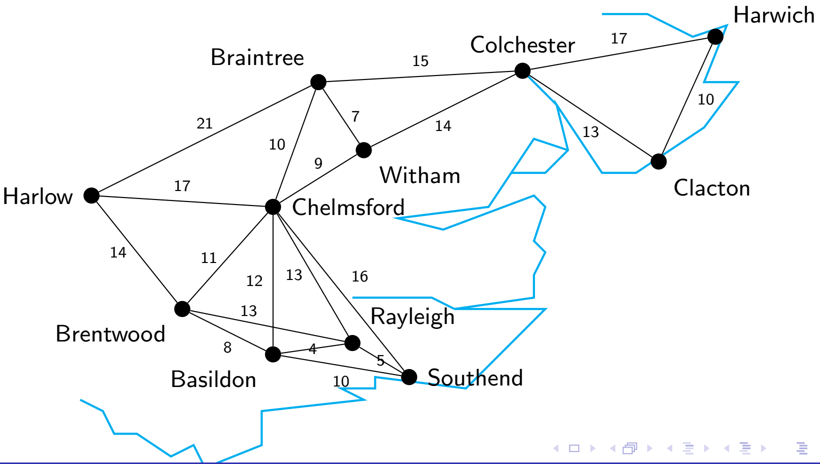
Prim's algorithm: running time

For an input graph with n vertices and e edges, using adjacency lists.

- The main loop runs n times (once per vertex) and calls `next()` each time. Total time for `next()` is $O(n \log n)$.
- `for each edge...` processes each edge once in total.
- For each edge, it can call `insert()` or `setPriority()`: each call takes time $O(\log n)$.
- Total running time is $O((n + e) \log n)$.
- This is $O(e \log n)$ if the input is connected (a connected graph must have at least $n - 1$ edges).

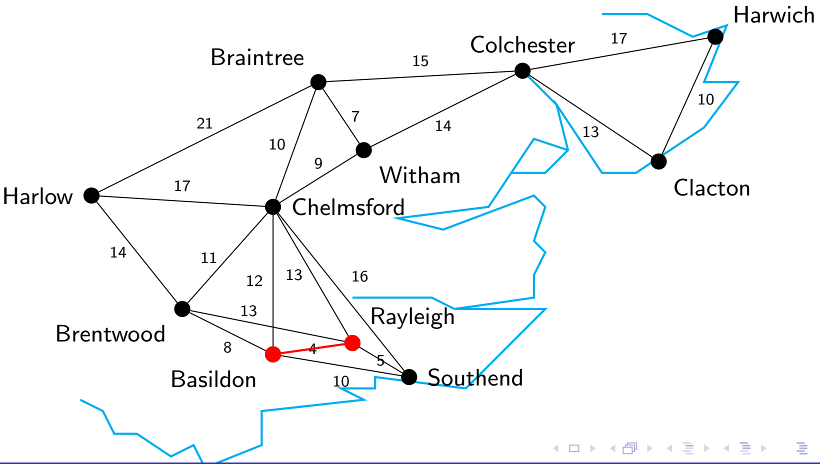
Kruskal's algorithm

1. Start with an empty subgraph.
2. Add to it the least-weight edge that doesn't create a cycle.
3. Repeat 2. until we have a spanning tree.



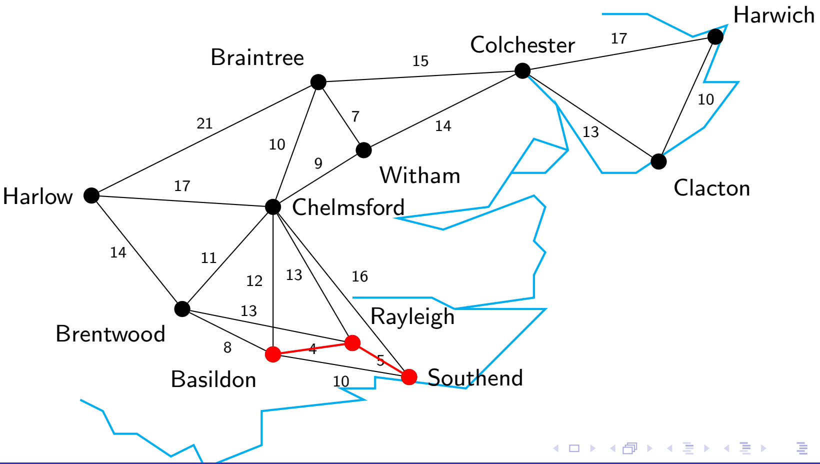
Kruskal's algorithm

1. Start with an empty subgraph.
2. Add to it the least-weight edge that doesn't create a cycle.
3. Repeat 2. until we have a spanning tree.



