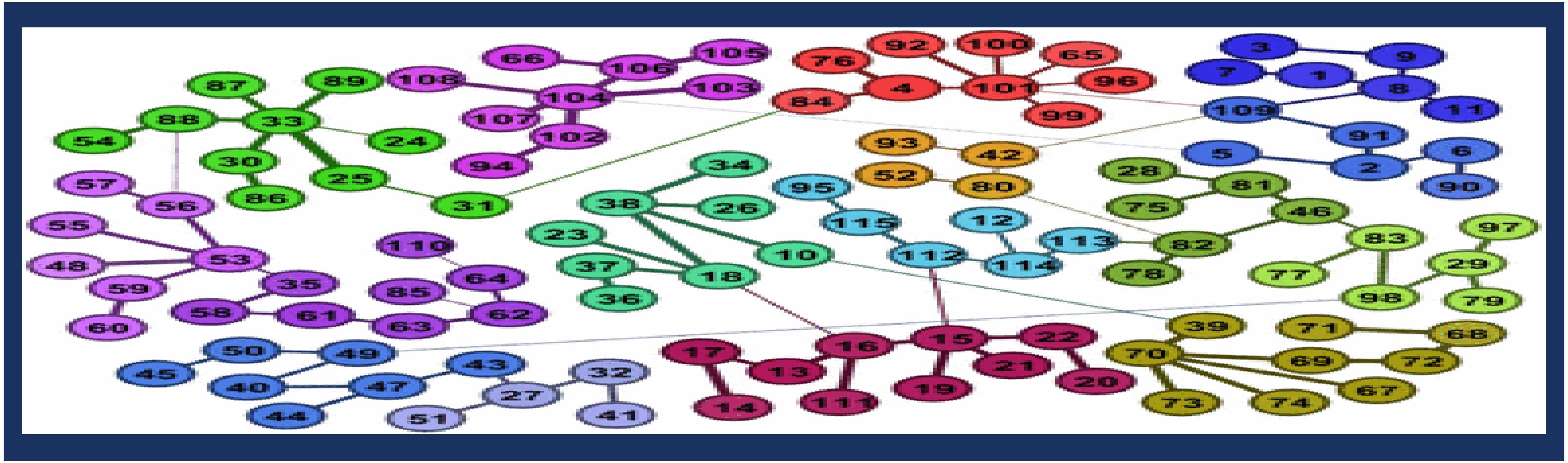
MINIMUM SPANNING TREES

HND THILINI



MINIMUM SPANNING TREES - APPLICATIONS

- Minimum spanning trees have direct applications in the design of networks, including:
    - computer networks
    - telecommunications networks
    - transportation networks
    - water supply networks and
    - electrical grids.

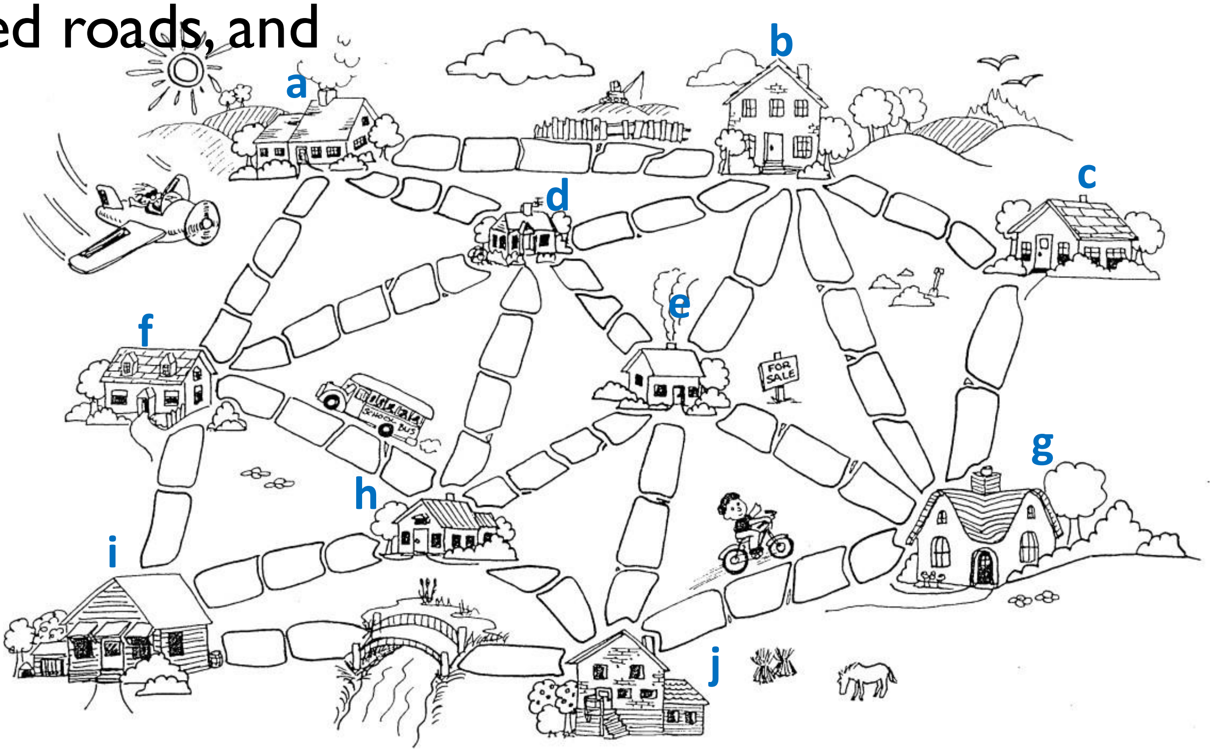
MINIMUM SPANNING TREES - APPLICATIONS

Example 1:

Designing a road network for a city with following conditions:

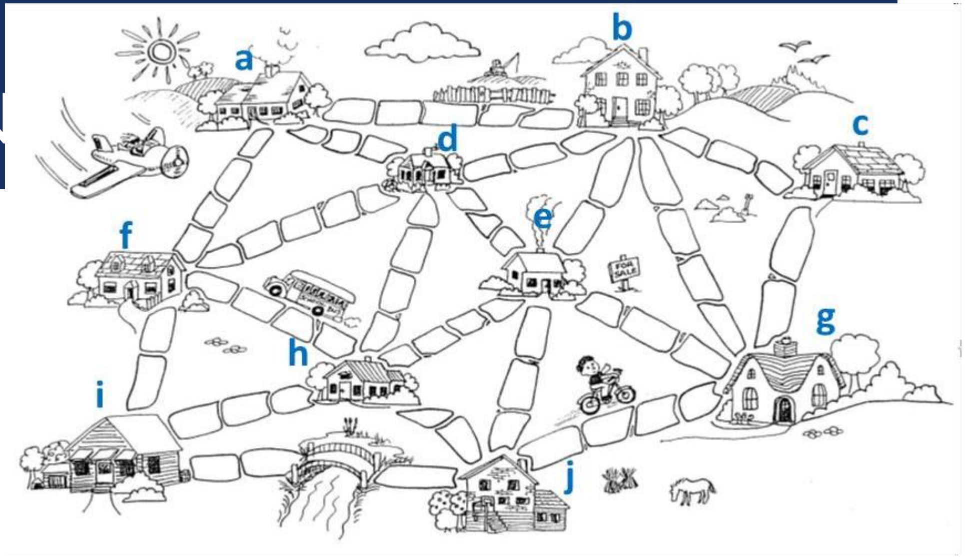
i) Enough streets must be paved so that it is possible for everyone to travel from their house to anyone else's house only along paved roads, and

ii) The paving should cost as little as possible.



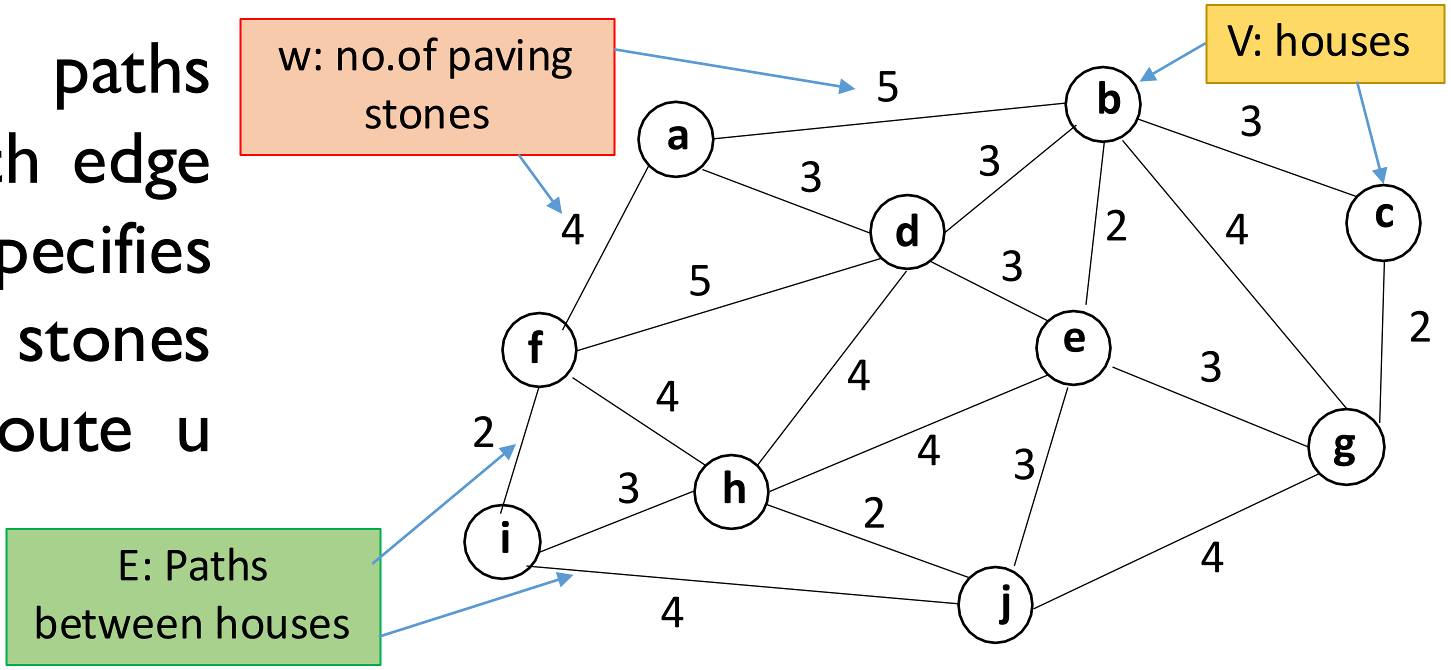
- The layout of the city is given.
- The number of paving stones between each house represents the cost of paving that route.
- Find the best route that connects all the houses, however with few paving stones as possible.

MINIMUM SPANNING TREES - APPLICATION



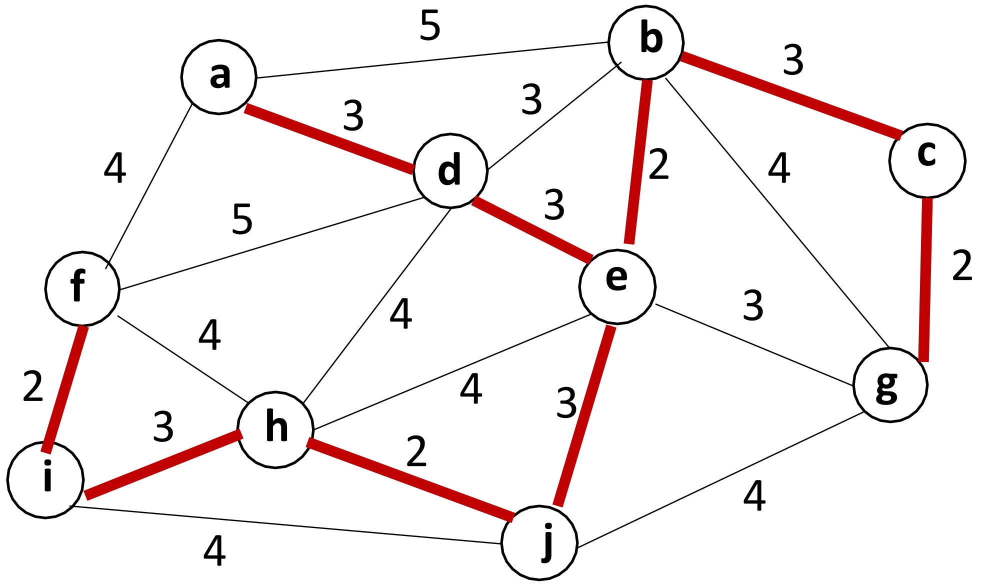
In finding a solution, model road network as a connected, undirected graph G =(V, E), where

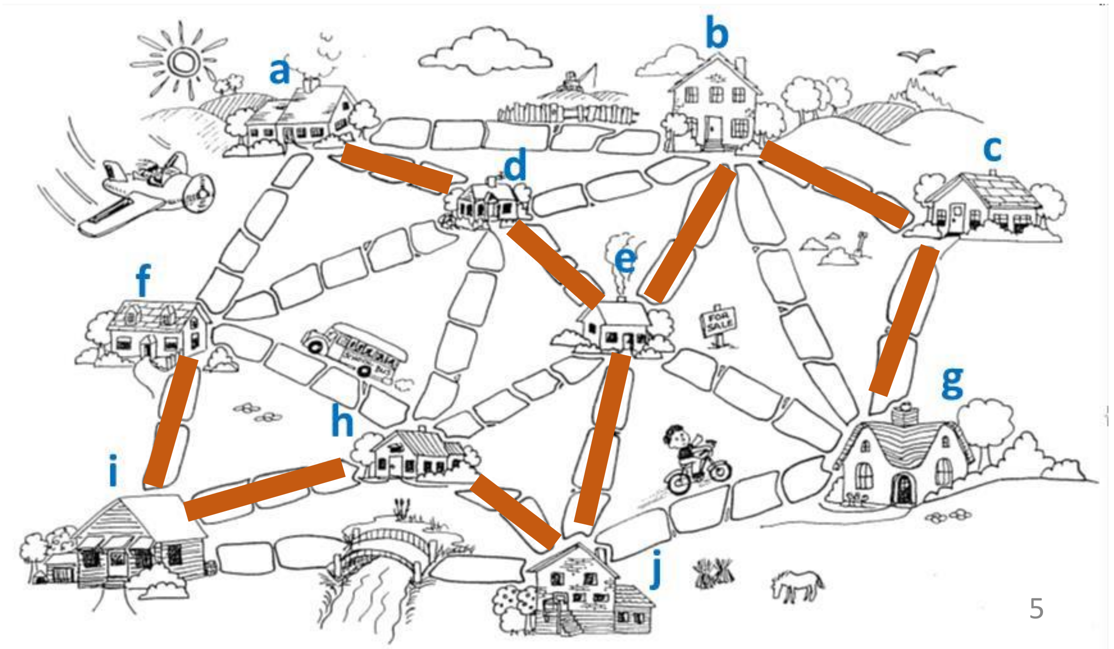
- V is the set of houses,
- E is the set of possible paths between houses, and for each edge (u, v) ∈ E, a weight w(u, v) specifies the cost (amount of paving stones needed) to connect the route u and v.



MINIMUM SPANNING TREES - APPLICATIONS

- Solution to road network problem could be found by computing an acyclic subset T ⊆ E of graph G that connects all of the vertices and whose total weight is the minimum value. This acyclic T forms a tree, which is known as a spanning tree since T "spans" the graph G.





MINIMUM SPANNING TREES - APPLICATIONS

Example 2:

- Building cable networks that join n locations with minimum cost.

Translating a Problem into MST

- Each location of the network must be connected using the least amount of cables.

Modeling the Problem

- Model the problem as an undirected graph $G = (V, E, W)$, where V is the set of locations, E is the set of all possible interconnections between the pairs of locations and $w(e)$ is the length of the cable needed to connect the pair of vertices.
- Find a minimum spanning tree.

SPANNING TREE

A spanning tree for a connected undirected graph G=(V,E) is a subgraph of G that is a tree and contains all the vertices of G

Thus a spanning tree for G is a graph, T = (V', E') with the following properties:

- V' = V
- T is connected
- T is acyclic.

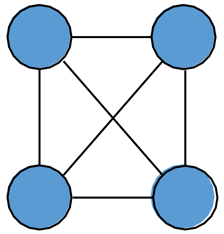
A spanning tree is called a tree because every acyclic undirected graph can be viewed as a general, unordered tree. Since the edges are undirected, any vertex may be chosen to serve as the root of the tree.

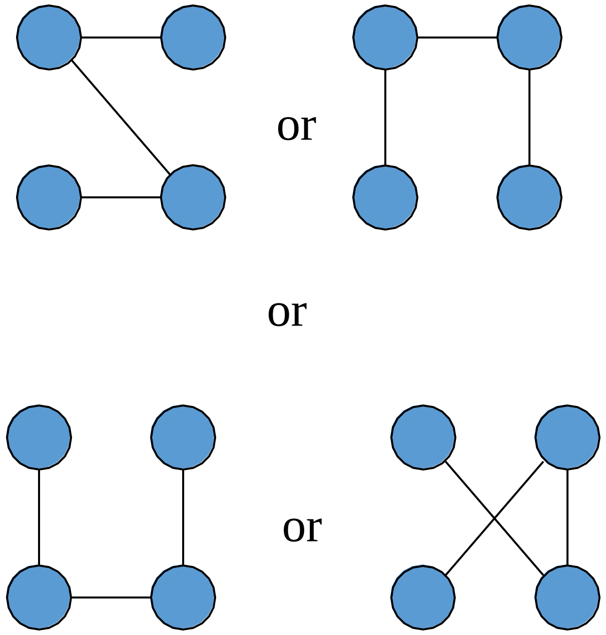
SPANNING TREE

A graph may have many spanning trees.

