DTS203TC
Design and Analysis of Algorithms

Lecture 14: Minimum Spanning Tree and Shortest Path

Dr. Qi Chen
School of AI and Advanced Computing

Learning Outcome

- Understand what Minimum Spanning Tree is
  - Able to apply Kruskal's algorithm to find minimum spanning tree
  - Able to apply Prim's algorithm to find minimum spanning tree

- Understand what Single-source shortest path is
  - Able to apply Dijkstra algorithm to solve shortest path problem
  - Able to apply Bellman-Ford algorithm to solve shortest path problem

Minimum Spanning tree (MST)

Given an undirected connected graph G

- The edges are labelled by weight

Spanning tree of G

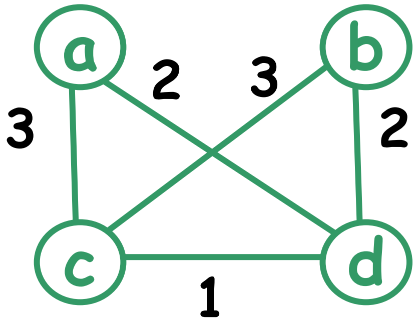
- a tree containing all vertices in G

Minimum spanning tree of G

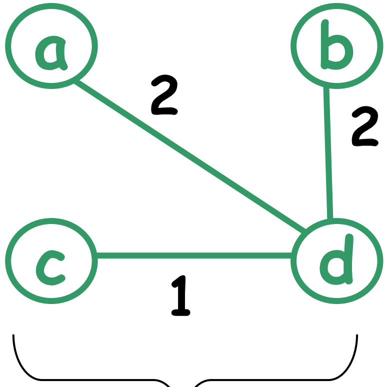
- a spanning tree of G with minimum weight

Examples

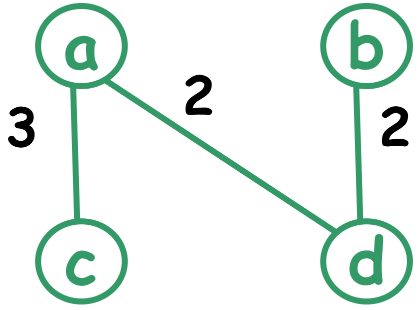
Graph G
(edge label is weight)

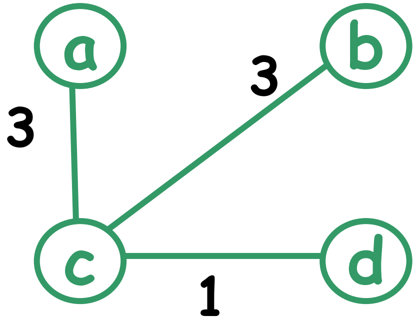


Spanning trees of G



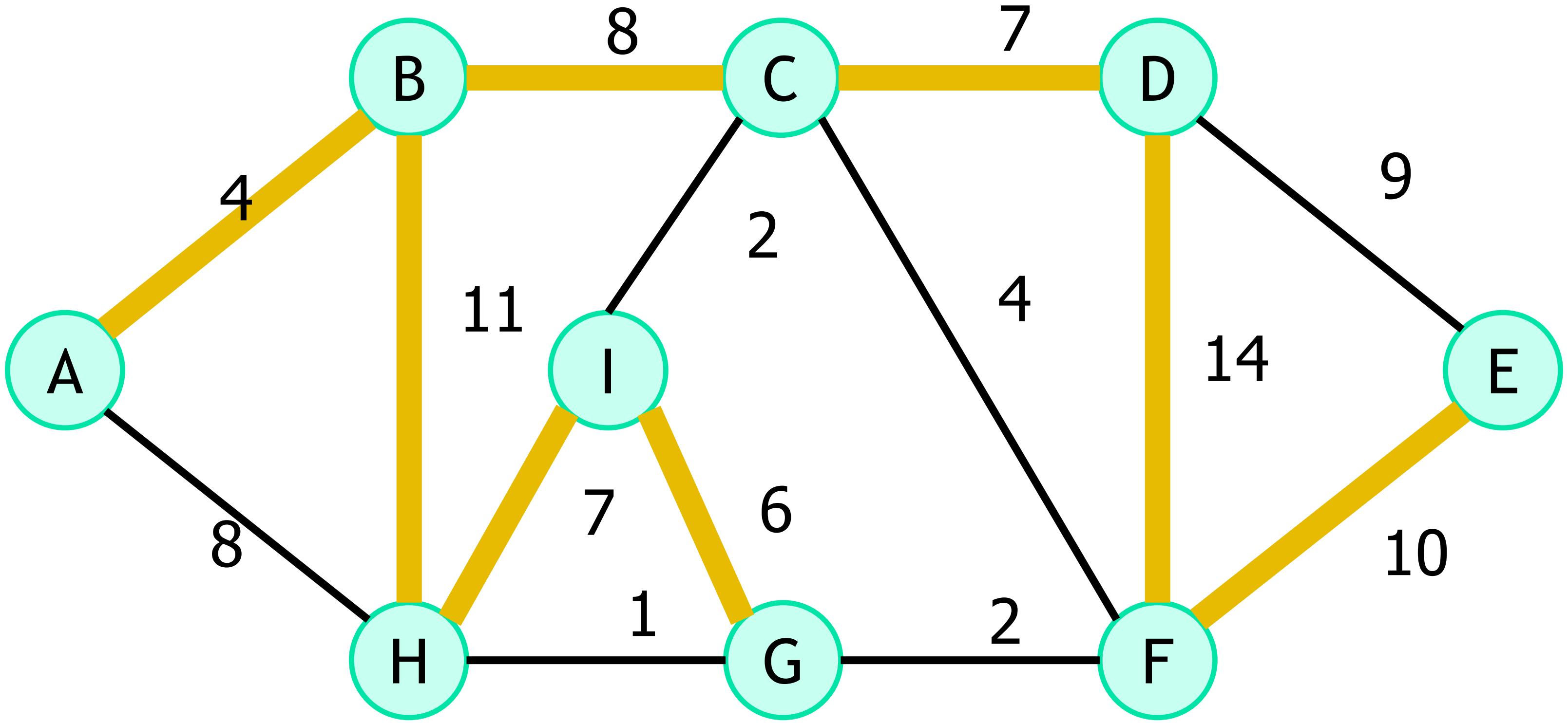
MST





Spanning Tree

The **cost** of a spanning tree is the sum of the weights on the edges.



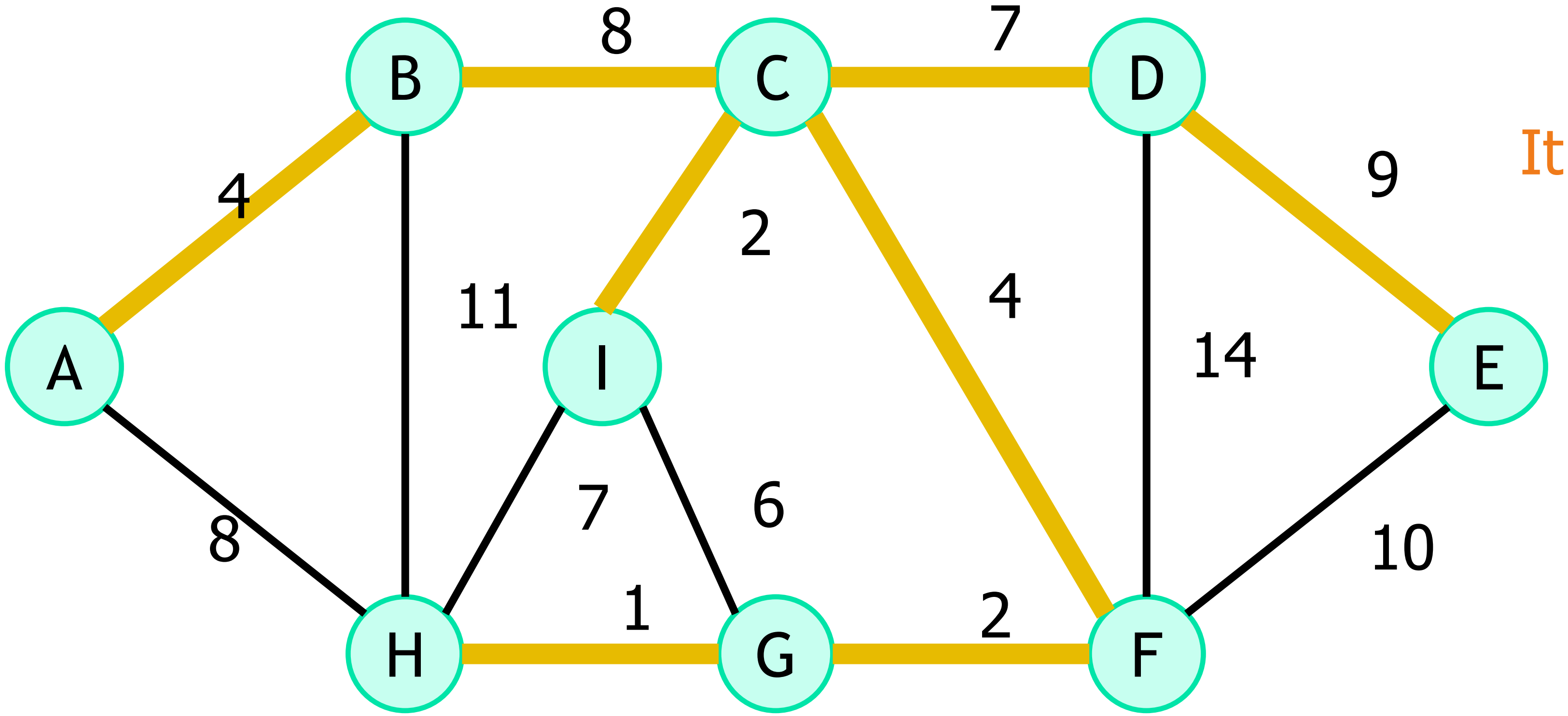
This is a spanning tree.

It has cost 67

A **tree** is a connected undirected graph with no cycles!

A **spanning tree** is a **tree** that connects all of the vertices.

Minimum Spanning Tree



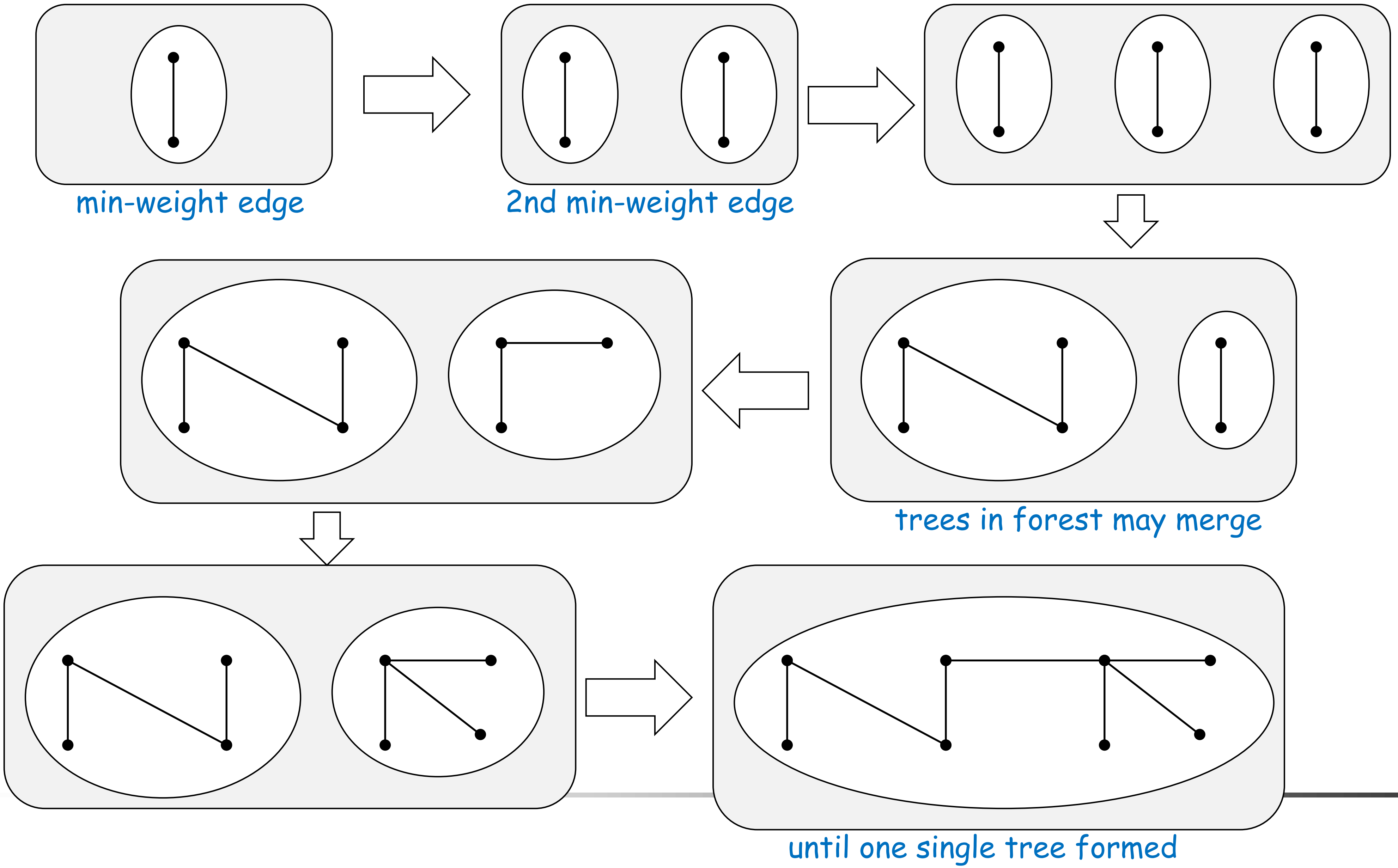
This is a minimum spanning tree.

It has cost 37

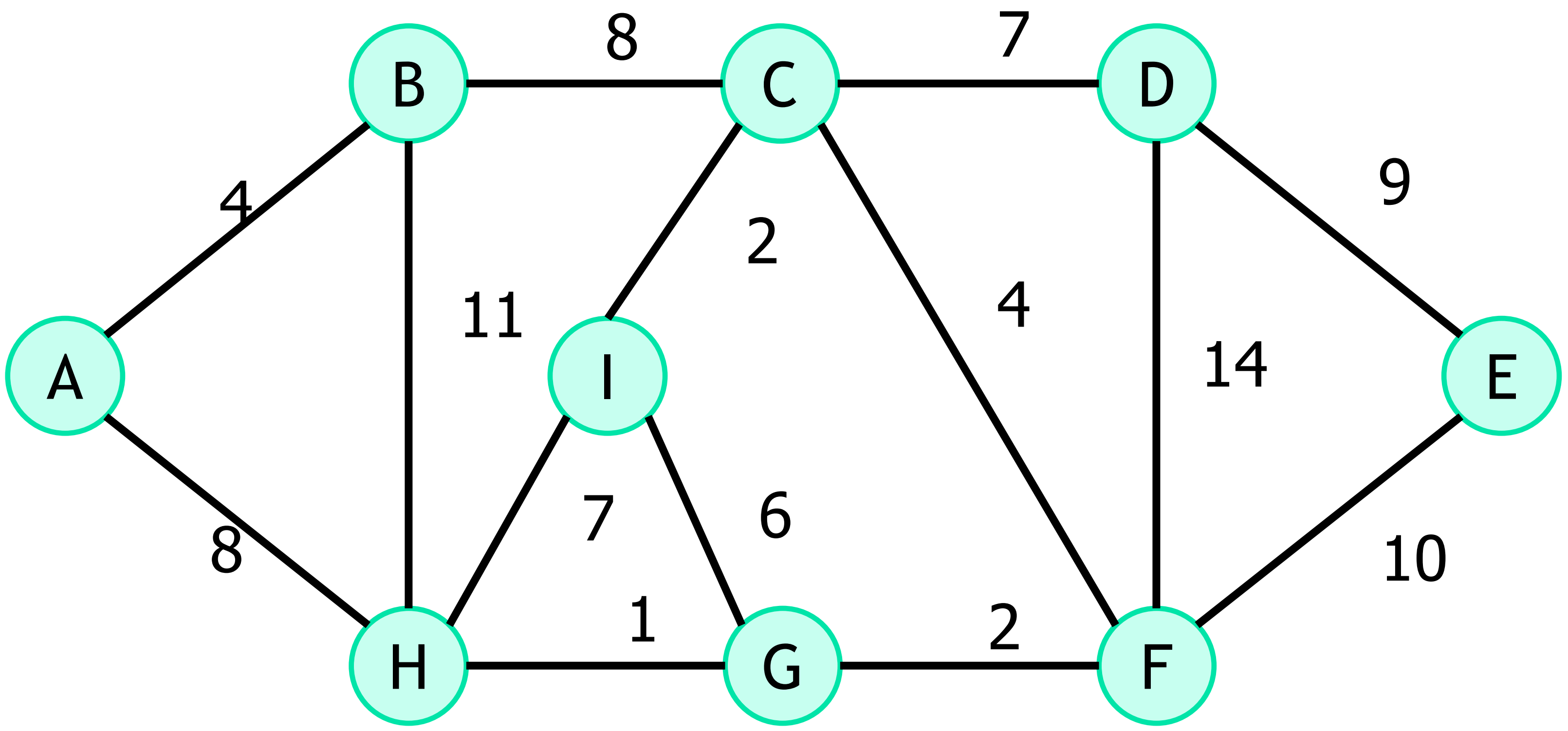
A **minimum spanning tree** is a **tree of minimal cost** that connects all of the vertices.

Kruskal's algorithm

Idea of Kruskal's algorithm - MST



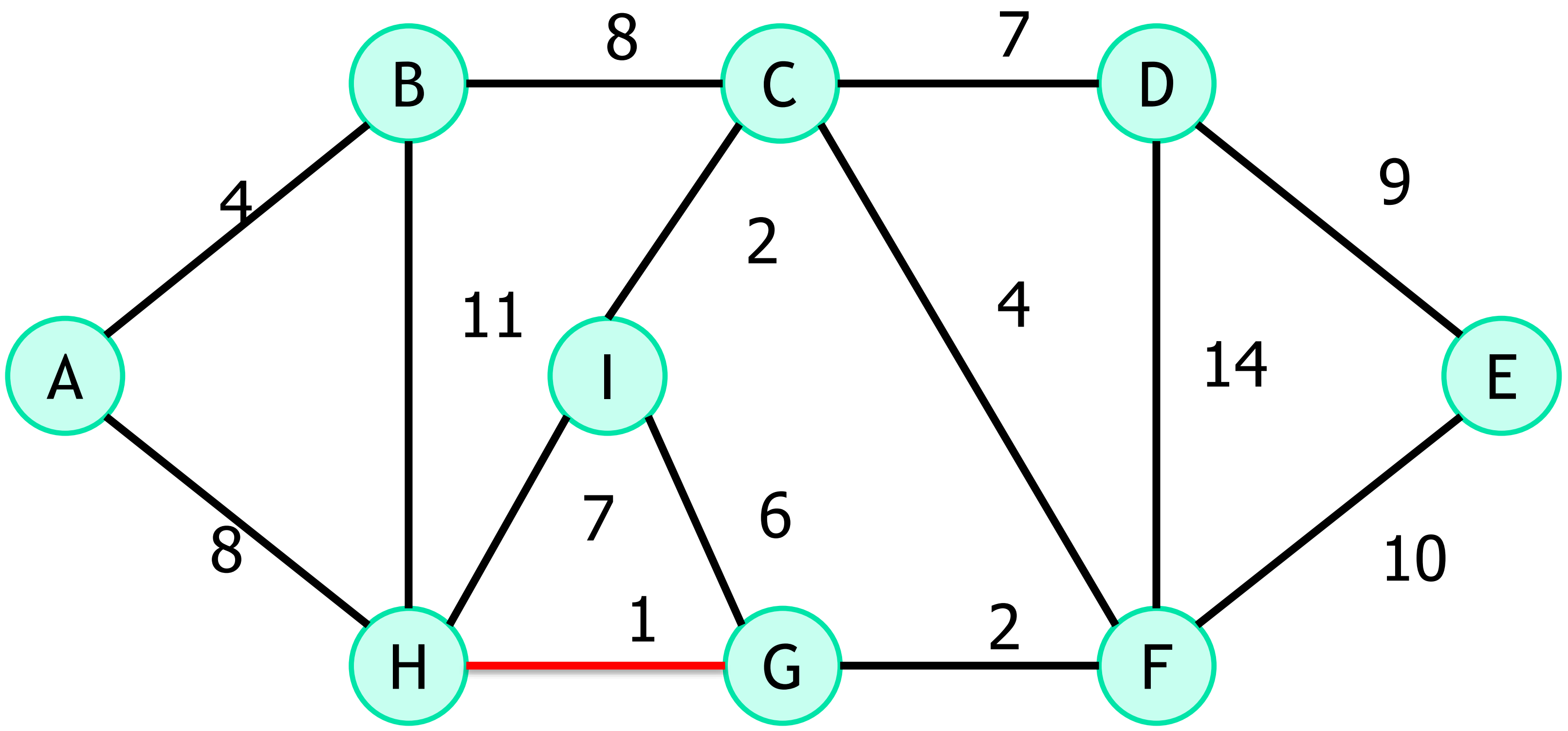
Kruskal's algorithm - MST



(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(i,g)	6
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

Arrange edges from smallest to largest weight

Kruskal's algorithm - MST

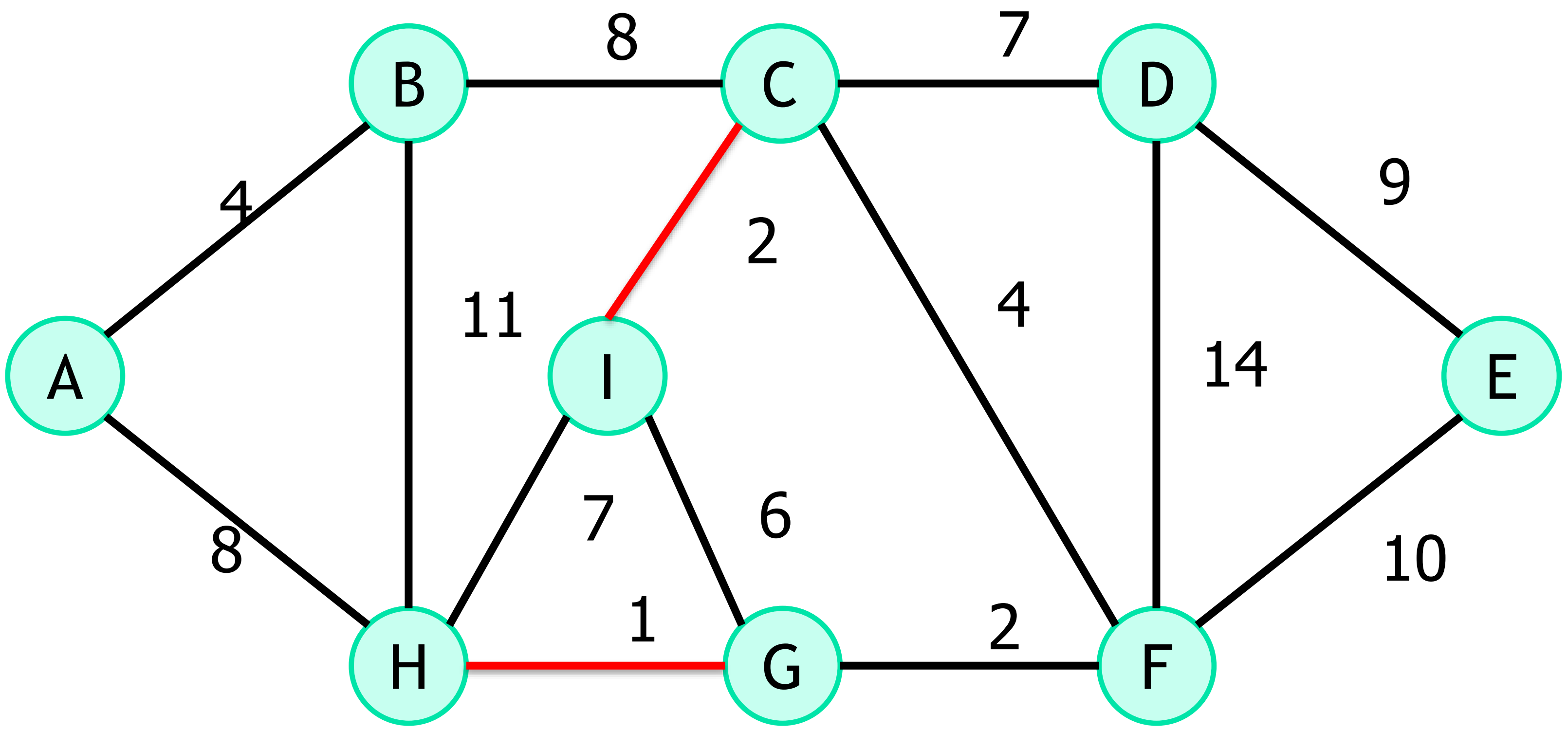


Choose the minimum weight edge

(h,g)	*1*
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(i,g)	6
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

italic: chosen

Kruskal's algorithm - MST

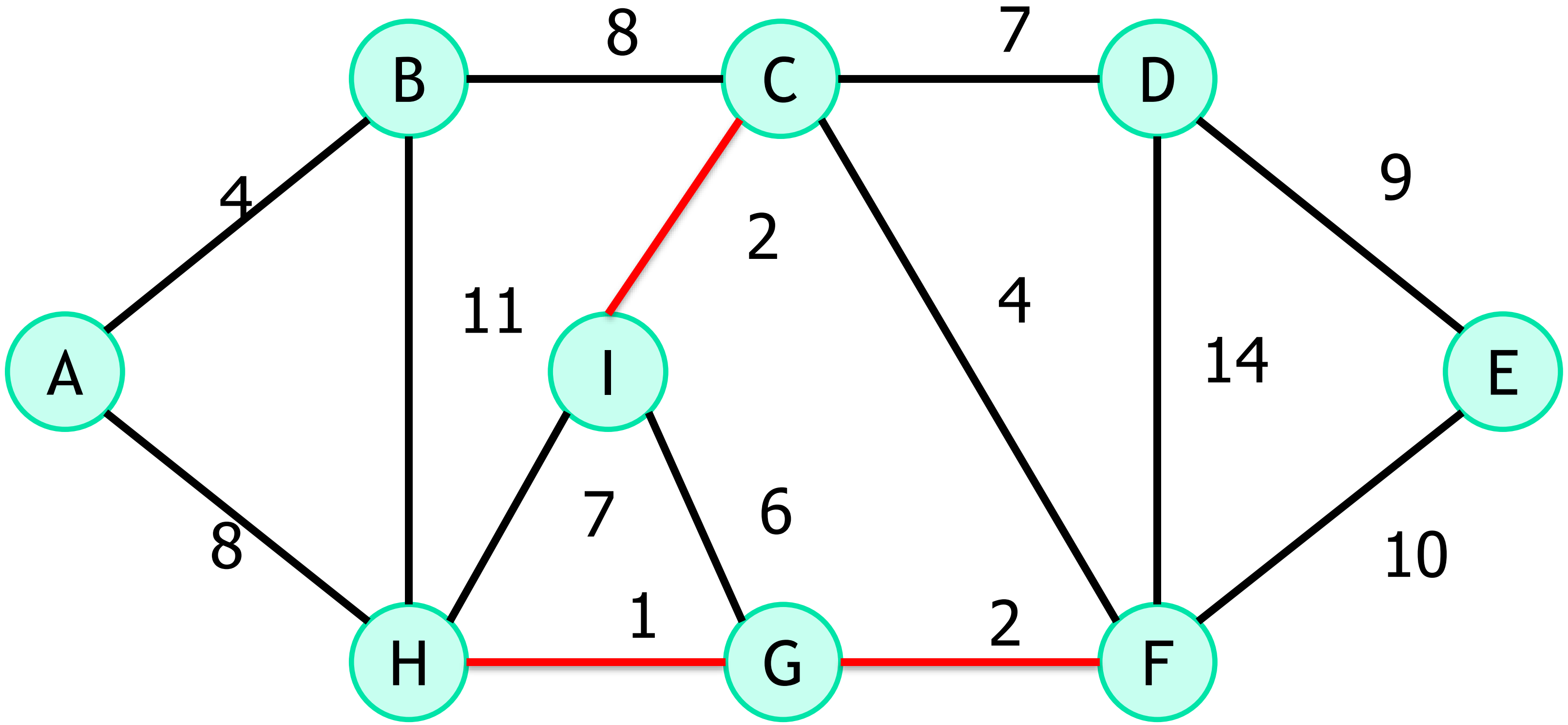


Choose the next minimum weight edge

(h,g)	*1*
(i,c)	*2*
(g,f)	2
(a,b)	4
(c,f)	4
(i,g)	6
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

italic: chosen

Kruskal's algorithm - MST

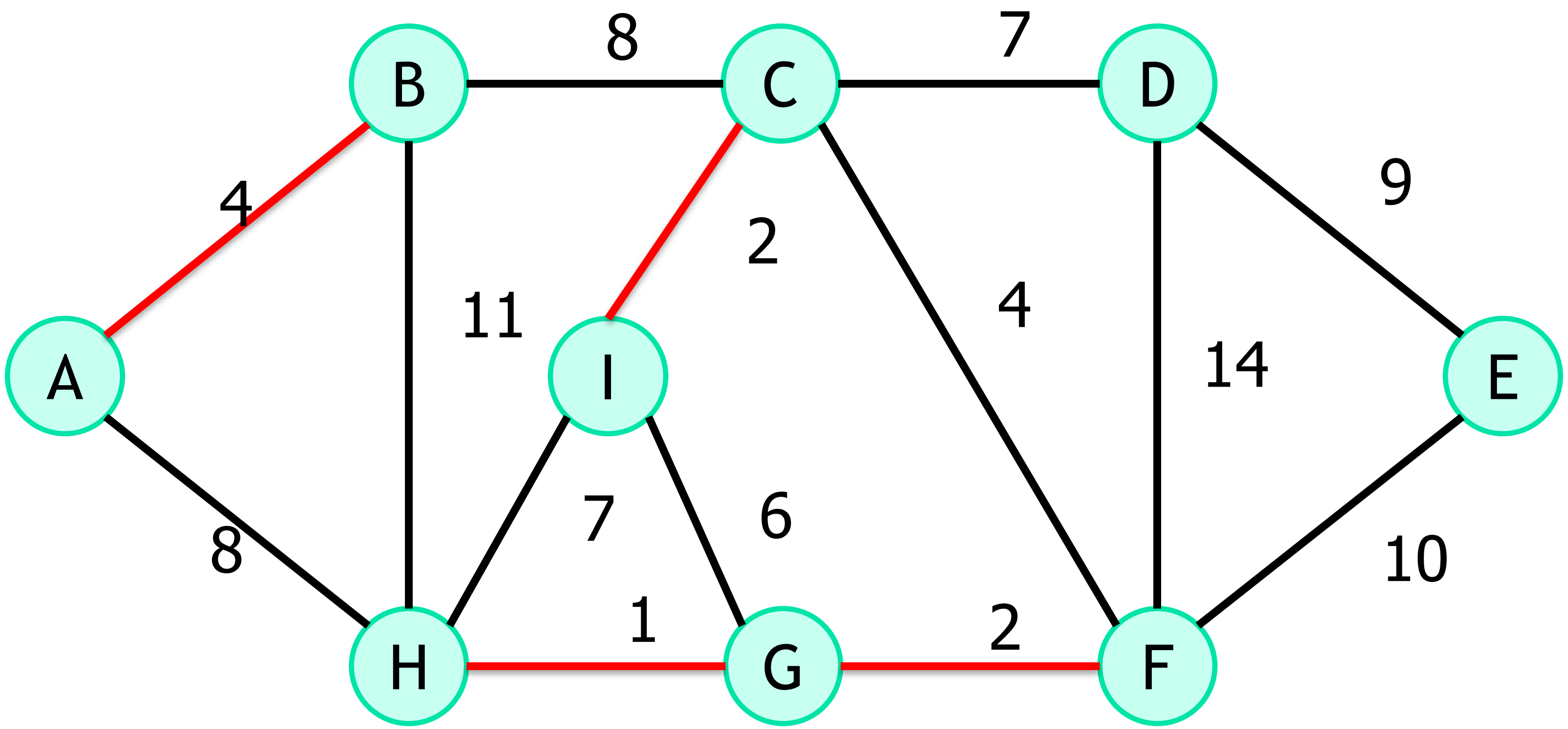


Continue as long as no cycle forms

(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	4
(c,f)	4
(i,g)	6
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

italic: chosen

Kruskal's algorithm - MST

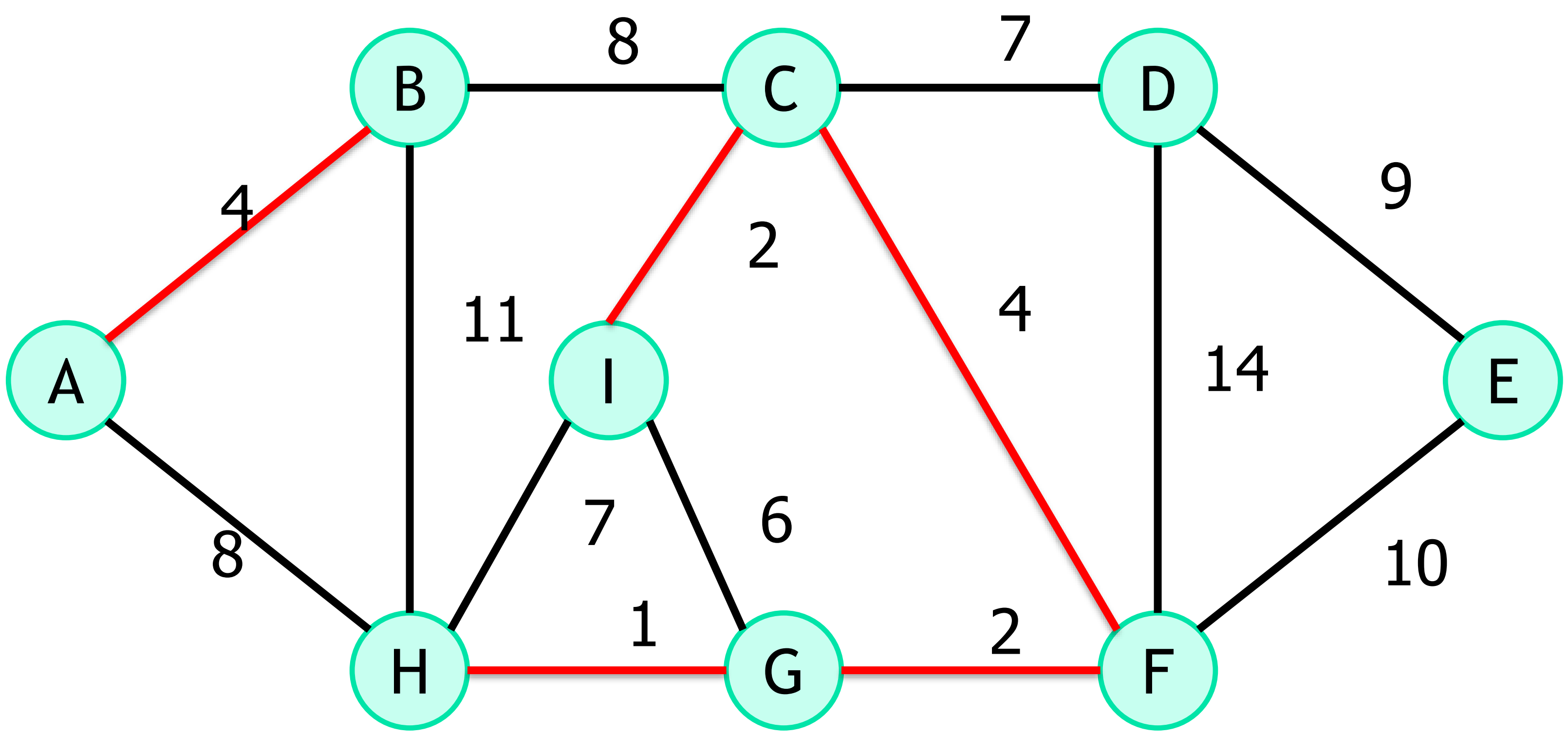


(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	4
(i,g)	6
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