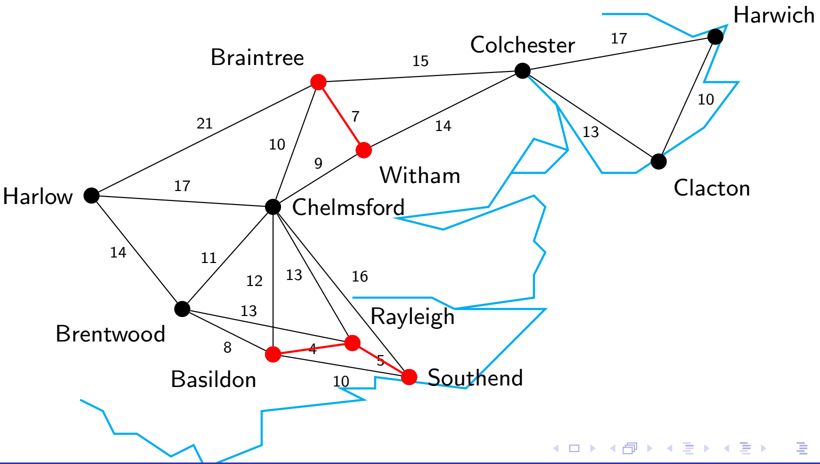
Kruskal's algorithm

1. Start with an empty subgraph.
2. Add to it the least-weight edge that doesn't create a cycle.
3. Repeat 2. until we have a spanning tree.



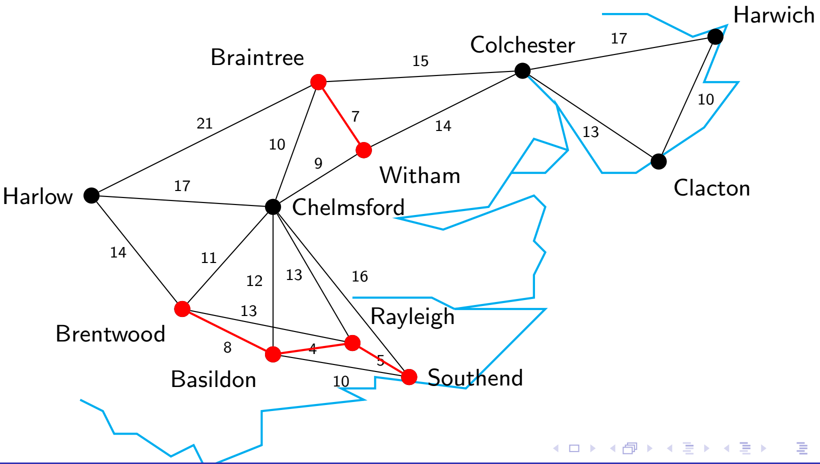
Kruskal's algorithm

1. Start with an empty subgraph.
2. Add to it the least-weight edge that doesn't create a cycle.
3. Repeat 2. until we have a spanning tree.



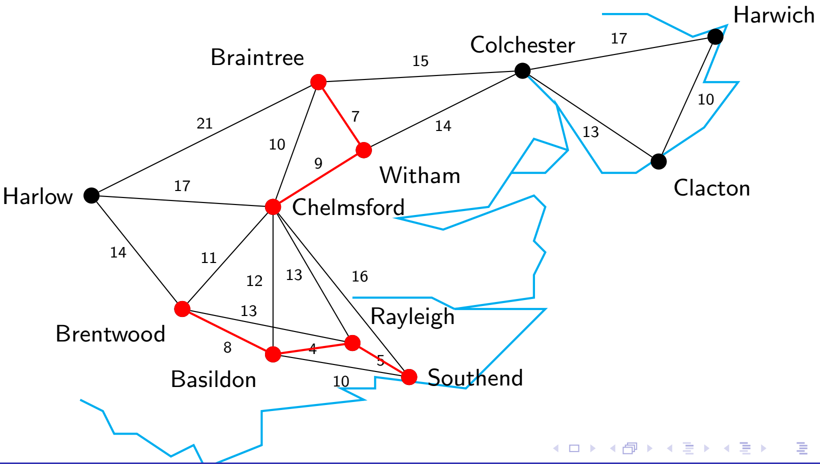
Kruskal's algorithm

1. Start with an empty subgraph.
2. Add to it the least-weight edge that doesn't create a cycle.
3. Repeat 2. until we have a spanning tree.