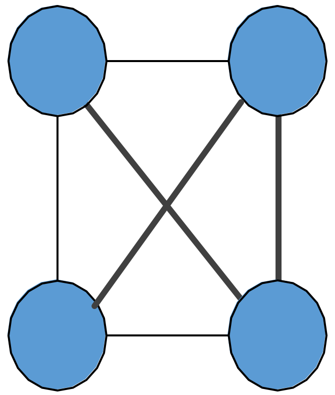
Some Spanning Trees from Graph A

Graph A

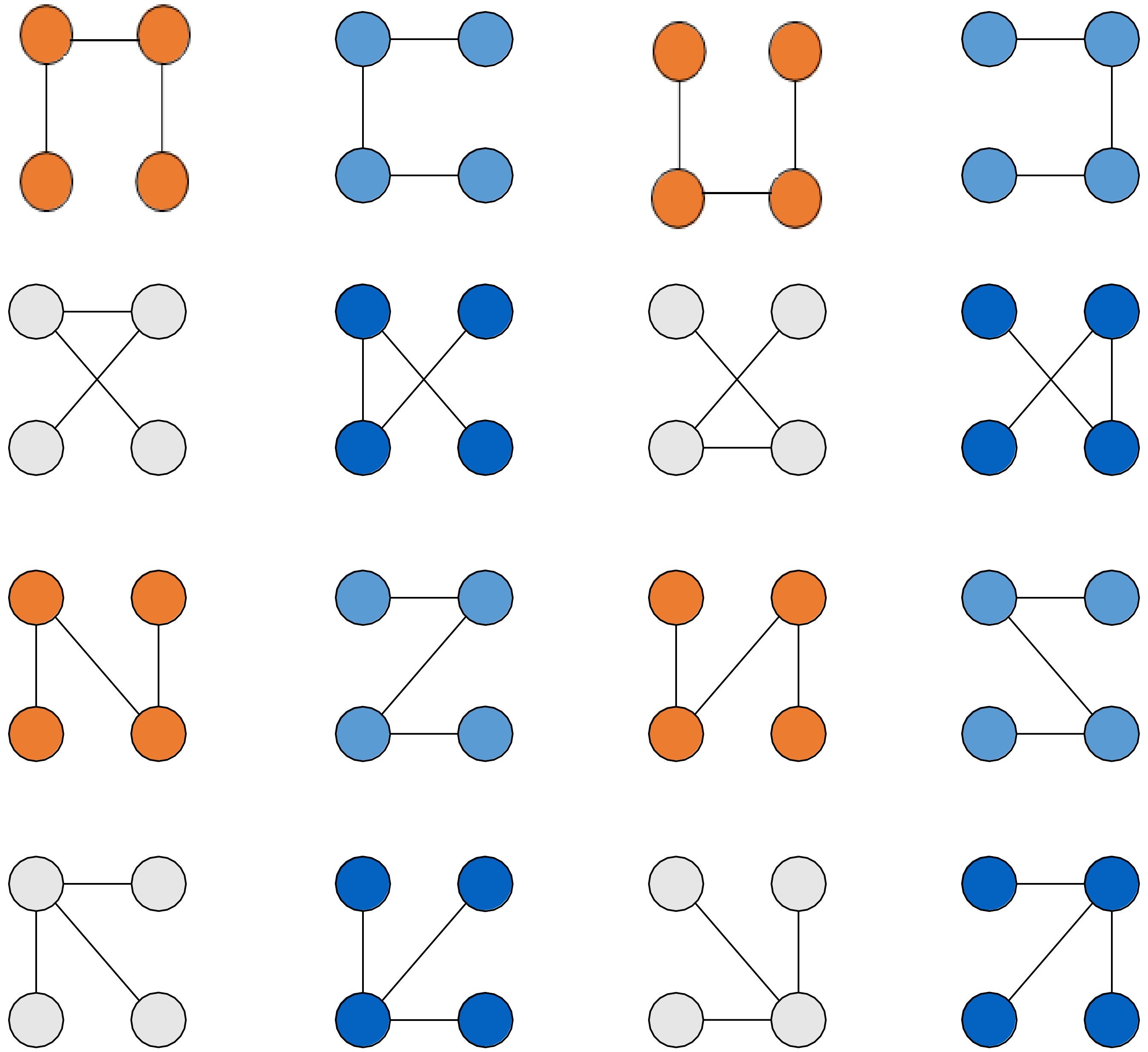




Complete Graph



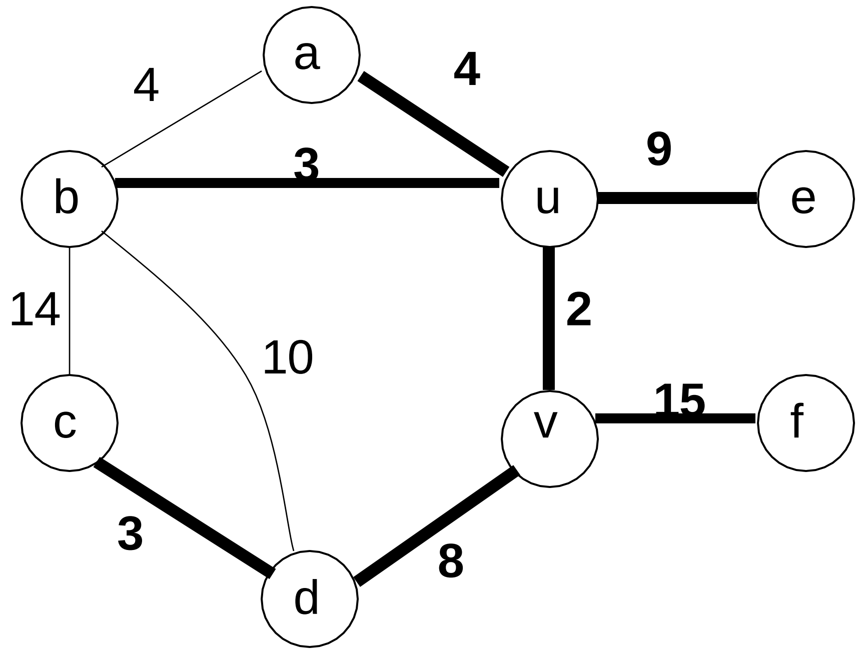
All 16 of its Spanning Trees



MINIMAL SPANNING TREE (MST)

A minimal spanning tree of a (connected undirected) weighted graph G = (V, E, w) is a sub-graph T = (V', E') of G such that

- T is a spanning tree and
- weight $w(T) = \sum_{e \in E'} w(e)$ is minimal.



∗ Can a graph have more then one minimum spanning tree?

UNDERLYING PRINCIPLES

Recall two of the defining properties of a tree:

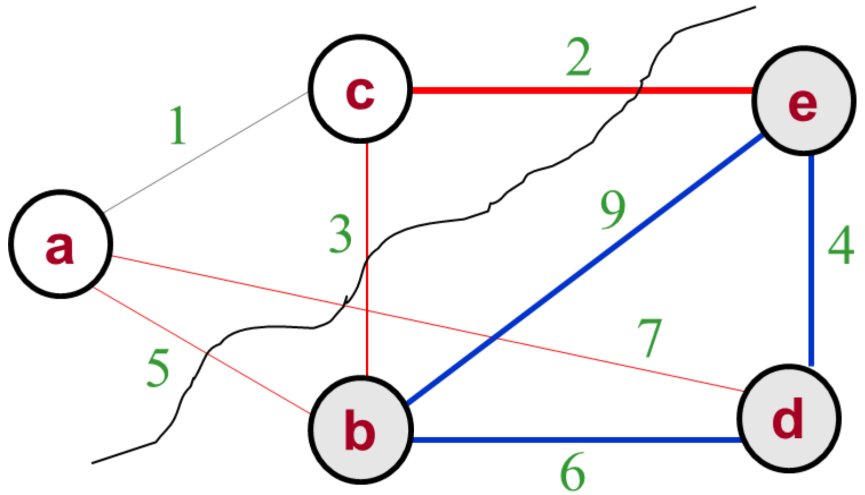
- Removing an edge from a tree breaks it into two separate sub-trees.
- Adding an edge that connects two vertices in a tree creates a unique cycle.

MST ALGORITHM – CUT PROPERTY

Given any cut in an edge-weighted graph (for simplicity assume that all edge weights are distinct), the crossing edge of minimum weight is in the MST of the graph.

A cut (S,V-S) of an undirected graph G = (V,E) is a partition of V.

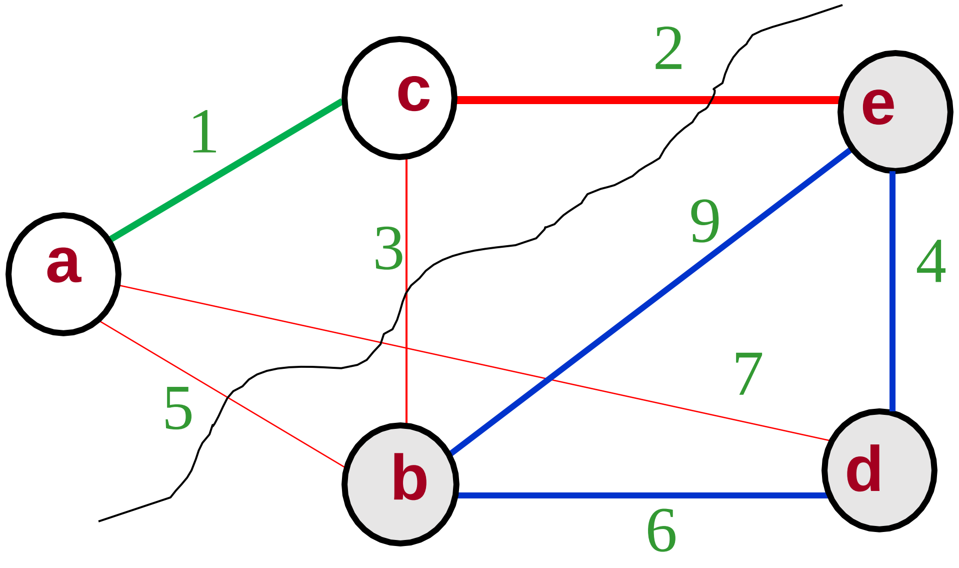
Vertices in the set S – gray, Vertices in (V-S) – white
Edges crossing the cut are connecting white vertices and gray vertices.



MST CONSTRUCTION

The cut property yields a simple greedy algorithm for finding an MST.

- Start with an empty set T of edges.
- Let E′ be the set of edges relevant to a cut which do not contain any edge from T
- As long as T is not a spanning tree, add a minimal-cost edge from E′ (known as light edge) to T.
- Different choices of E′ lead to different specific algorithms.



MST CONSTRUCTION

Given any cut the crossing edge of minimum weight is in the MST.

Proof:

Suppose min-weight crossing edge e is not in the MST.

- Adding e to the MST creates a cycle.
- Some other edge e’ in cycle must be a crossing edge.
- Removing e’ and adding e is also a spanning tree
- Since w(e) is less than w(e’) that spanning tree is lower weight.

Contradiction*

GENERIC ALGORITHM FOR MST

Input : connected weighted graph, G

Output : MST, T, for graph G

Greedy strategy in the generic algorithm

- Grow the MST one edge at a time.
- Manage a set of edges A, that is prior to each iteration, A is a subset of some MST
  - At each step determine an edge (u,v) that can be added to A without violating this invariant.
  - We call such an edge a **safe edge** for A, since it can be safely added to A while maintaining the invariant.

GENERIC ALGORITHM

```
Generic-MST(G,w)
1.  A ← 0
2.  while A does not form a spanning tree
3.    do find an edge (u,v) that is safe for A
4.           A ← A U { (u,v)}
5.  return A
```

Safe edge - a light edge satisfying a given property.

GREEDY CHOICE

Two algorithms to build a minimum spanning tree.

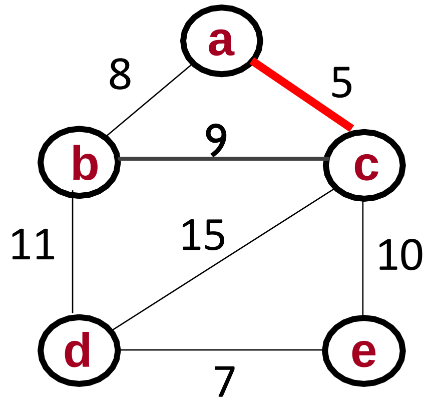
- MST can be grown from a forest of spanning trees : find a safe edge to be added to the growing forest by finding, of all the edges that connects two distinct trees in the forest. The choice is greedy because at each step it adds to the forest an edge of least possible weight. (**Kruskal's algorithm**)

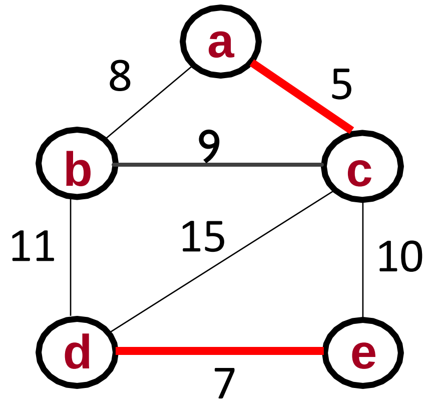
- MST can be grown from the current spanning tree: Each step adds to the tree A a light edge that connects A to an isolated vertex— one on which no edge of A is incident. (**Prim's algorithm**)

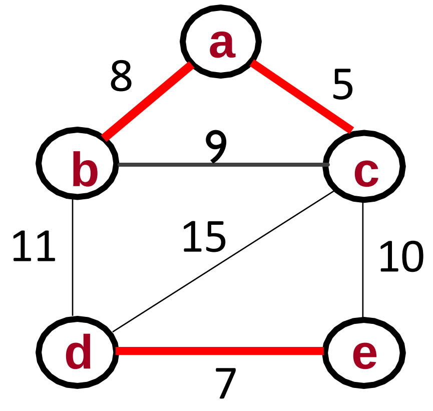
KRUSKAL’S ALGORITHM

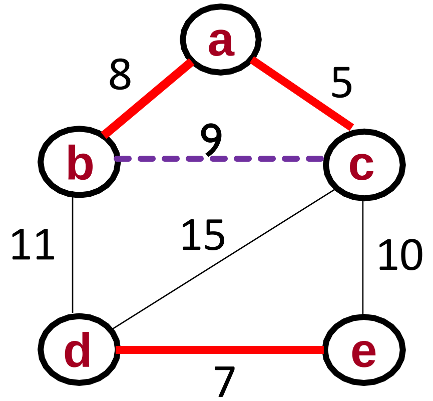
KRUSKAL'S ALGORITHM

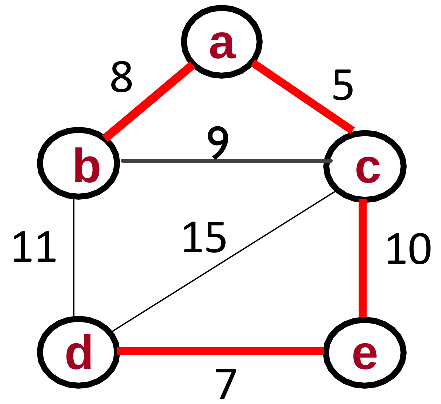
Kruskal's Algorithm is growing the MST as a forest.









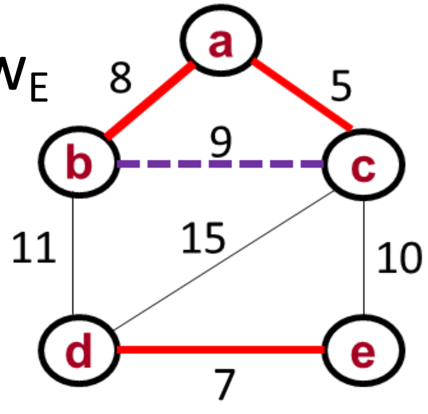


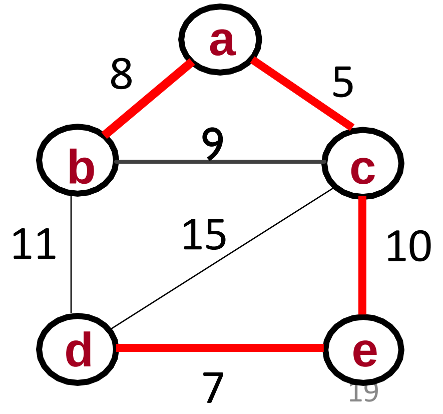
KRUSKAL'S ALGORITHM

$|E| \le |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V)$
$O(E \lg E) \Rightarrow$ **$O(E \lg V)$**

```
Kruskal ()
- Sort edges in order of increasing cost so that $w_1 < w_2 < \dots\dots\dots < w_E$
- T = Ø
- For i = 1 to E      O(E)
      if T U {i} has no cycles
            add i to T
- Return T
```

O(V)* (Need to search for edges in T only and contains ≤ V-1 no. of edges).





Running time : $O(E \lg V) + O(VE) \Rightarrow O(VE)$

* - cycle checking dominates the running time.

KRUSKAL'S ALGORITHM

- Runtime can be improved by using Union-Find data structure.
- Motivation: O(1) time cycle checks in Kruskal's algorithm.
- Union-Find data structure keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets and supports two useful operations:
  - **MakeSet(v):** create the set {v}.
  - **Find-set(v):** Determine which subset a particular element v is in. This can be used for determining if two elements are in the same subset – O(1).
  - **Union (u,v):** Join two subsets (i.e. the set that u is in and the set that v is in) into a single subset – O(lg v).

KRUSKAL’S ALGORITHM

- Initialize the set A to the empty set
- Create |V| trees (sets), one containing each vertex.
- Sort the edges in increasing order of weight.
- Take the edges in the sorted order.
- For each edge (u,v), check whether the endpoints (vertices) u and v belong to the same tree – Check for cycles [Find-set(u) & Find-set(v)].
- It is safe to connect two vertices if they belong to different trees. If so the edge (u,v) is added to A – Union(u,v)
- Vertices in the two trees are merged.