Continue as long as no cycle forms

italic: chosen

Kruskal's algorithm - MST

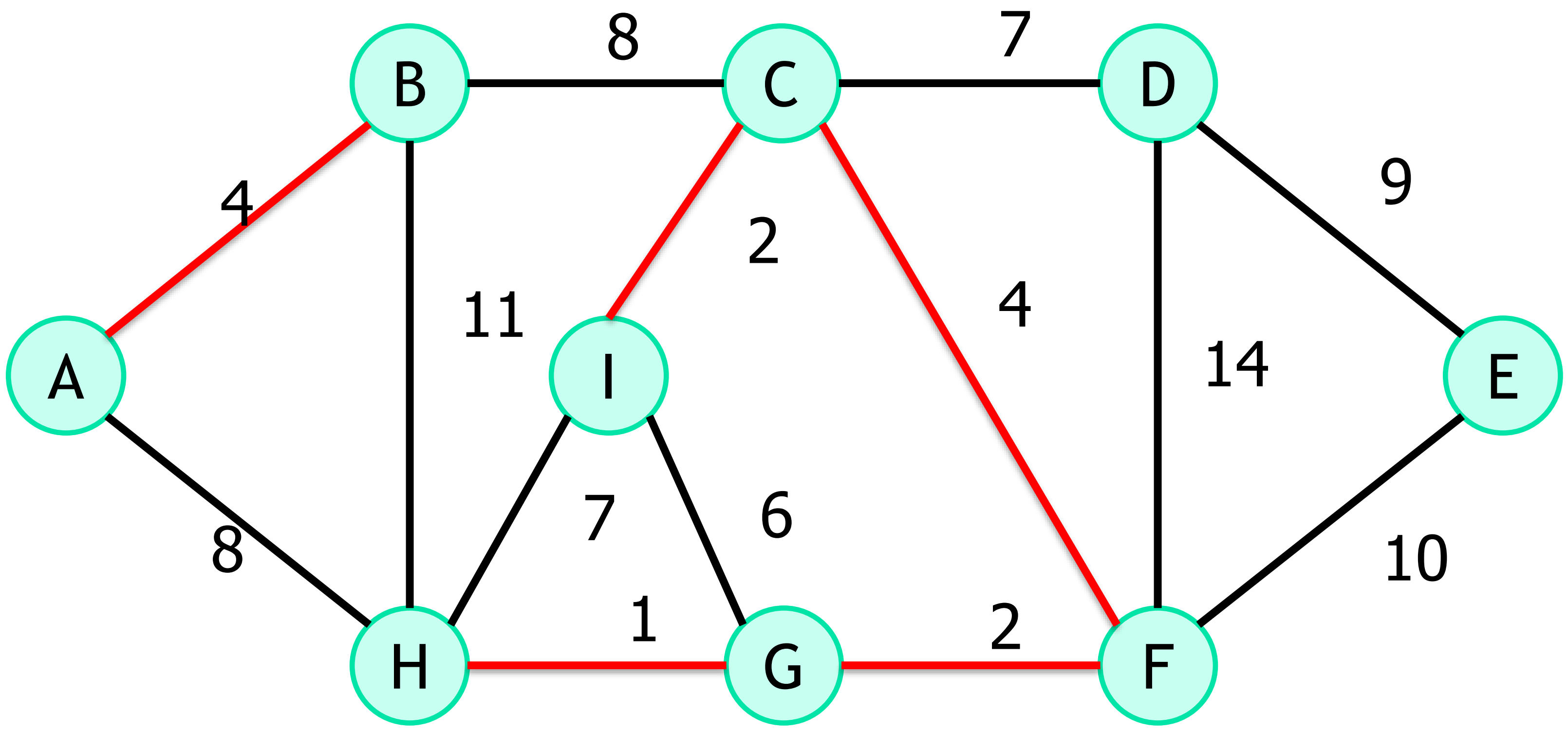


Continue as long as no cycle forms

(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	*4*
(i,g)	6
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

italic: chosen

Kruskal's algorithm - MST

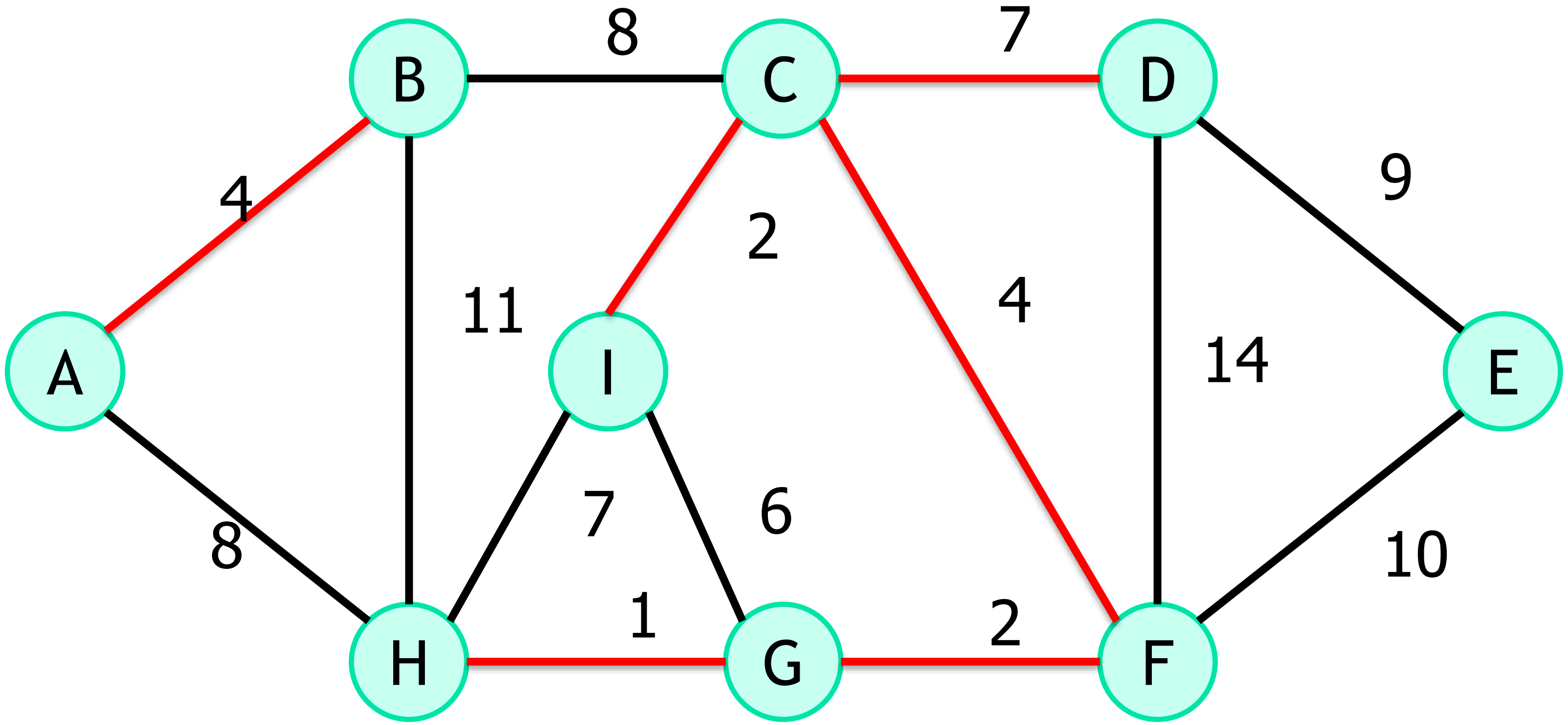


(i,g) cannot be included, otherwise, a cycle is formed

(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	*4*
~~(i,g)~~	~~6~~
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

italic: chosen

Kruskal’s algorithm - MST

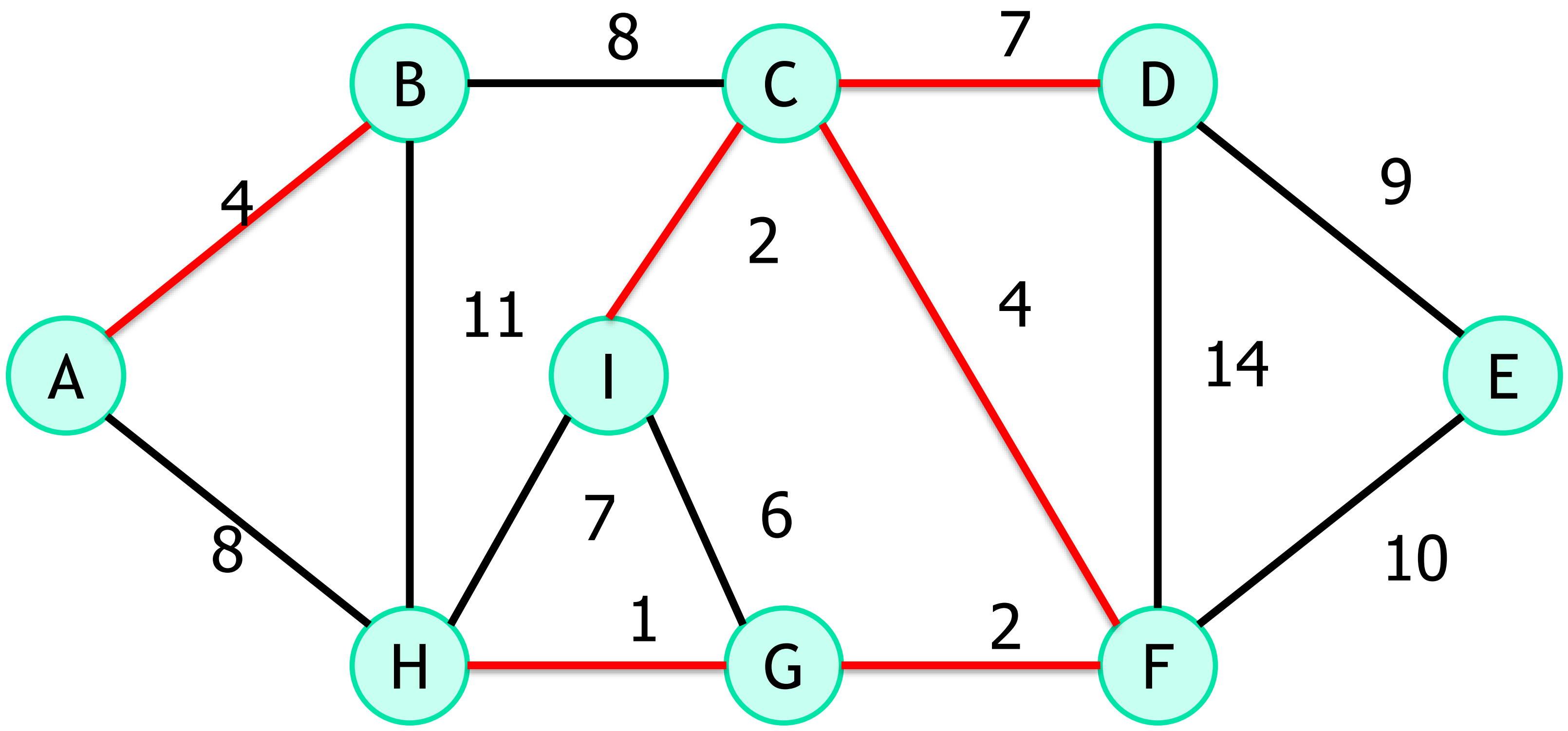


(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	*4*
~~(i,g)~~	~~6~~
(c,d)	*7*
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

italic: chosen

Continue as long as no cycle forms

Kruskal's algorithm - MST

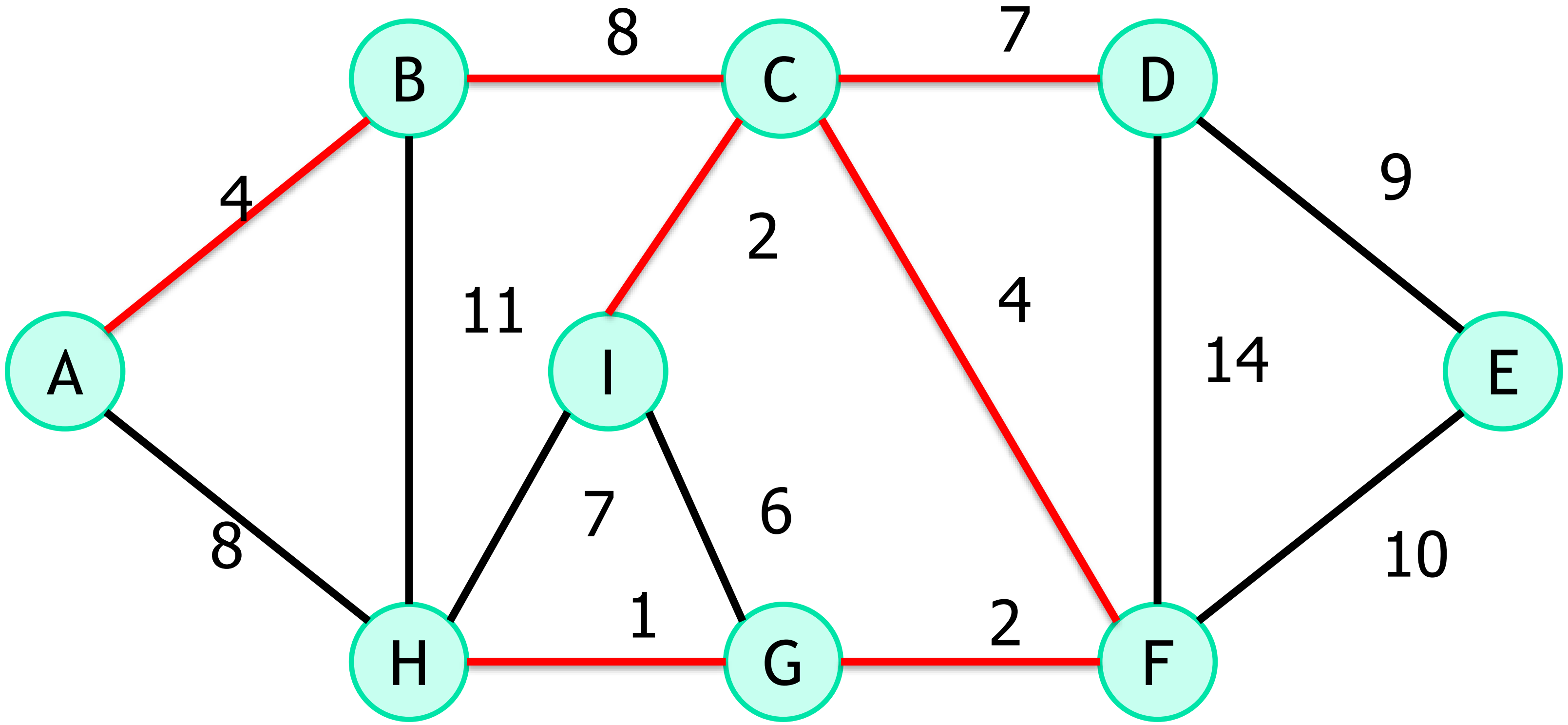


(h,i) cannot be included, otherwise, a cycle is formed

(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	*4*
~~(i,g)~~	~~6~~
(c,d)	*7*
~~(h,i)~~	~~7~~
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

italic: chosen

Kruskal’s algorithm - MST

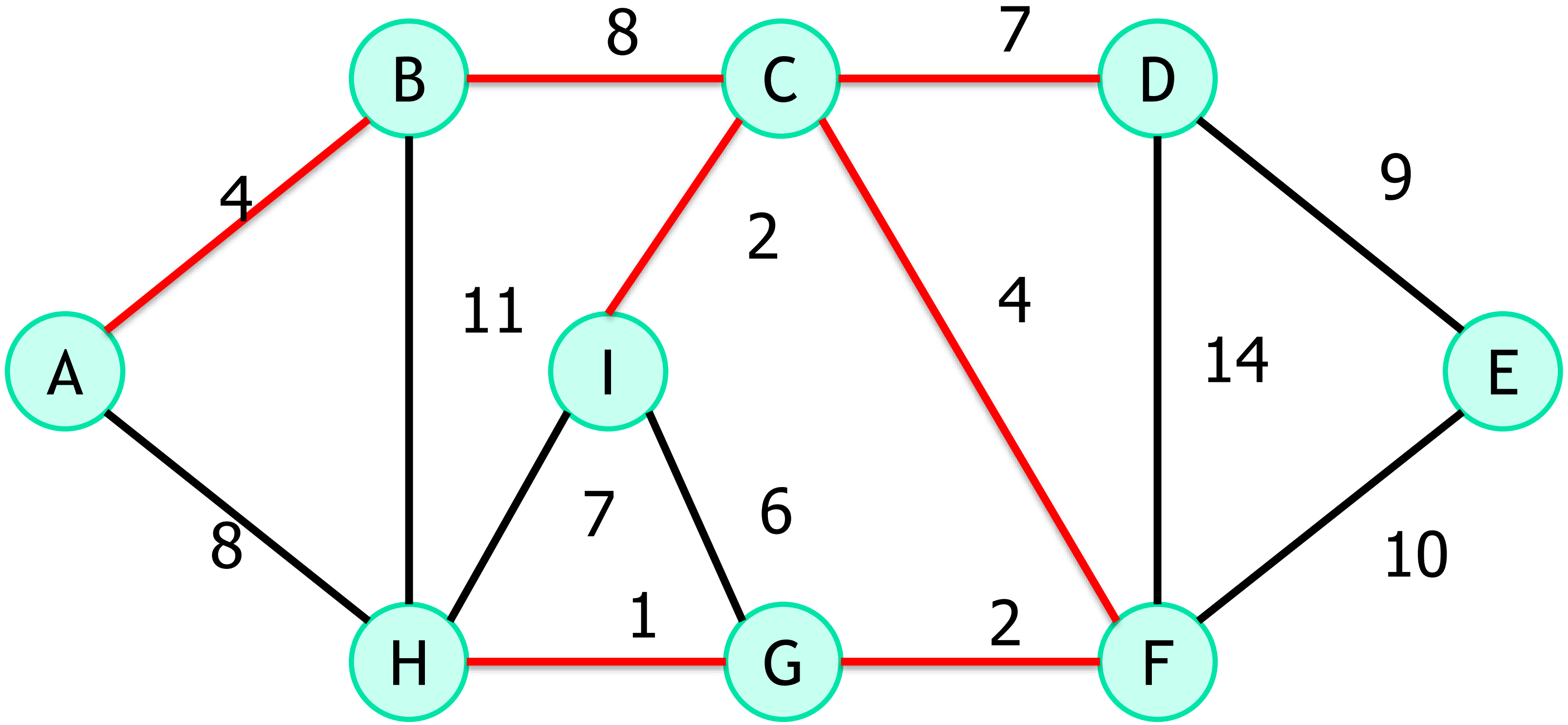


(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	*4*
~~(i,g)~~	~~6~~
(c,d)	*7*
~~(h,i)~~	~~7~~
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

italic: chosen

Choose the next minimum weight edge

Kruskal's algorithm - MST

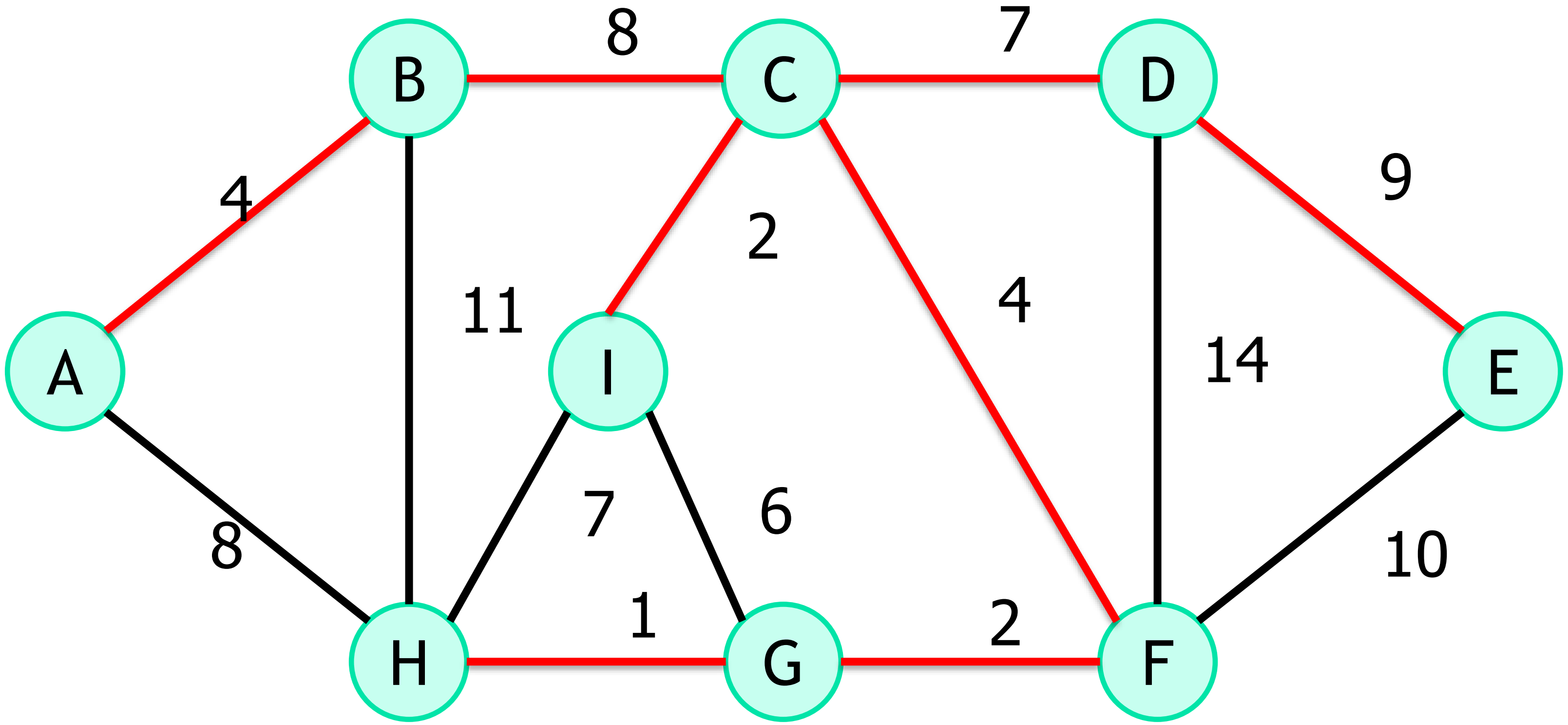


(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	*4*
~~(i,g)~~	~~6~~
(c,d)	*7*
~~(h,i)~~	~~7~~
(b,c)	*8*
~~(a,h)~~	~~8~~
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

italic: chosen

(a,h) cannot be included, otherwise, a cycle is formed

Kruskal’s algorithm - MST

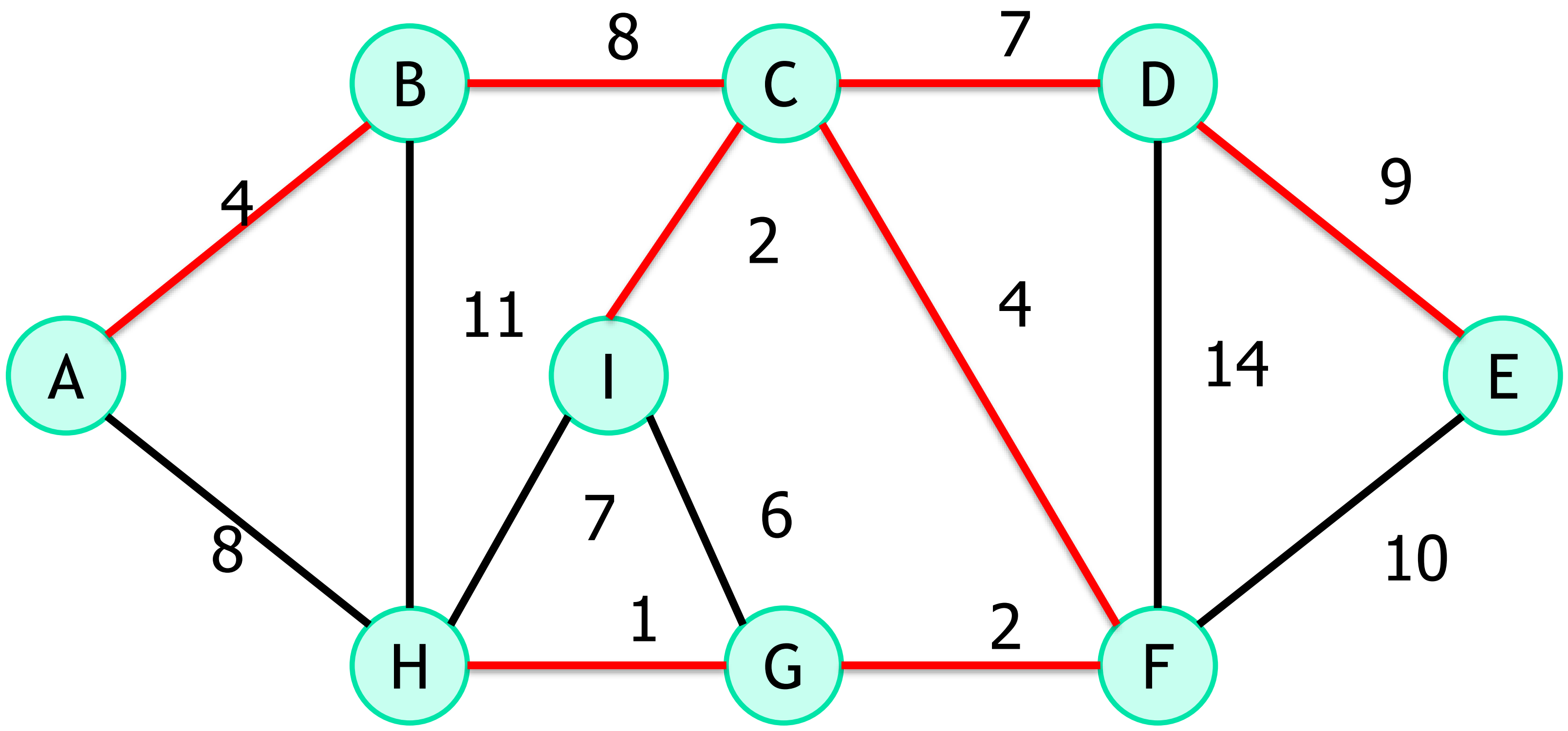


(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	*4*
~~(i,g)~~	~~6~~
(c,d)	*7*
~~(h,i)~~	~~7~~
(b,c)	8
~~(a,h)~~	~~8~~
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14

italic: chosen

Choose the next minimum weight edge

Kruskal’s algorithm - MST

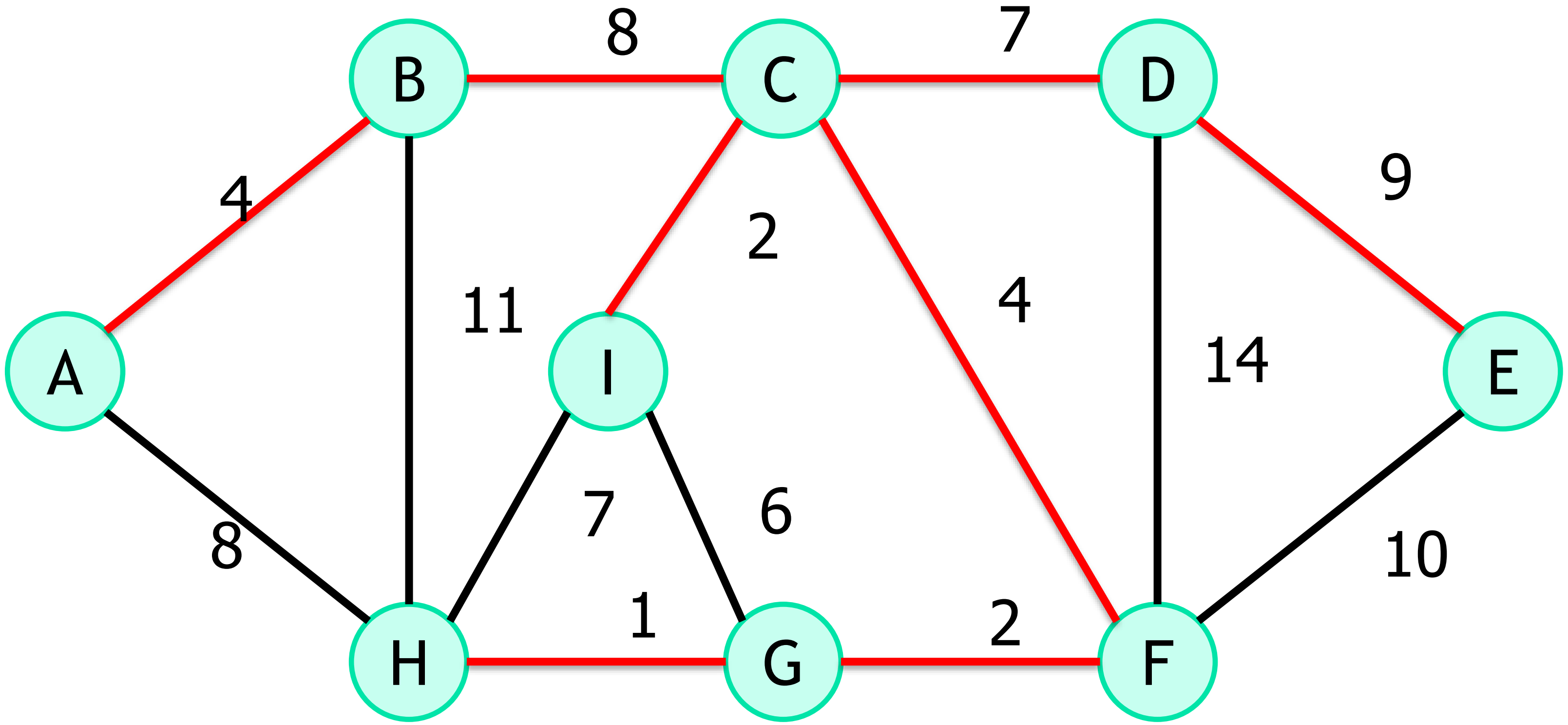


(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	*4*
~~(i,g)~~	~~6~~
(c,d)	*7*
~~(h,i)~~	~~7~~
(b,c)	*8*
~~(a,h)~~	~~8~~
(d,e)	*9*
~~(f,e)~~	~~10~~
(b,h)	11
(d,f)	14

italic: chosen

(f,e) cannot be included, otherwise, a cycle is formed

Kruskal’s algorithm - MST

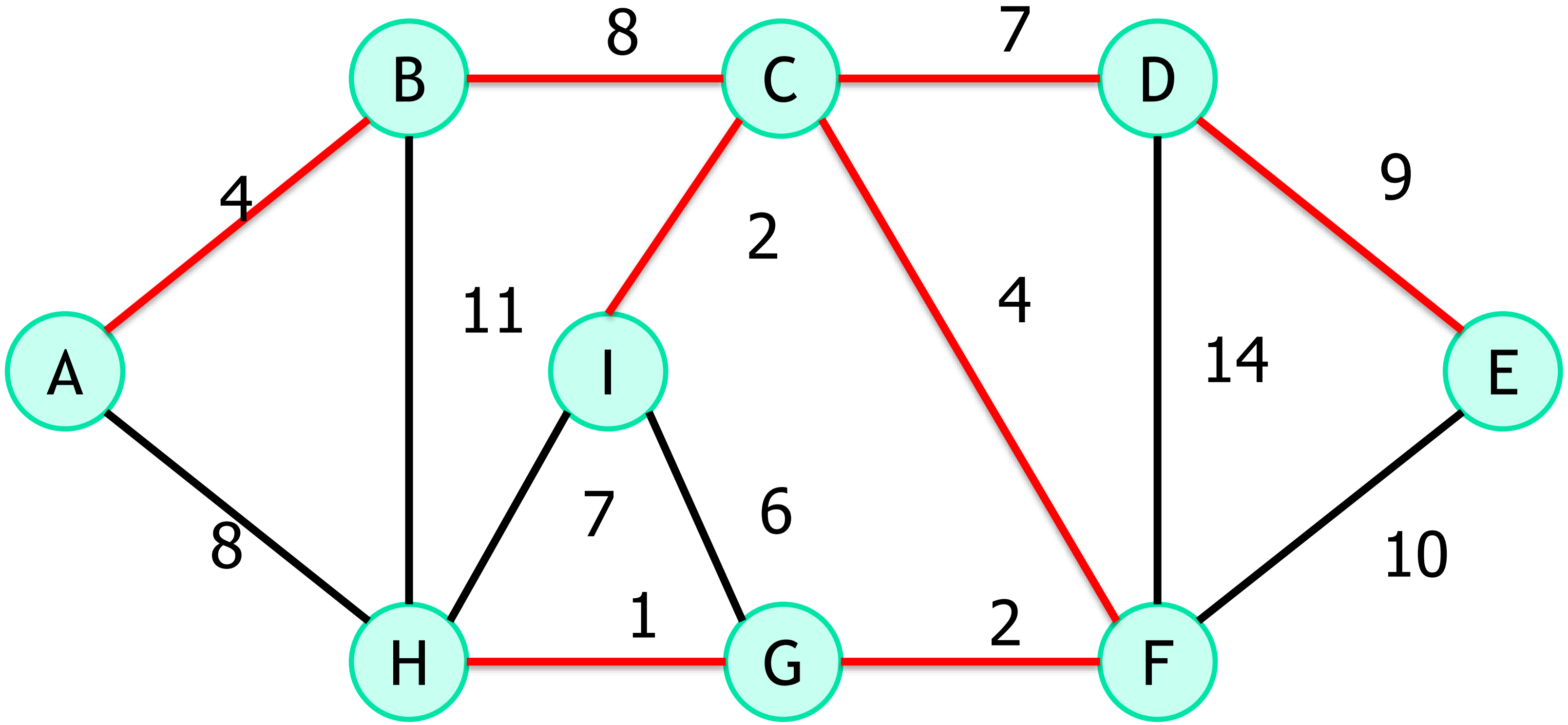


(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	*4*
~~(i,g)~~	~~6~~
(c,d)	*7*
~~(h,i)~~	~~7~~
(b,c)	8
~~(a,h)~~	~~8~~
(d,e)	9
~~(f,e)~~	~~10~~
~~(b,h)~~	~~11~~
(d,f)	14

italic: chosen

(b,h) cannot be included, otherwise, a cycle is formed

Kruskal’s algorithm - MST

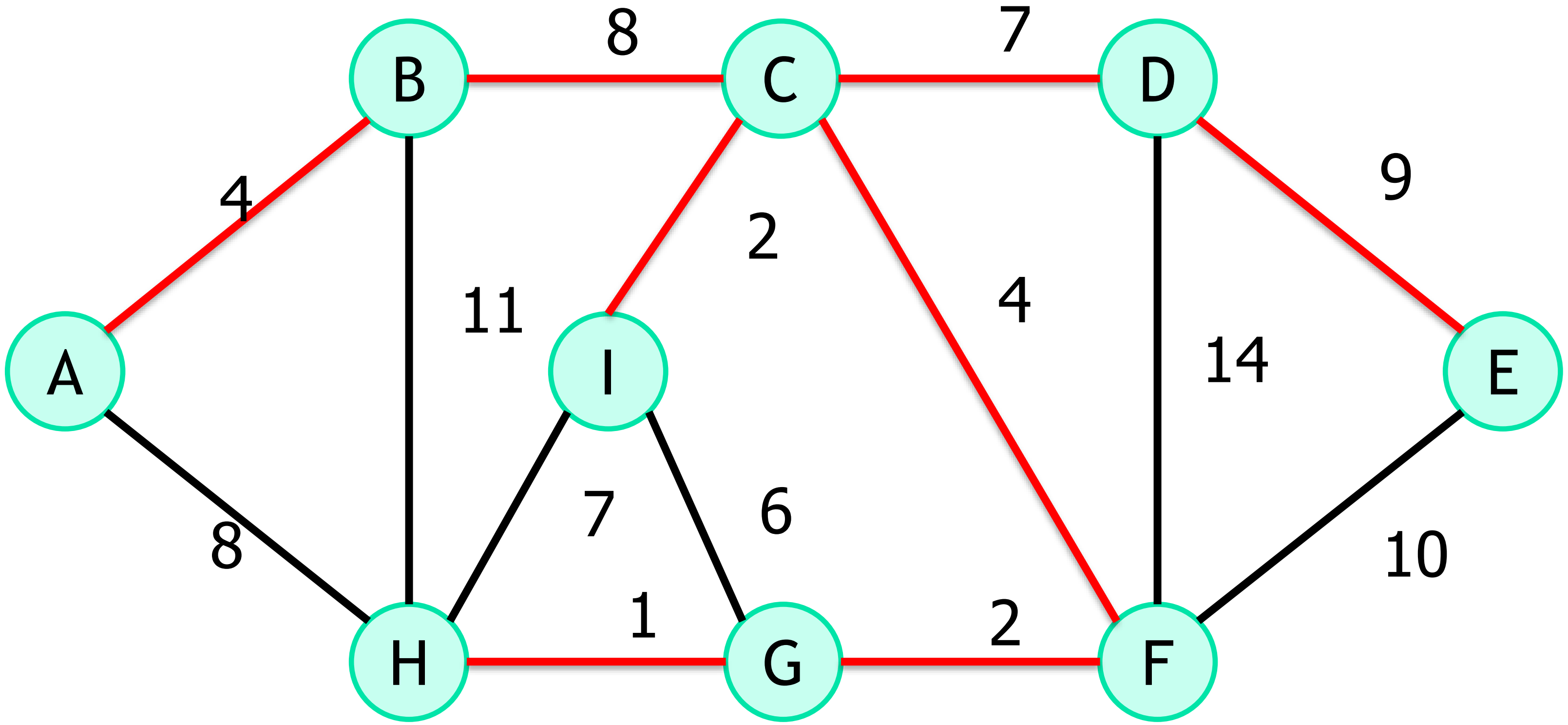


(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	*4*
~~(i,g)~~	~~6~~
(c,d)	*7*
~~(h,i)~~	~~7~~
(b,c)	8
~~(a,h)~~	~~8~~
(d,e)	9
~~(f,e)~~	~~10~~
~~(b,h)~~	~~11~~
~~(d,f)~~	~~14~~

italic: chosen

(d,f) cannot be included, otherwise, a cycle is formed

Kruskal's algorithm - MST

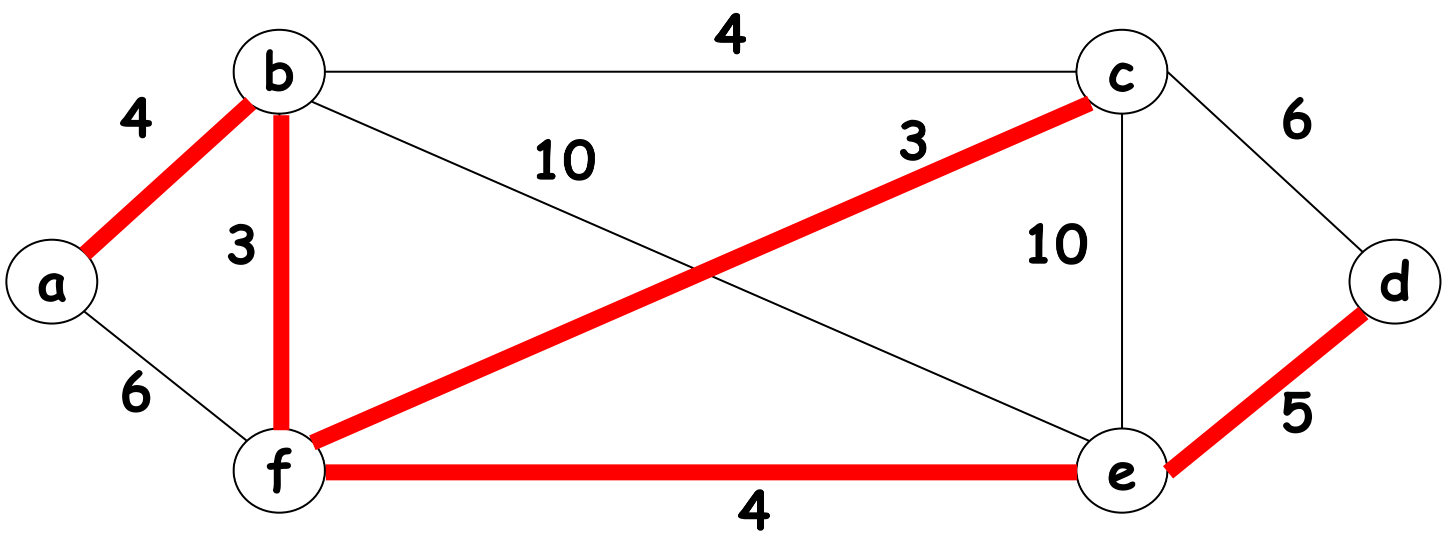


(h,g)	*1*
(i,c)	*2*
(g,f)	*2*
(a,b)	*4*
(c,f)	*4*
~~(i,g)~~	~~6~~
(c,d)	*7*
~~(h,i)~~	~~7~~
(b,c)	8
~~(a,h)~~	~~8~~
(d,e)	9
~~(f,e)~~	~~10~~
~~(b,h)~~	~~11~~
~~(d,f)~~	~~14~~