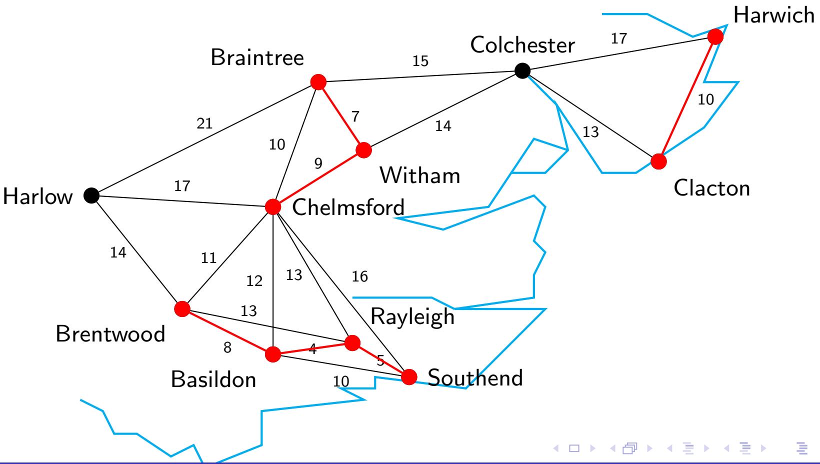
Kruskal's algorithm

1. Start with an empty subgraph.
2. Add to it the least-weight edge that doesn't create a cycle.
3. Repeat 2. until we have a spanning tree.



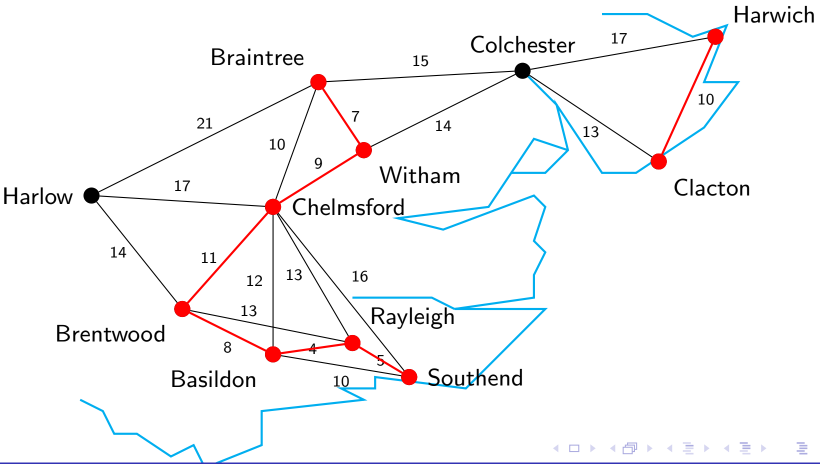
Kruskal's algorithm

1. Start with an empty subgraph.
2. Add to it the least-weight edge that doesn't create a cycle.
3. Repeat 2. until we have a spanning tree.



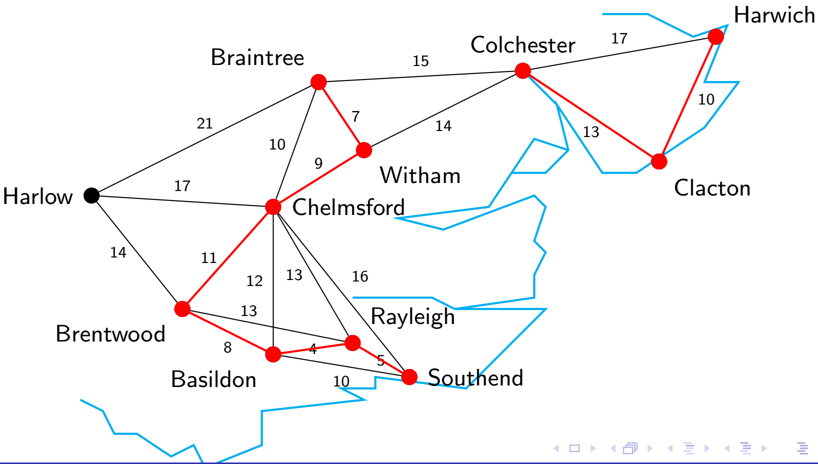
Kruskal's algorithm

1. Start with an empty subgraph.
2. Add to it the least-weight edge that doesn't create a cycle.
3. Repeat 2. until we have a spanning tree.