KRUSKAL'S ALGORITHM

MST-Kruskal(G,w)

A – Tree

w - weight

```
1.  A ← 0
2.  For each vertex v ∈ V[G]
3.      do Make-Set(v)
4.  sort the edges of E into nondecreasing order by weight w
5.  for each edge (u,v) ∈ E, taken in nondecreasing order by weight w
6.      do if Find-Set(u)   ≠ Find-Set(v)
7.          then A ← A U { (u,v)}
8.              Union (u,v)
9.  return A
```

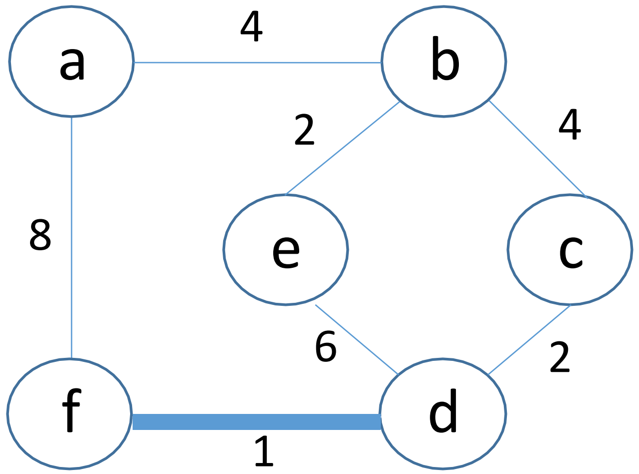
KRUSKAL'S ALGORITHM EXAMPLE

Initially A = { }

Sets – {a} {b} {c} {d} {e} {f}

E – Sorted in Ascending Order

(f, d)	(b, e)	(c, d)	(a, b)	(b, c)	(e, d)	(a, f)



Step 1

Take (f,d) ; Find-Set(f) ≠ Find-Set(d) => **add (f,d) to A**

A = {(f,d)}

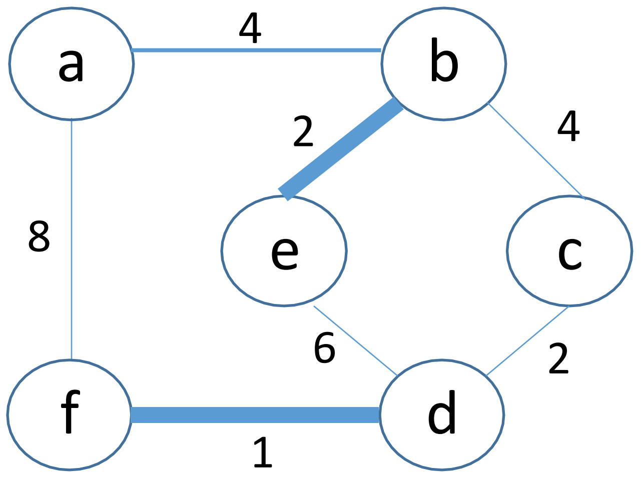
Combine Set(f) & Set(d)

Sets - {a} {b} {c} {e} {f,d}

KRUSKAL'S ALGORITHM EXAMPLE

Step 2

(f, d)	(b, e)	(c, d)	(a, b)	(b, c)	(e, d)	(a, f)



Take (b,e) ; Find-Set(b) ≠ Find-Set(e) => **add (b,e) to A**

A = {(f,d), (b,e)}

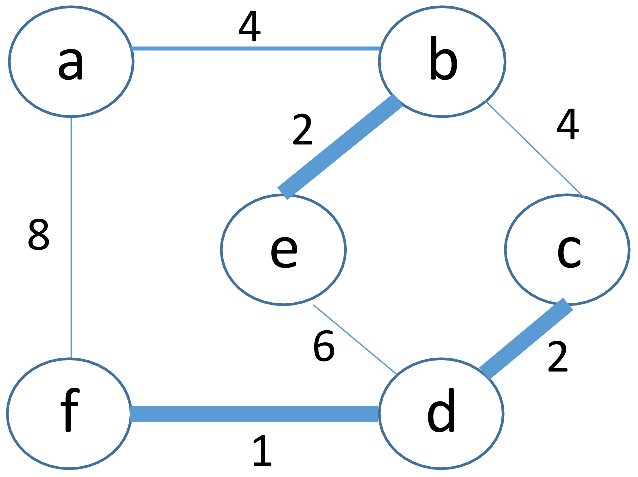
Combine Set(b) & Set(e)

Sets - {a} {b,e} {c} {f,d}

KRUSKAL'S ALGORITHM EXAMPLE

Step 3

(f, d)	(b, e)	(c, d)	(a, b)	(b, c)	(e, d)	(a, f)



Take (c,d) ; Find-Set(c) ≠ Find-Set(d) => **add (c,d) to A**

A = {(f,d), (b,e),(c,d)}

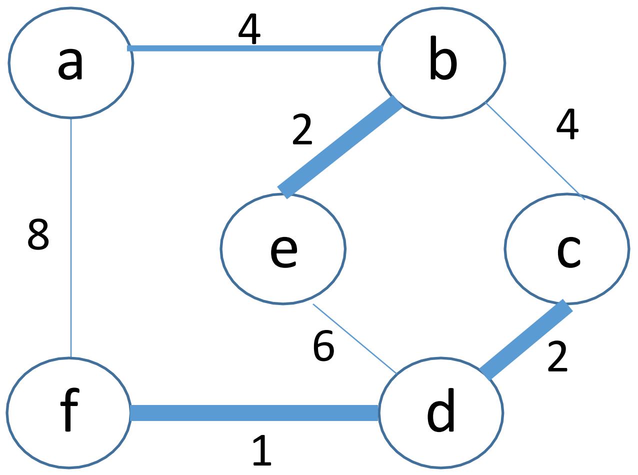
Combine Set(c) & Set(d)

Sets - {a} {b,e} {f,d,c}

KRUSKAL'S ALGORITHM EXAMPLE

Step 4

(f, d)	(b, e)	(c, d)	(a, b)	(b, c)	(e, d)	(a, f)



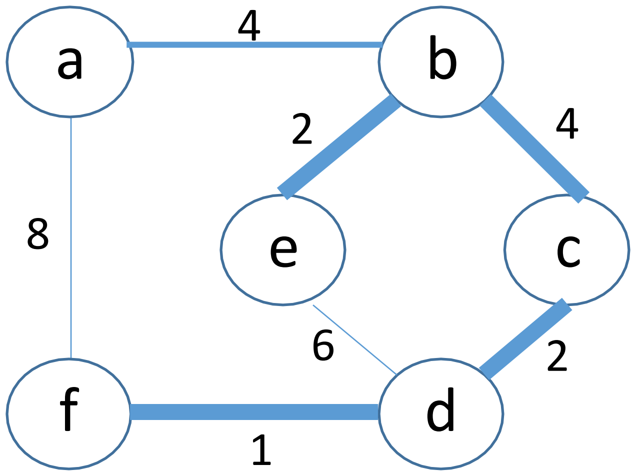
Take (a,b) ; Find-Set(a) ≠ Find-Set(b) => **add (a,b) to A**

A = {(f,d), (b,e),(c,d),(a,b)}

Combine Set(a) & Set(b)

Sets - {b,e,a} {f,d,c}

KRUSKAL'S ALGORITHM EXAMPLE



Step 5

(f, d)	(b, e)	(c, d)	(a, b)	(b, c)	(e, d)	(a, f)

Take (b,c) ; Find-Set(b) ≠ Find-Set(c) => **add (b,c) to A**

A = {(f,d), (b,e),(c,d),(a,b),(b,c)}

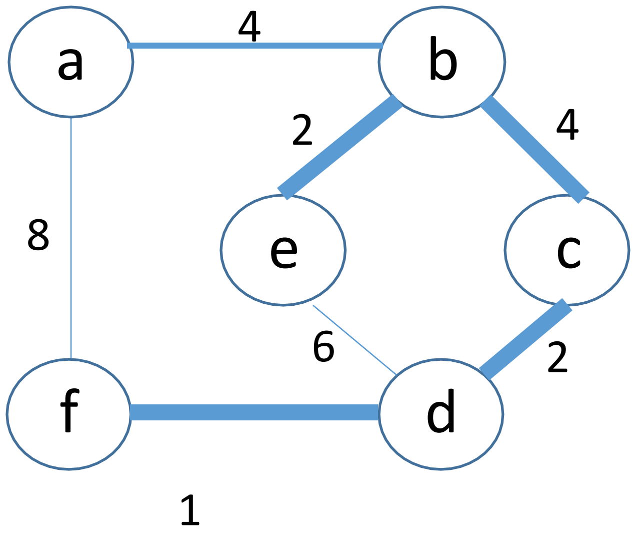
Combine Set(b) & Set(c)

Sets - {b,e,a,f,d,c}

KRUSKAL’S ALGORITHM EXAMPLE

Step 6

(f, d)	(b, e)	(c, d)	(a, b)	(b, c)	(e, d)	(a, f)



Take (e,d) ; Find-Set(e) = Find-Set(d) => **Ignore**

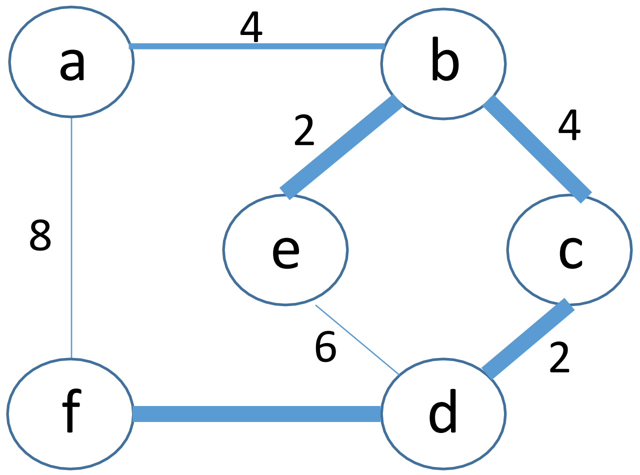
A = {(f,d), (b,e),(c,d),(a,b),(b,c)}

Sets - {b,e,a,f,d,c}

KRUSKAL'S ALGORITHM EXAMPLE

Step 7

(f, d)	(b, e)	(c, d)	(a, b)	(b, c)	(e, d)	(a, f)

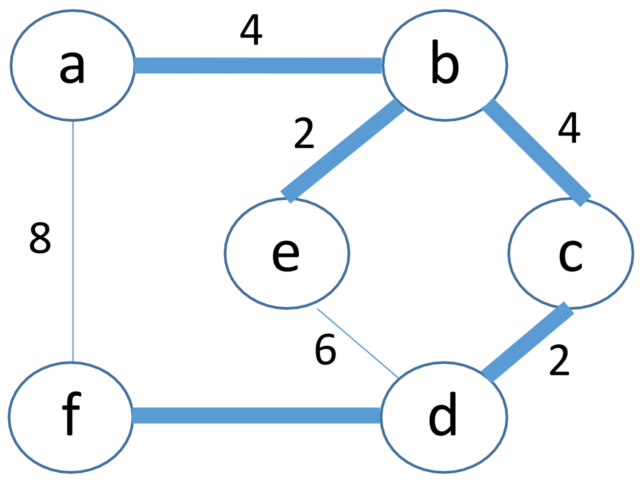


Take (a,f) ; Find-Set(a) = Find-Set(f) => **Ignore**

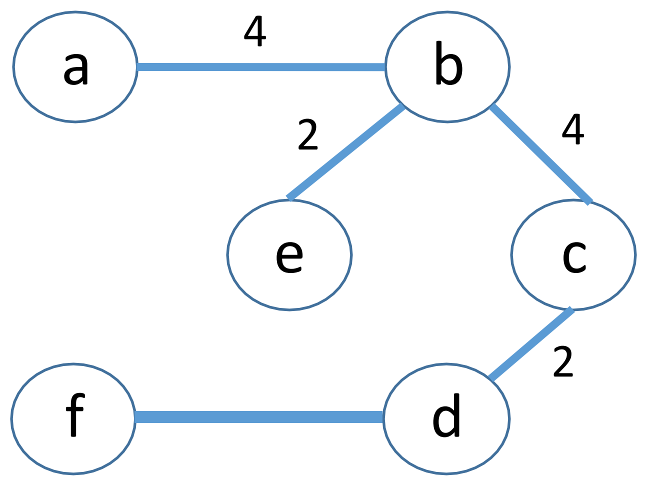
A = {(f,d), (b,e),(c,d),(a,b),(b,c)}

Sets - {b,e,a,f,d,c}

KRUSKAL'S ALGORITHM EXAMPLE

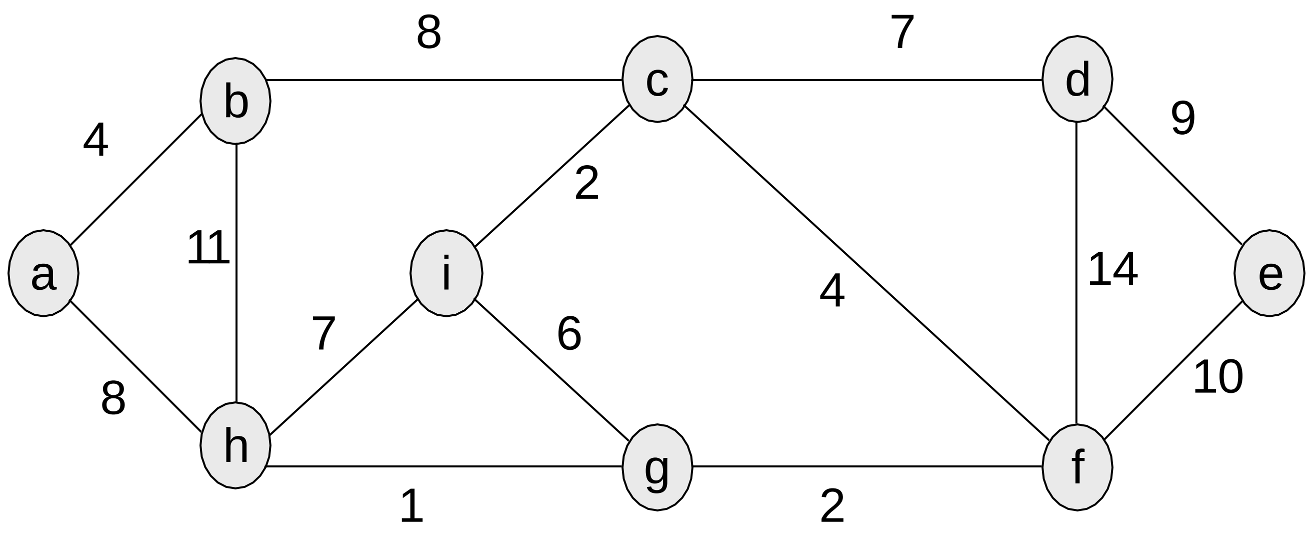


Graph



MST

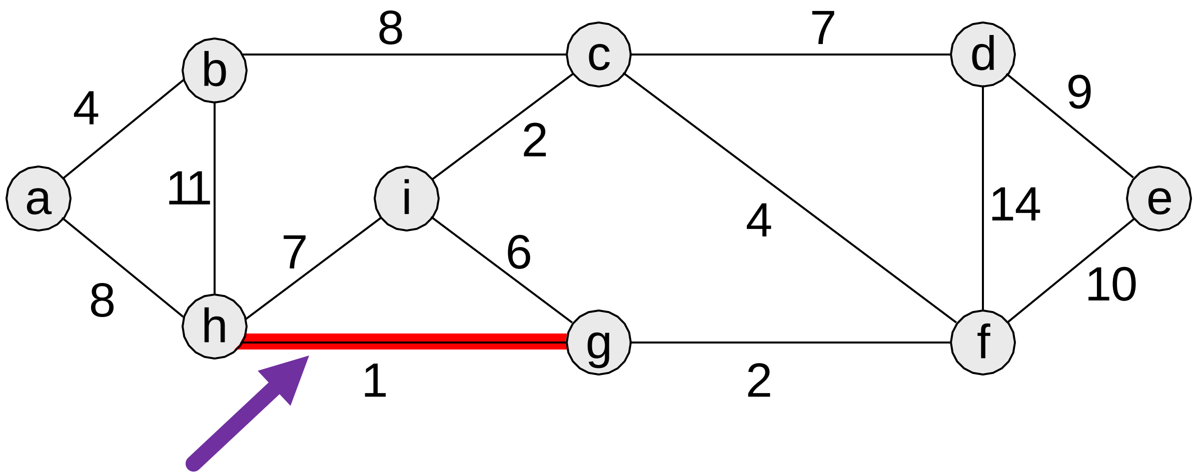
KRUSKAL'S ALGORITHM - PROBLEM



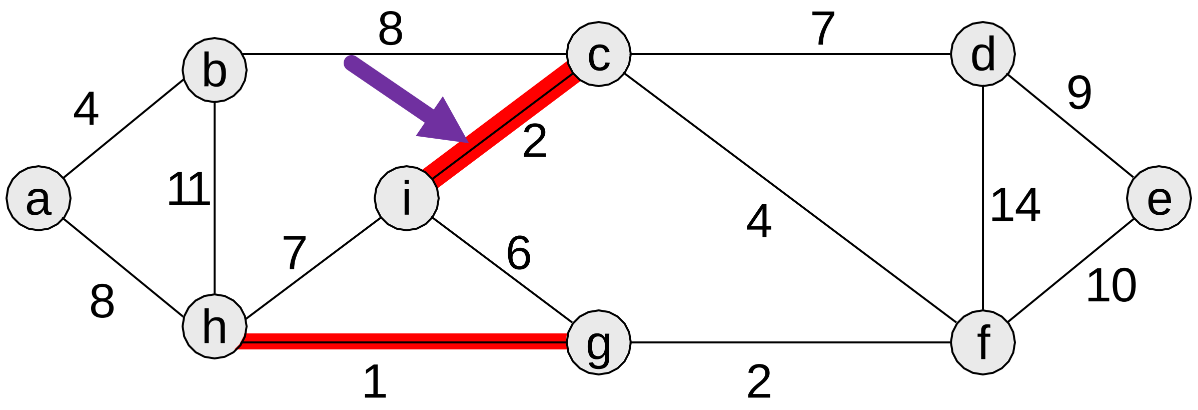
* Same example is given for Prim's in order to understand how the MST is constructed with respect to the two algorithms.