MST is found when all edges are examined

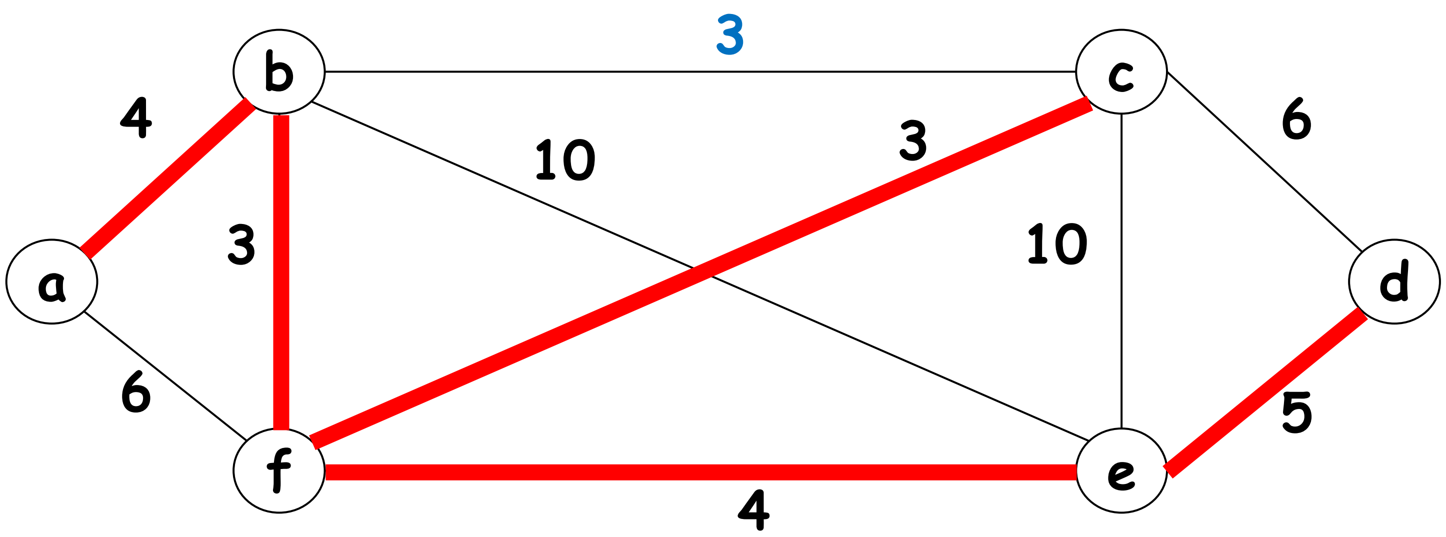
italic: chosen

Exercise – Find MST for this graph



order of (edges) selection: (b,f), (c,f), (a,b), (f,e), (e,d)

Exercise – Find MST for this graph



Kruskal's algorithm

m is the number of edges in the graph
n is the number of vertices in the graph

```
slowKruskal(G = (V,E)):
    Sort the edges in E by non-decreasing weight.
    MST = {}
    for e in E (in sorted order):
        if adding e to MST won't cause a cycle:
            add e to MST.
    return MST
```

m iterations through this loop

How do we check this?

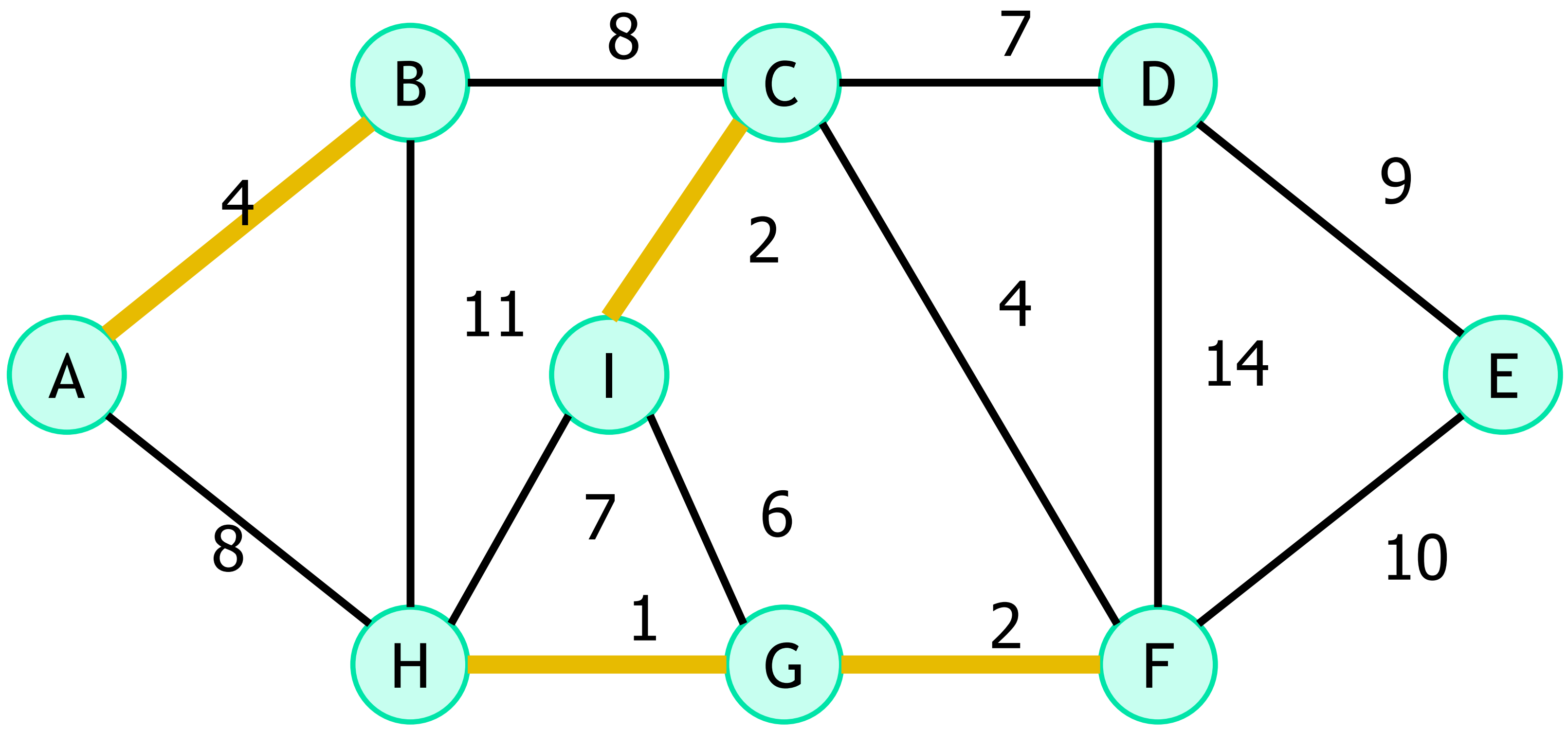
<u>Naively, the running time is O(mn):</u>

- For each of m iterations of the for loop:
    - Check if adding e would cause a cycle…

Kruskal's algorithm

At each step of Kruskal's, we are maintaining a forest.

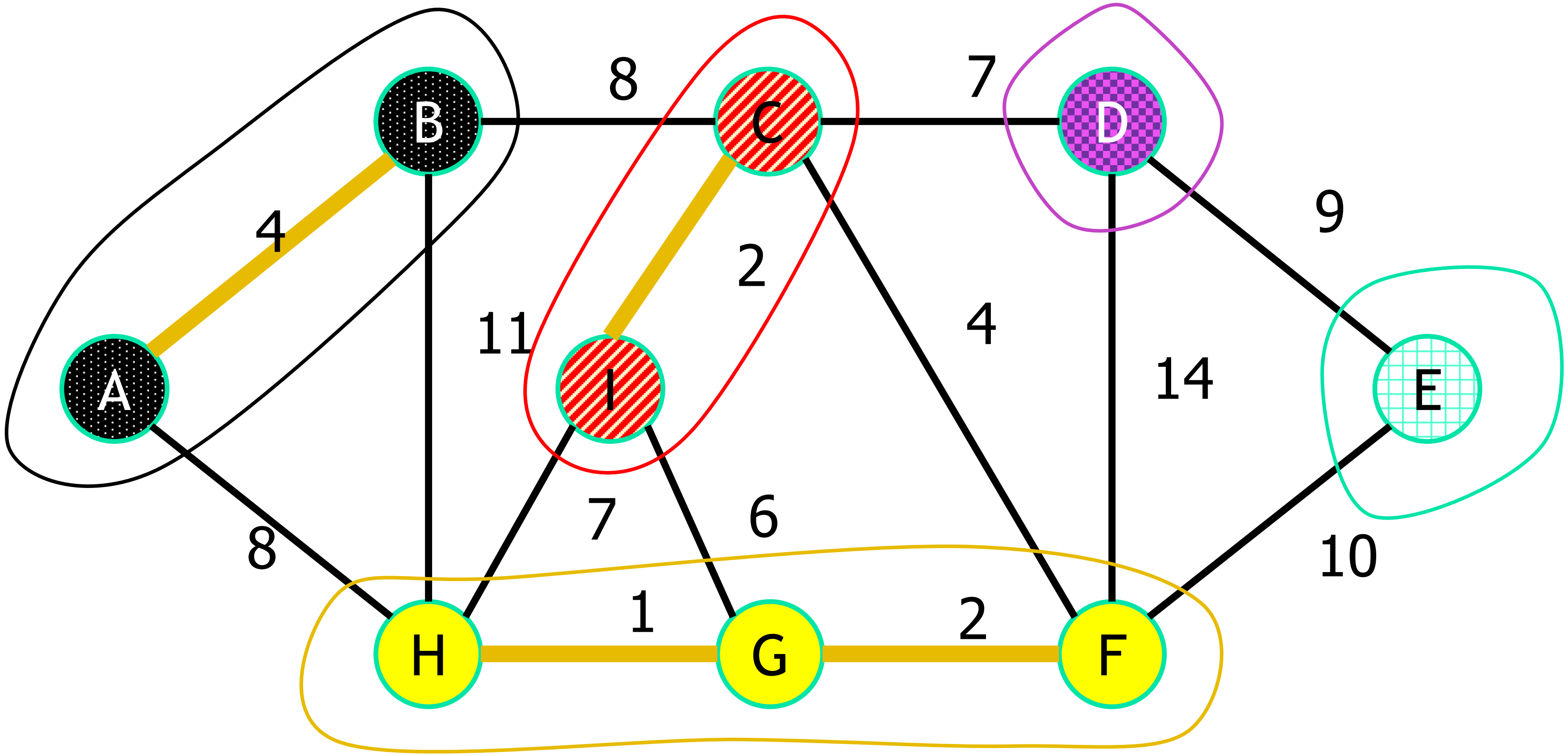
A **forest** is a collection of disjoint trees



Kruskal's algorithm

At each step of Kruskal's, we are maintaining a forest.

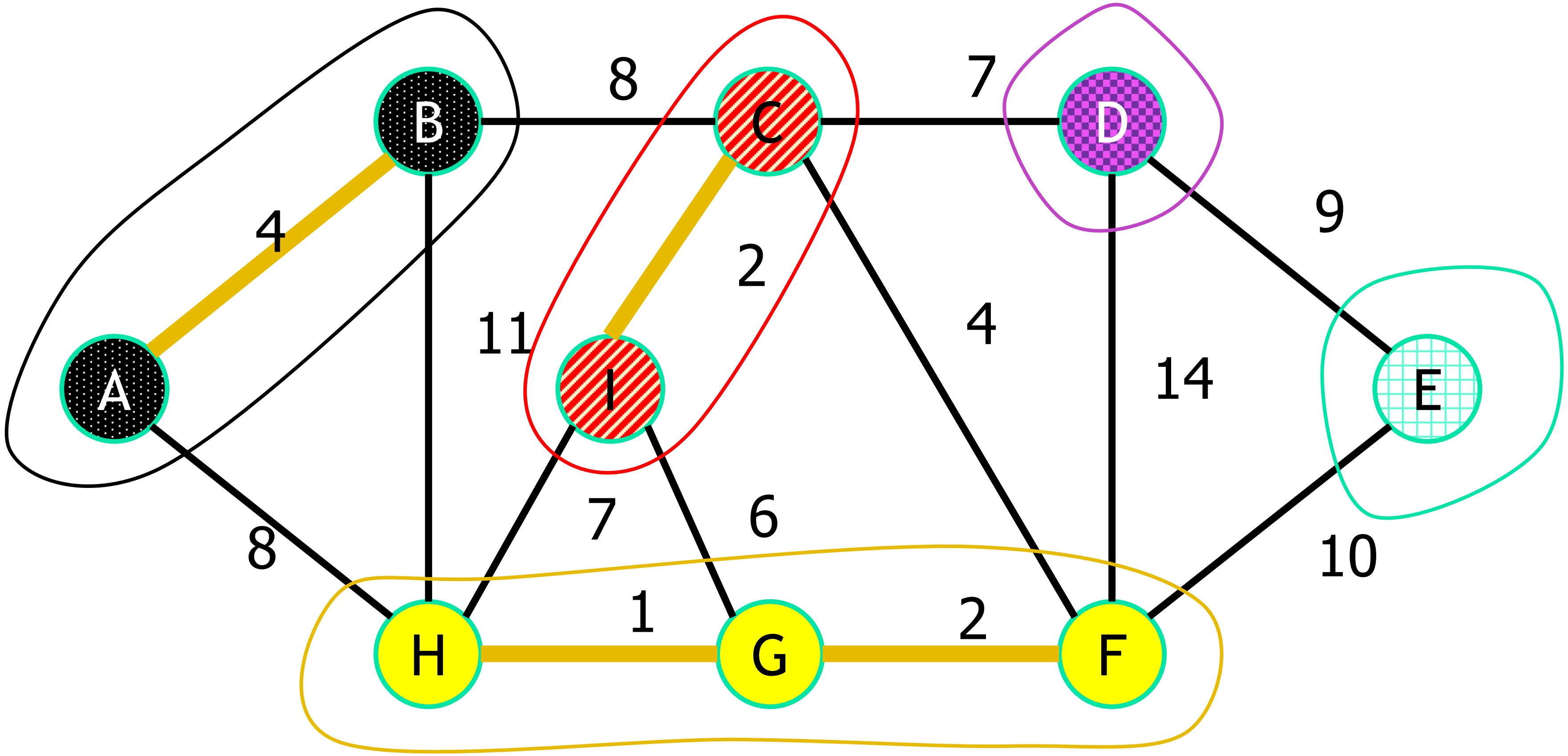
A **forest** is a collection of disjoint trees



Xi'an Jiaotong-Liverpool University 西交利物浦大学

Kruskal's algorithm

When we add an edge, we merge two trees:

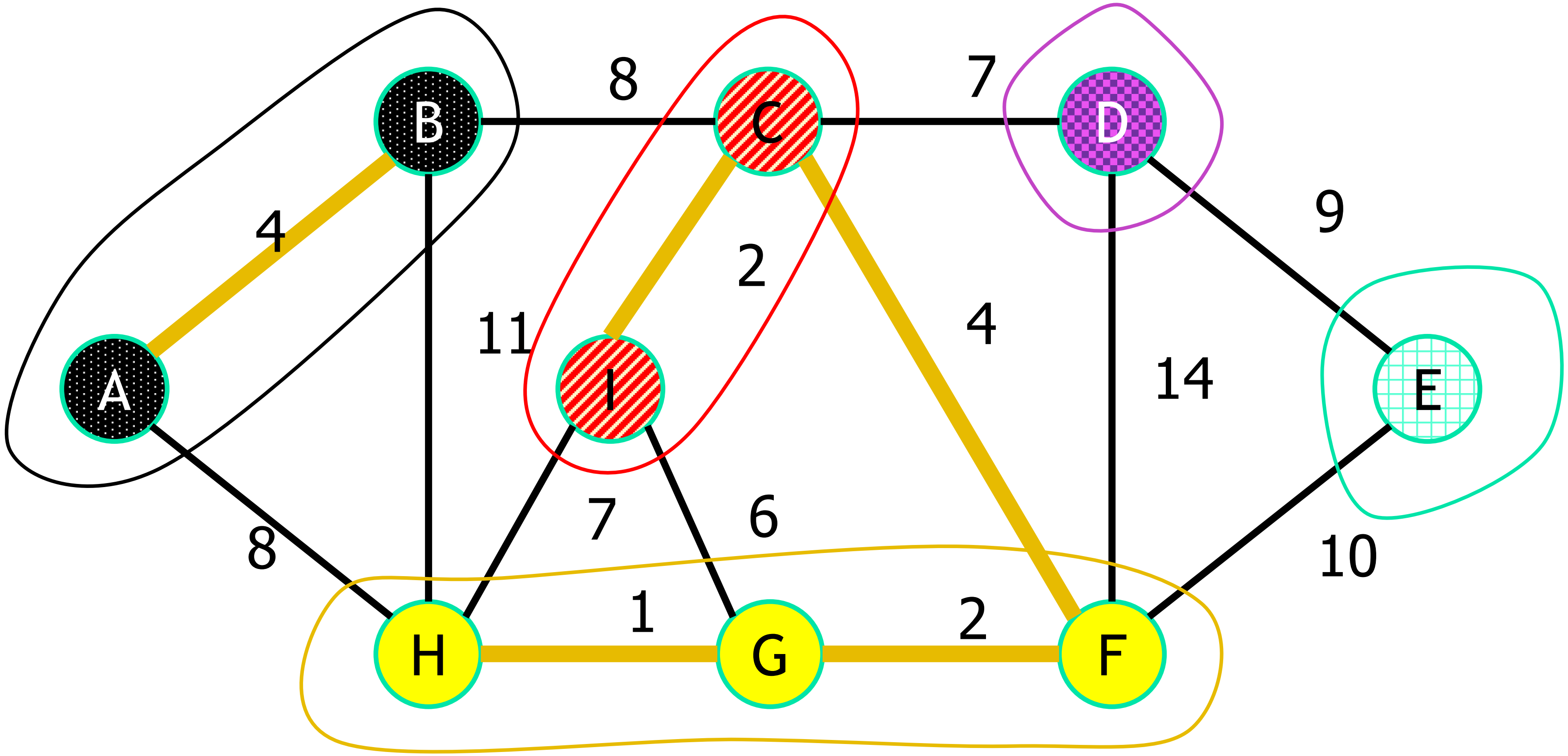


A **forest** is a collection of disjoint trees

Kruskal's algorithm

When we add an edge, we merge two trees:

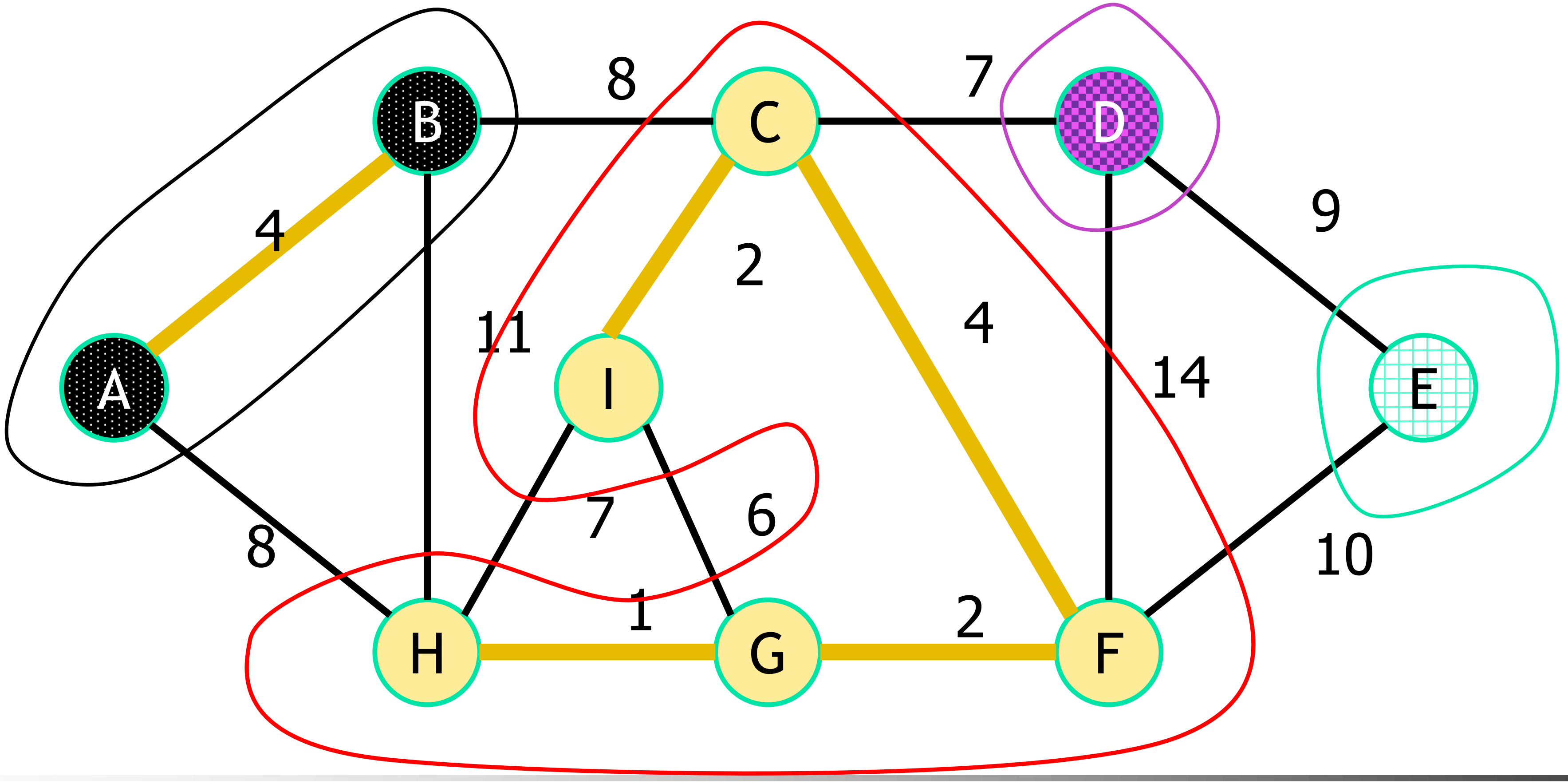
A **forest** is a collection of disjoint trees



Kruskal’s algorithm

When we add an edge, we merge two trees.
We never add an edge within a tree since that would create a cycle

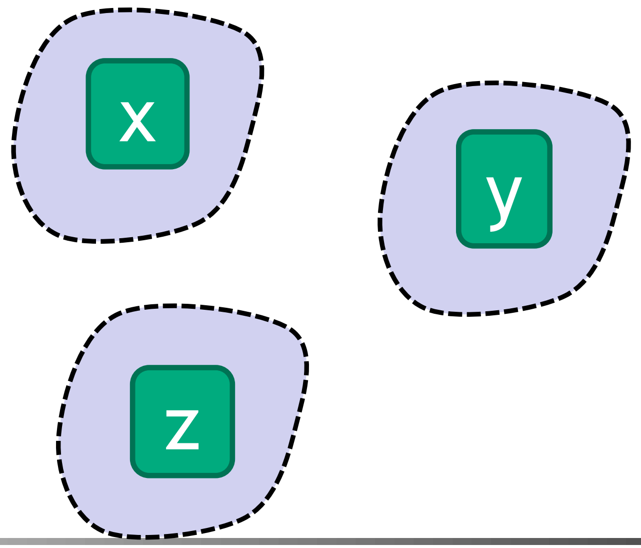
A **forest** is a collection of disjoint trees



Disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)
```

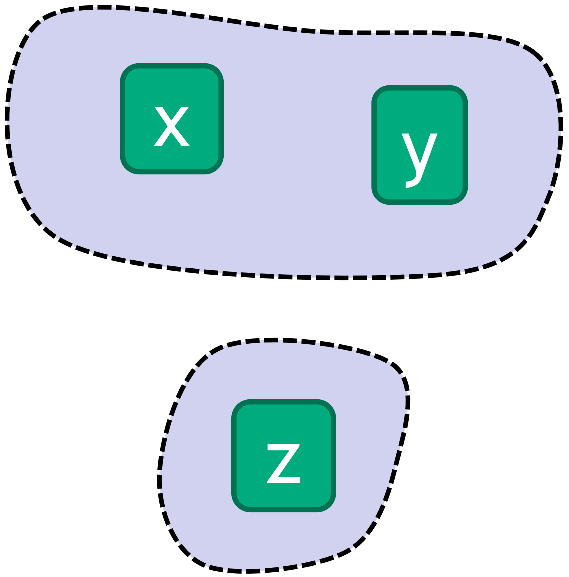


Disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)
```



Disjoint-set data structure

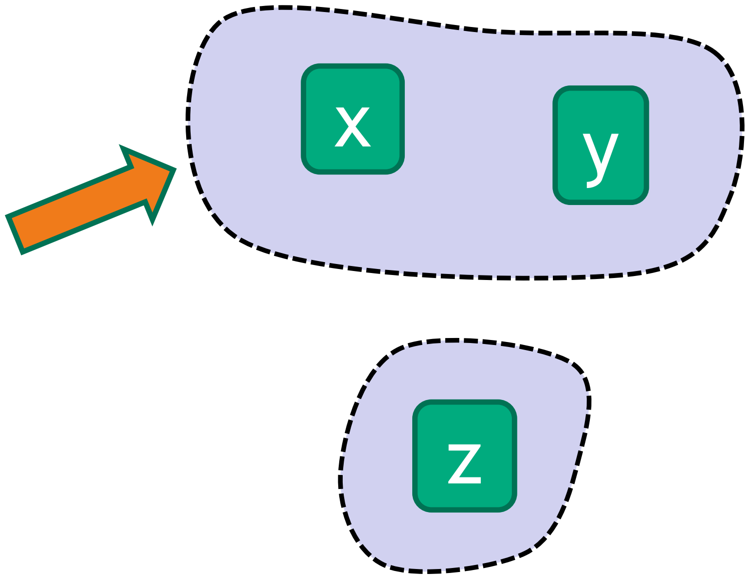
- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