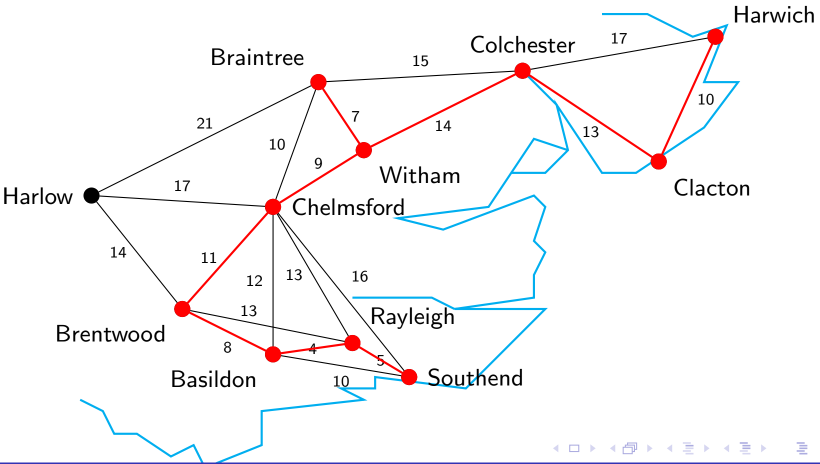
Kruskal's algorithm

1. Start with an empty subgraph.
2. Add to it the least-weight edge that doesn't create a cycle.
3. Repeat 2. until we have a spanning tree.



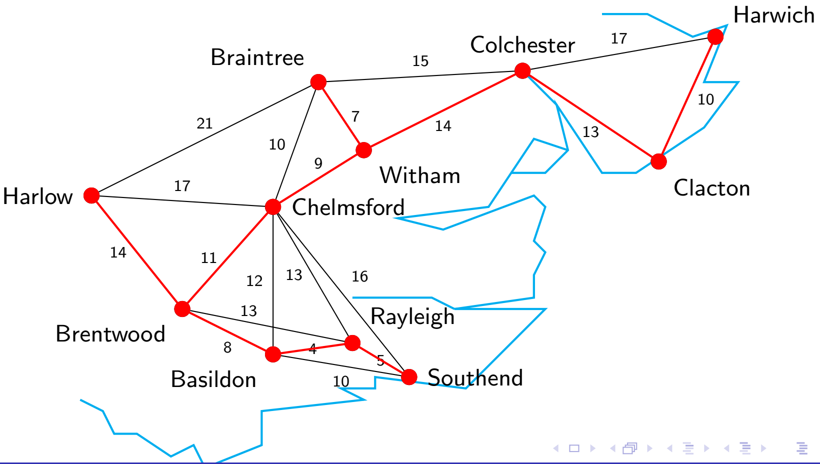
Kruskal's algorithm

1. Start with an empty subgraph.
2. Add to it the least-weight edge that doesn't create a cycle.
3. Repeat 2. until we have a spanning tree.



Kruskal's algorithm

1. Start with an empty subgraph.
2. Add to it the least-weight edge that doesn't create a cycle.
3. Repeat 2. until we have a spanning tree.



Kruskal's algorithm: implementation

We need a data structure that lets us check if two vertices are in the same component.

Kruskal's algorithm: implementation

We need a data structure that lets us check if two vertices are in the same component.

Simplest way is:

- for each component, keep a list of its vertices;
- use an array to map each vertex to a reference to its list.

Kruskal's algorithm: implementation

We need a data structure that lets us check if two vertices are in the same component.

Simplest way is:

- for each component, keep a list of its vertices;
- use an array to map each vertex to a reference to its list.

To merge components, add the smaller list to the larger and update the references in the array.

Kruskal's algorithm: implementation

We need a data structure that lets us check if two vertices are in the same component.