KRUSKAL'S ALGORITHM - SOLUTION

(a)

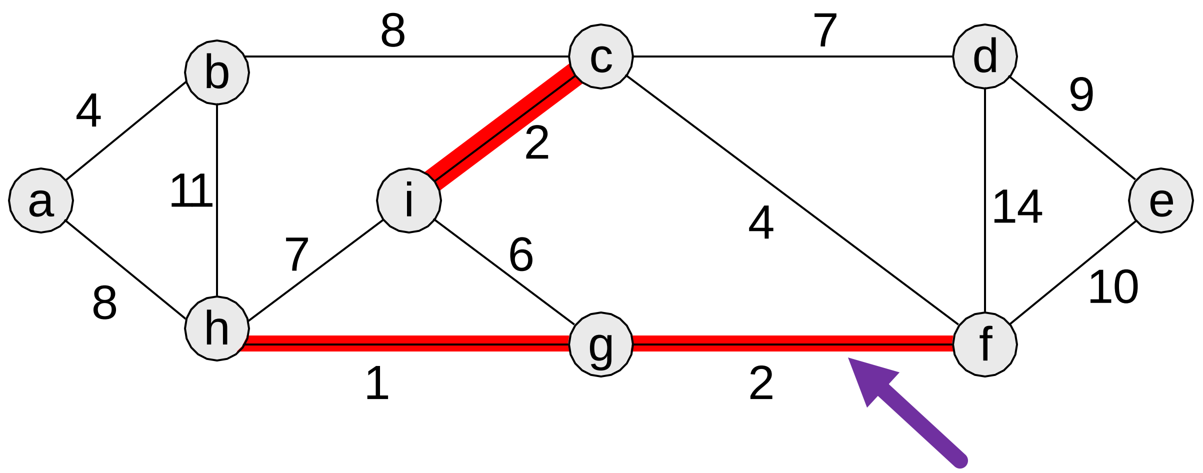


(b)

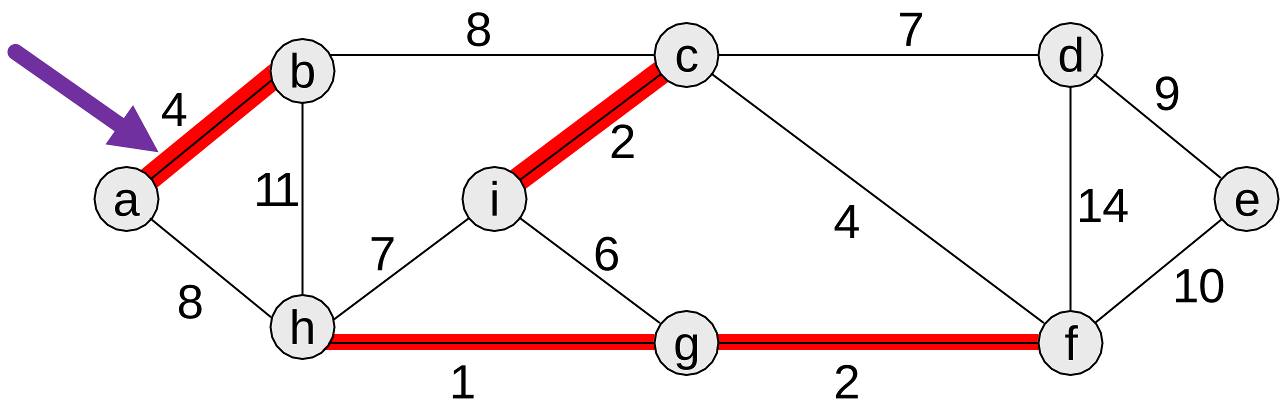


KRUSKAL'S ALGORITHM - SOLUTION

(c)

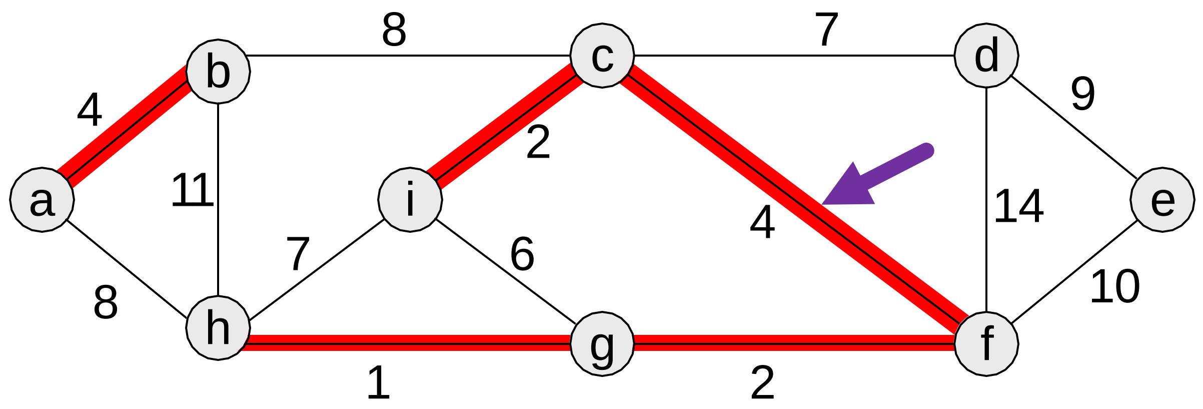


(d)

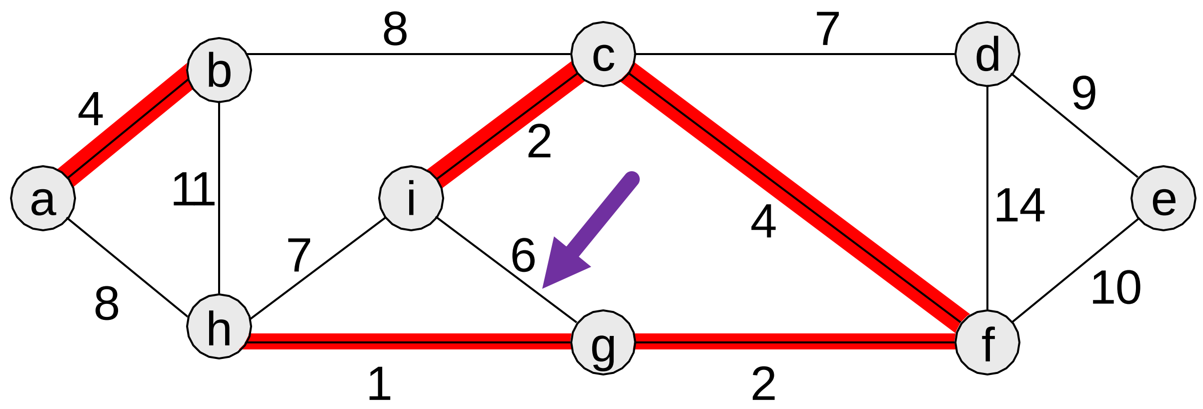


KRUSKAL'S ALGORITHM - SOLUTION

(e)

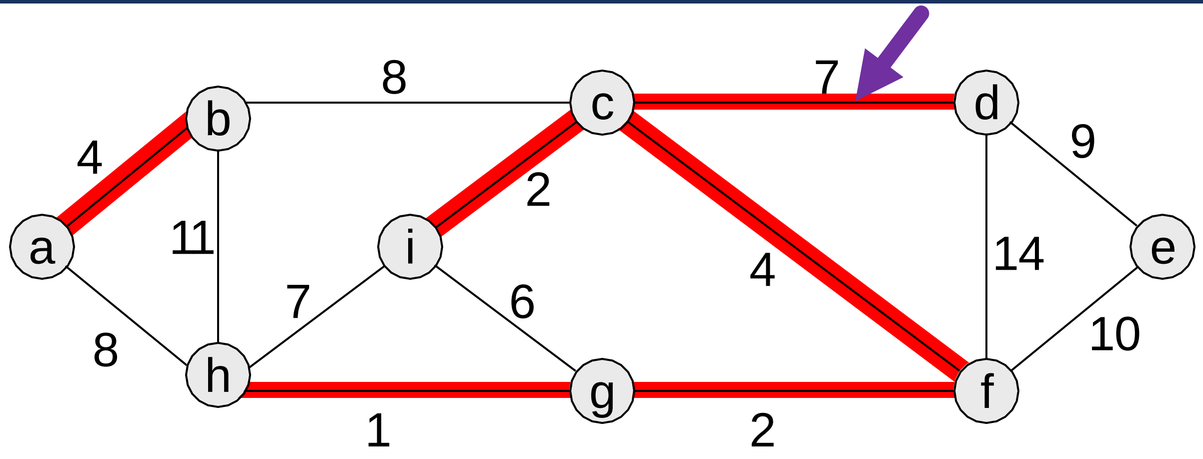


(f)

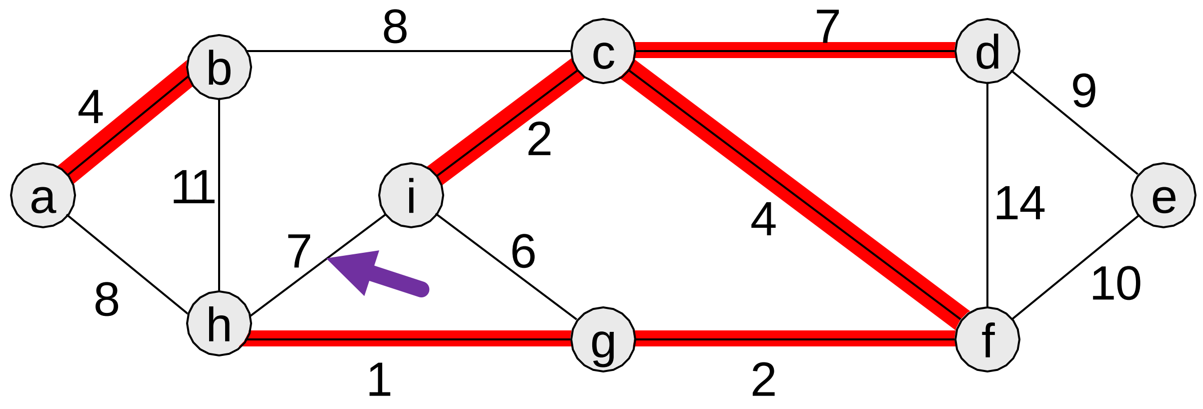


KRUSKAL'S ALGORITHM - SOLUTION

(g)

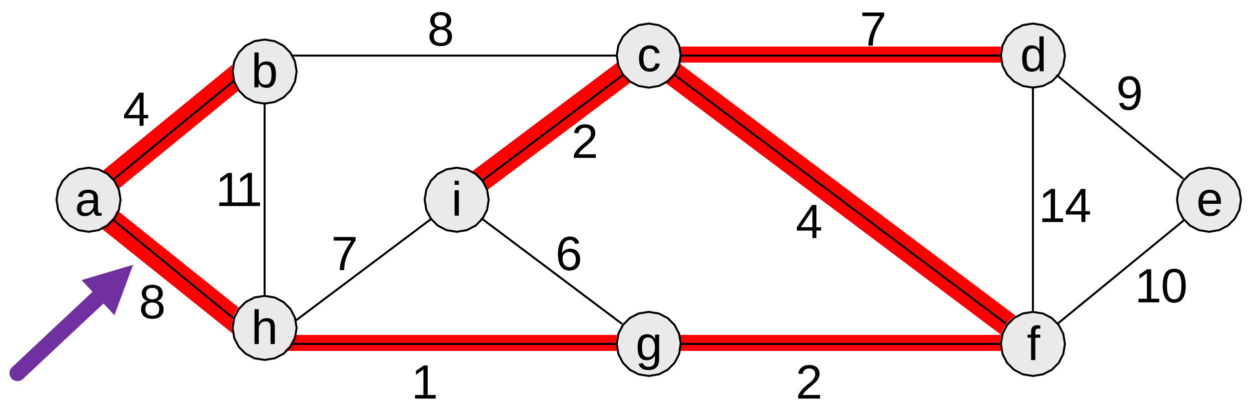


(h)

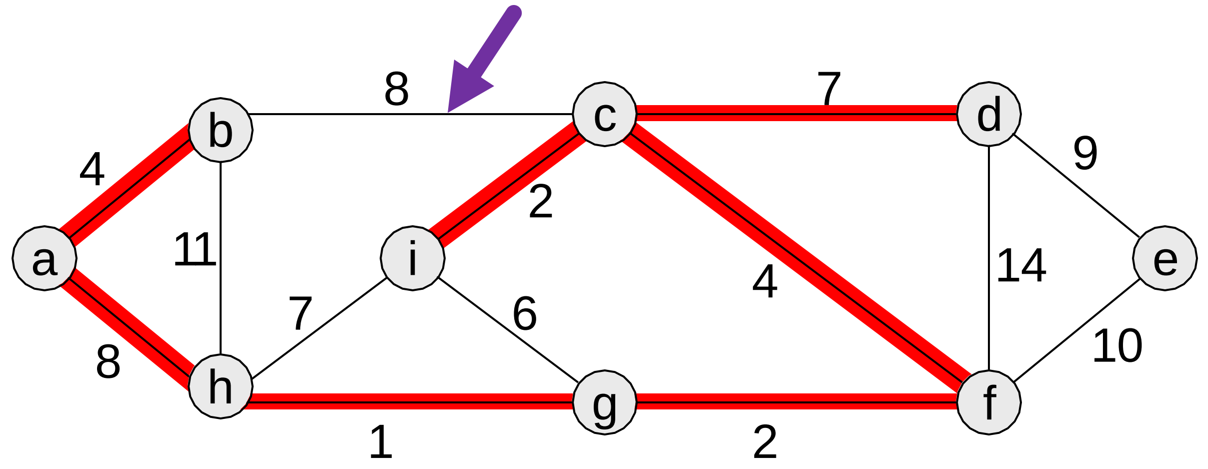


KRUSKAL’S ALGORITHM - SOLUTION

(i)

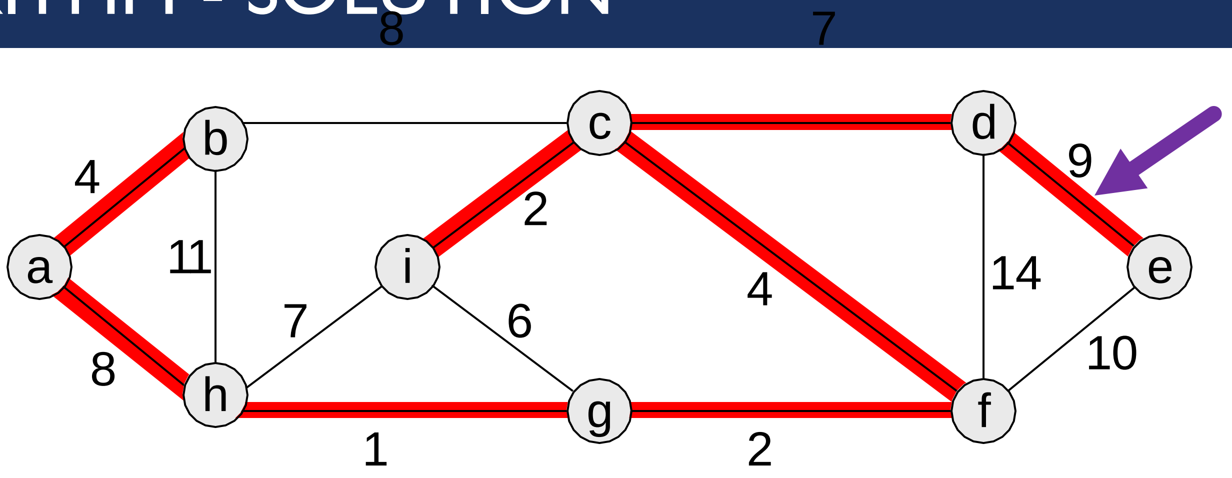


(j)

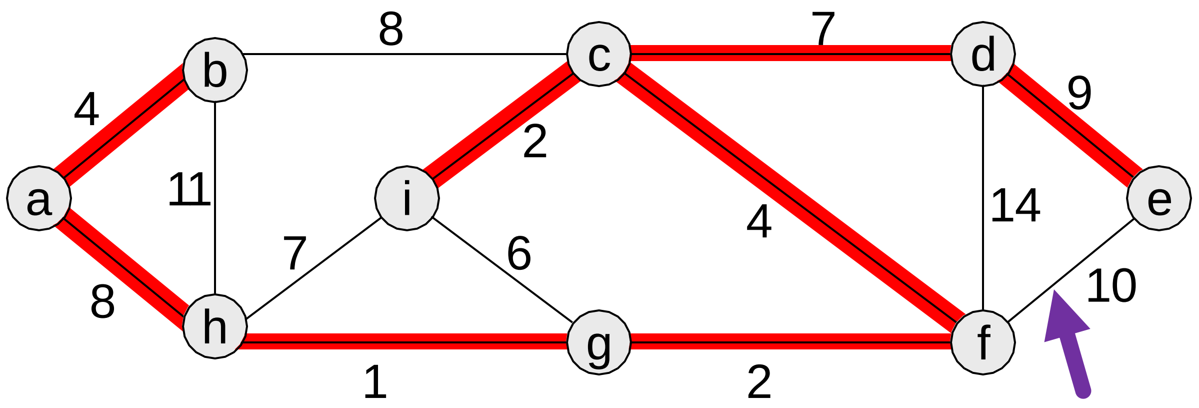


KRUSKAL'S ALGORITHM - SOLUTION

(k)

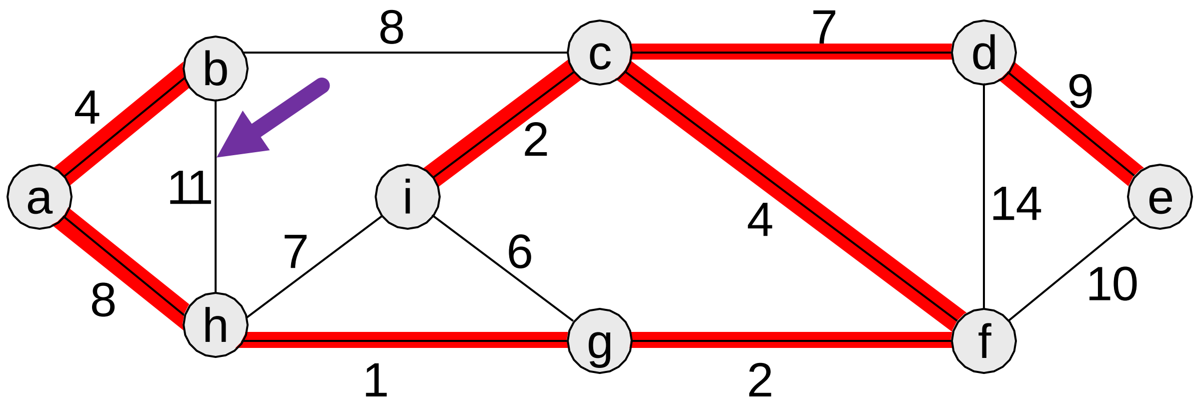


(l)

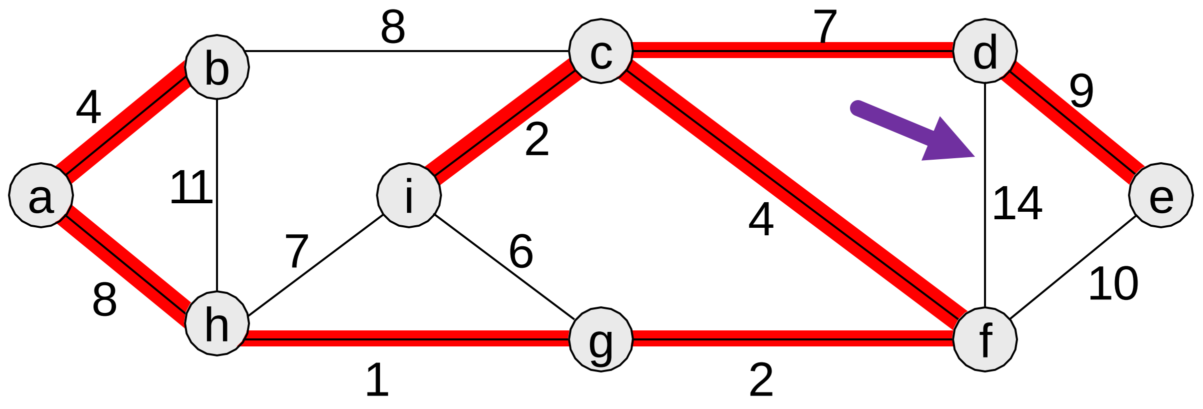


KRUSKAL'S ALGORITHM - SOLUTION

(m)



(n)



KRUSKAL'S ALGORITHM – RUNTIME ANALYSIS

Run time depends on the operations on the disjoint sets data structure:

- Initialize the set A: $O(1)$
- First for loop MAKE-SETs $O(1) - V$
- Sort E: $O(E \lg E)$ $= O(E \lg V)$

$|E| \leq |V|^2 \Rightarrow \lg |E| = O(2 \lg V)$
$= O(\lg V)$

- Second for loop:
  - FIND-SETs $= O(1) - E \Rightarrow O(E)$
  - UNIONs $= O(\lg V) - V \Rightarrow O(V \lg V)$

Therefore, total run time is $O(E \lg V)$.