Refer to Textbook Chapter 21 for more info

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)

  find(x)
```



Kruskal pseudo-code

```
kruskal(G = (V,E)):
    Sort E by weight in non-decreasing order
    MST = {}                          // initialize an empty tree
    for v in V:
        makeSet(v)                    // put each vertex in its own tree in the forest
    for (u,v) in E:                   // go through the edges in sorted order
        if find(u) != find(v):        // if u and v are not in the same tree
            add (u,v) to MST
            union(u,v)                // merge u's tree with v's tree
    return MST
```

Running time

- Sorting the edges takes O(mlogn)
- For the rest:
  - n calls to **makeSet**
    - put each vertex in its own set
  - 2m calls to **find**
    - for each edge, **find** its endpoints
  - n calls to **union**
    - we will never add more than n-1 edges to the tree,
    - so we will never call **union** more than n-1 times.
- Total running time:
  - Worst-case O(mlogn).

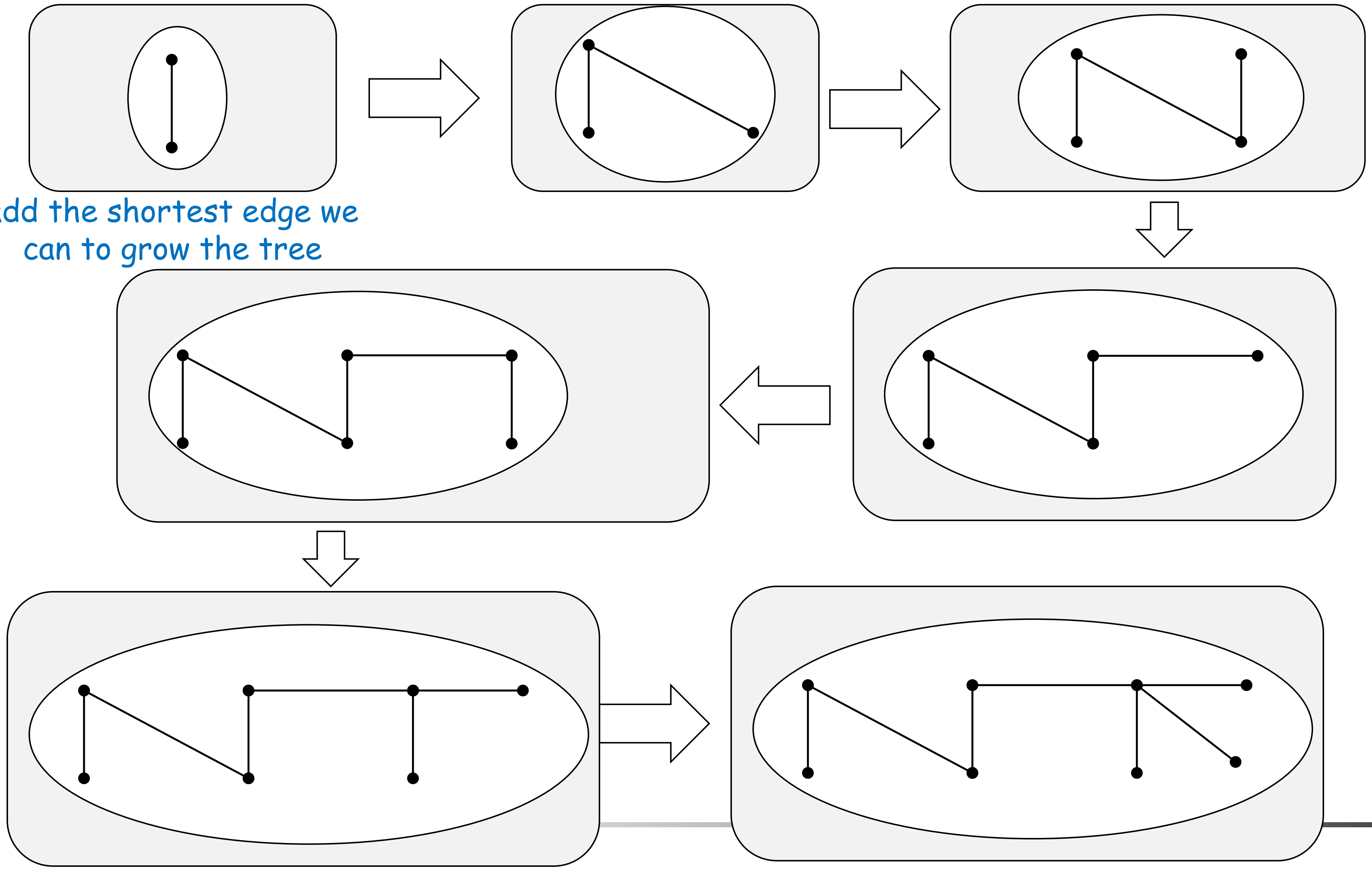
In any graph, $m = O(n^2)$
Therefore,
$O(m\log m)=O(m\log n^2)$
$=O(m\log n)$

In practice, each of makeSet, find, and union run in constant time*

*technically, they run in *amortized time* $O(\alpha(n))$, where $\alpha(n)$ is the *inverse Ackermann function*. $\alpha(n) \leq 4$ provided that n is smaller than the number of atoms in the universe.

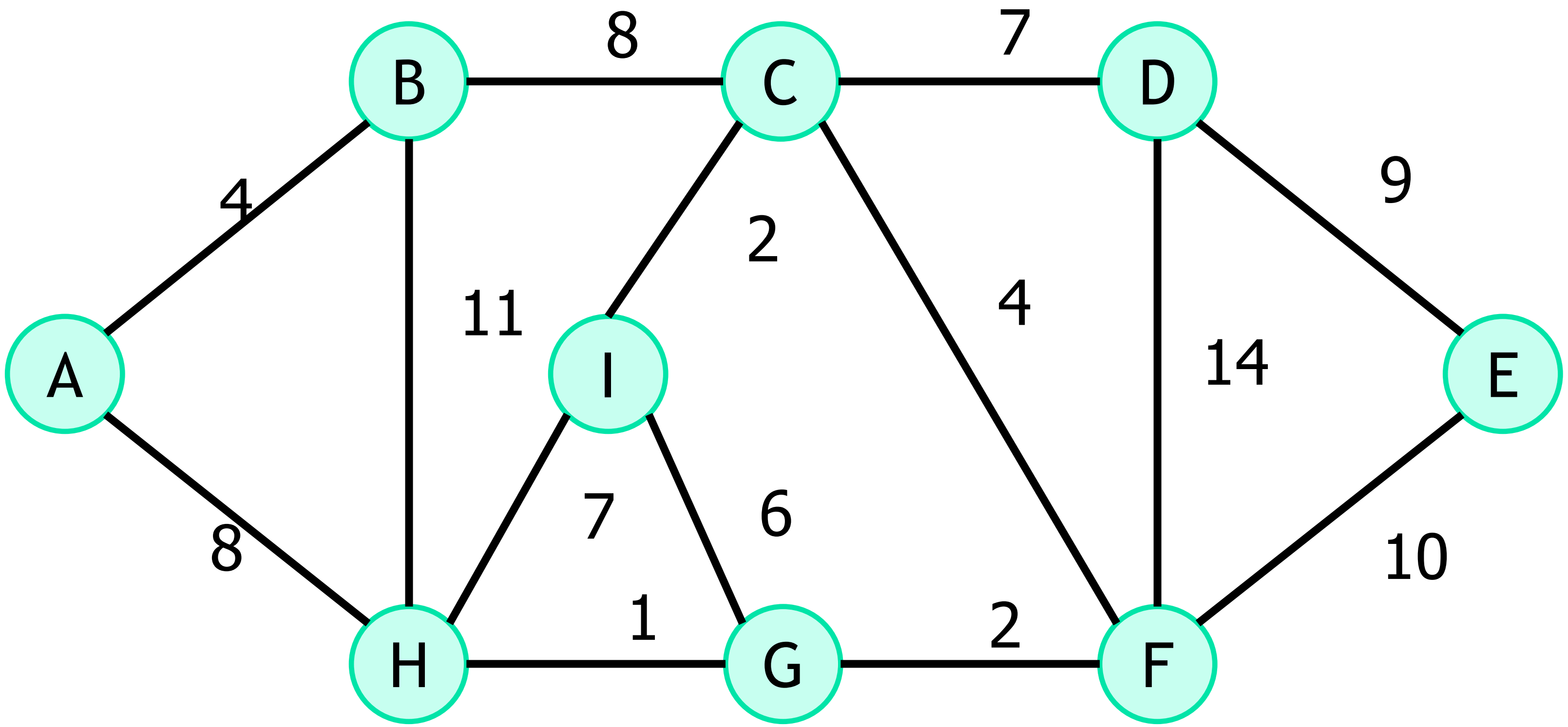
Prim’s Algorithm

Idea of Prim's Algorithm - MST



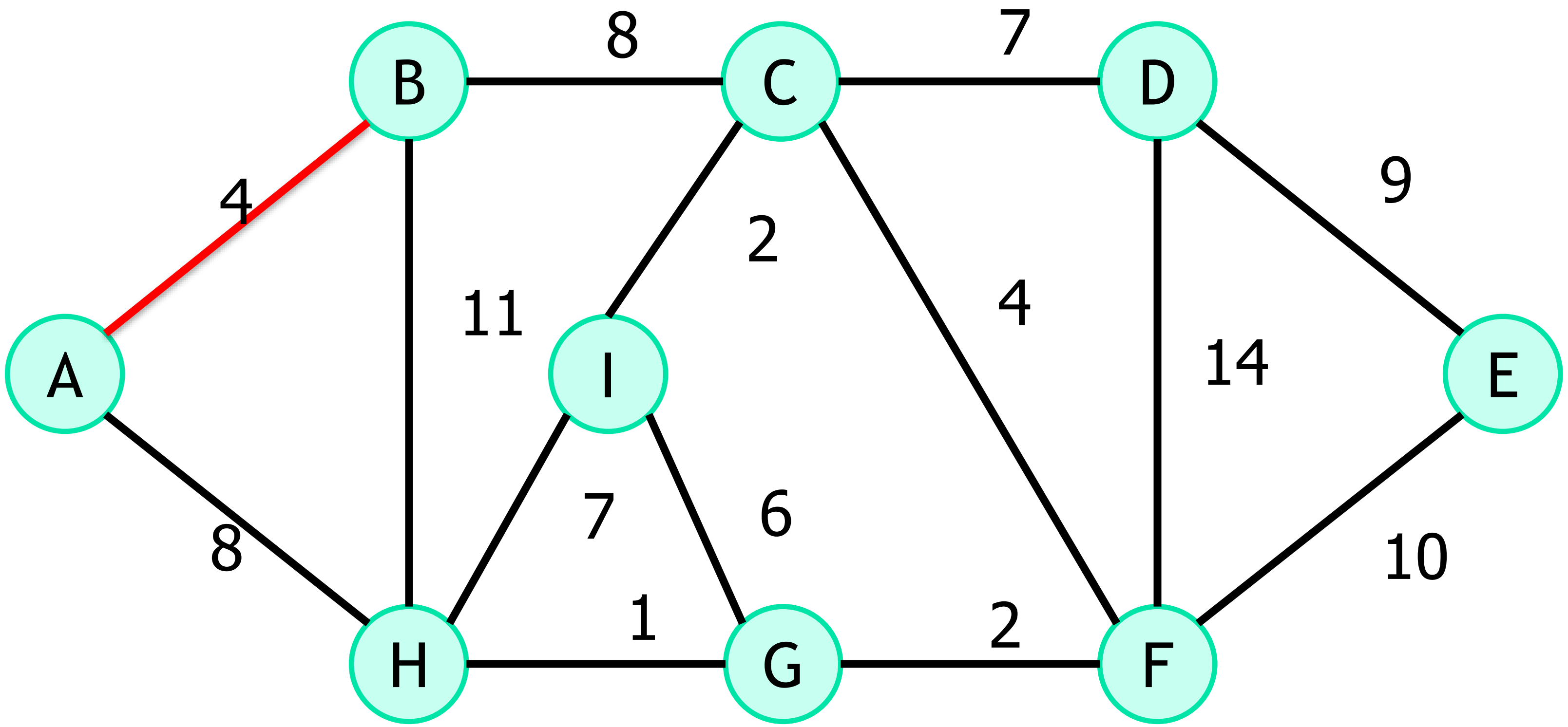
Prim's Algorithm

- Start growing a tree, greedily add the shortest edge we can to grow the tree.



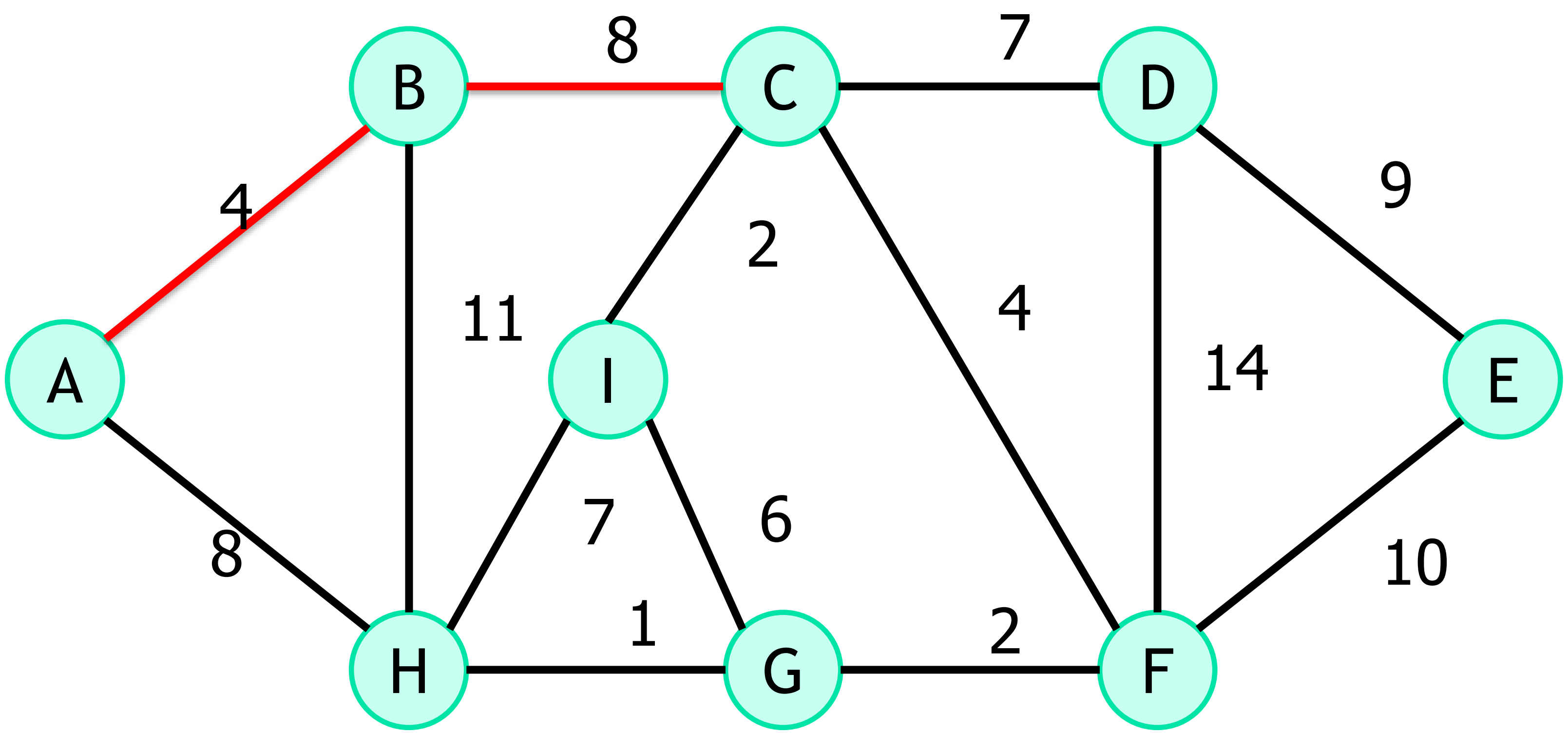
Prim’s Algorithm

- Start growing a tree, greedily add the shortest edge we can to grow the tree.



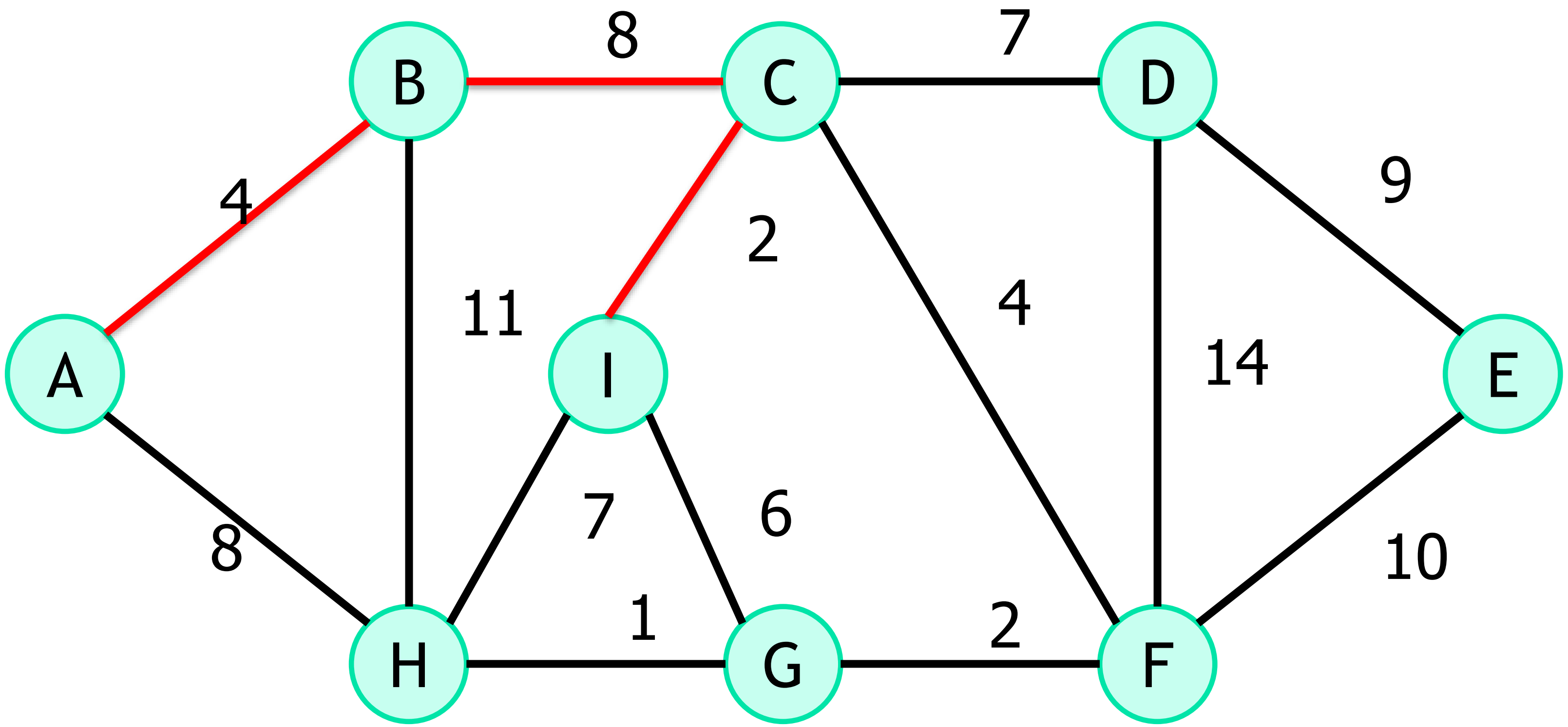
Prim's Algorithm

- Start growing a tree, greedily add the shortest edge we can to grow the tree.



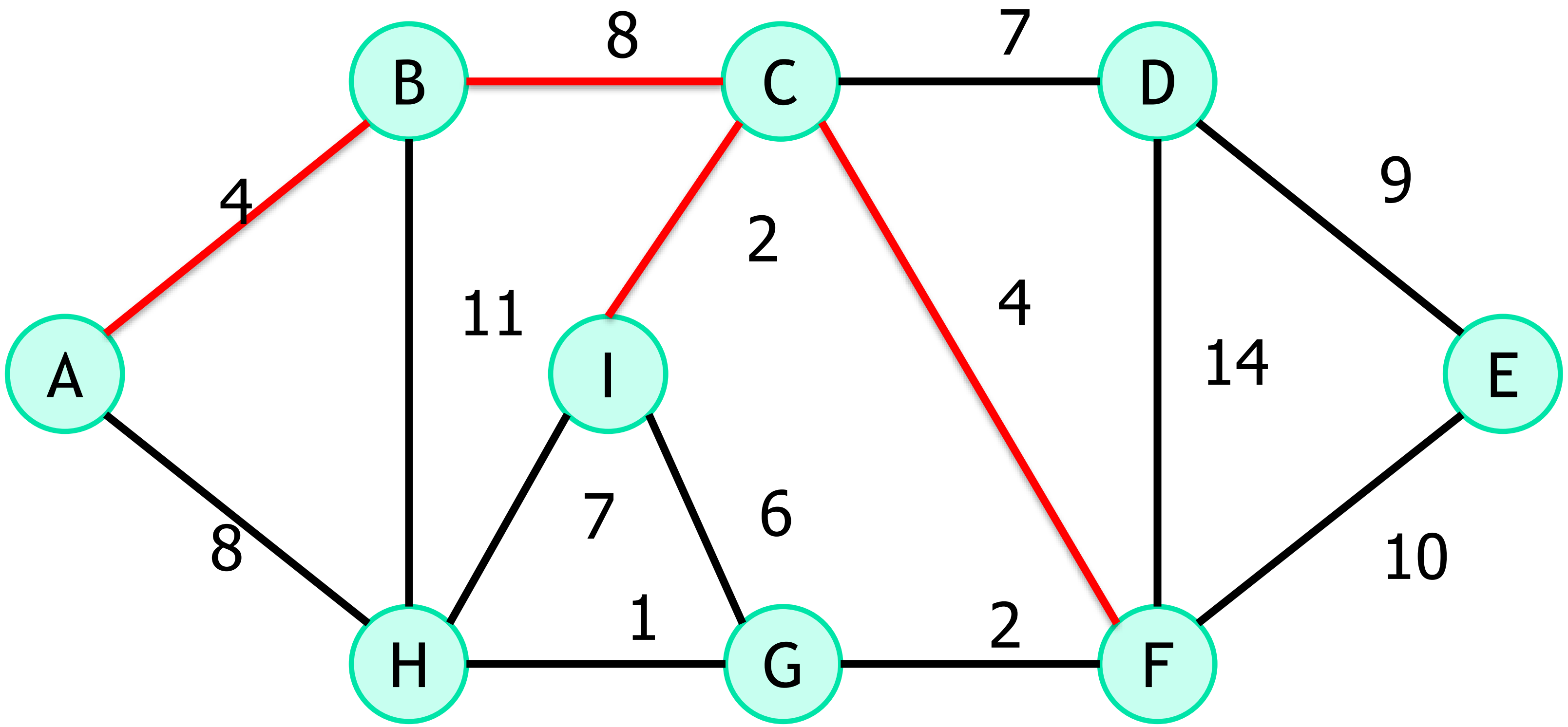
Prim’s Algorithm

- Start growing a tree, greedily add the shortest edge we can to grow the tree.



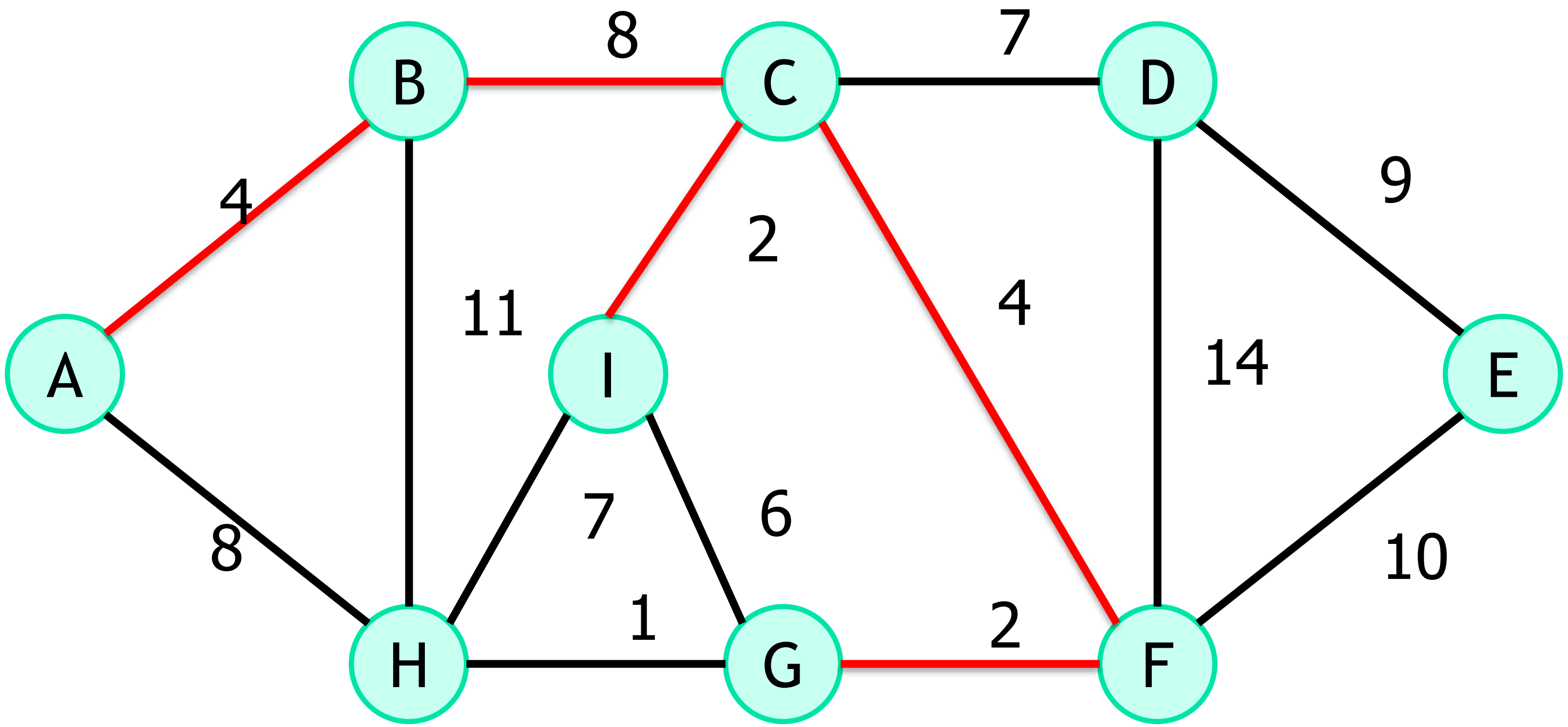
Prim's Algorithm

- Start growing a tree, greedily add the shortest edge we can to grow the tree.



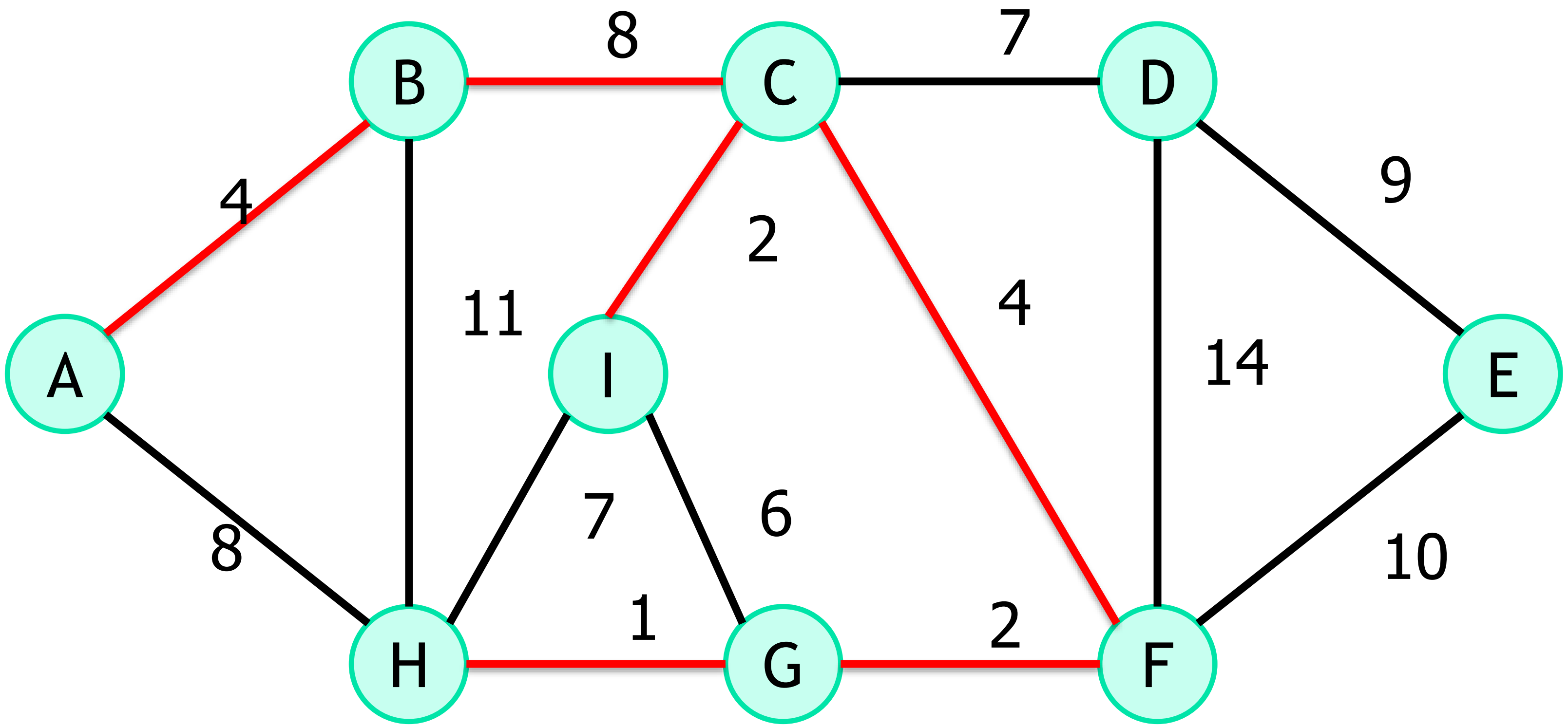
Prim’s Algorithm

- Start growing a tree, greedily add the shortest edge we can to grow the tree.



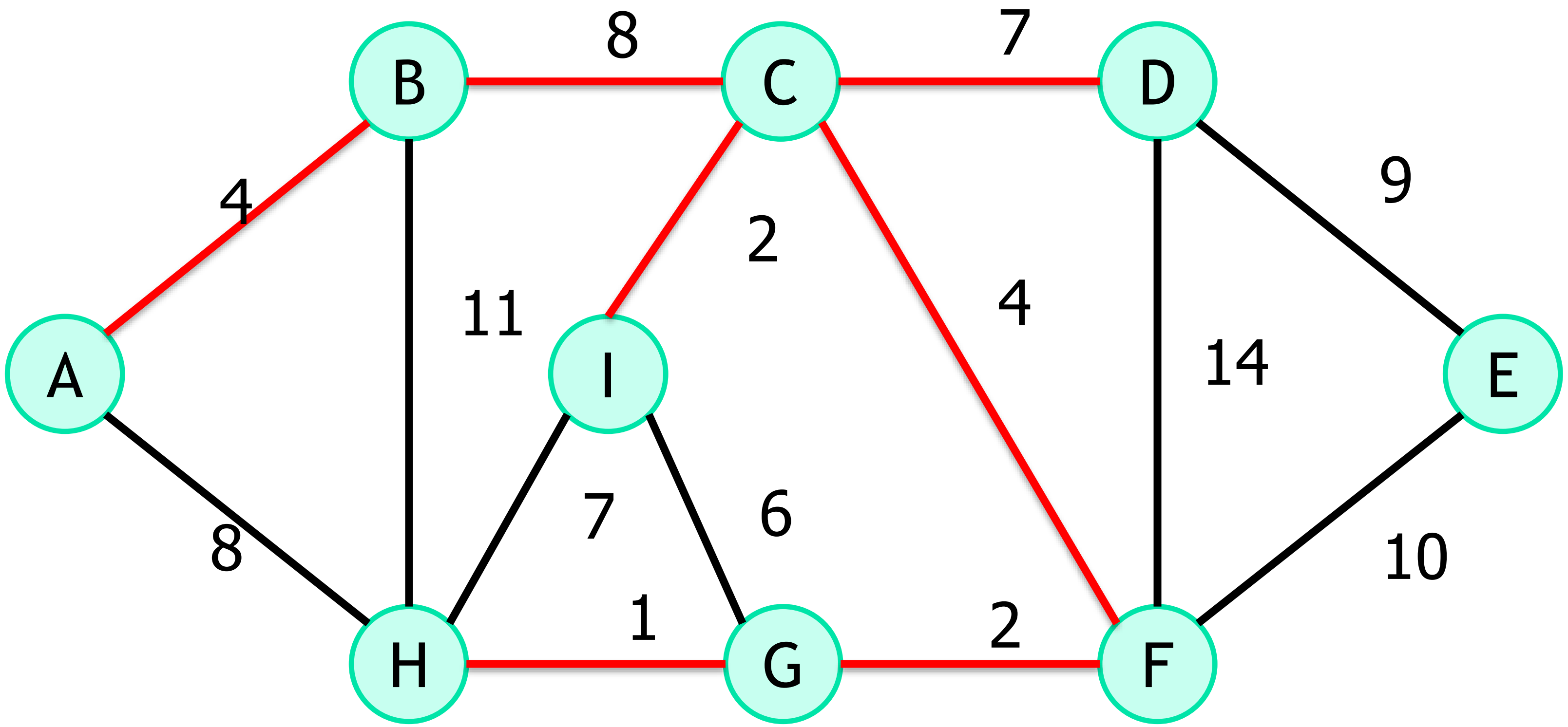
Prim’s Algorithm

- Start growing a tree, greedily add the shortest edge we can to grow the tree.



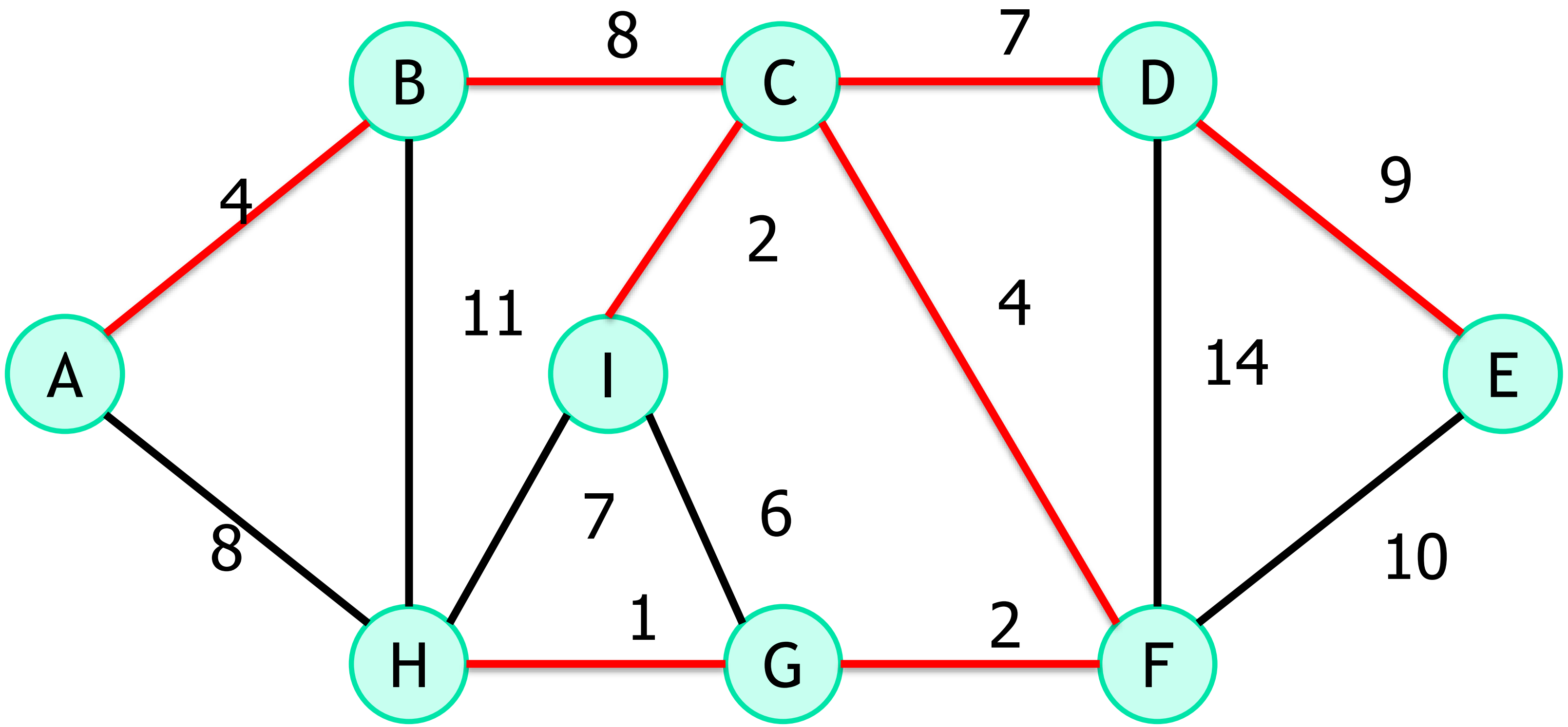
Prim’s Algorithm

- Start growing a tree, greedily add the shortest edge we can to grow the tree.



Prim’s Algorithm

- Start growing a tree, greedily add the shortest edge we can to grow the tree.



Prim's algorithm