Simplest way is:

- for each component, keep a list of its vertices;
- use an array to map each vertex to a reference to its list.

To merge components, add the smaller list to the larger and update the references in the array.

Each time we merge, the shorter list doubles (or more) in length, so a vertex can't be moved more than $\log n$ times.

Kruskal's algorithm: pseudocode

For an input graph G with n edges.

```
mst = new graph with n vertices;
L = list of G's edges sorted by increasing weight;

for each edge (x, y, w) in L, in order
    if (componentOf(x) != componentOf(y))
        mst.addEdge (x, y, w);
        mergeComponents (x, y);

return mst;
```

Kruskal's algorithm: running time

For an input graph with n vertices and e edges, using adjacency lists.

- Sorting the edges takes time $O(e \log e) = O(e \log(n^2)) = O(e \log n)$.

Kruskal's algorithm: running time

For an input graph with n vertices and e edges, using adjacency lists.

- Sorting the edges takes time $O(e \log e) = O(e \log(n^2)) = O(e \log n)$.
- In total, all the `mergeComponents()` operations could move each vertex $\log n$ times.

Kruskal's algorithm: running time

For an input graph with n vertices and e edges, using adjacency lists.

- Sorting the edges takes time $O(e \log e) = O(e \log(n^2)) = O(e \log n)$.
- In total, all the `mergeComponents()` operations could move each vertex $\log n$ times.
- Total time is $O((n + e) \log n) = O(e \log n)$.

MSTs: which algorithm to use?

Our implementations of Prim and Kruskal each take time $O(e \log n)$, so which should we use?

MSTs: which algorithm to use?

Our implementations of Prim and Kruskal each take time $O(e \log n)$, so which should we use?

- By using more sophisticated heaps, Prim can be improved to $O(e + n \log n)$.
- This is faster than $O(e \log n)$ if $e > O(n)$.

MSTs: which algorithm to use?

Our implementations of Prim and Kruskal each take time $O(e \log n)$, so which should we use?

- By using more sophisticated heaps, Prim can be improved to $O(e + n \log n)$.
- This is faster than $O(e \log n)$ if $e > O(n)$.
- If there are only k possible different edge weights, Kruskal can use counting sort in time $O(k + e)$.
- If $k = O(e)$, this is time $O(e)$ and a sophisticated version of `mergeComponents()` allows a total running time of almost $O(e)$.

MSTs: which algorithm to use?

Our implementations of Prim and Kruskal each take time $O(e \log n)$, so which should we use?

- By using more sophisticated heaps, Prim can be improved to $O(e + n \log n)$.
- This is faster than $O(e \log n)$ if $e > O(n)$.
- If there are only k possible different edge weights, Kruskal can use counting sort in time $O(k + e)$.
- If $k = O(e)$, this is time $O(e)$ and a sophisticated version of `mergeComponents()` allows a total running time of almost $O(e)$.

Conclusion:

- If there are only $O(e)$ possible edge-weights, use Kruskal.
- Otherwise, if $e > O(n)$ (usually true), use Prim.
- Otherwise, use either.

Minimum spanning trees: summary

- A minimum spanning tree (MST) is a tree that includes all a graph's vertices and has the least possible total edge weight.

Minimum spanning trees: summary

- A minimum spanning tree (MST) is a tree that includes all a graph's vertices and has the least possible total edge weight.
- Prim's algorithm builds an MST by adding the least-weight edge that extends the current tree.
- Kruskal's algorithm adds the least-weight edge from the graph that doesn't make a cycle.

Minimum spanning trees: summary

- A minimum spanning tree (MST) is a tree that includes all a graph's vertices and has the least possible total edge weight.
- Prim's algorithm builds an MST by adding the least-weight edge that extends the current tree.
- Kruskal's algorithm adds the least-weight edge from the graph that doesn't make a cycle.
- Both run in time $O(e \log n)$.

Minimum spanning trees: summary

- A minimum spanning tree (MST) is a tree that includes all a graph's vertices and has the least possible total edge weight.
- Prim's algorithm builds an MST by adding the least-weight edge that extends the current tree.
- Kruskal's algorithm adds the least-weight edge from the graph that doesn't make a cycle.
- Both run in time $O(e \log n)$.
- Prim can be improved to $O(e + n \log n)$ for "most" graphs.
- Kruskal can be improved to almost $O(e)$ if few possible edge weights.