PRIM'S ALGORITHM

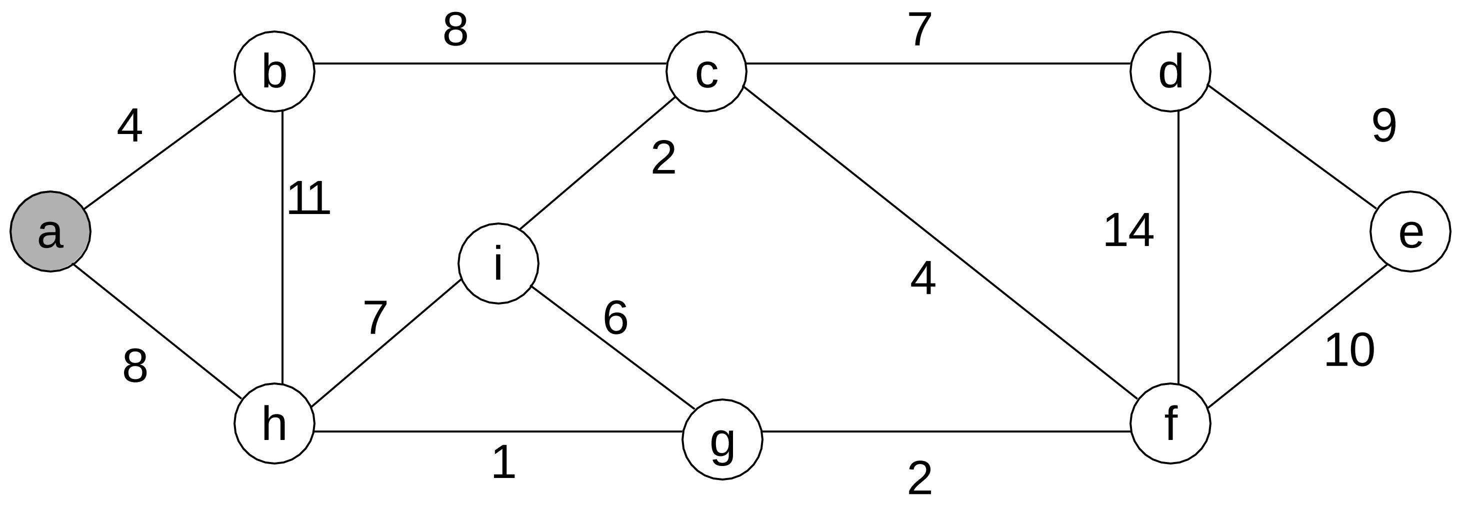
PRIM'S ALGORITHM

- Edges in the set A always form a single tree.
- Tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V.
- At each step a light edge is added to the tree A. The algorithm implicitly maintains the set A.
- This strategy is greedy.

PRIM'S ALGORITHM

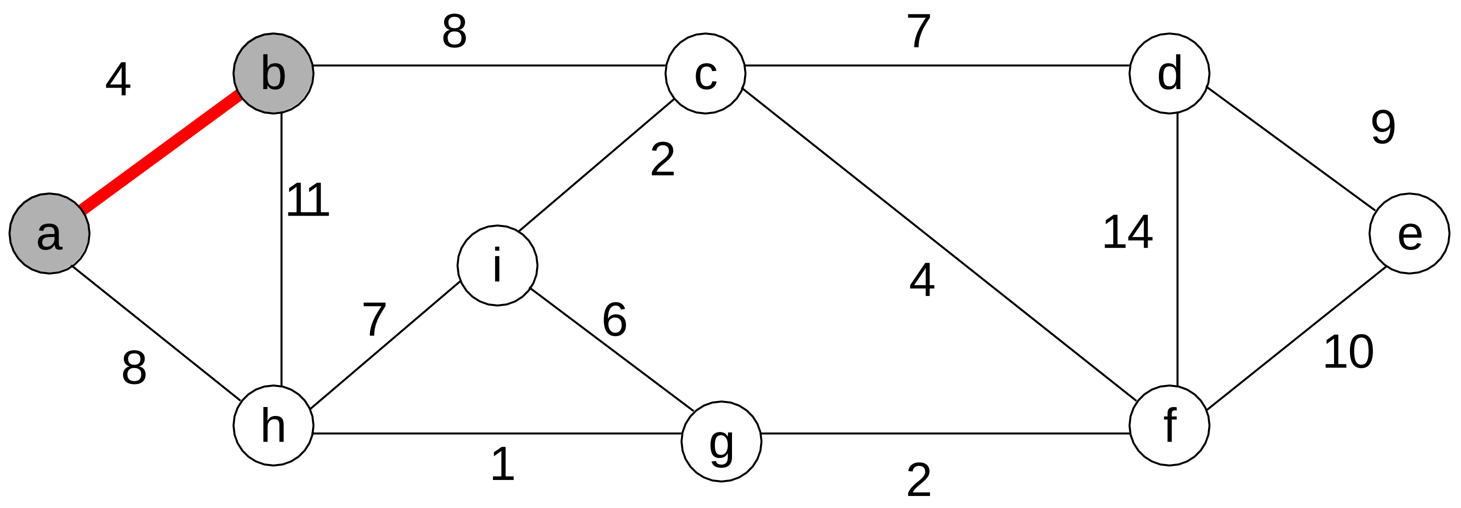
```
MST-PRIM (G, w, r)
1   for each u ∈ V [G]
2         do key [u] ← ∞
3              π [u] ← NIL
4      key [r ] ← 0
5      Q ← V [G]
6      while Q ≠ ∅
7         do u ← EXTRACT-MIN (Q)
8              for each v ∈ Adj[u]
9                 do if v ∈ Q and w(u, v) < key[v]
10                     then π[v] ← u
11                          key[v] ← w(u, v)
```

PRIM’S ALGORITHM - PROBLEM

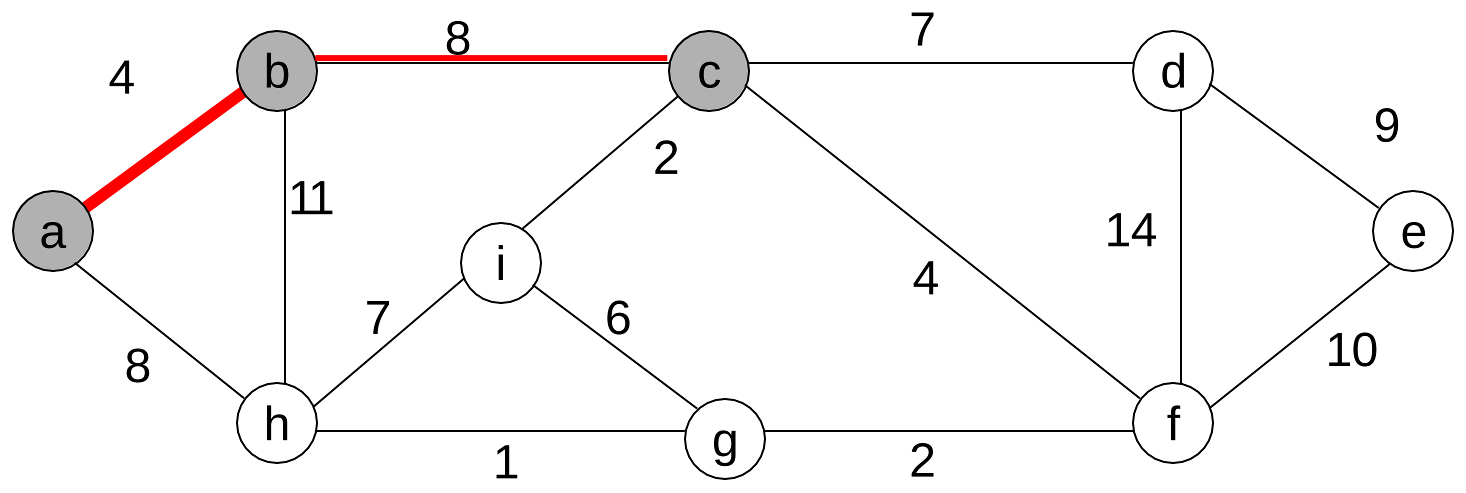


PRIM'S ALGORITHM - SOLUTION

a)

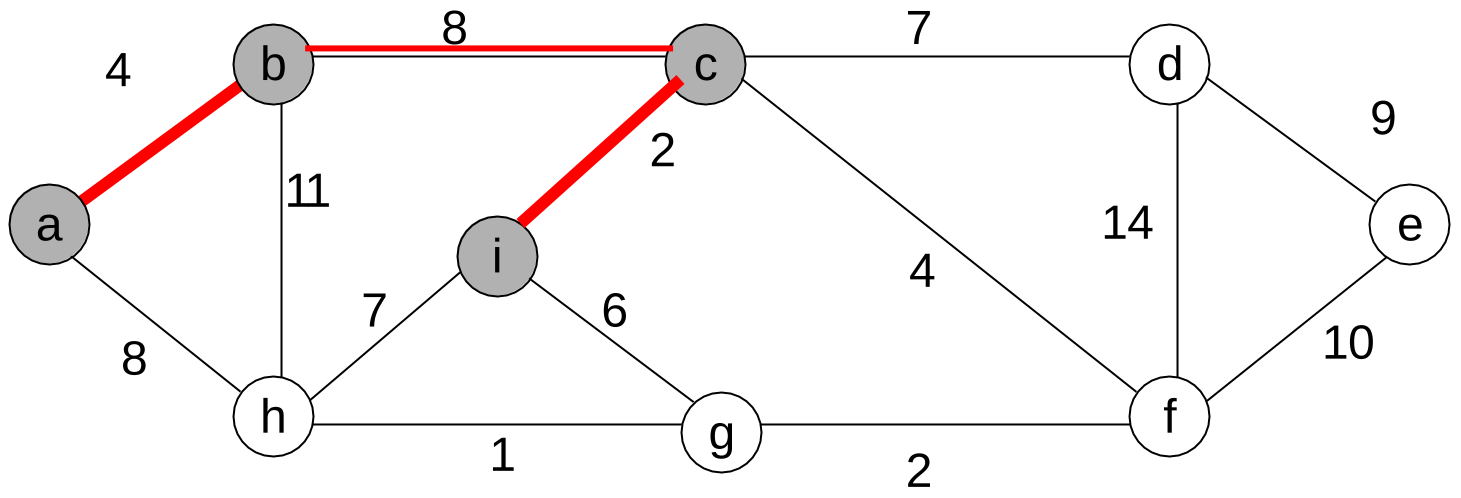


b)

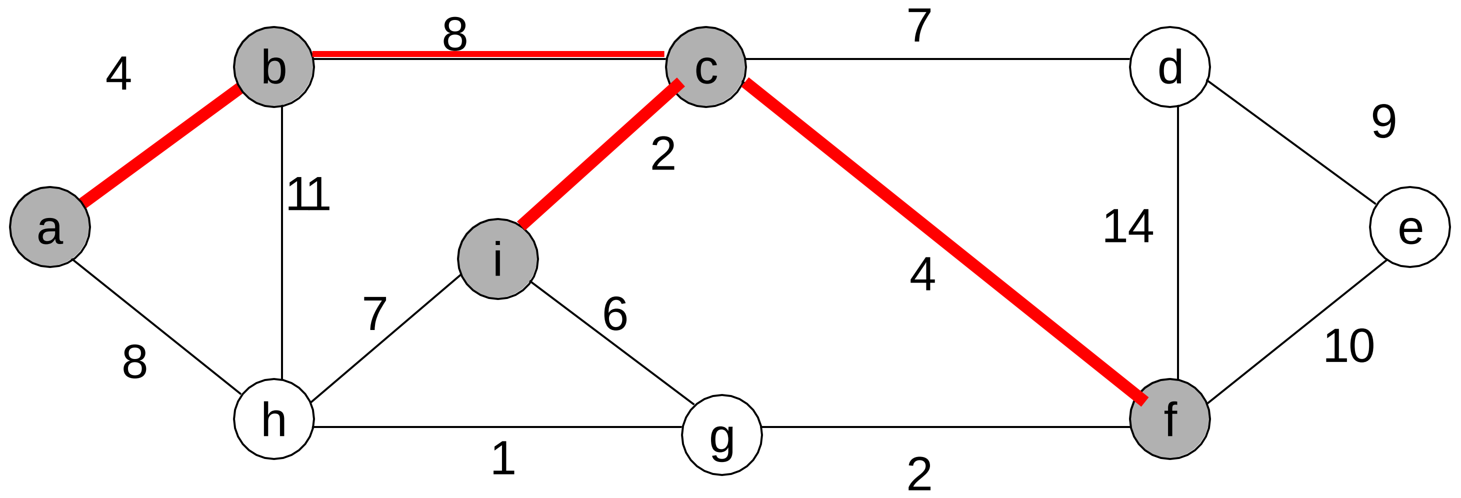


PRIM'S ALGORITHM - SOLUTION

c)

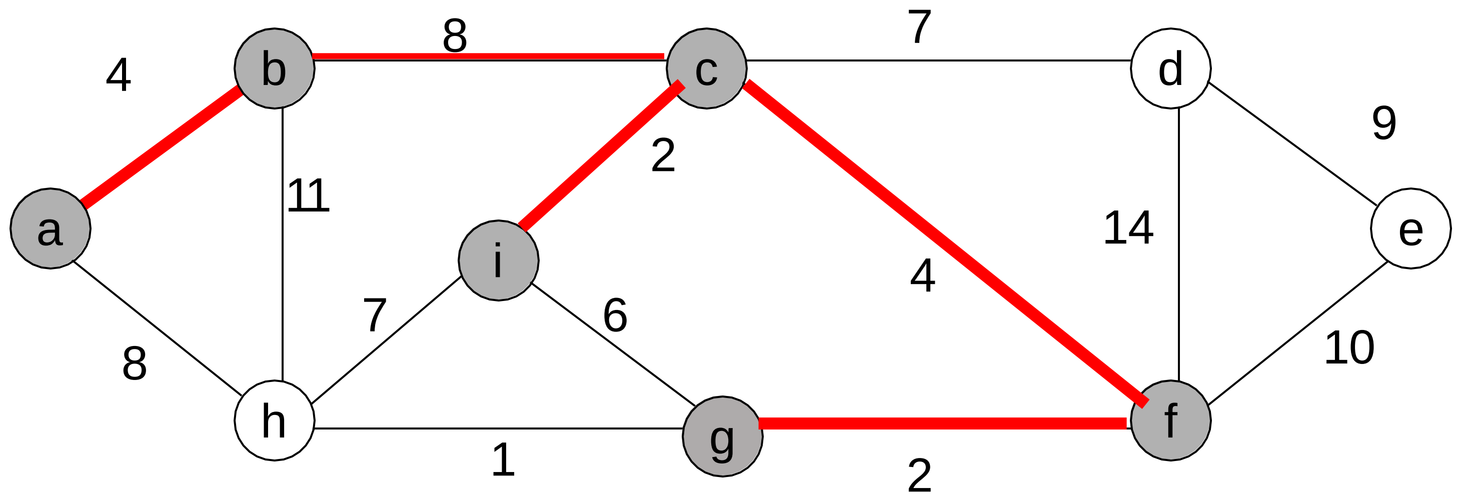


d)

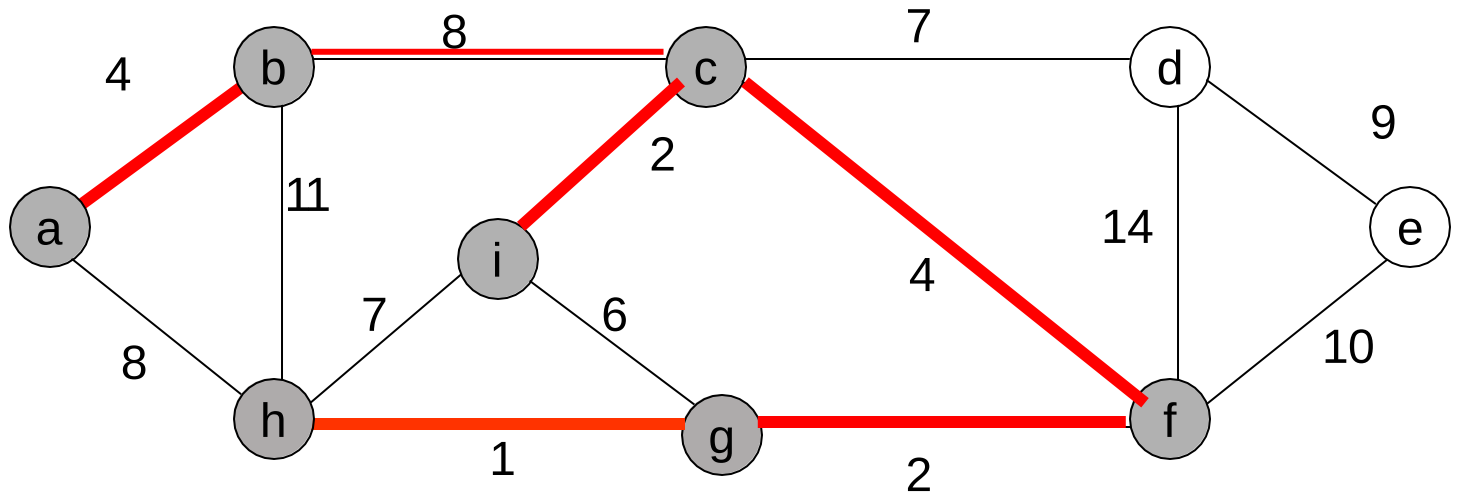


PRIM’S ALGORITHM - SOLUTION

e)