```
slowPrim( G = (V,E), starting vertex s ):
    MST = {}
    verticesVisited = {s}
    while |verticesVisited| < |V|:
        find the lightest edge (x,v) in E so that:
            x is in verticesVisited
            v is not in verticesVisited
        add (x,v) to MST
        add v to verticesVisited
    return MST
```

n iterations of this while loop.

Maybe take time m to go through all the edges and find the lightest.

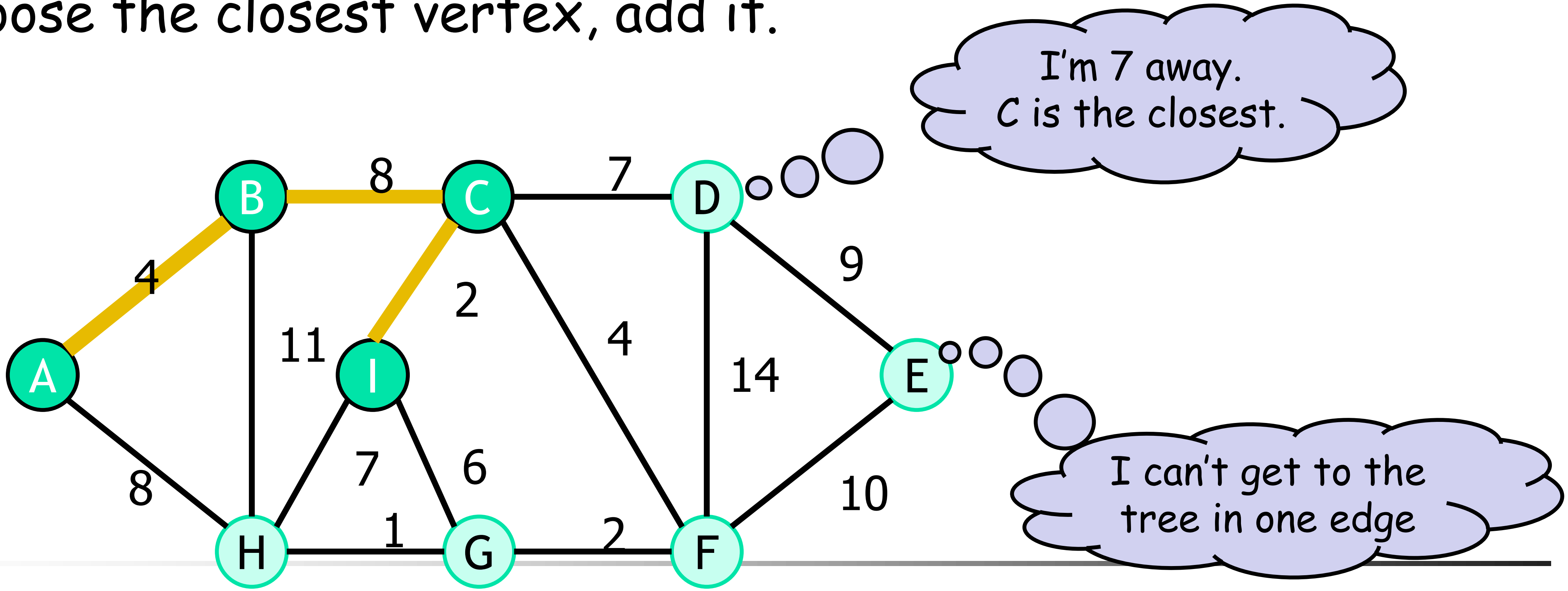
<u>Naively, the running time is O(nm):</u>

- For each of n-1 iterations of the while loop:
  - Maybe go through all the edges.

Efficient implementation

- Each vertex keeps:
  - the **distance** from itself to the ***growing spanning tree***
  - ***how to get there.***
- Choose the closest vertex, add it.

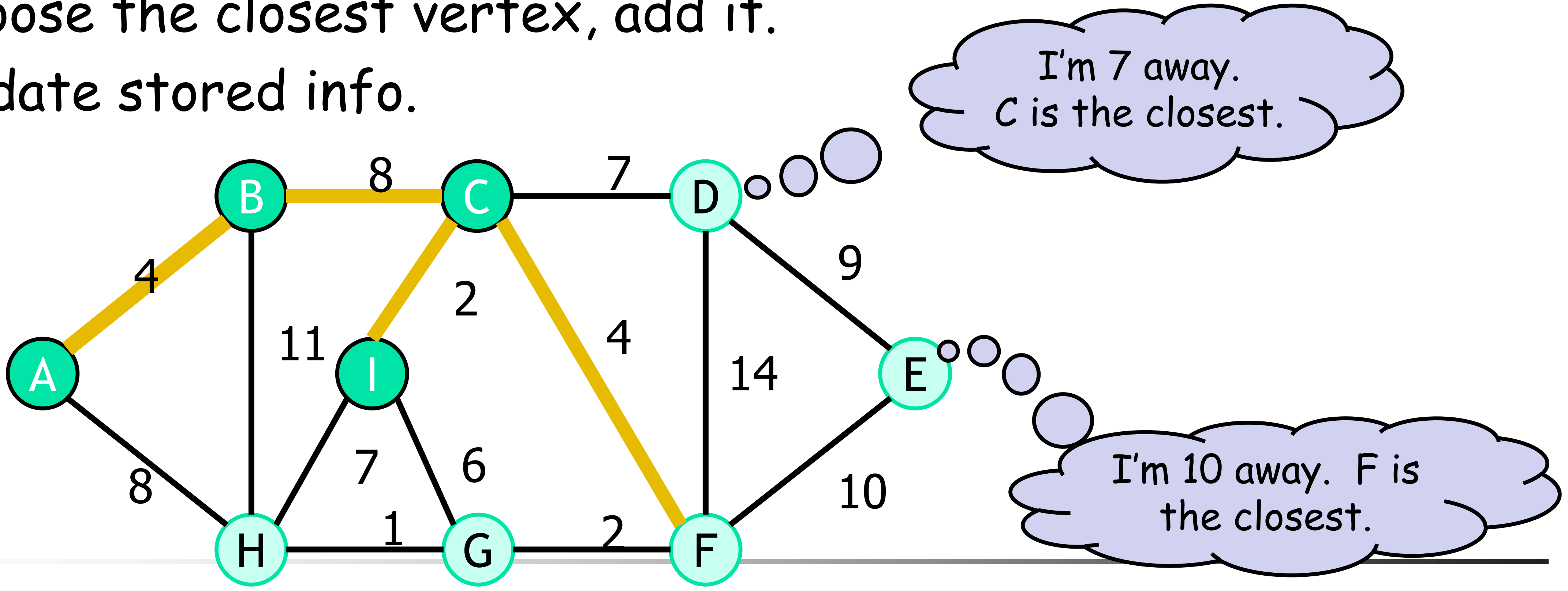
if you can get there in one edge.



Efficient implementation

- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree**
  - **how to get there**.
- Choose the closest vertex, add it.
- Update stored info.

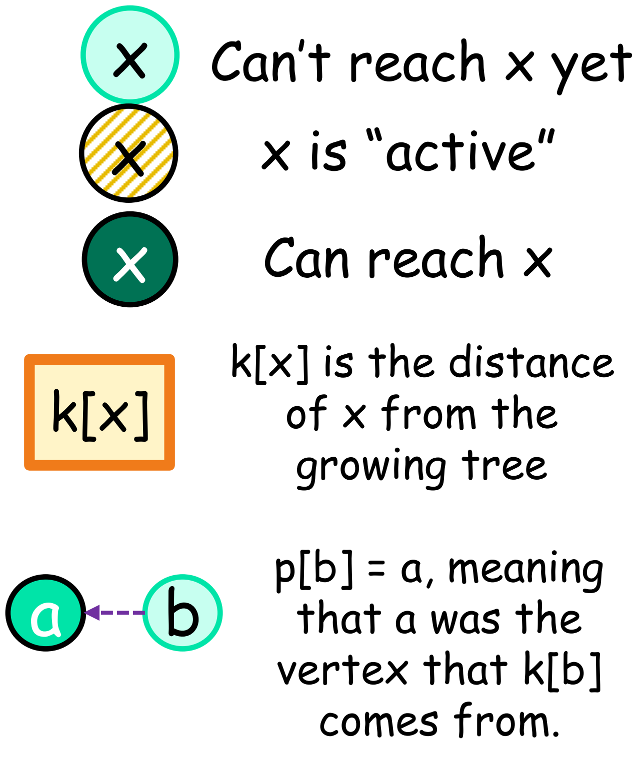
if you can get there in one edge.

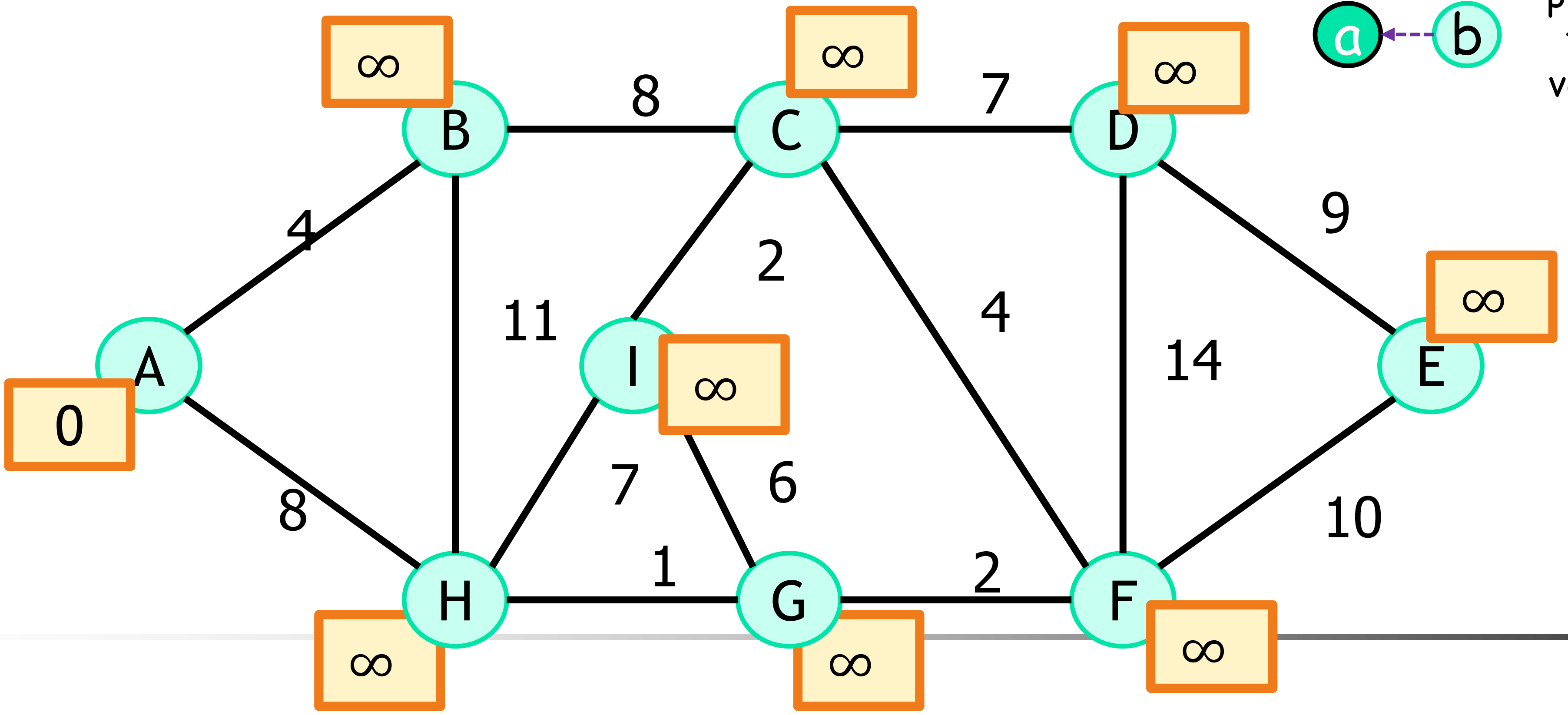


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**





Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.

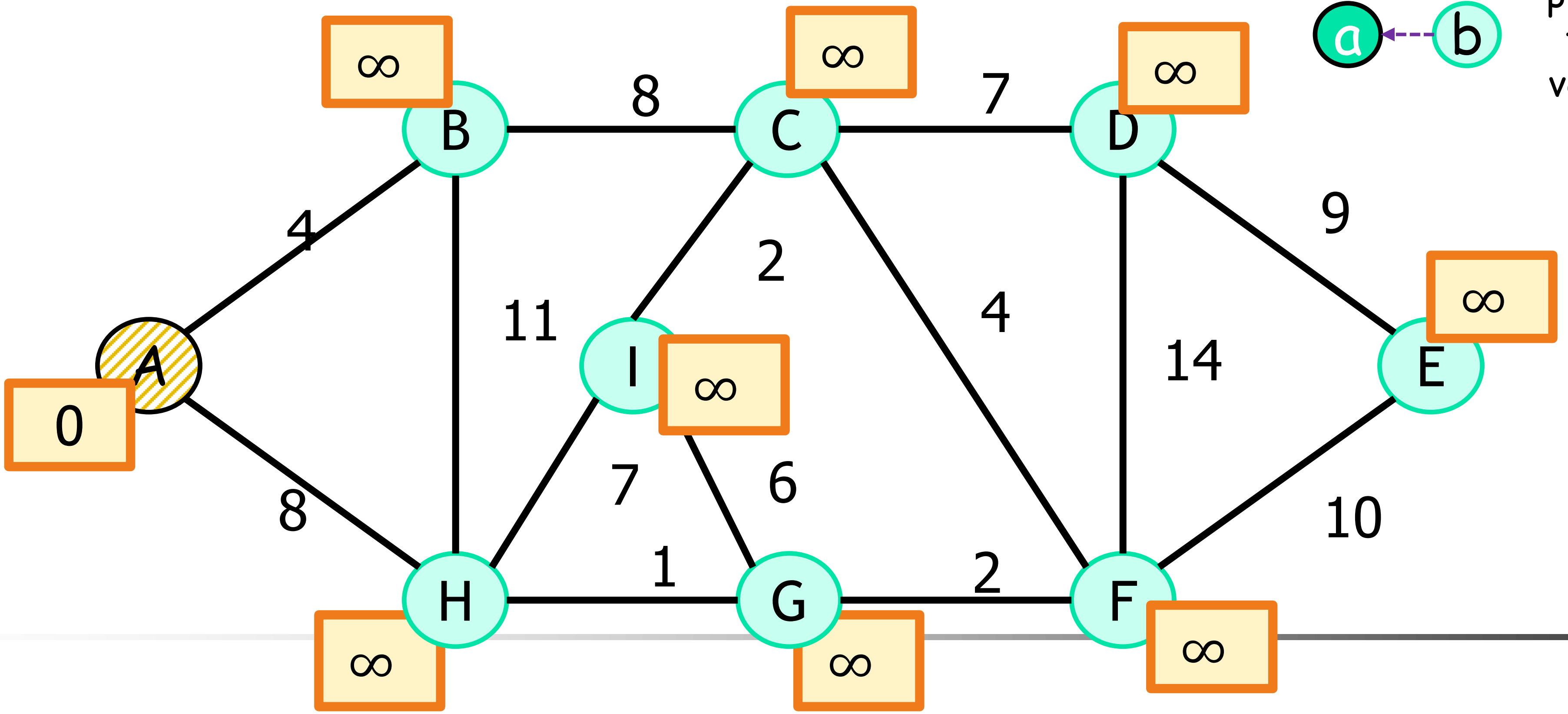
x Can't reach x yet

x x is "active"

x Can reach x

k[x] k[x] is the distance of x from the growing tree

a b p[b] = a, meaning that a was the vertex that k[b] comes from.

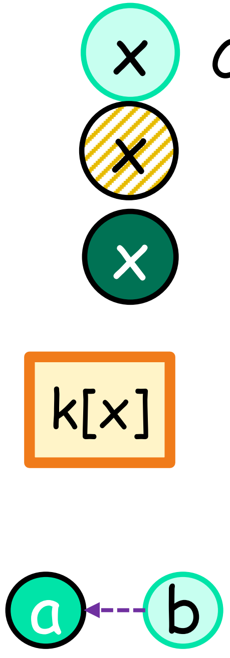


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  k[v] = min(k[v], weight(u,v))
  if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



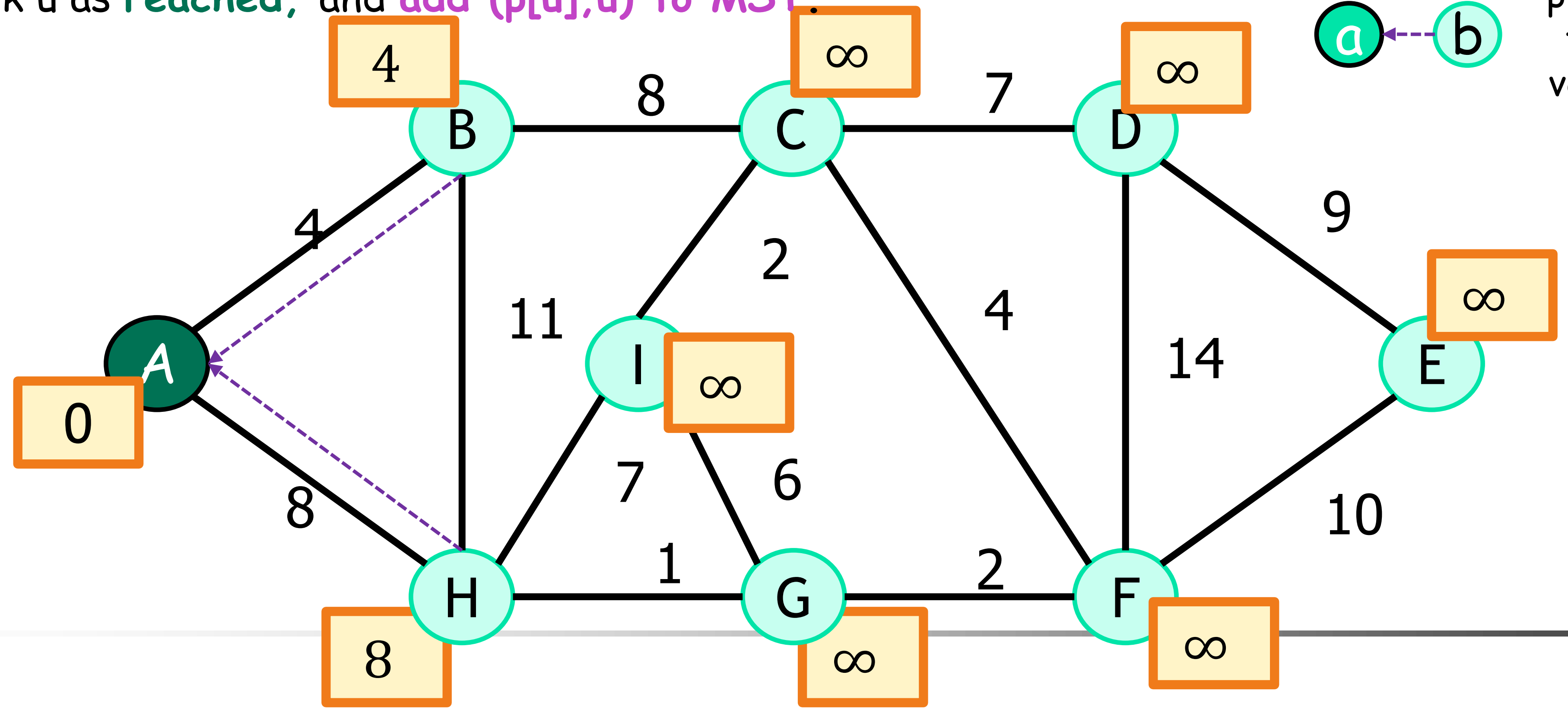
Can't reach x yet

x is "active"

Can reach x

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.

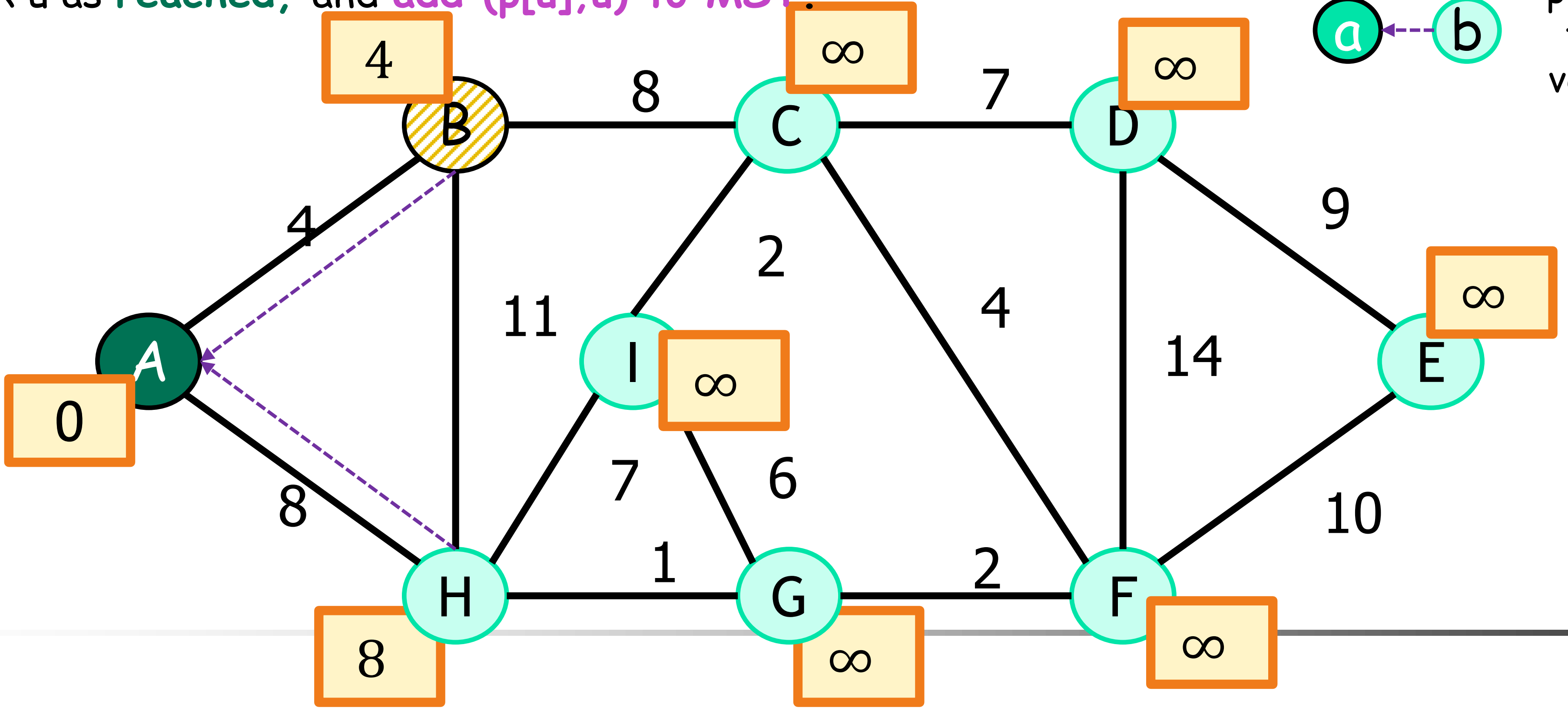


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  k[v] = min(k[v], weight(u,v))
  if k[v] updated, p[v] = u
- Mark u as **reached**, and **add (p[u],u) to MST**.



x Can't reach x yet

x x is "active"

x Can reach x

k[x] k[x] is the distance of x from the growing tree

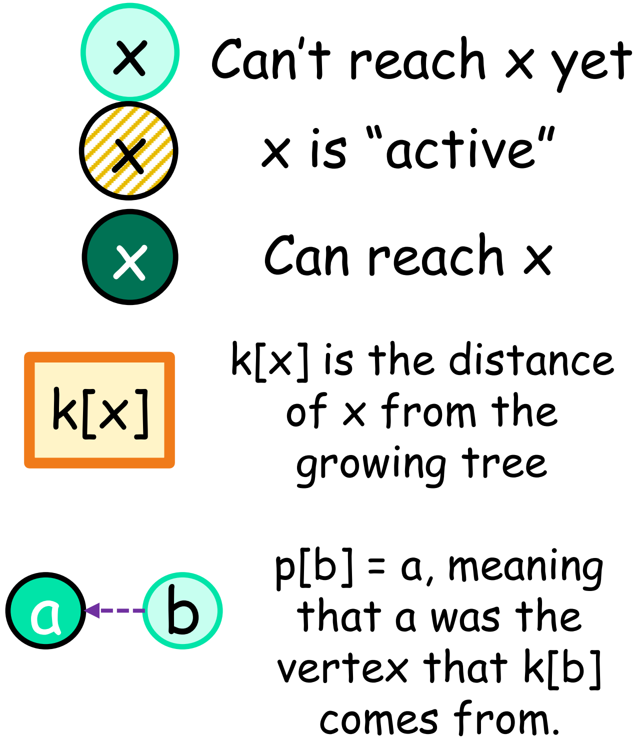
a b p[b] = a, meaning that a was the vertex that k[b] comes from.

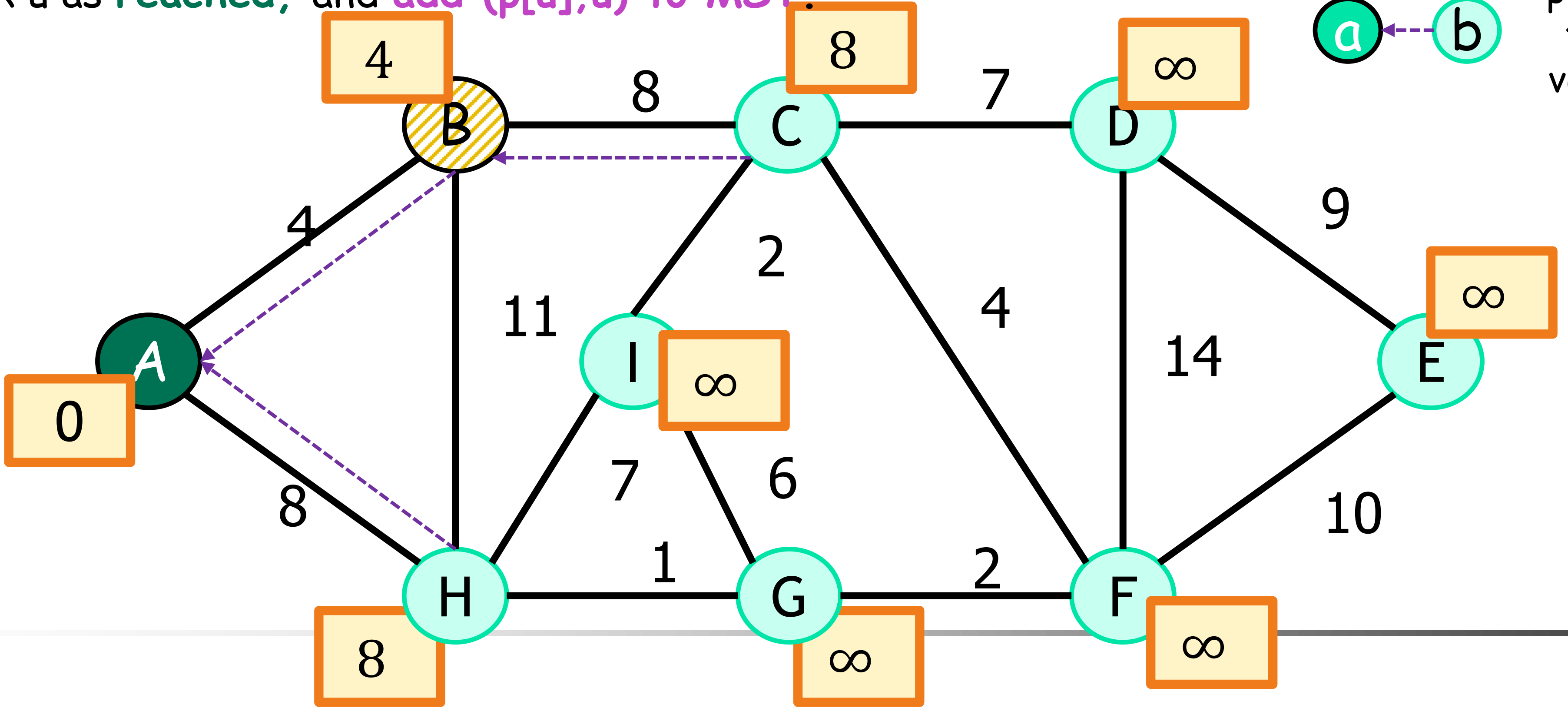
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached**:

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  k[v] = min(k[v], weight(u,v))
  if k[v] updated, p[v] = u
- Mark u as **reached**, and **add (p[u],u) to MST**.



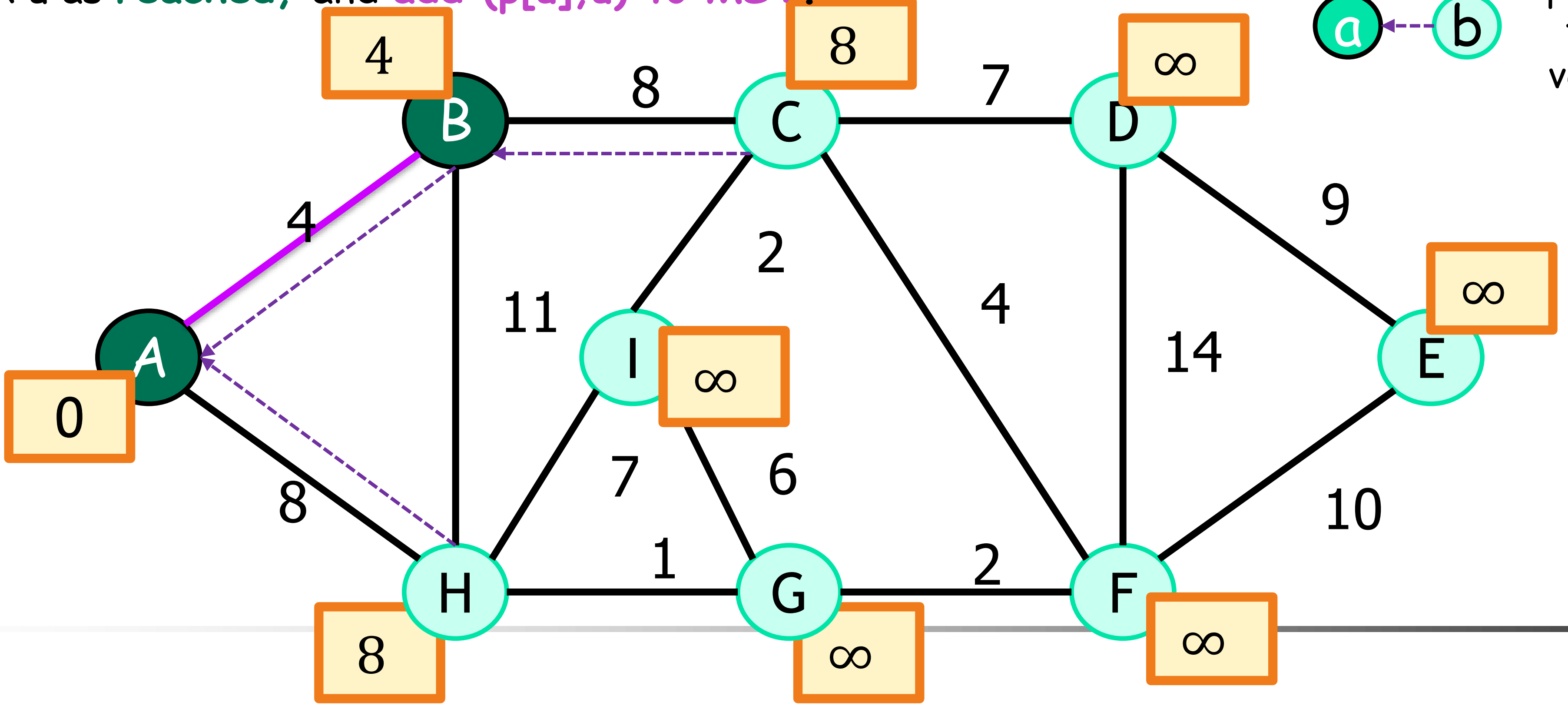


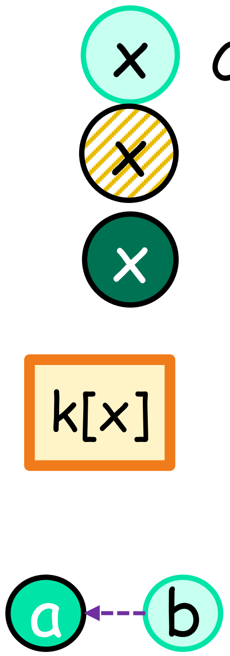
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached**:

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  k[v] = min(k[v], weight(u,v))
  if k[v] updated, p[v] = u
- Mark u as **reached**, and **add (p[u],u) to MST**.





Can't reach x yet

x is "active"

Can reach x

k[x] is the distance of x from the growing tree

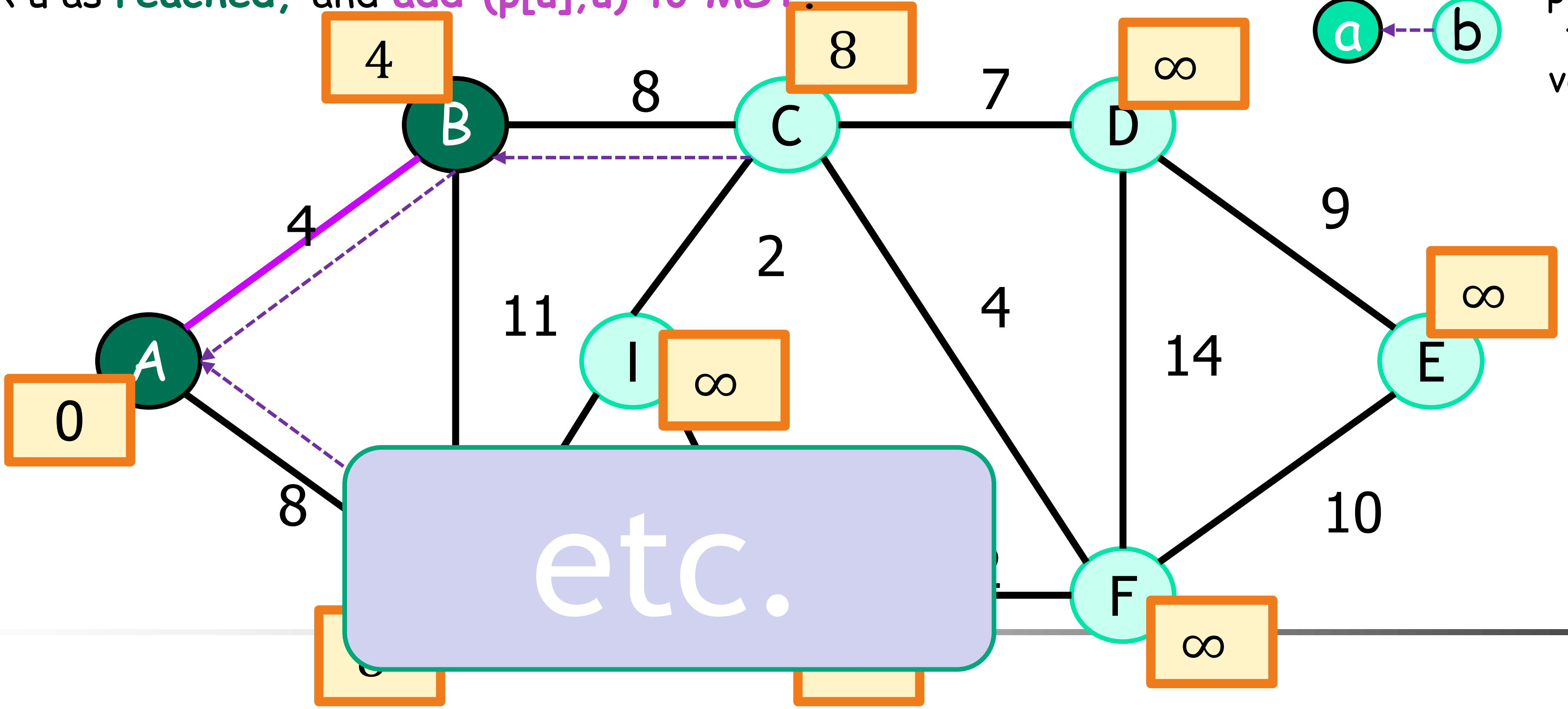
p[b] = a, meaning that a was the vertex that k[b] comes from.

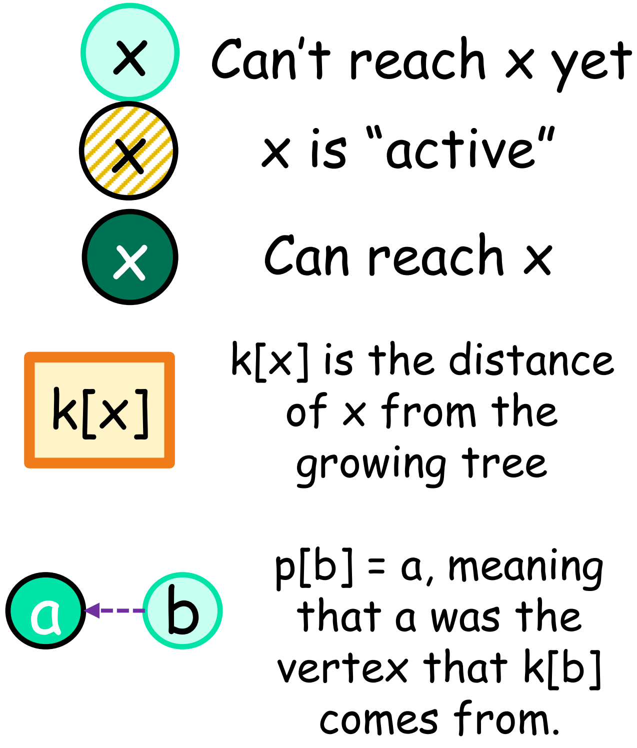
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached**:

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  k[v] = min(k[v], weight(u,v))
  if k[v] updated, p[v] = u
- Mark u as **reached**, and **add (p[u],u) to MST**.





Prim's algorithm

```
Prim( G = (V,E), starting vertex s):
    key(v) = ∞, ∀v ∈ V
    key(s) = 0
    Q = (key(v), v), ∀v ∈ V
    p(v) = NULL, ∀v ∈ V
    A = ∅
    while Q is not empty:
        u = ExtractMin(Q)
        if u != s then
            A = A ∪ {(p(u), u)}
        for each neighbor v of u:
            if v ∈ Q and w(u, v) < key(v):
                key(v) = w(u, v)
                DecreaseKey(key(v), v)
                p(v) = u
return A
```

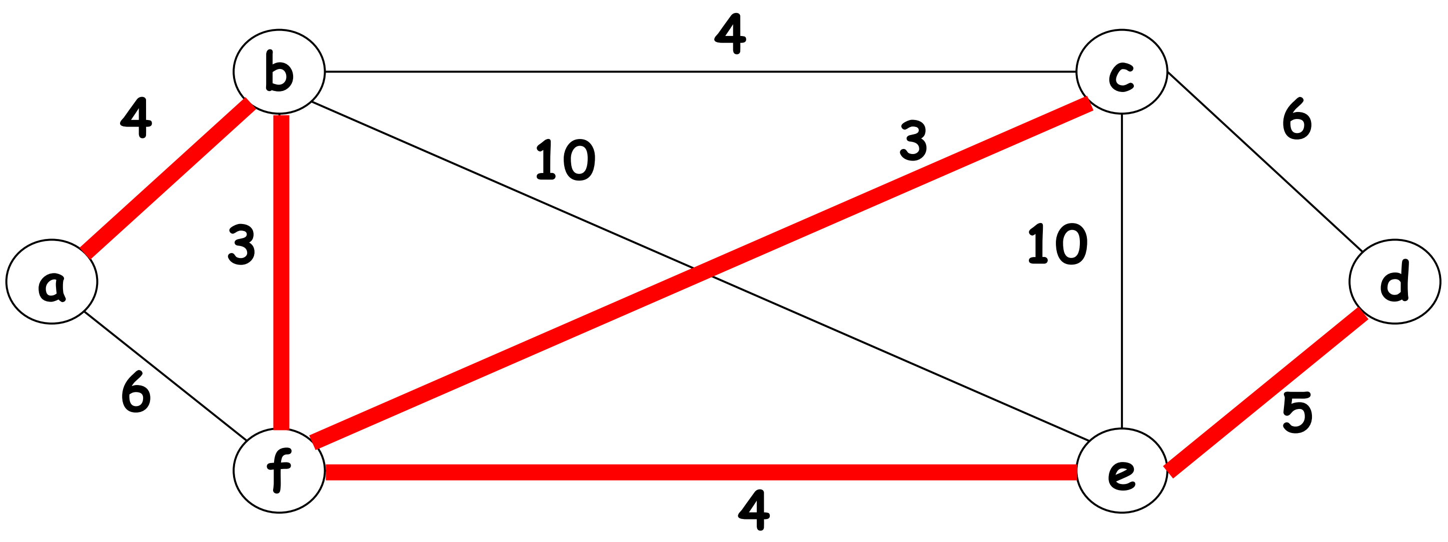
If a binary min-heap is used:

ExtractMin: $O(\log n)$

DecreaseKey: $O(\log n)$

Total: $O(n\log n + m\log n) = O(m\log n)$

Exercise – Find MST for this graph



order of (edges) selection: (a,b), (b,f), (c,f), (f,e), (e,d)

Prim and Kruskal

- Both Prim and Kruskal are greedy algorithms for MST.

- Kruskal:
  - Grows a forest.
  - Time O(mlogn) with a disjoint-set data structure

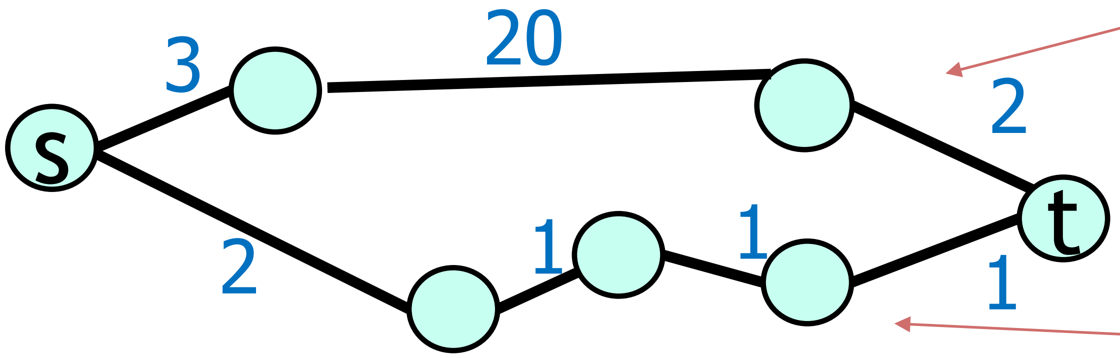
- Prim:
  - Grows a tree.
  - Time O(mlogn) with a binary min-heap
  - Time O(m + nlogn) with a Fibonacci heap*

* Refer to textbook Chapter 19

Shortest path

Shortest path problem

- What is the shortest path between u and v in a weighted graph?
  - the **cost** of a path is the sum of the weights along that path
  - The **shortest path** is the one with the minimum cost.



This path from s to t has cost 25.

This path is shorter, it has cost 5.

- The **distance** d(u,v) between two vertices u and v is the cost of the the shortest path between u and v.

Single-source shortest-paths

Consider a directed connected graph G

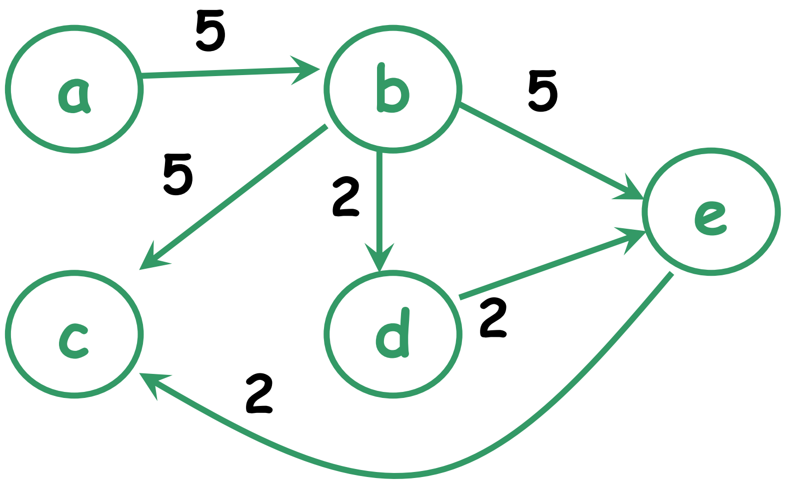
- The edges are labelled by weight

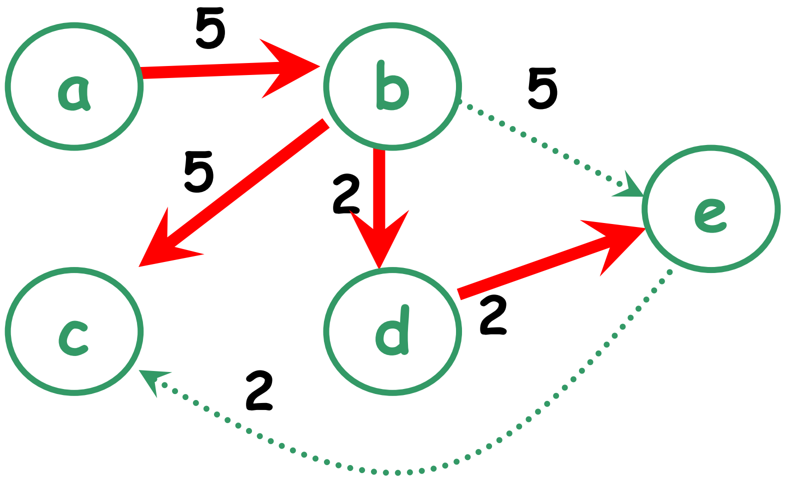
Given a particular vertex called the ***source***

- Find **shortest paths** from the source to all other vertices (shortest path means the total weight of the path is the smallest)

Example

Directed Graph G
(edge label is weight)
a is source vertex

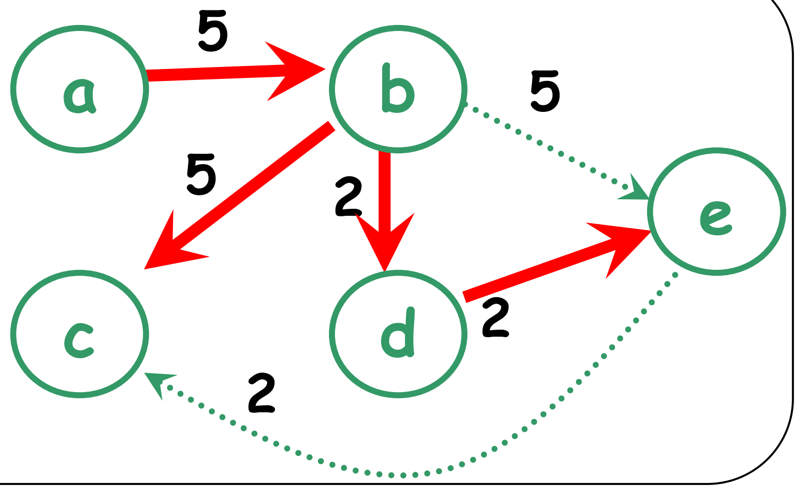




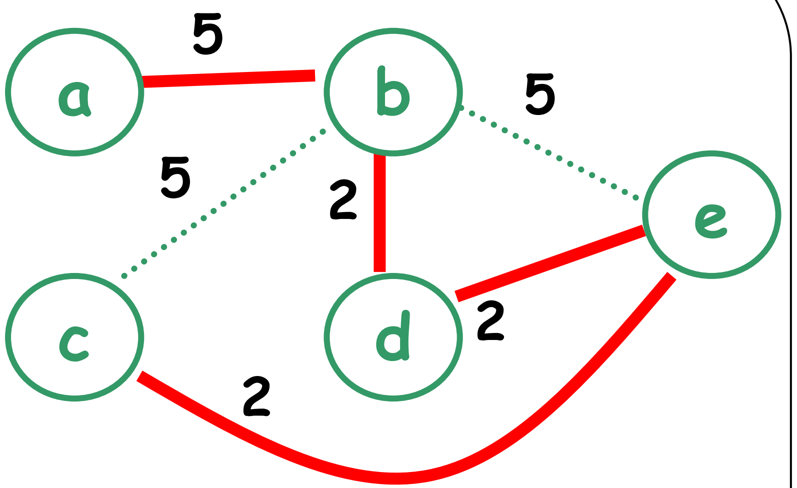
thick lines: shortest path
dotted lines: not in shortest path

Single-source shortest paths vs MST

Shortest paths from a



What is the difference between MST and shortest paths from a?

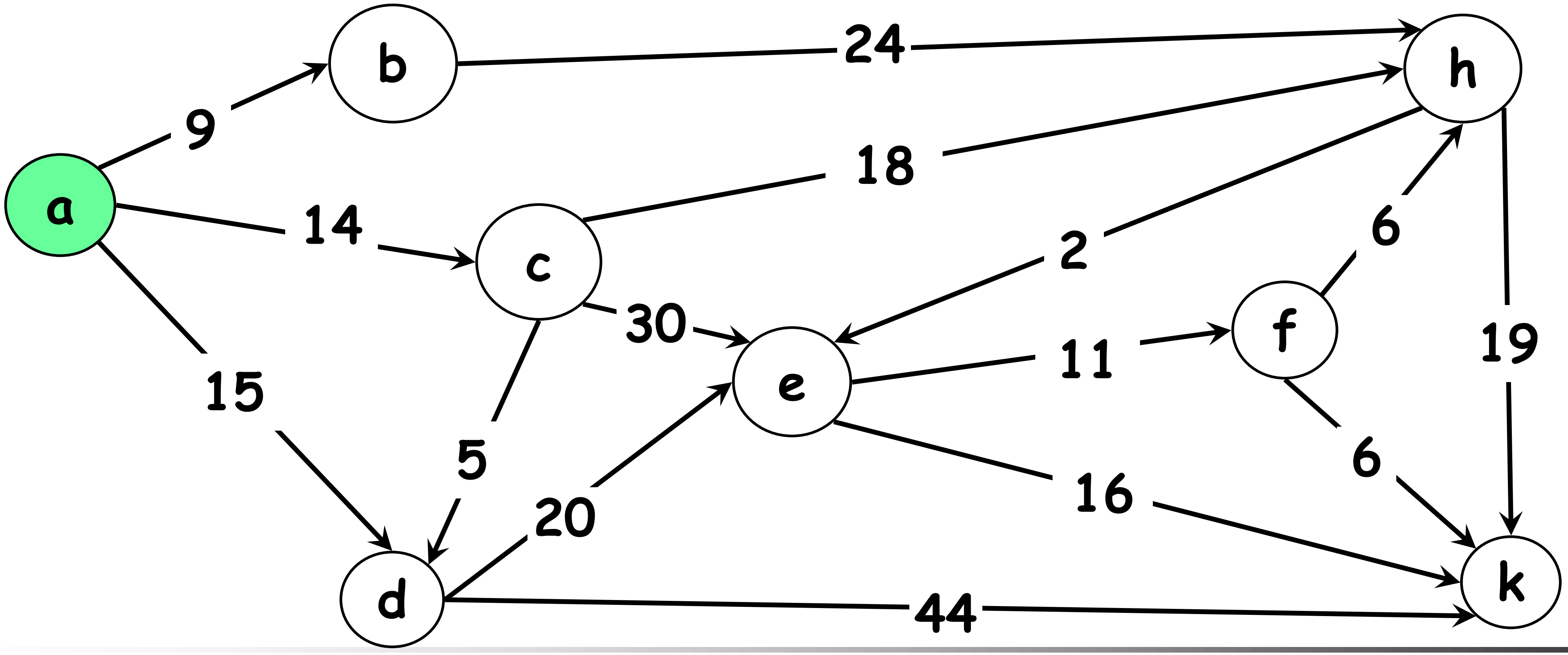


MST

Dijkstra algorithm

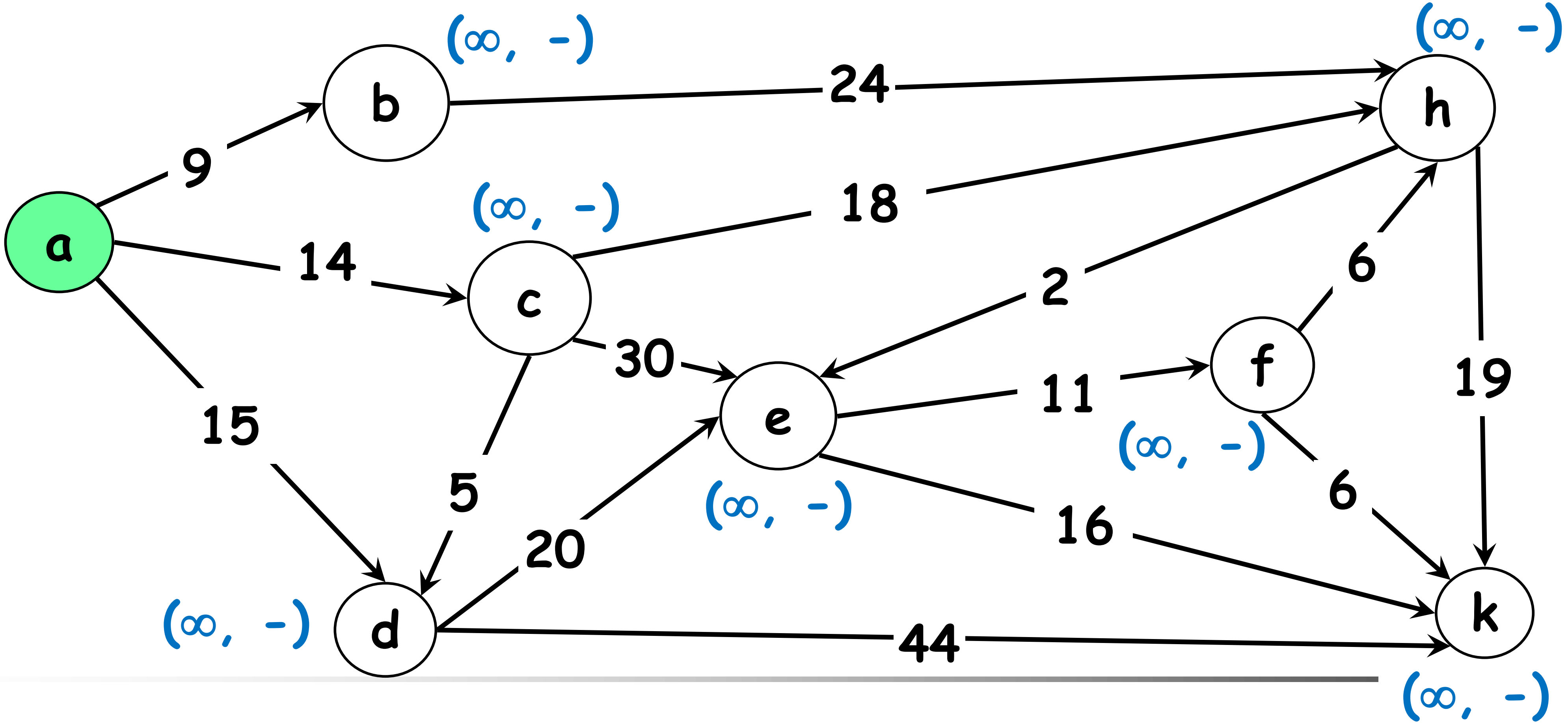
Dijkstra's algorithm

Suppose vertex *a* is the source, we now show how Dijkstra's algorithm works



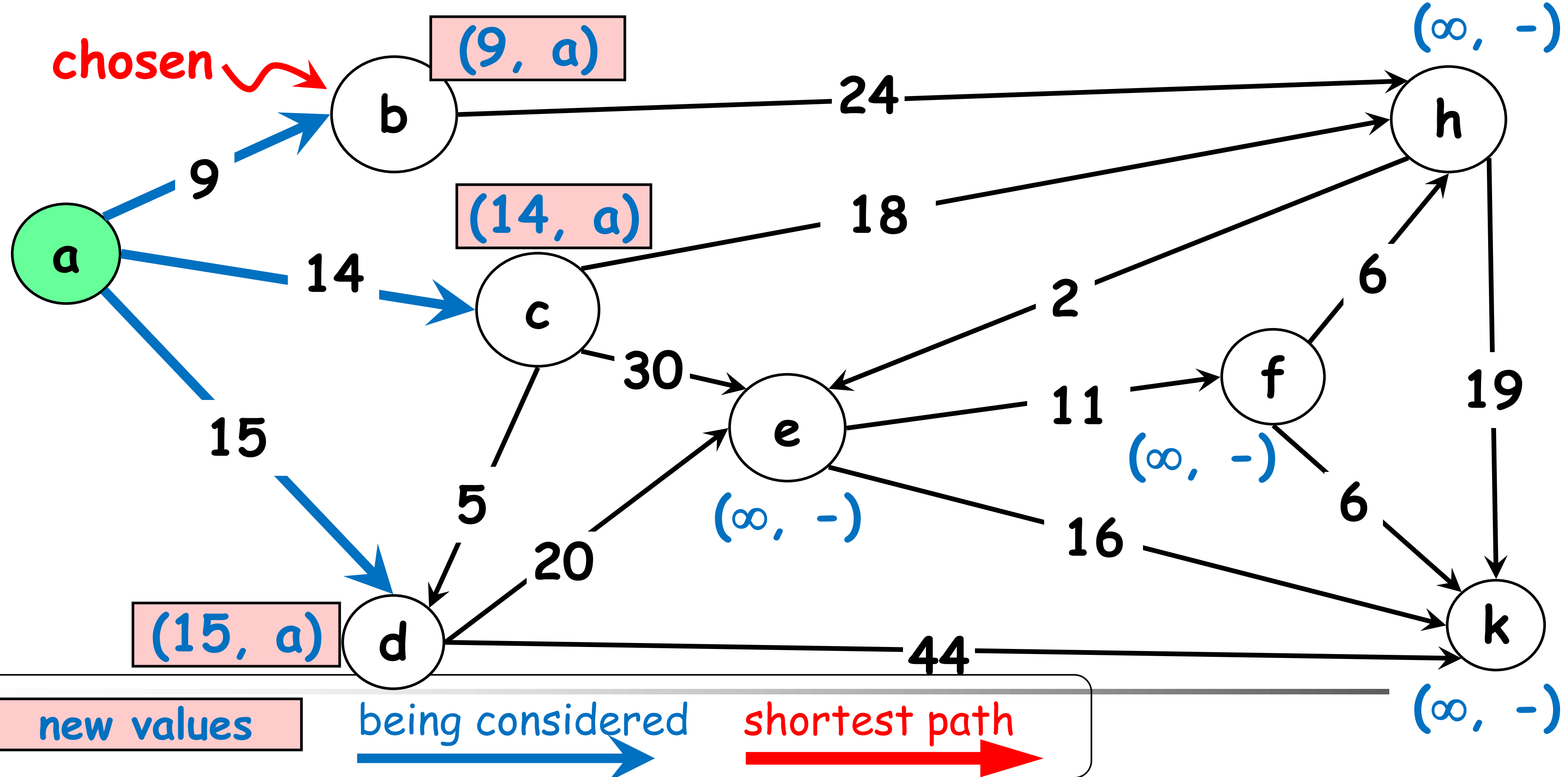
Dijkstra's algorithm

Every vertex *v* keeps 2 labels: (1) the weight of the current shortest path from *a*; (2) the vertex leading to *v* on that path, initially as (∞, -)



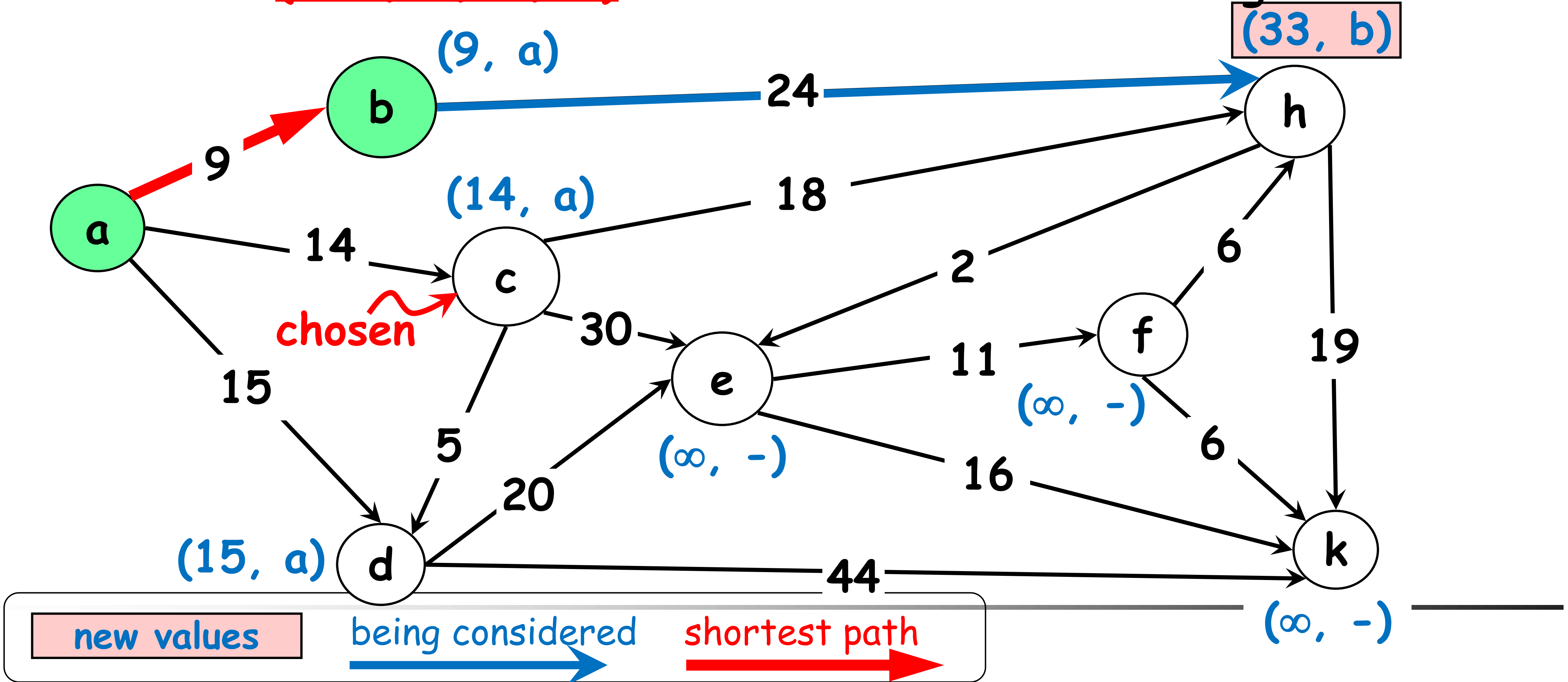
Dijkstra's algorithm

For every neighbor *u* of *a*, update the weight to the weight of *(a,u)* and the leading vertex to *a*. Choose from b, c, d the one with the smallest such weight.



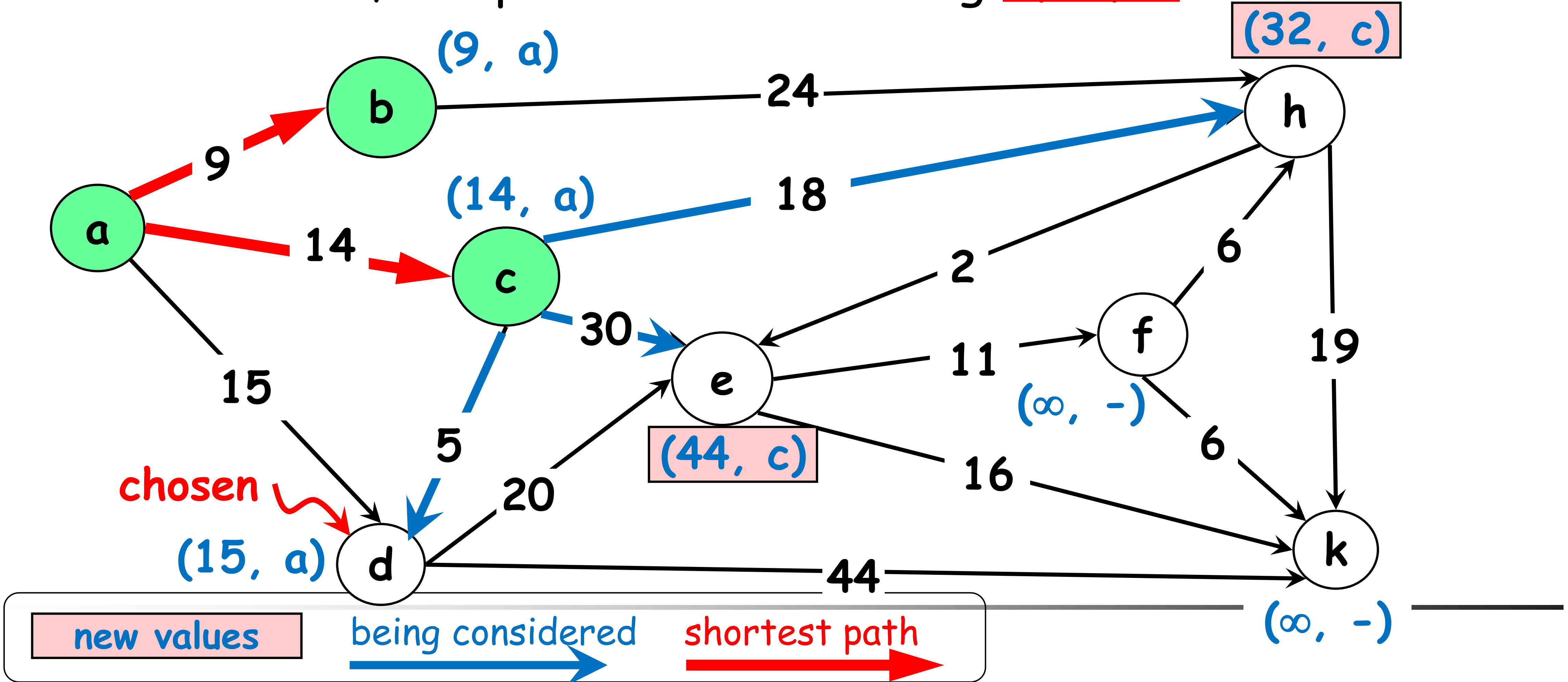
Dijkstra's algorithm

For every un-chosen neighbor of vertex **b**, update the weight and leading vertex. Choose from **ALL** un-chosen vertices (i.e., c, d, h) the one with smallest weight.



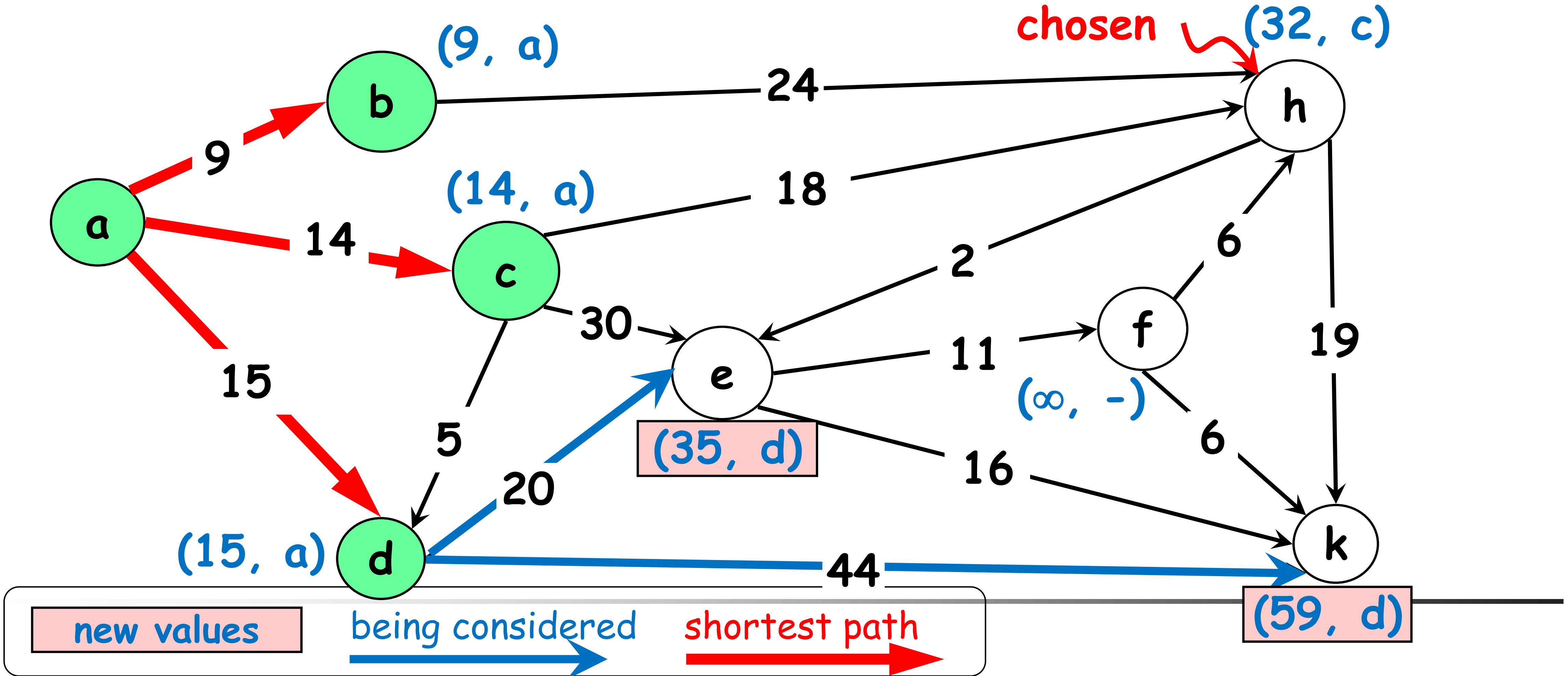
Dijkstra’s algorithm

If a new path with smallest weight is discovered, e.g., for vertices *e*, *h*, the weight is updated. Otherwise, like vertex *d*, no update. Choose among <u>*d*, *e*, *h*</u>.



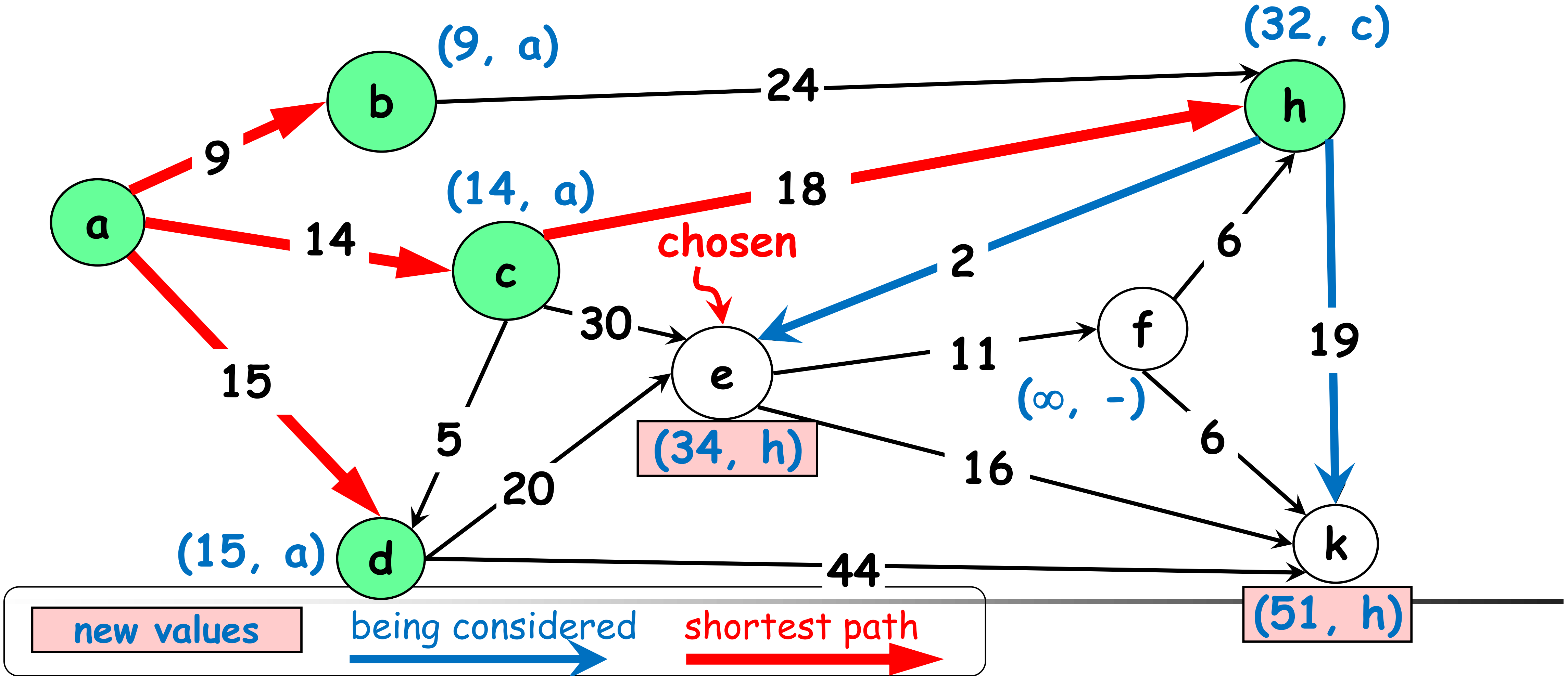
Dijkstra's algorithm

Repeat the procedure. After **d** is chosen, the weight of **e** and **k** is updated. Choose among <u>**e, h, k**</u>. Next vertex chosen is **h**.



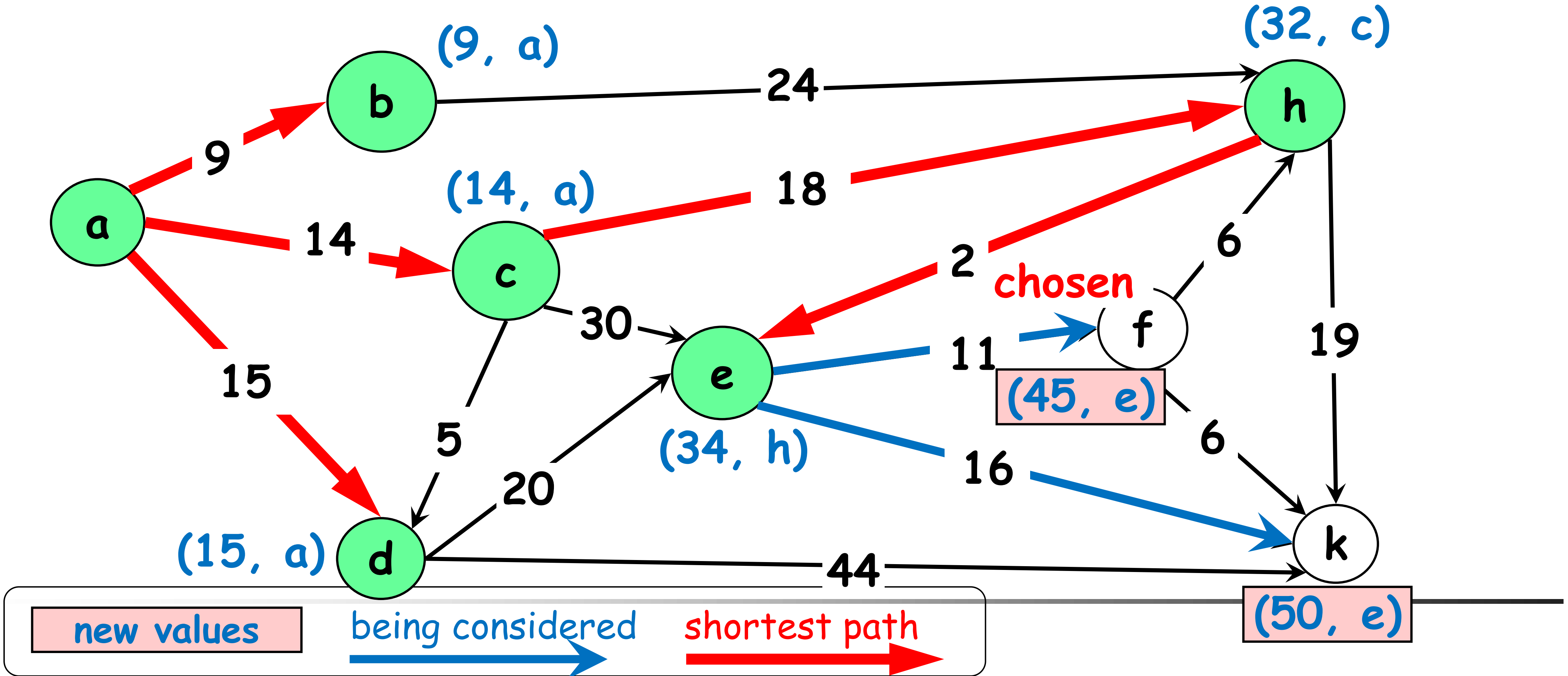
Dijkstra's algorithm

After ***h*** is chosen, the weight of ***e*** and ***k*** is updated again. Choose among ***e***, ***k***. Next vertex chosen is ***e***.



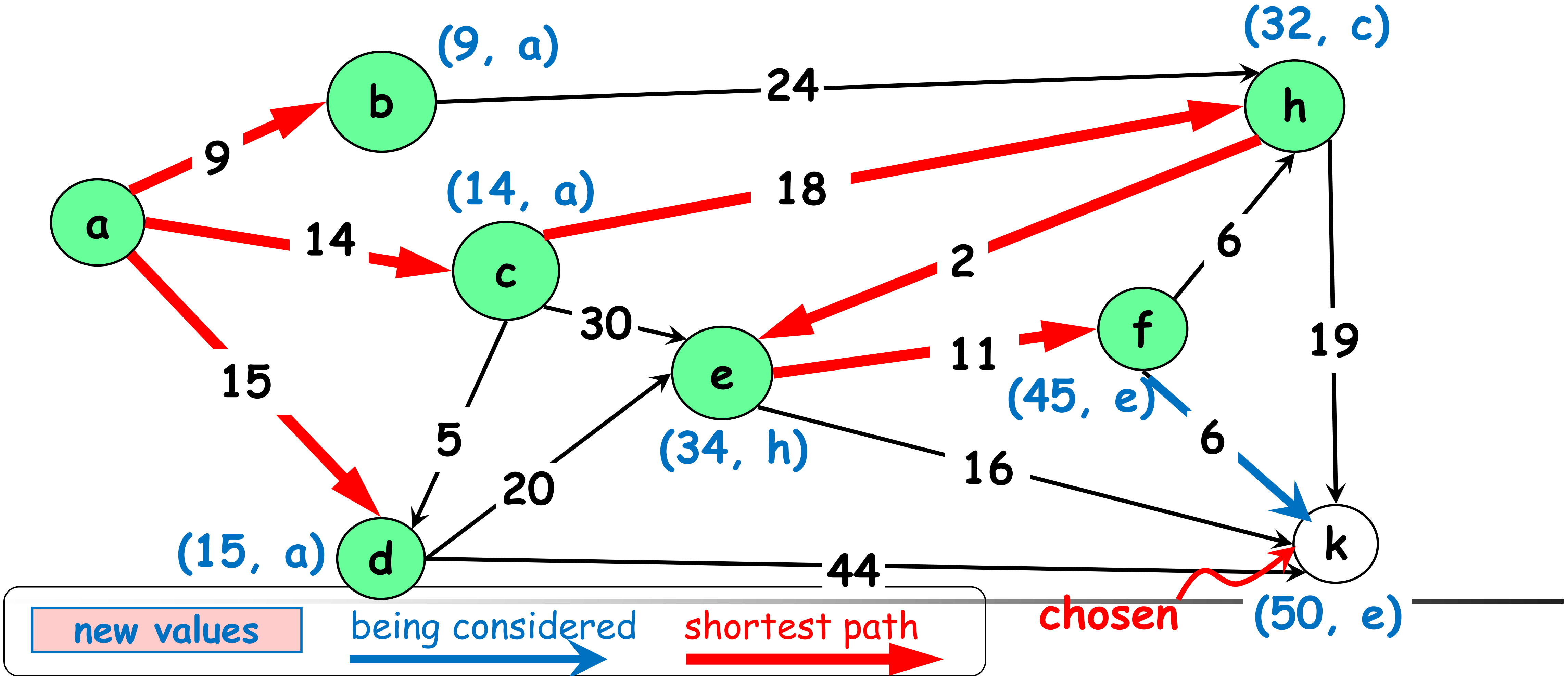
Dijkstra's algorithm

After *e* is chosen, the weight of *f* and *k* is updated again. Choose among *f, k*. Next vertex chosen is *f*.



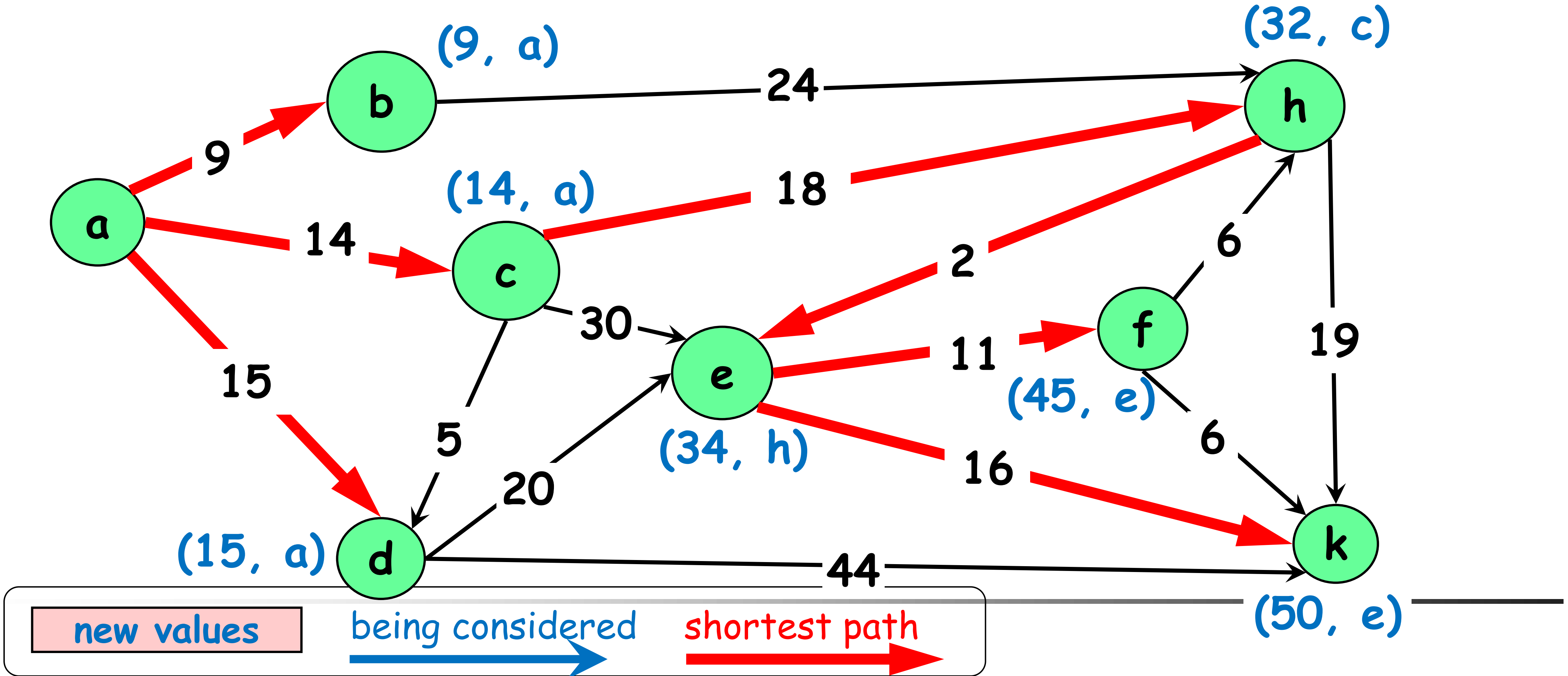
Dijkstra's algorithm

After *f* is chosen, it is NOT necessary to update the weight of *k*. The final vertex chosen is *k*.

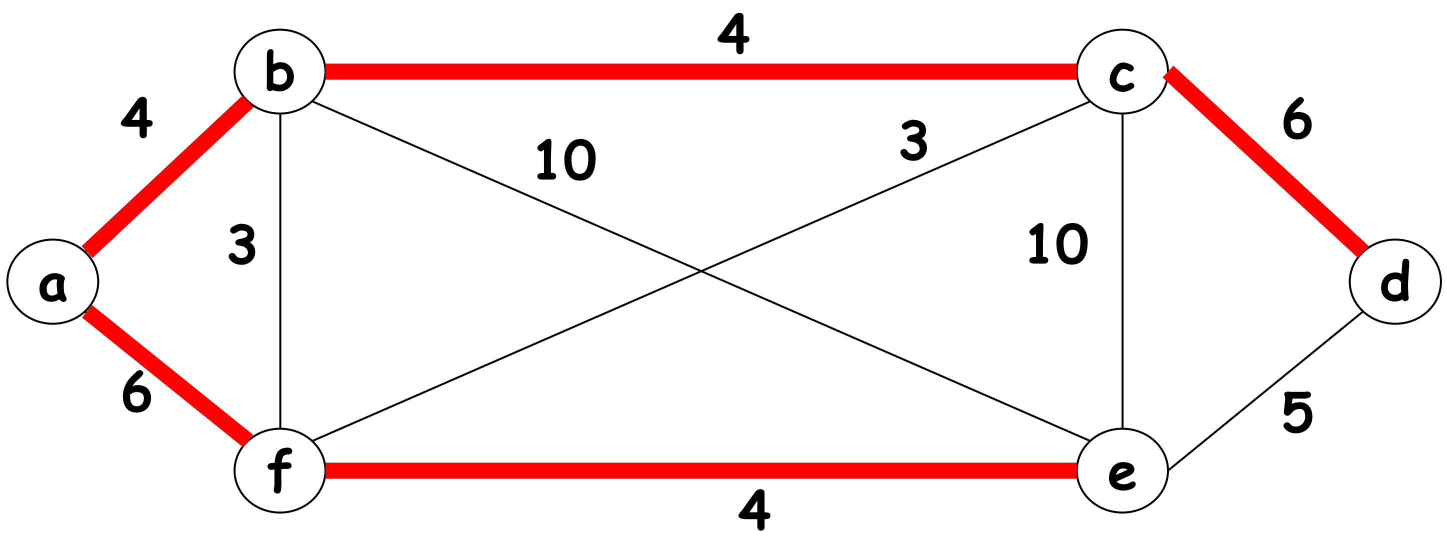


Dijkstra's algorithm

At this point, all vertices are chosen, and the shortest path from *a* to every vertex is discovered.



Exercise – Shortest paths from *a*



Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	~~∞~~ 4	~~-~~ a
c	~~∞~~ 8	~~-~~ b
d	~~∞~~ 14	~~-~~ c
e	~~∞~~ ~~14~~ 10	~~-~~ ~~b~~ f
f	~~∞~~ 6	~~-~~ a

order of (edges) selection: (a,b), (a,f), (b,c), (f,e), (c,d)

Dijkstra’s algorithm