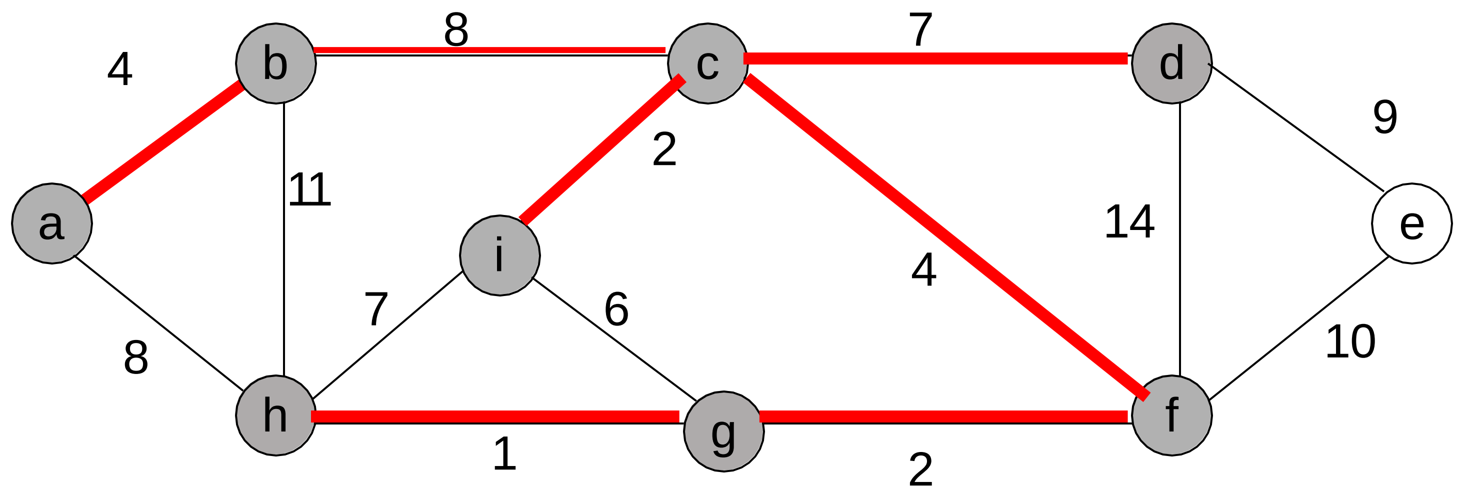


f)

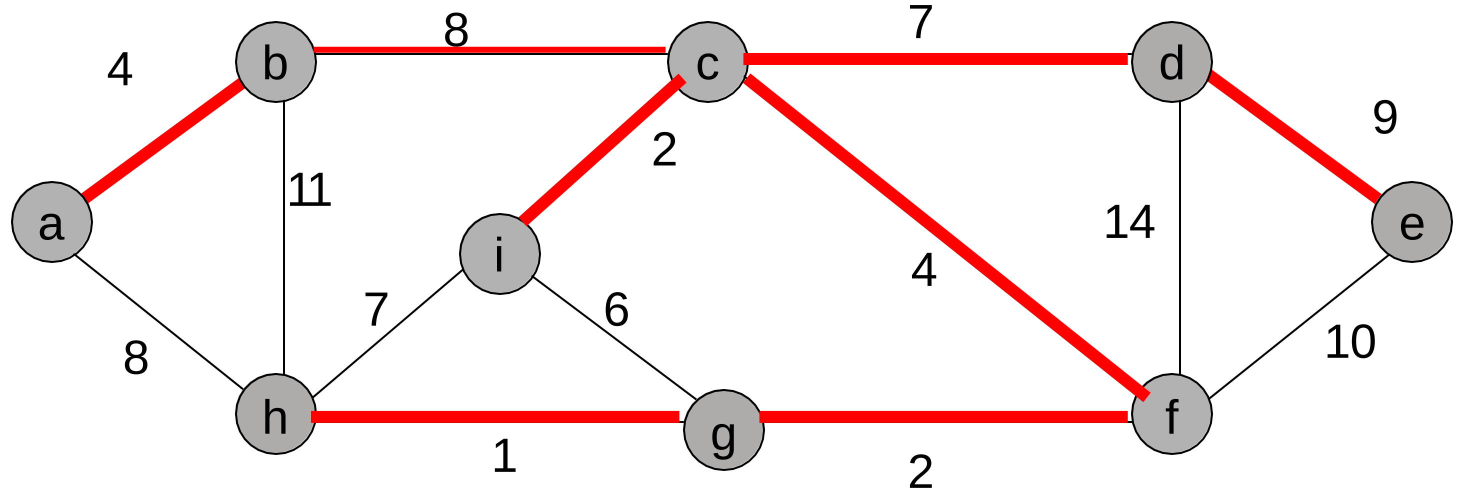


PRIM'S ALGORITHM - SOLUTION

g)



h)



PRIM’S ALGORITHM - ANALYSIS

- Maintains a min-priority queue by calling three priority queue operations:
  - INSERT
  - EXTRACT-MIN
  - DECREASE-KEY

- Running time of Prim’s Algorithm depends on how the min-priority queue is implemented.

PRIM'S ALGORITHM - ANALYSIS

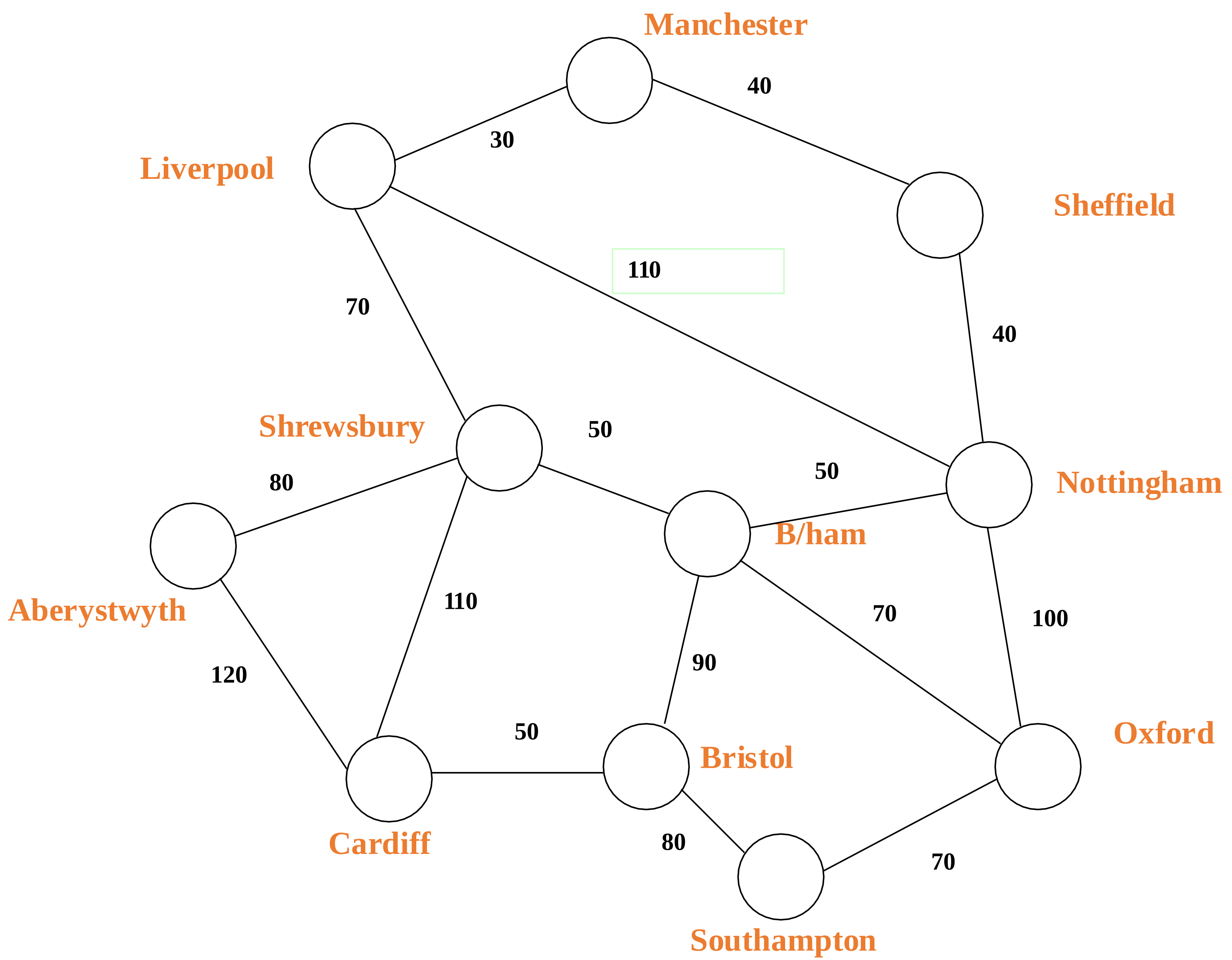
- Binary min-heap

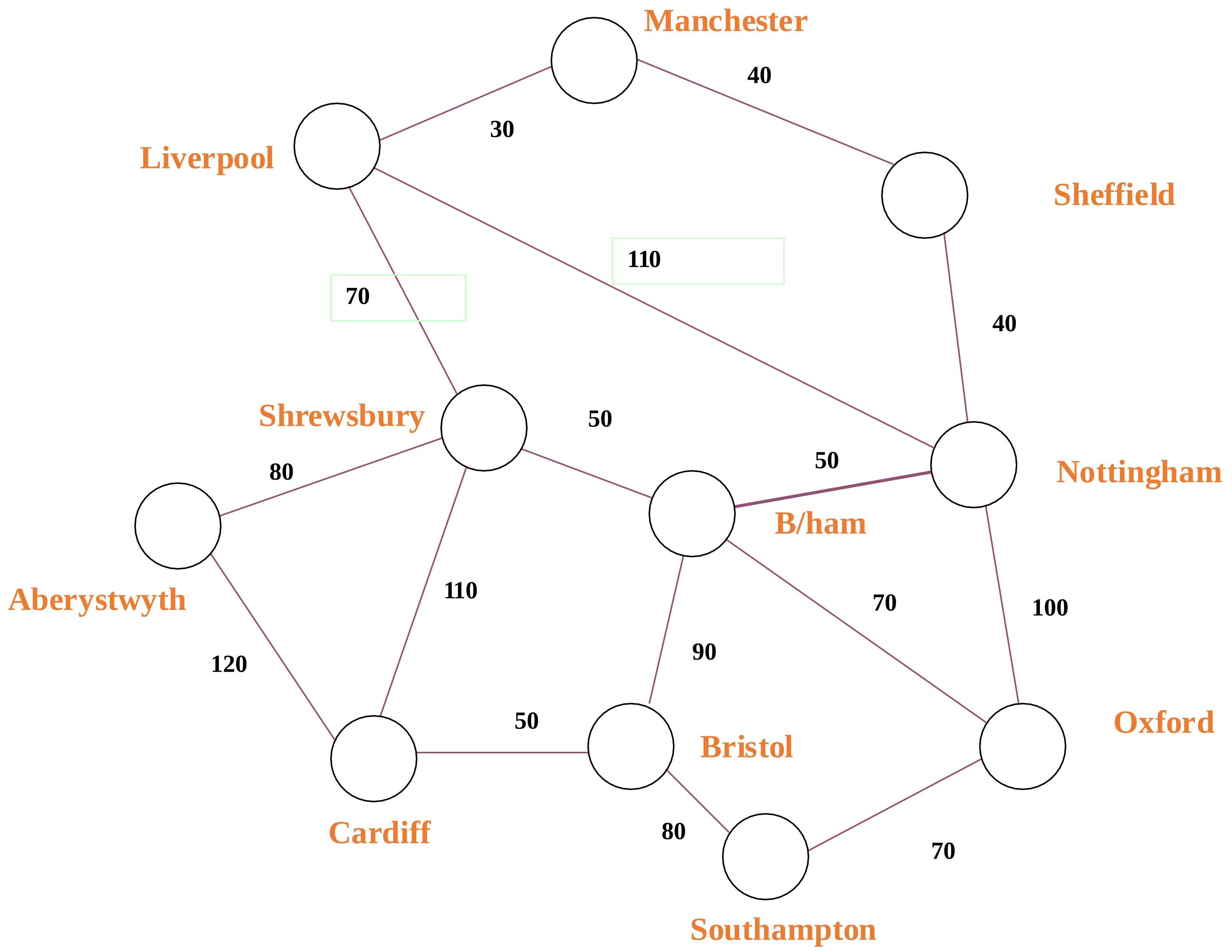
Building min binary heap	
	O(V)
DECREASE-KEY	O(lg V) - E
EXTRACT-MIN	O(lg V) - V
Total time	O ((V + E) lg V)
	= O (E lg V)