```
Dijkstra( G = (V,E), starting vertex s):
    key(v) = ∞, ∀v ∈ V
    key(s) = 0
    Q = (key(v), v), ∀v ∈ V
    p(v) = NULL, ∀v ∈ V
    A = ∅
    while Q is not empty:
        u = ExtractMin(Q)
        if u != s then
            A = A ∪ {(p(u), u)}
        for each neighbor v of u:
            if v ∈ Q and key(v) > key(u) + w(u, v) :
                key(v) = key(u) + w(u, v)
                DecreaseKey(key(v), v)
                p(v) = u
return A
```

If a binary min-heap is used:
ExtractMin: $O(\log n)$
DecreaseKey: $O(\log n)$
Total: $O(n\log n+m\log n)=O(m\log n)$

Bellman-Ford Algorithm

Bellman-Ford Algorithm

- Basic idea:
  - Instead of picking the u with the smallest key(u) to update, just update all of the u's by relaxation.
- Can handle negative edge weights.
- Slower than Dijkstra's algorithm

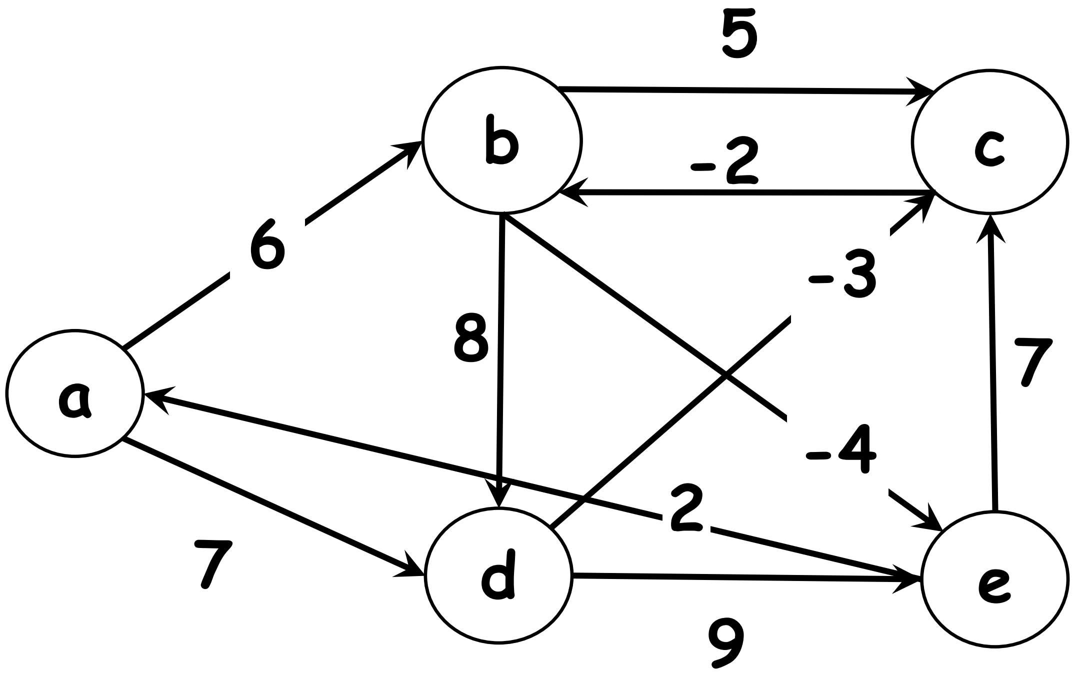
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 1

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	∞	-
c	∞	-
d	∞	-
e	∞	-



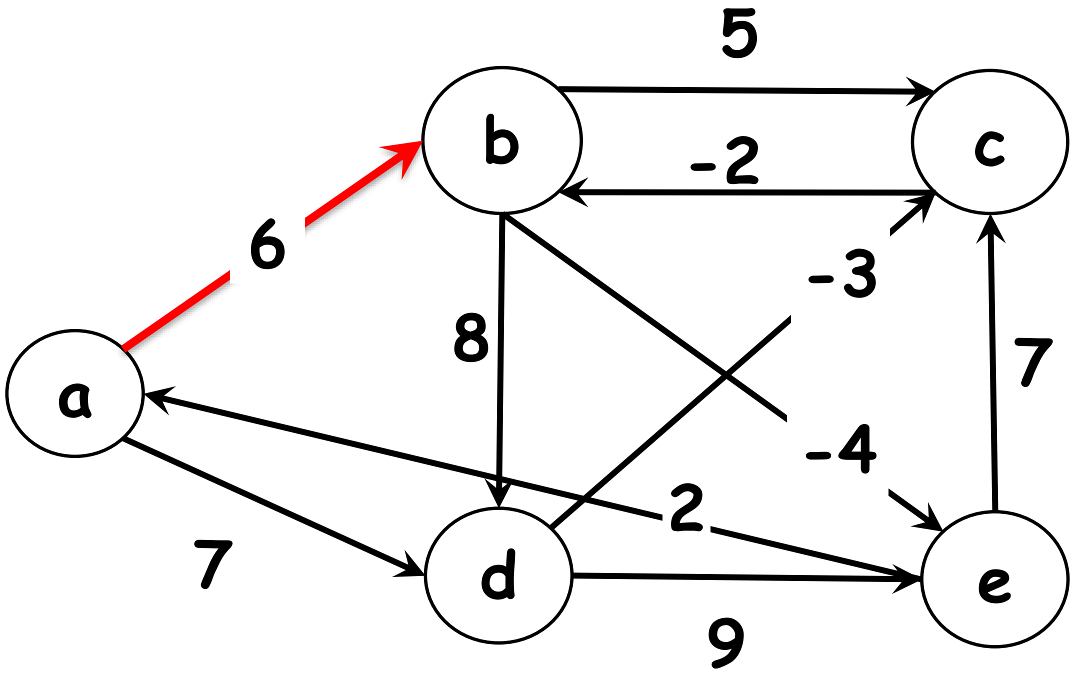
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 1

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	∞	-
d	∞	-
e	∞	-

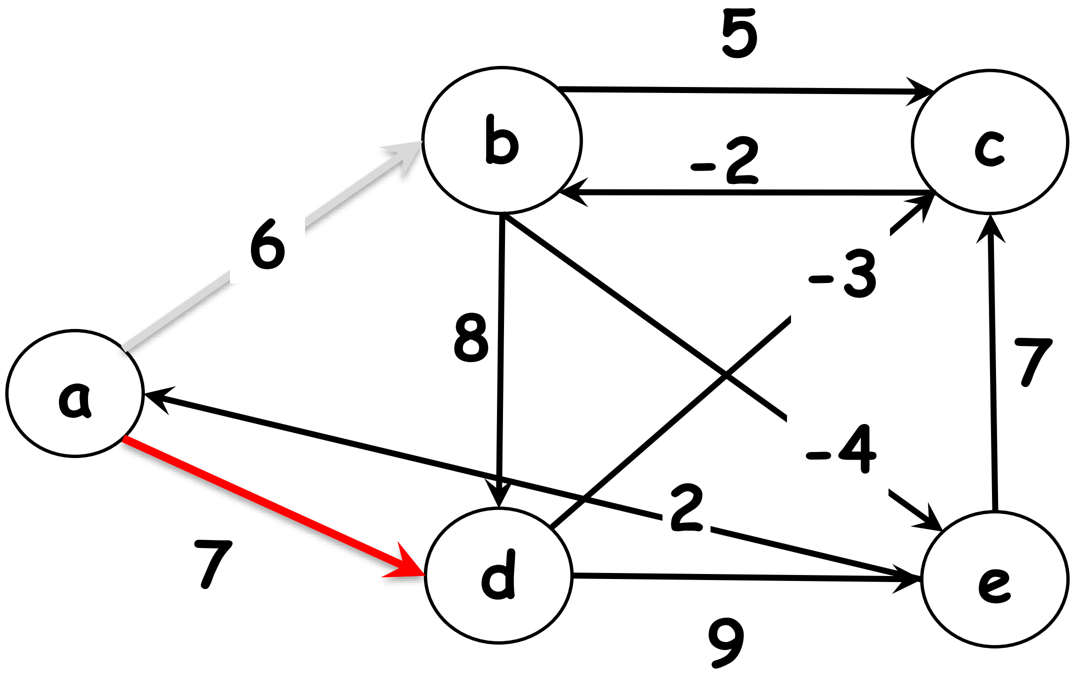


Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 1   E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	∞	-
d	7	a
e	∞	-



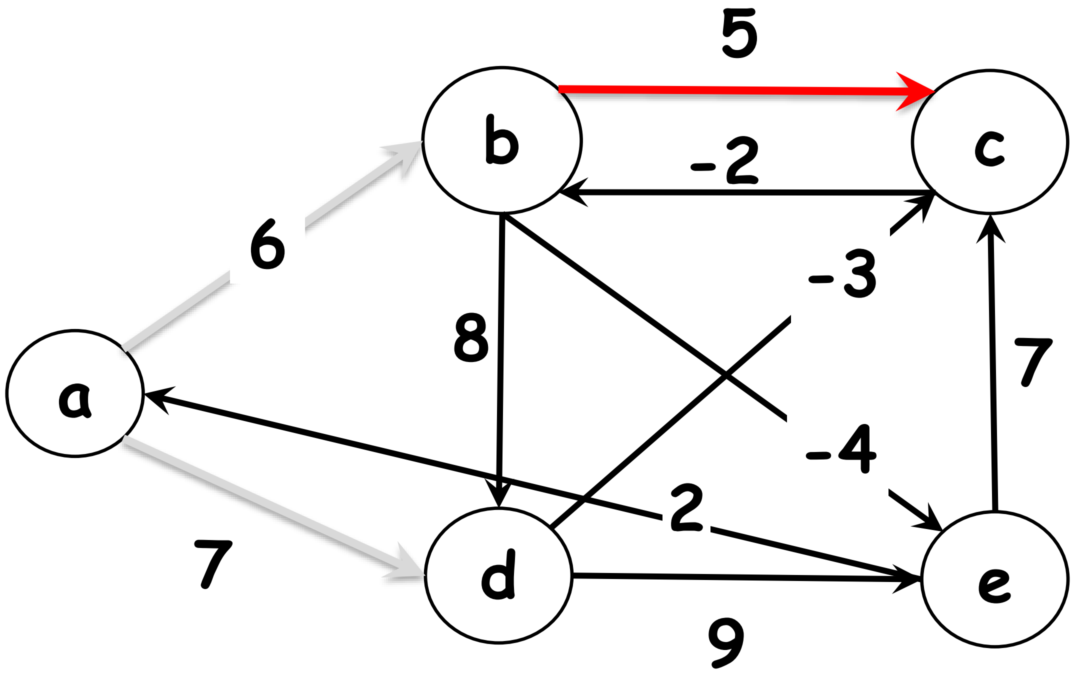
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 1

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	11	b
d	7	a
e	∞	-



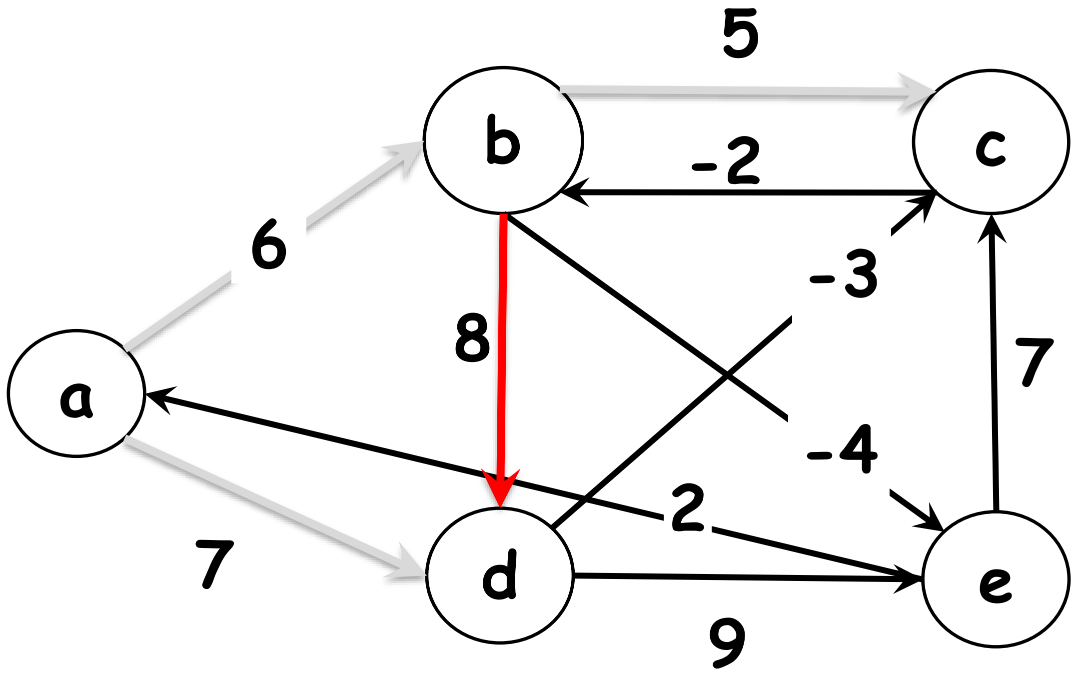
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 1

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	11	b
d	7	a
e	∞	-



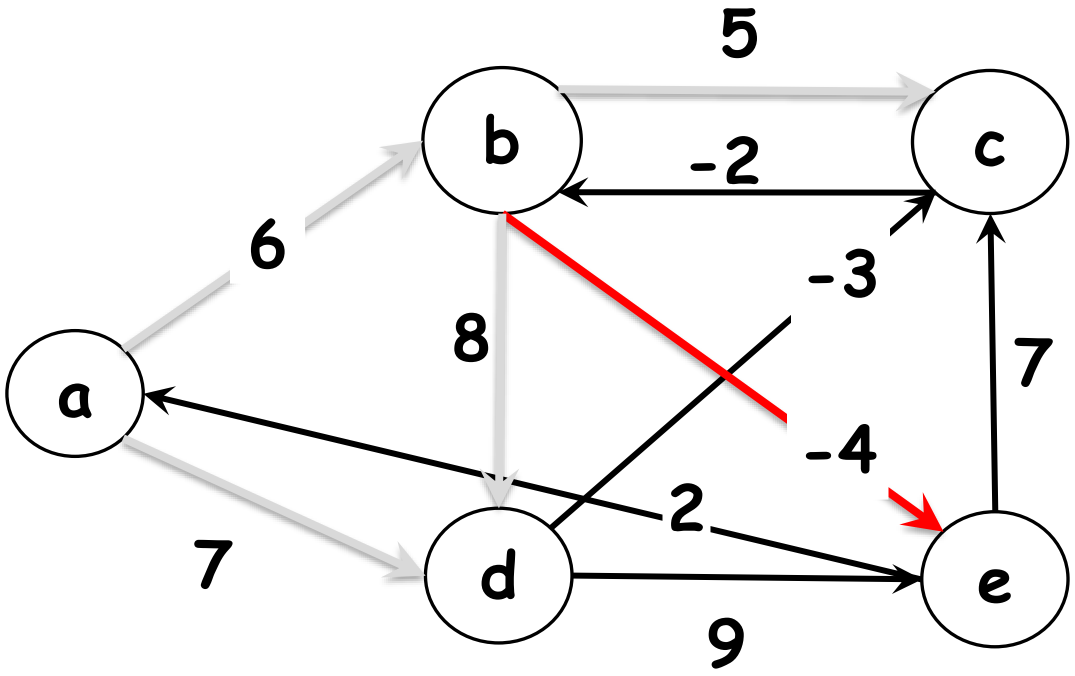
Bellman-Ford Algorithm

```
for i=1,...,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 1

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	11	b
d	7	a
e	2	b



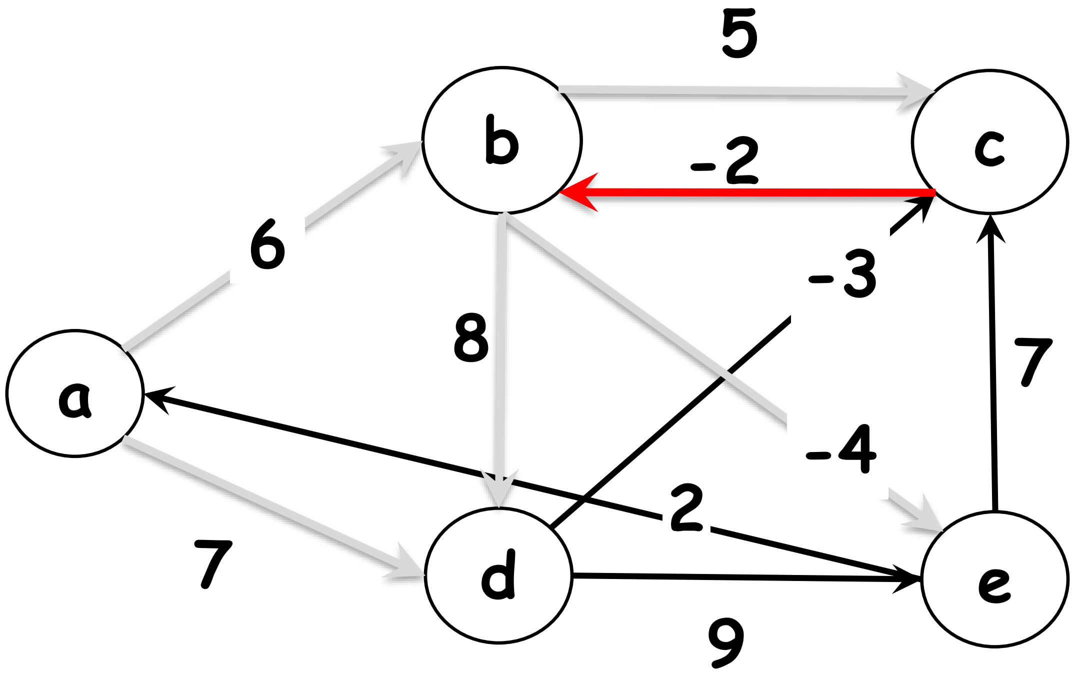
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 1

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	11	b
d	7	a
e	2	b



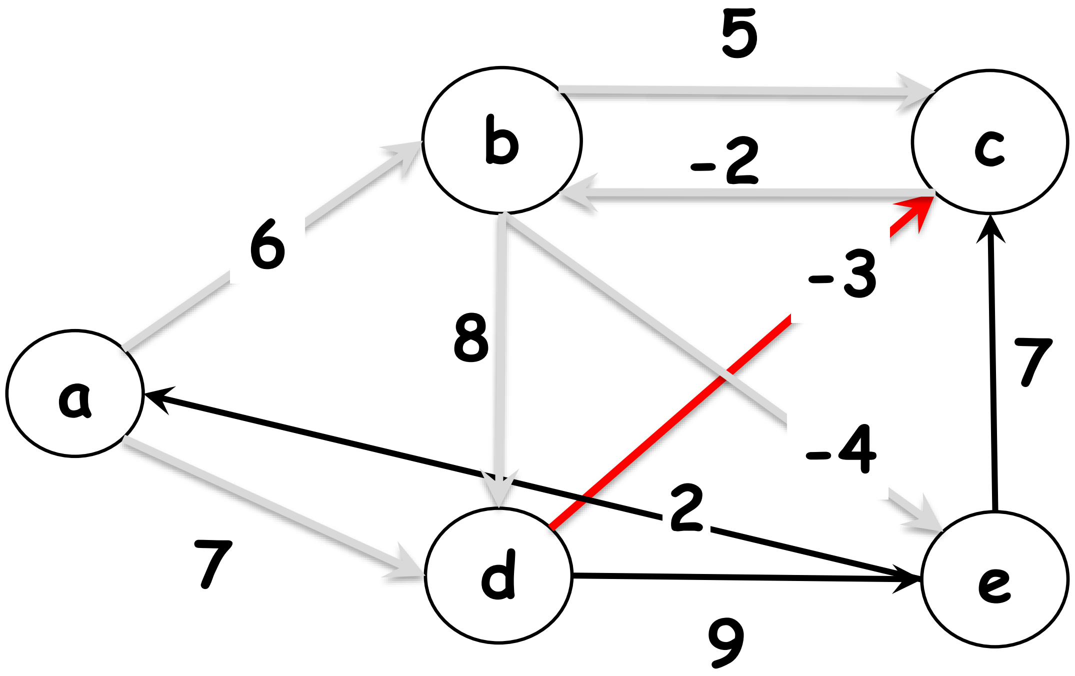
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 1

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	4	d
d	7	a
e	2	b

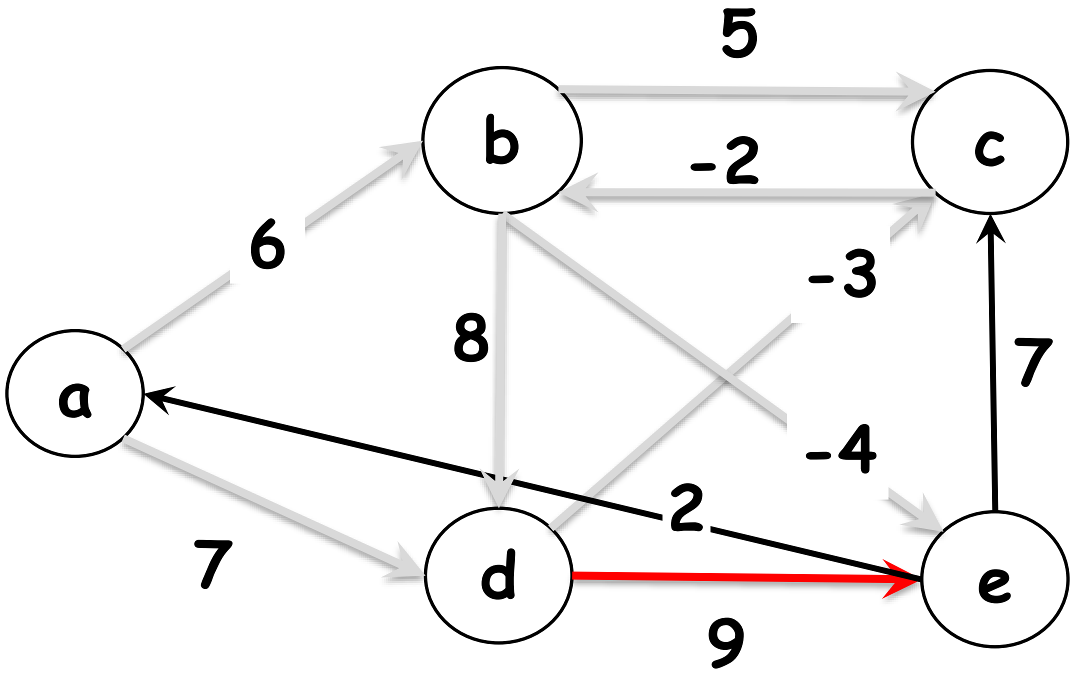


Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 1   E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e), (e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	4	d
d	7	a
e	2	b



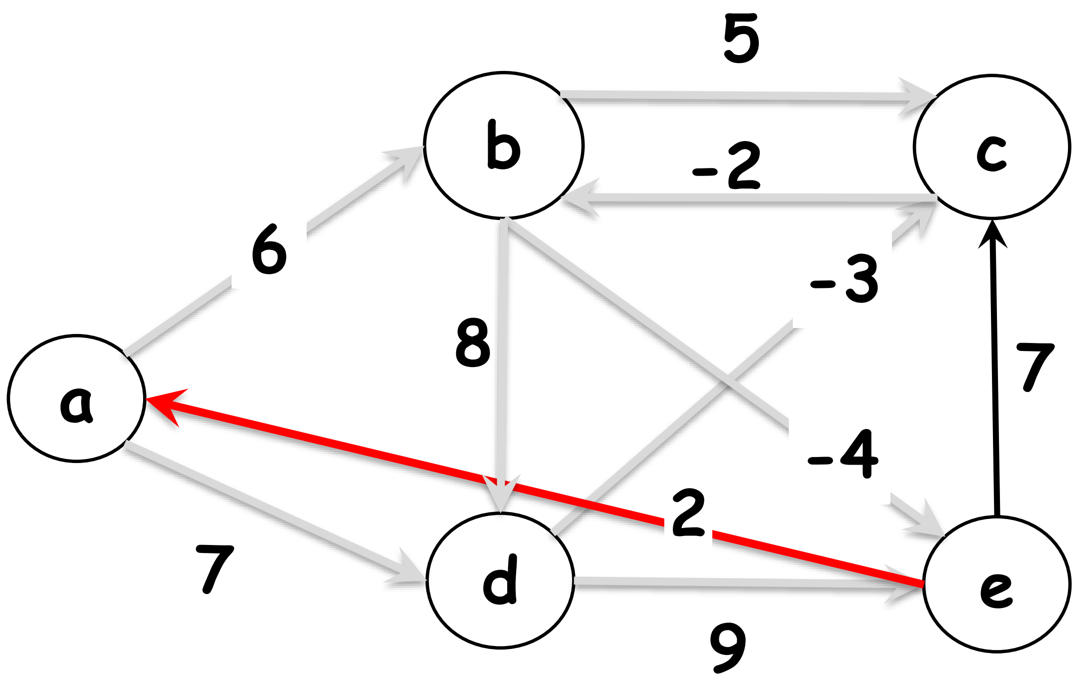
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 1

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e), (e,a), (e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	4	d
d	7	a
e	2	b



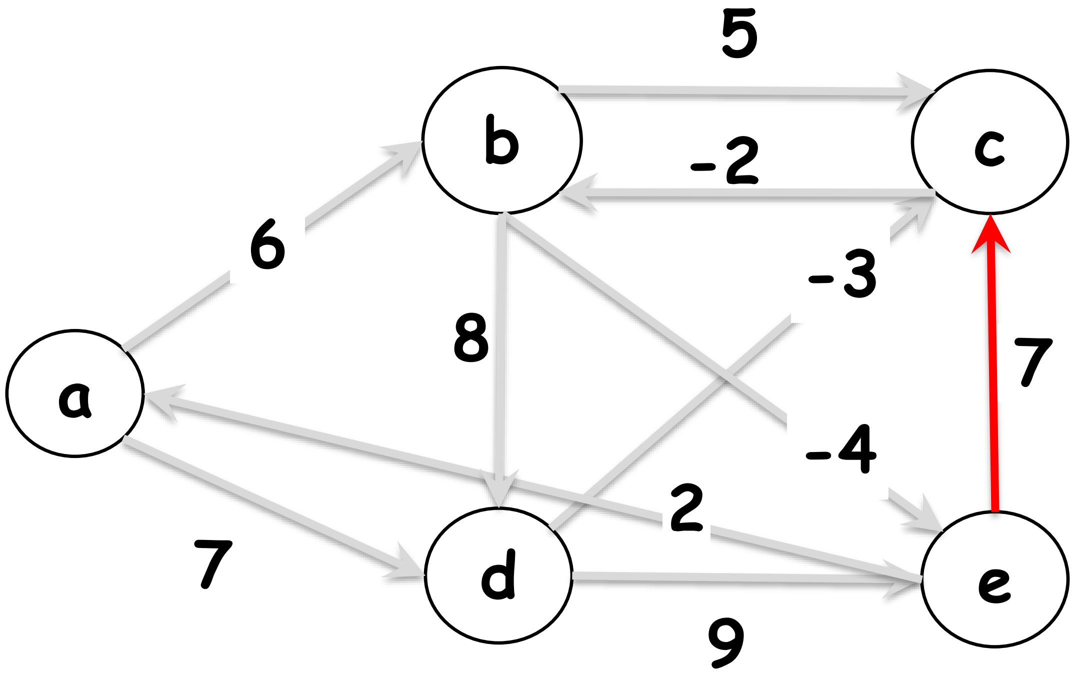
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 1

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e), (e,a), (e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	4	d
d	7	a
e	2	b



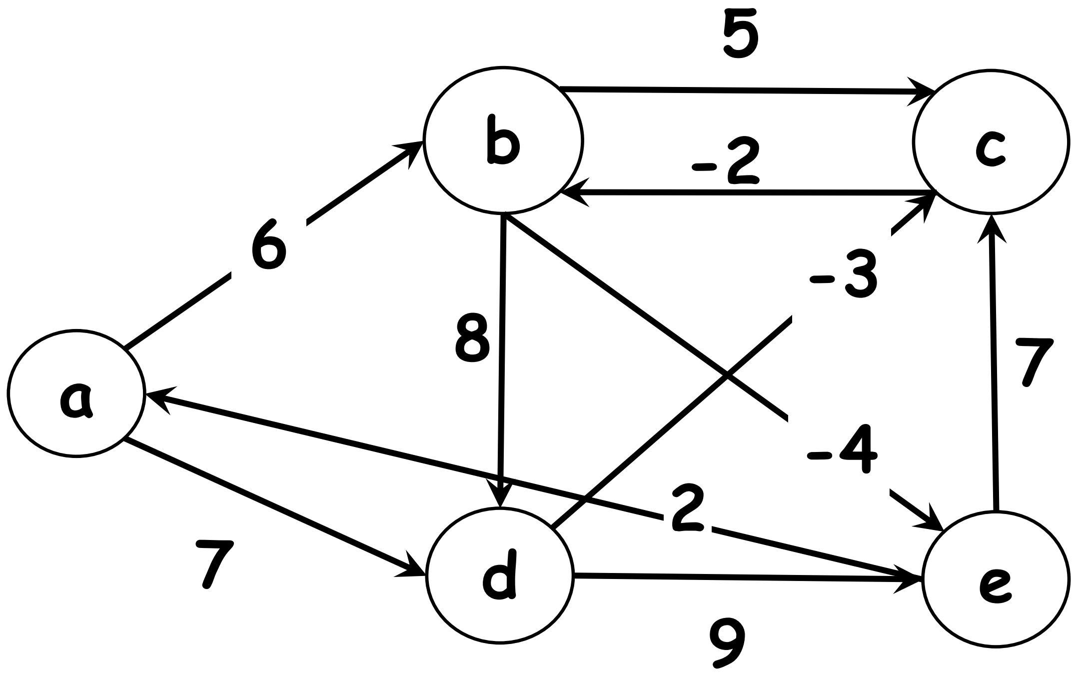
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 2

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	4	d
d	7	a
e	2	b



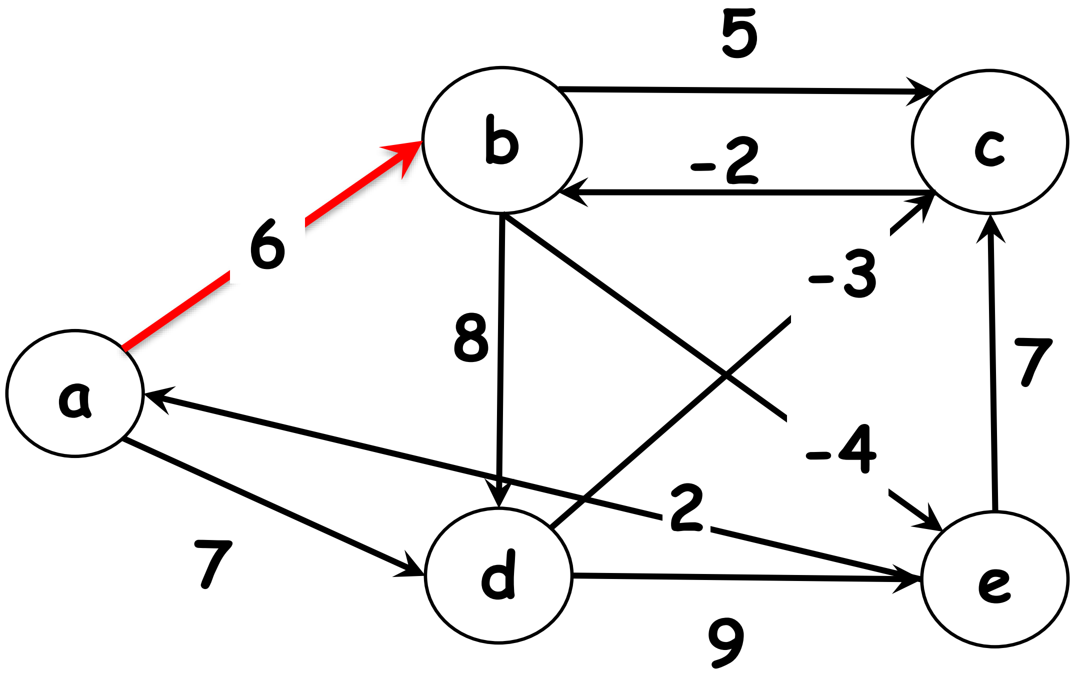
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 2

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	4	d
d	7	a
e	2	b



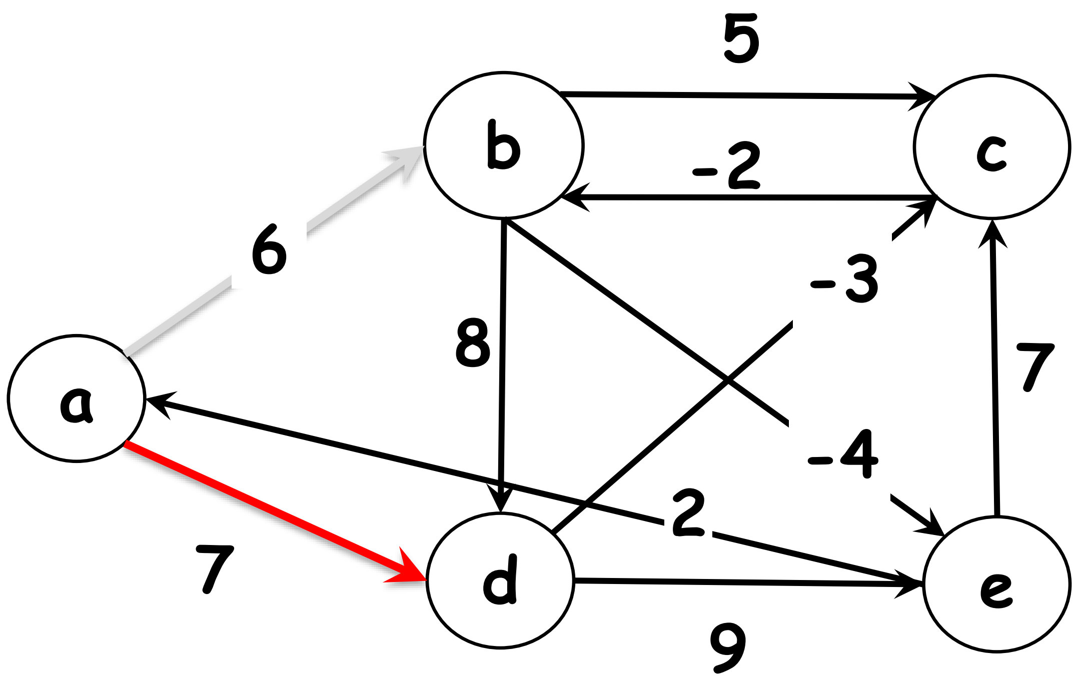
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 2

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	4	d
d	7	a
e	2	b



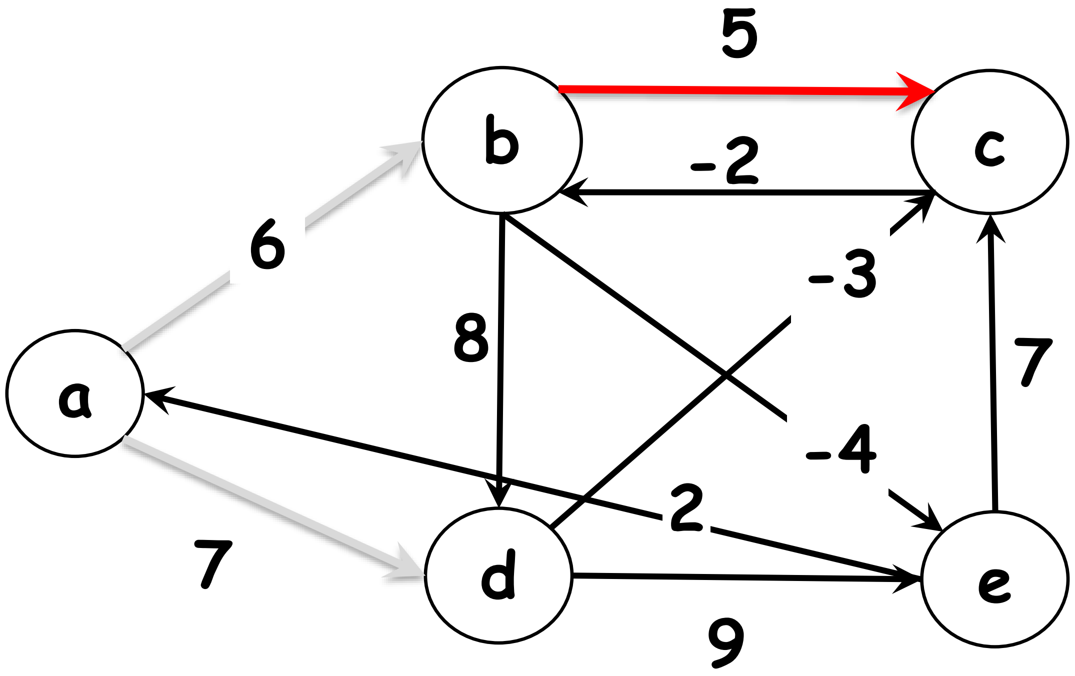
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 2

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	4	d
d	7	a
e	2	b



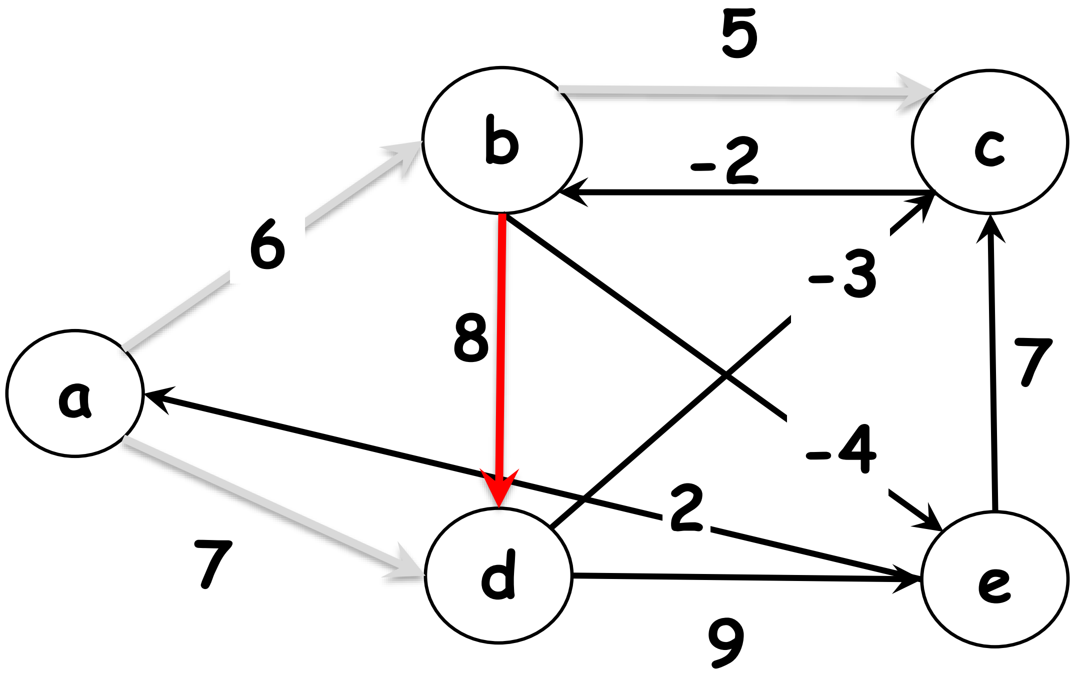
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 2

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	4	d
d	7	a
e	2	b

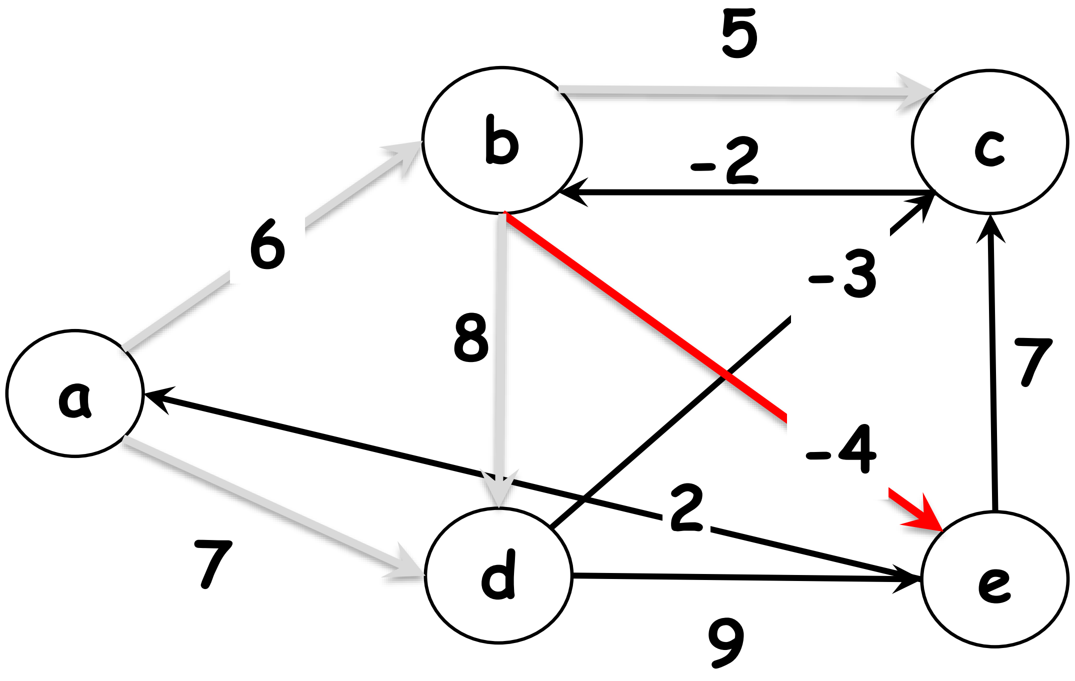


Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 2   E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	6	a
c	4	d
d	7	a
e	2	b



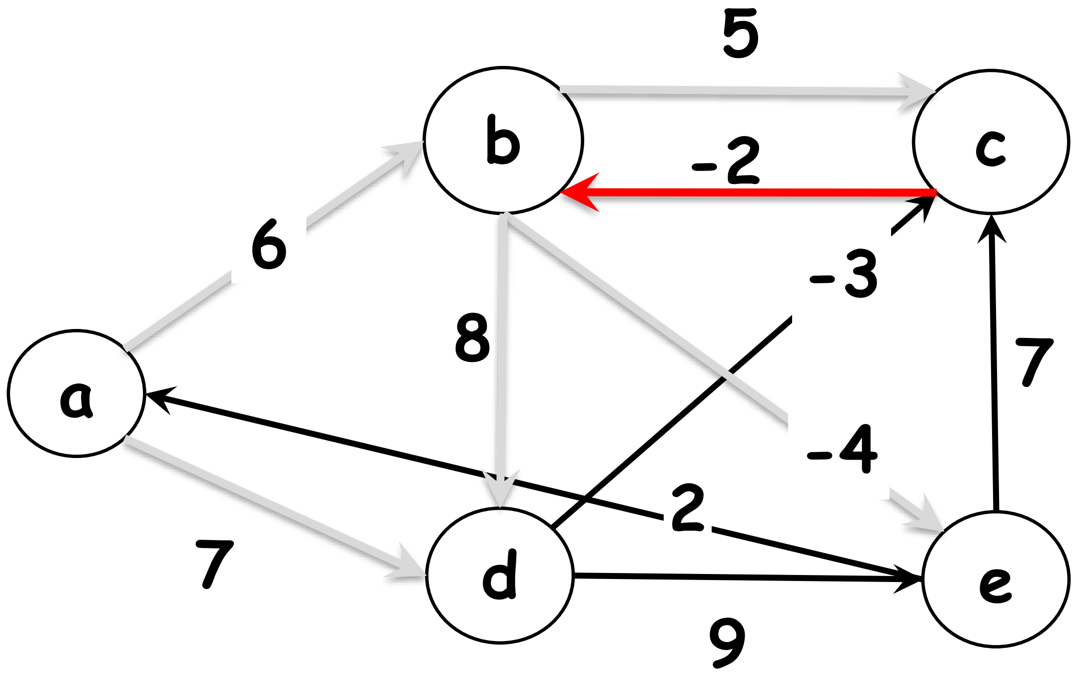
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 2

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	2	c
c	4	d
d	7	a
e	2	b



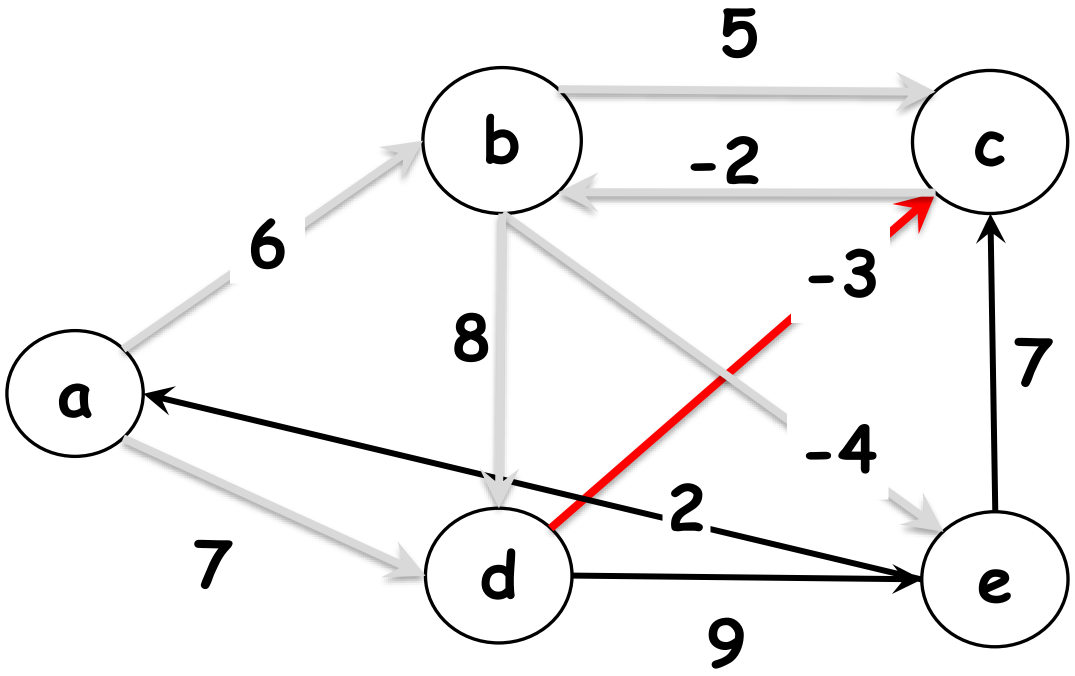
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 2

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e),(e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	2	c
c	4	d
d	7	a
e	2	b



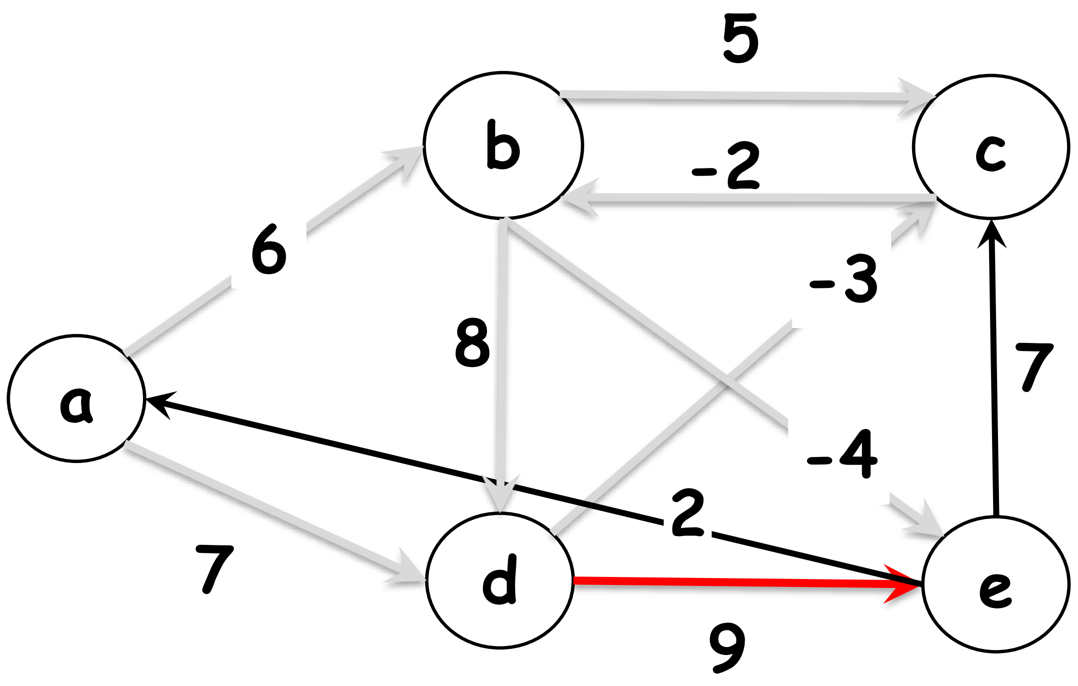
Bellman-Ford Algorithm

```
for i=1,...,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 2

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e), (e,a),(e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	2	c
c	4	d
d	7	a
e	2	b



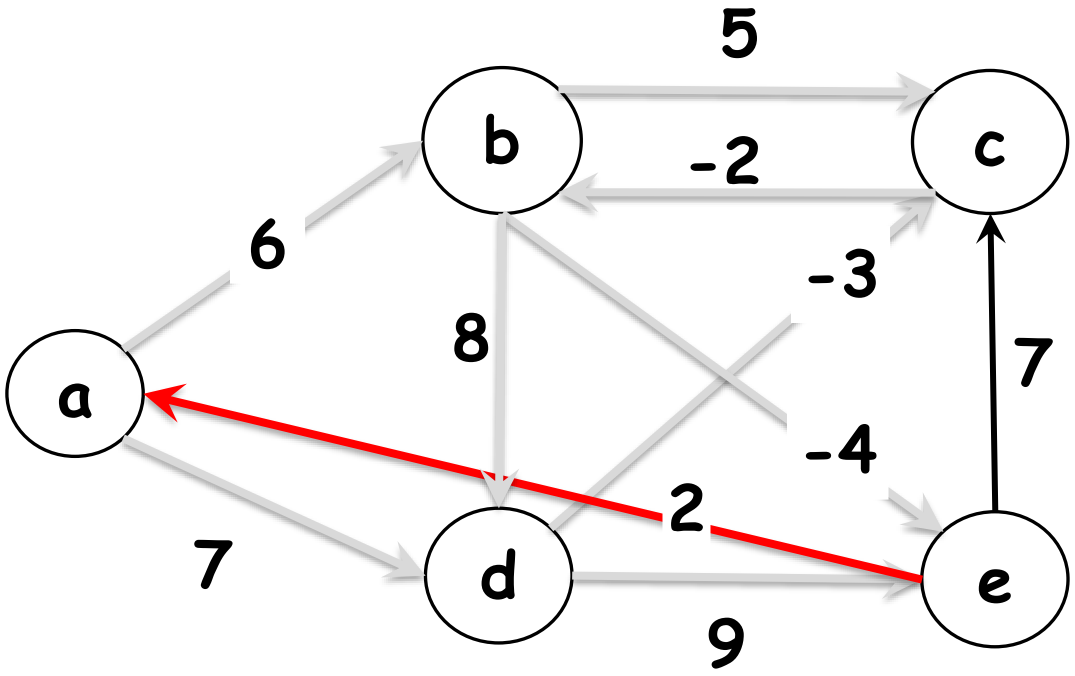
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 2

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e), (e,a), (e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	2	c
c	4	d
d	7	a
e	2	b



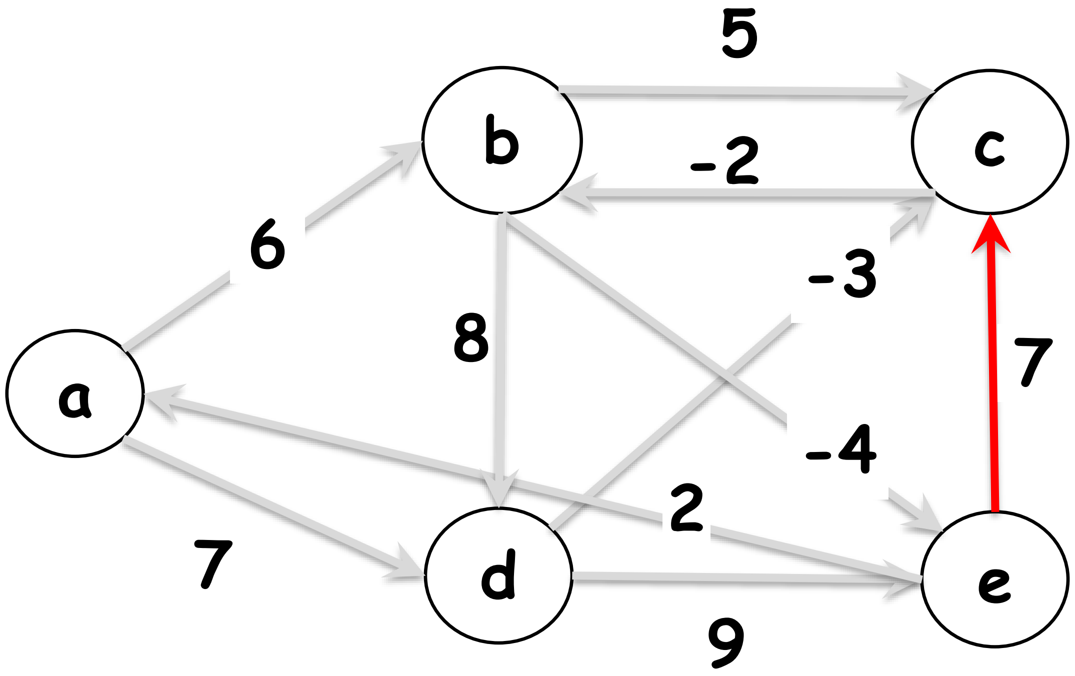
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
            key(v) = key(u) + w(u, v)
            p(v) = u
```

i = 2

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e), (e,a), (e,c)}

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	2	c
c	4	d
d	7	a
e	2	b



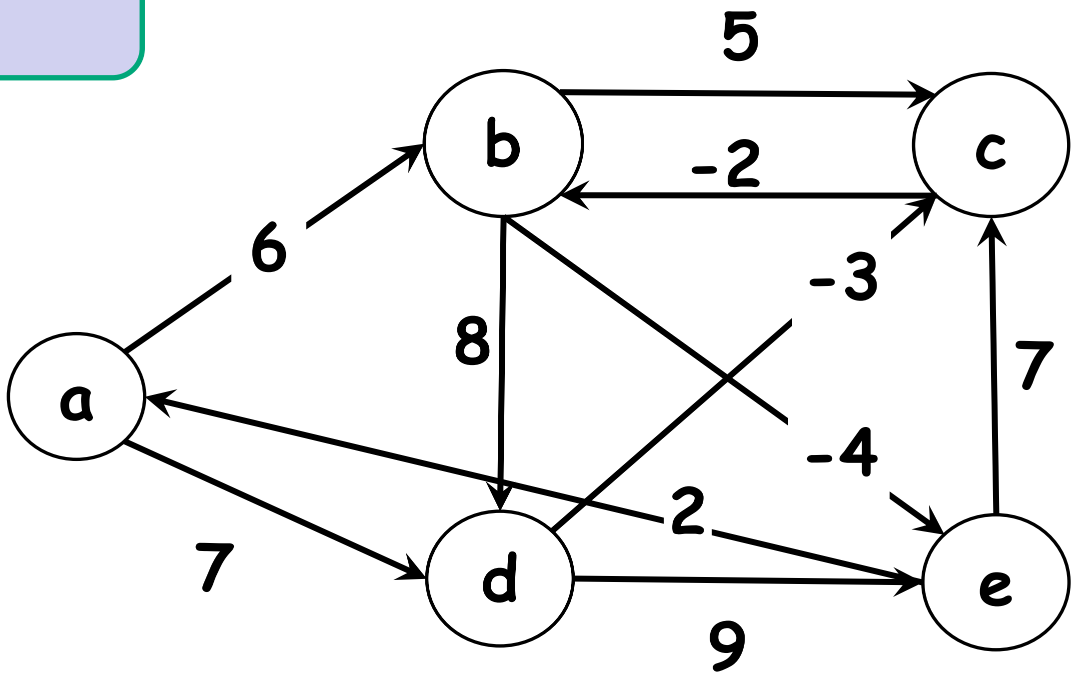
Bellman-Ford Algorithm

```
for i=1,…,n-1:
    for each edge (u,v) in E:
        if key(v) > key(u) + w(u, v) :
```

E = {(a,b), (a,d), (b,c), (b,d), (b,e), (c,b), (d,c), (d,e), (e,a), (e,c)}

Then, i=3,4

Vertex	Shortest Distance from a	Previous vertex
a	0	-
b	2	c
c	4	d
d	7	a
e	-2	b



Bellman-Ford algorithm

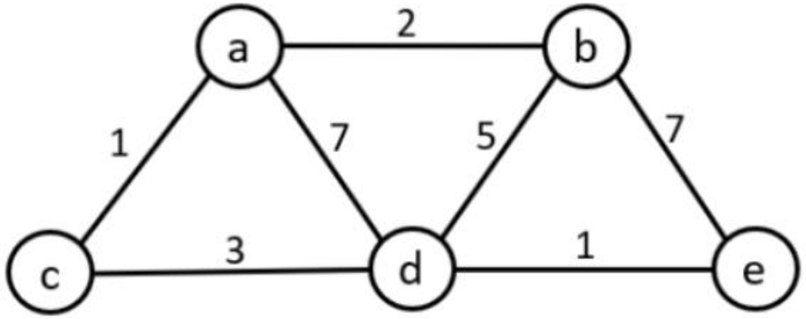
```
Bellman-Ford( G = (V,E), starting vertex s):
    key(v) = ∞, ∀v ∈ V
    key(s) = 0
    p(v) = NULL, ∀v ∈ V
    for i=1,…,n-1:
        for each edge (u,v) in E:
            if key(v) > key(u) + w(u, v) :
                key(v) = key(u) + w(u, v)
                p(v) = u
return key, p
```

$O(nm)$

Summary: shortest path

- BFS:
  - (+) O(n+m)
  - (-) only unweighted graphs
- Dijkstra's algorithm:
  - (+) weighted graphs
  - (+) O(mlogn) if you implement it right.
  - (-) no negative edge weights
- The Bellman-Ford algorithm:
  - (+) weighted graphs, even with negative weights
  - (-) O(nm)

Exercise



i. Draw the adjacency matrix of this graph **[3 marks]**

ii. Step through Dijkstra's Algorithm to calculate the single source shortest path from vertex *a* to every other vertex. You need to show your steps in the table below for full credit. Show your steps by crossing through values that are replaced by a new value. **[8 marks]**

Vertex	Distance	Previous Vertex
a	0	
b		
c		
d		
e		

iii. What is the shortest path from a to e? **[3 marks]**

iv. Give a Minimum Spanning Tree (MST) of the graph above. **[3 marks]**

Learning Outcome

- Understand what Minimum Spanning Tree is
  - Able to apply Kruskal's algorithm to find minimum spanning tree
  - Able to apply Prim's algorithm to find minimum spanning tree

- Understand what Single-source shortest path is
  - Able to apply Dijkstra algorithm to solve shortest path problem
  - Able to apply Bellman-Ford algorithm to solve shortest path problem