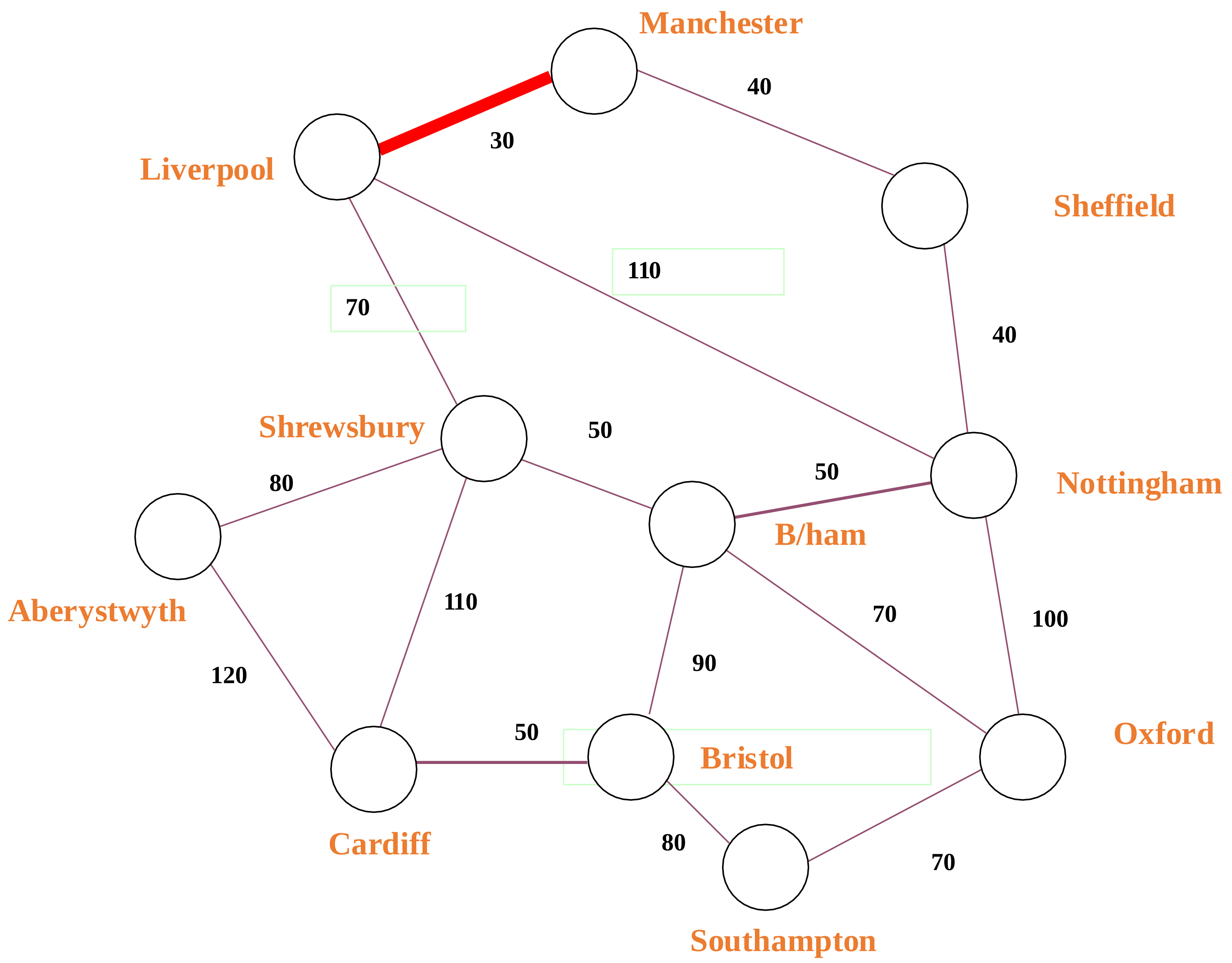
PRIM'S ALGORITHM - ANALYSIS

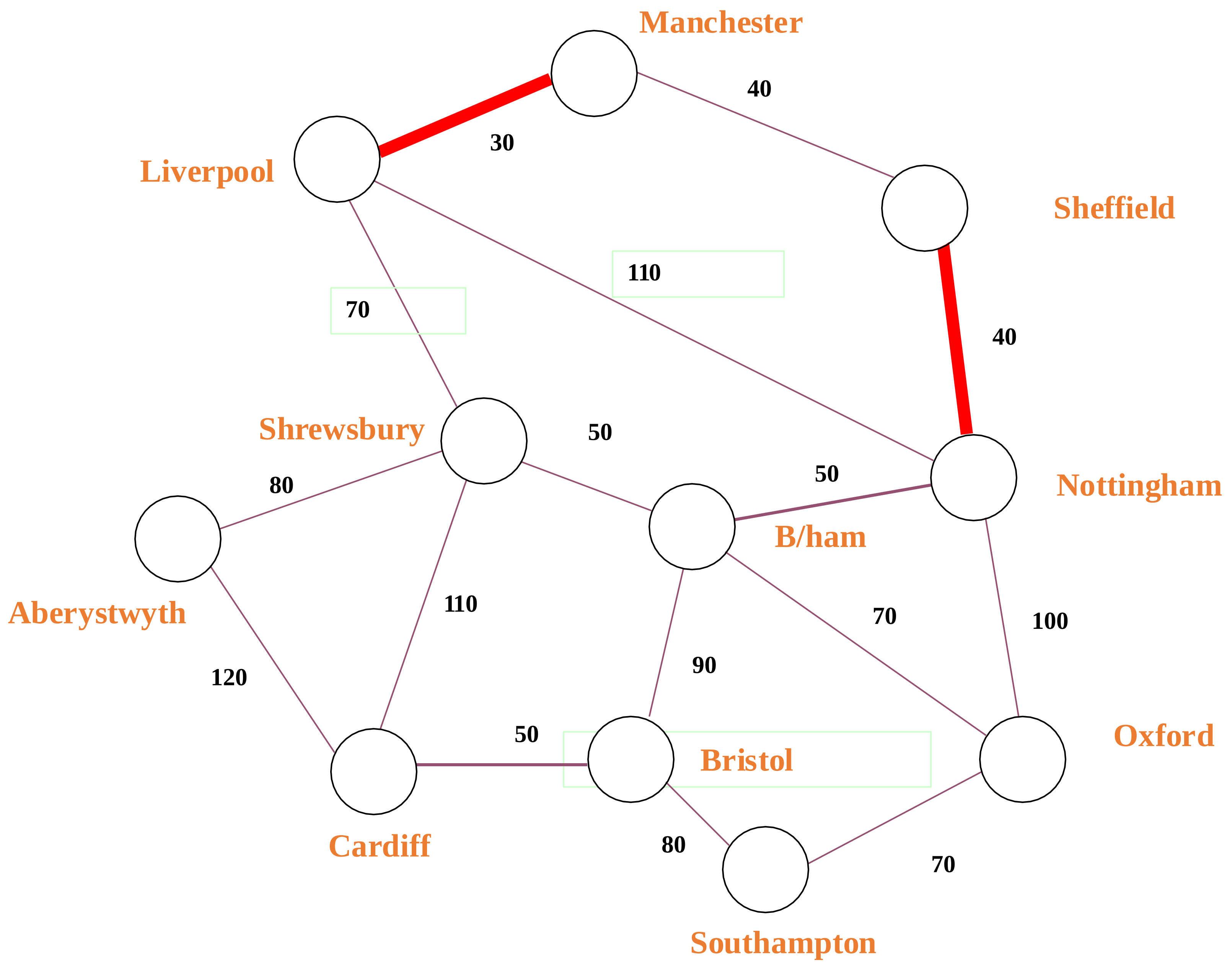
- Fibonacci heap

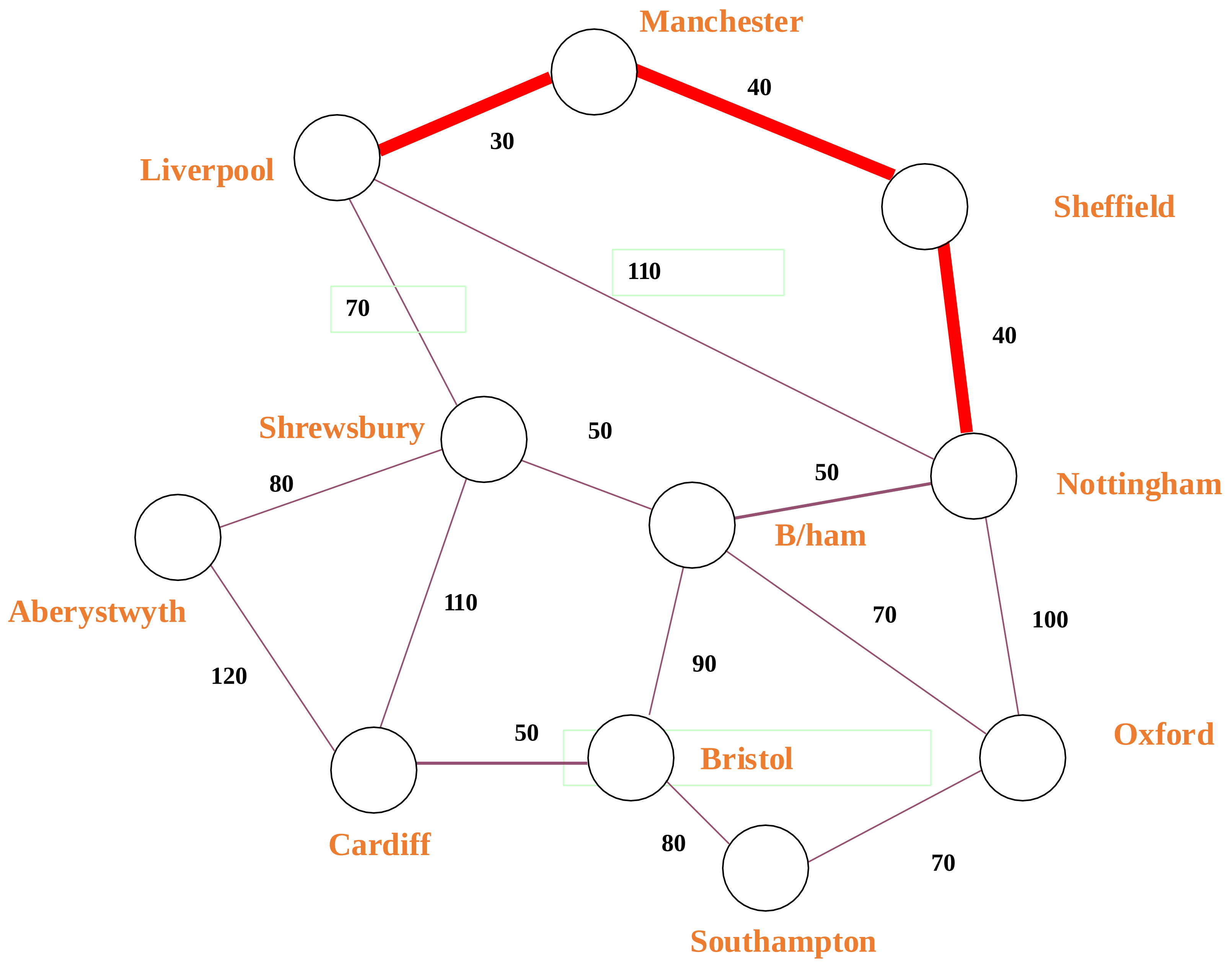
Building Fibonacci heap	O(V)
DECREASE-KEY	O(1) - E
EXTRACT-MIN	O(lg V) - V
Total time	O (V lg V + E)

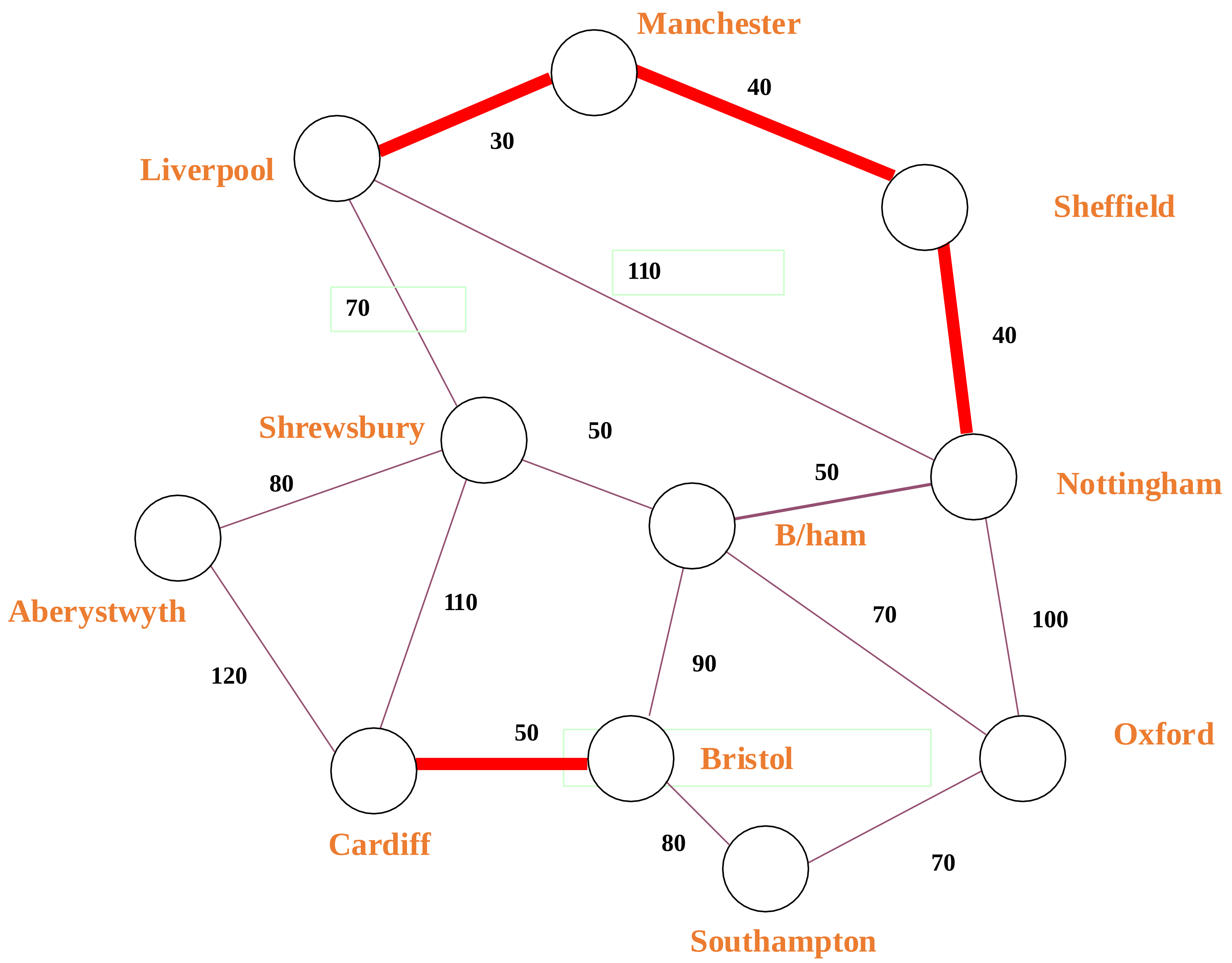

Manchester
40
30
Liverpool
Sheffield
110
70
40
Shrewsbury
50
50
80
Nottingham
B/ham
Aberystwyth
110
70
100
90
120
50
Oxford
Bristol
Cardiff
80
70
Southampton

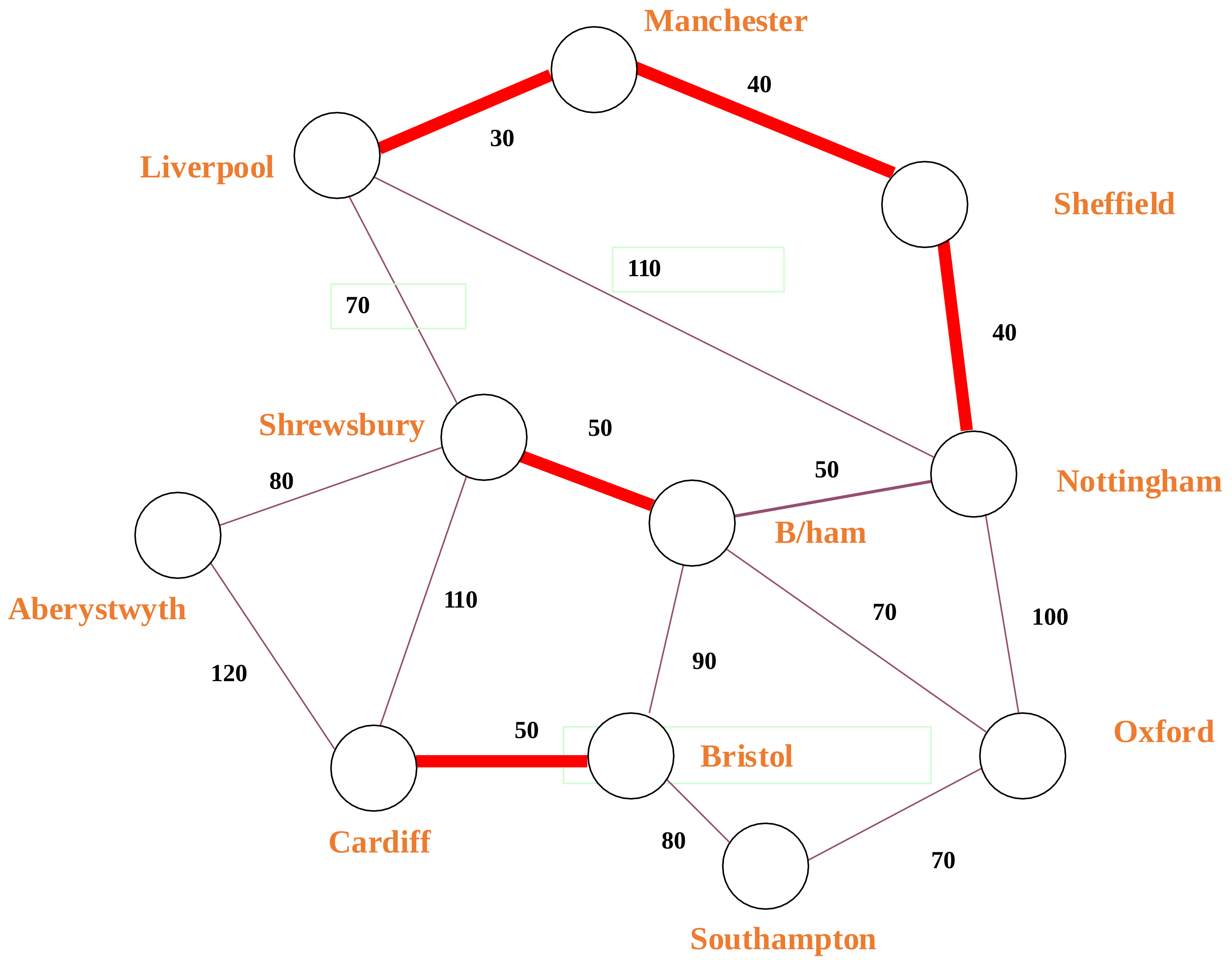

Manchester
40
30
Liverpool
Sheffield
110
70
40
Shrewsbury
50
50
Nottingham
80
B/ham
Aberystwyth
110
70
100
90
120
50
Oxford
Bristol
Cardiff
80
70
Southampton

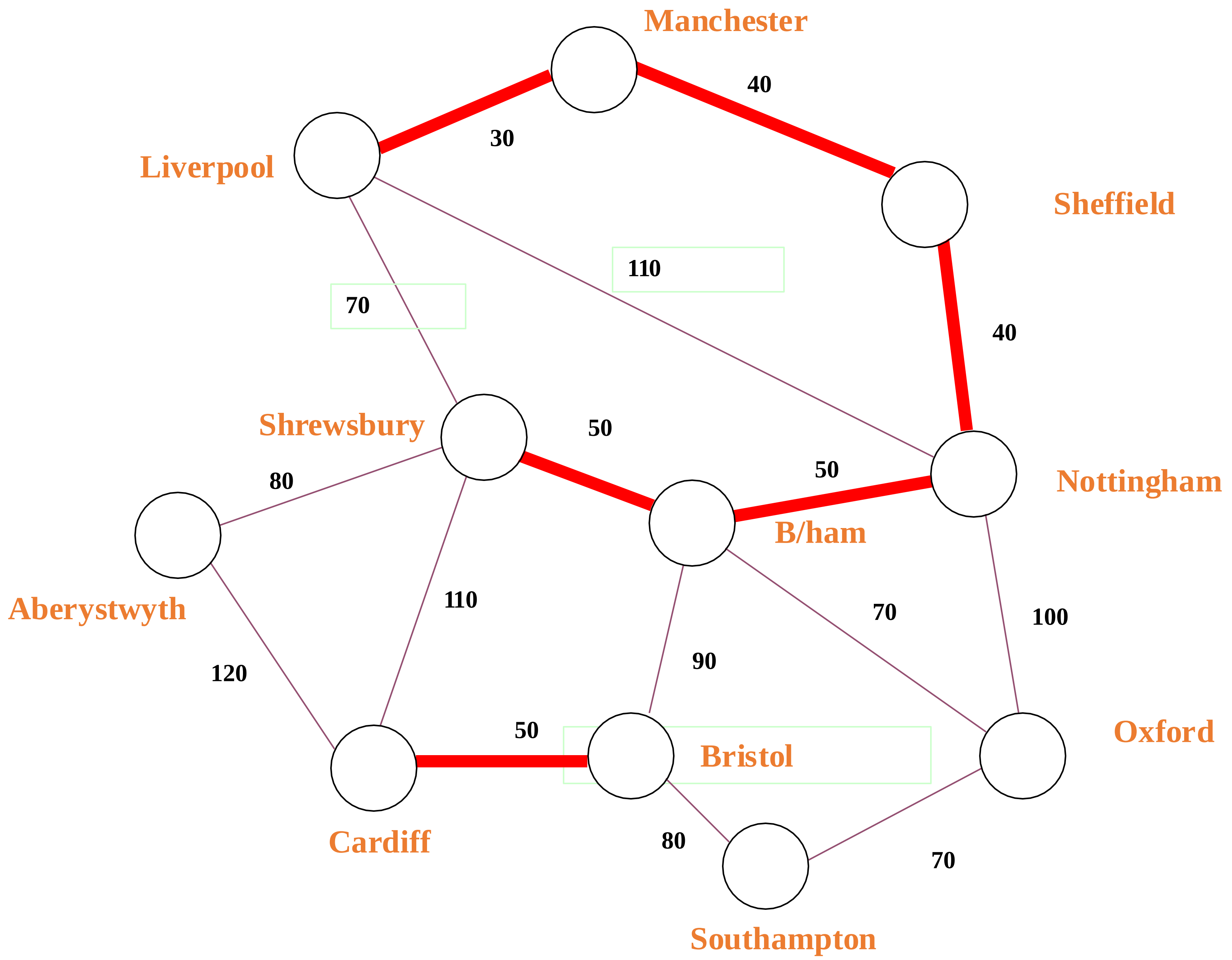

Manchester
40
30
Liverpool
Sheffield
110
70
40
Shrewsbury
50
50
Nottingham
80
B/ham
Aberystwyth
110
70
100
90
120
50
Oxford
Bristol
Cardiff
80
70
Southampton

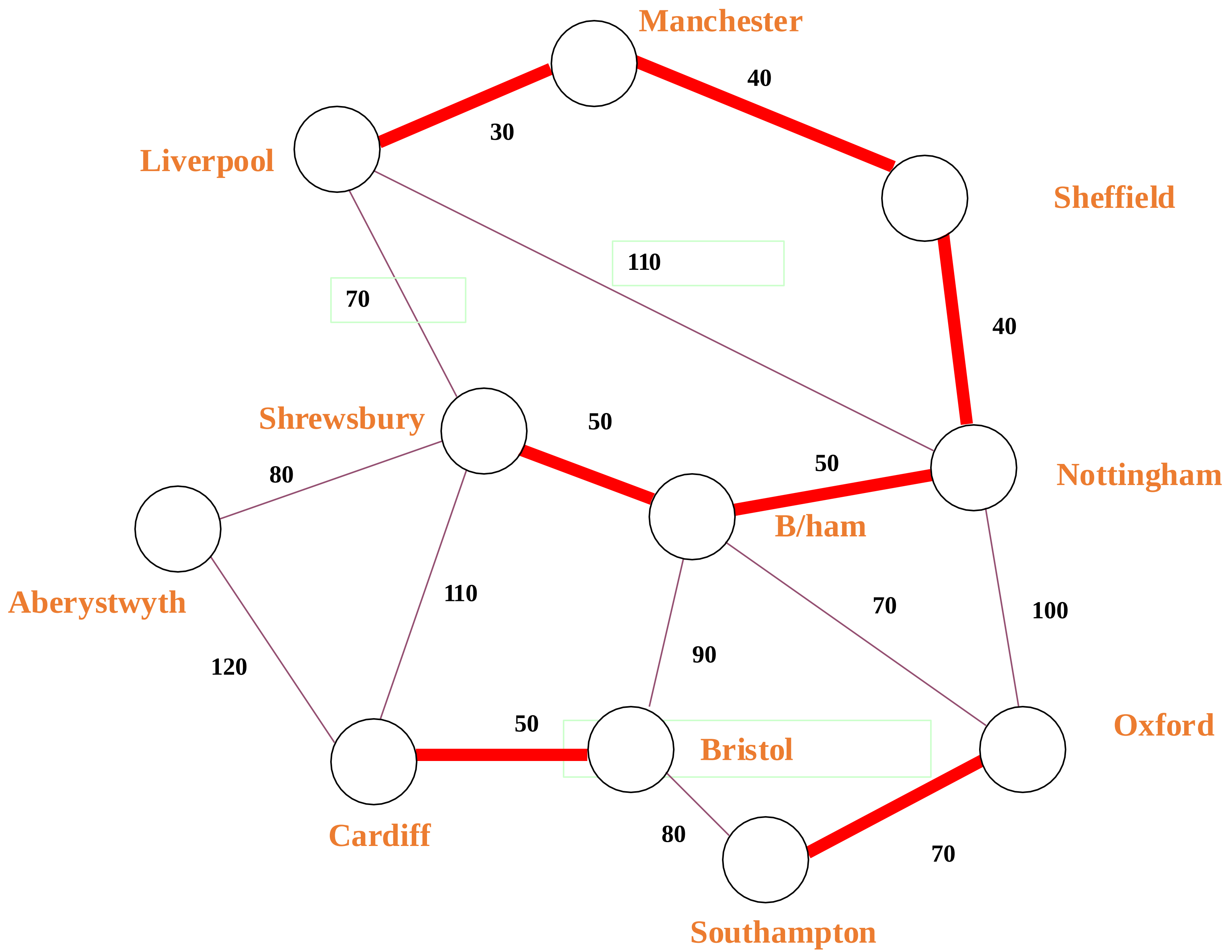

Manchester
40
Liverpool
30
Sheffield
110
70
40
Shrewsbury
50
50
Nottingham
80
B/ham
Aberystwyth
110
70
100
90
120
Oxford
50
Bristol
Cardiff
80
70
Southampton

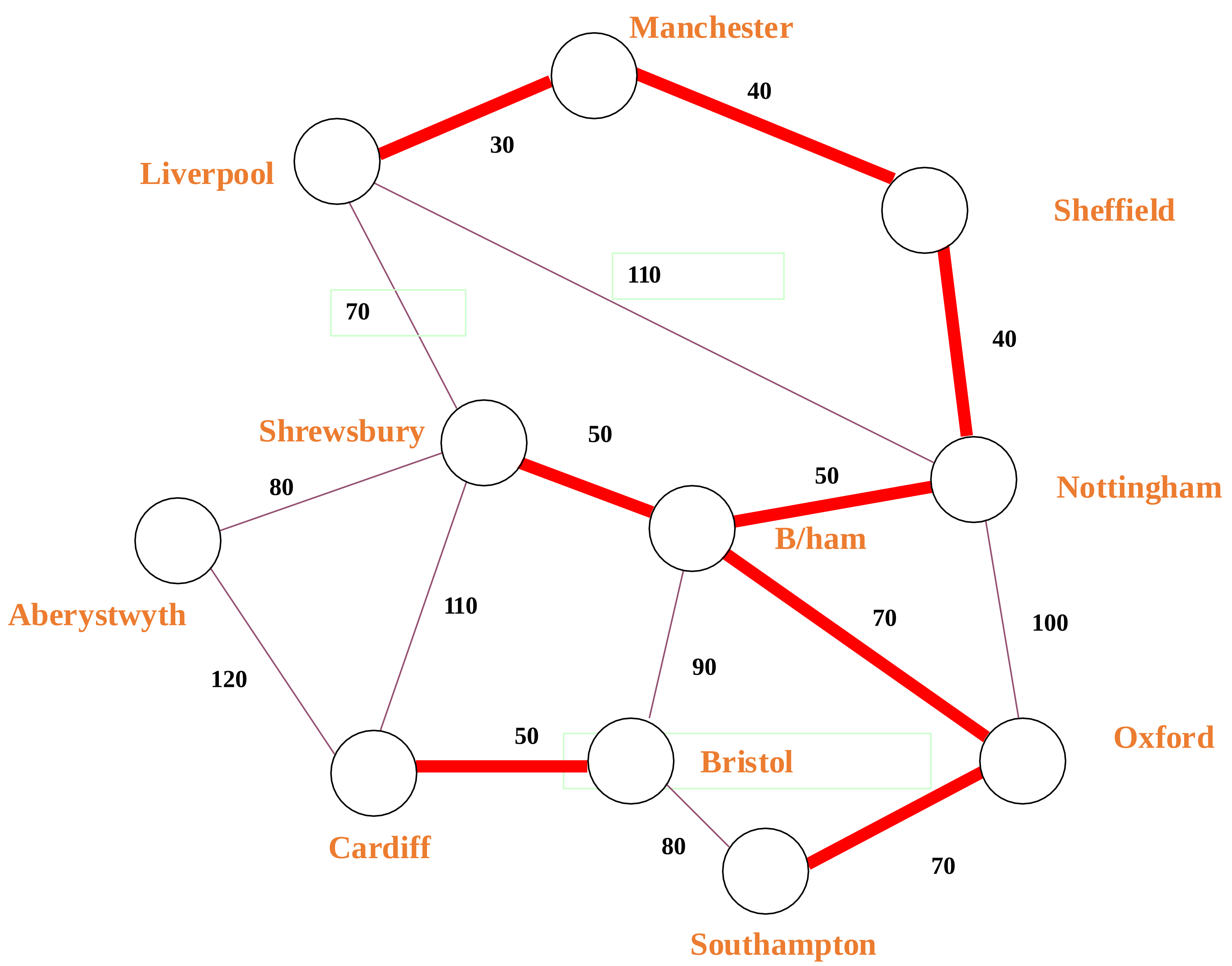

Manchester
40
30
Liverpool
Sheffield
110
70
40
Shrewsbury
50
50
80
Nottingham
B/ham
Aberystwyth
110
70
100
90
120
Oxford
50
Bristol
Cardiff
80
70
Southampton

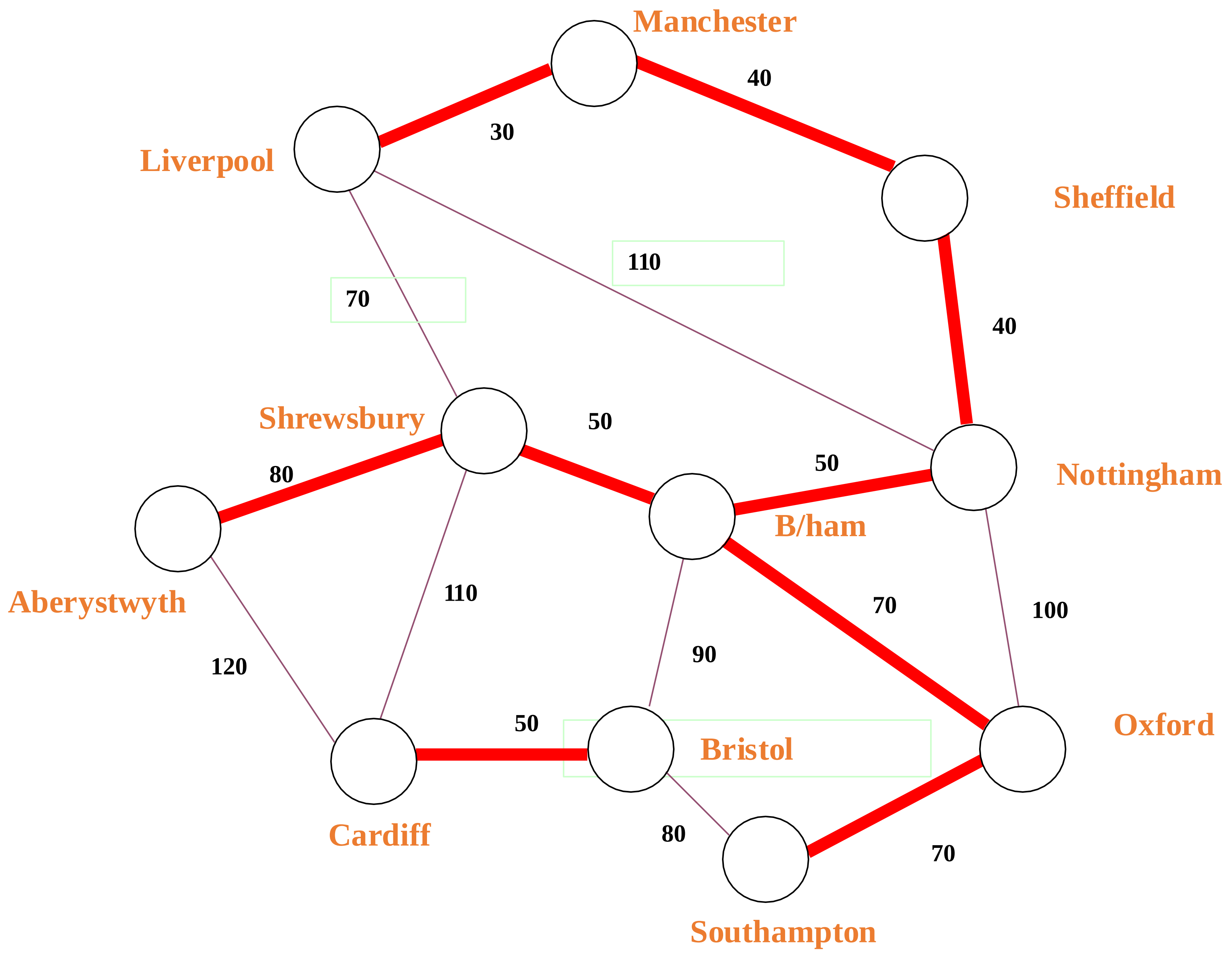

Manchester
40
30
Liverpool
Sheffield
110
70
40
Shrewsbury
50
50
Nottingham
80
B/ham
Aberystwyth
110
70
100
90
120
50
Bristol
Oxford
Cardiff
80
70
Southampton

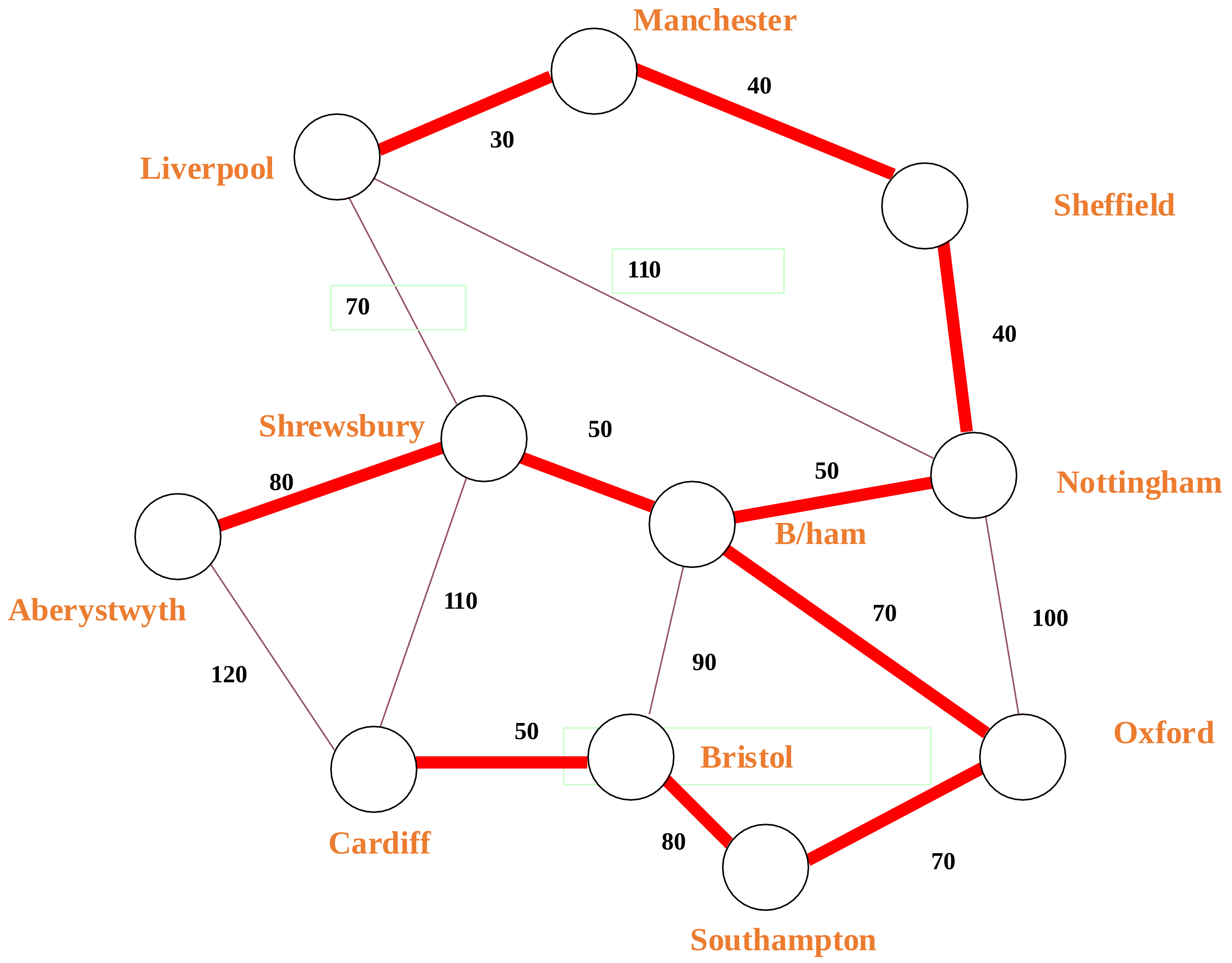

Manchester
Liverpool
Sheffield
Shrewsbury
Nottingham
B/ham
Aberystwyth
Oxford
Bristol
Cardiff
Southampton
30
40
110
70
40
50
50
80
110
70
100
90
120
50
80
70


Manchester
40
30
Liverpool
Sheffield
110
70
40
Shrewsbury
50
50
Nottingham
80
B/ham
Aberystwyth
110
70
100
90
120
Oxford
50
Bristol
Cardiff
80
70
Southampton

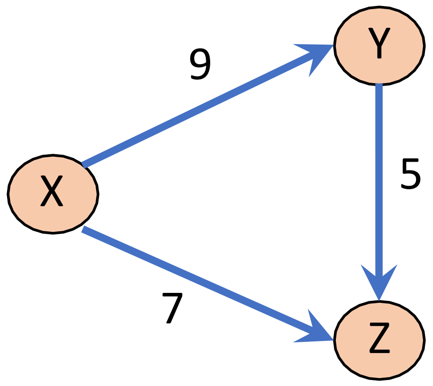

Manchester
40
30
Liverpool
Sheffield
110
70
40
Shrewsbury
50
50
80
Nottingham
B/ham
Aberystwyth
110
70
100
90
120
Oxford
50
Bristol
Cardiff
80
70
Southampton


Manchester
Liverpool
Sheffield
Shrewsbury
Nottingham
B/ham
Aberystwyth
Oxford
Bristol
Cardiff
Southampton
30
40
110
70
40
50
50
80
110
70
100
90
120
50
80
70


Manchester
40
30
Liverpool
Sheffield
110
70
40
Shrewsbury
50
50
Nottingham
80
B/ham
Aberystwyth
110
70
100
90
120
Oxford
50
Bristol
Cardiff
80
70
Southampton


Manchester
Liverpool
Sheffield
Shrewsbury
Nottingham
B/ham
Aberystwyth
Oxford
Bristol
Cardiff
Southampton
30
40
110
70
40
50
50
80
110
70
100
90
120
50
80
70

USING MST ALGORITHM FOR DIRECTED GRAPH



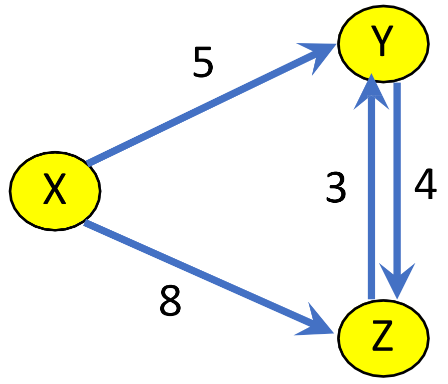
The graph has two spanning trees

i) X →Y→Z, with weight 9+5 and

ii) X→Z and X→Y with weight 9+7.

- Starting with node X Prim's algorithm will consider the two edges (X,Z) and (X,Y) and choose the lightest one which is (X,Z).
- Then the spanning tree considered will be (ii) and it is not the minimum spanning tree.

USING MST ALGORITHM FOR DIRECTED GRAPH



The graph has the following three spanning trees,

i) X →Y → Z with weight 5 + 4,

ii) X → Z →Y with weight 8 + 3, and

iii) X → Z , X →Y with weight 8 + 5.

- Kruskal will choose the lightest edge Z →Y.
- This will select (ii) as the spanning tree, and it is not the best one.