DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

GRAPH PATH FINDING

MINIMUM SPANNING TREE (MST)

Course Instructor: Dr. Shreya Ghosh

DIJKSTRA'S ALGORITHM (REVISIT)

```
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \emptyset$
$Q \leftarrow V$          ⊳ $Q$ is a priority queue maintaining $V - S$
```

DIJKSTRA'S ALGORITHM (REVISIT)

```
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$        ⊳ $Q$ is a priority queue maintaining $V - S$
```

```
while $Q \neq \varnothing$
    do $u \leftarrow$ EXTRACT-MIN$(Q)$
        $S \leftarrow S \cup \{u\}$
        for each $v \in Adj[u]$
            do if $d[v] > d[u] + w(u, v)$
                then $d[v] \leftarrow d[u] + w(u, v)$
                    $p[v] \leftarrow u$
```

DIJKSTRA'S ALGORITHM (REVISIT)

```
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$          ⊳ $Q$ is a priority queue maintaining $V - S$
```

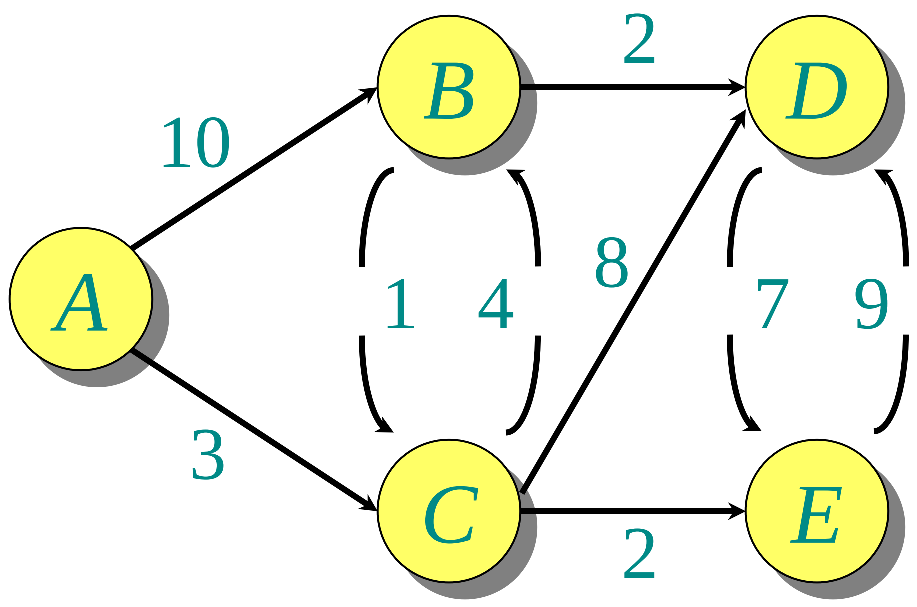
```
while $Q \neq \varnothing$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
        $S \leftarrow S \cup \{u\}$
        for each $v \in Adj[u]$
            do if $d[v] > d[u] + w(u, v)$
                then $d[v] \leftarrow d[u] + w(u, v)$
                $p[v] \leftarrow u$
```

relaxation step

Implicit DECREASE-KEY

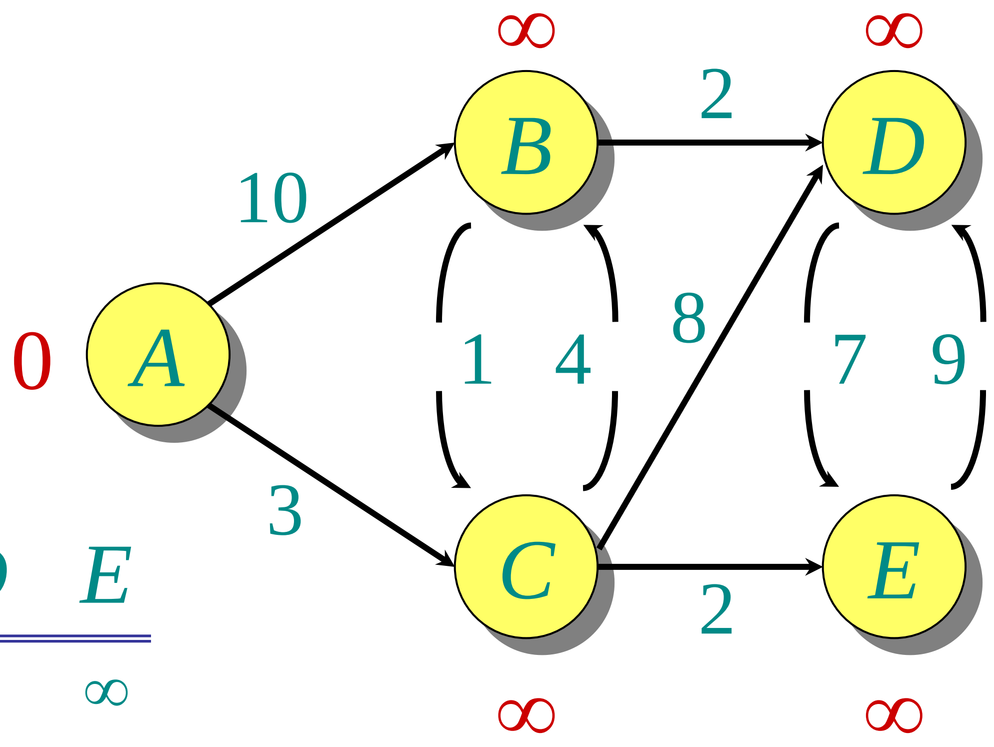
Example of Dijkstra's algorithm

Graph with nonnegative edge weights:



Example of Dijkstra’s algorithm

Initialize:

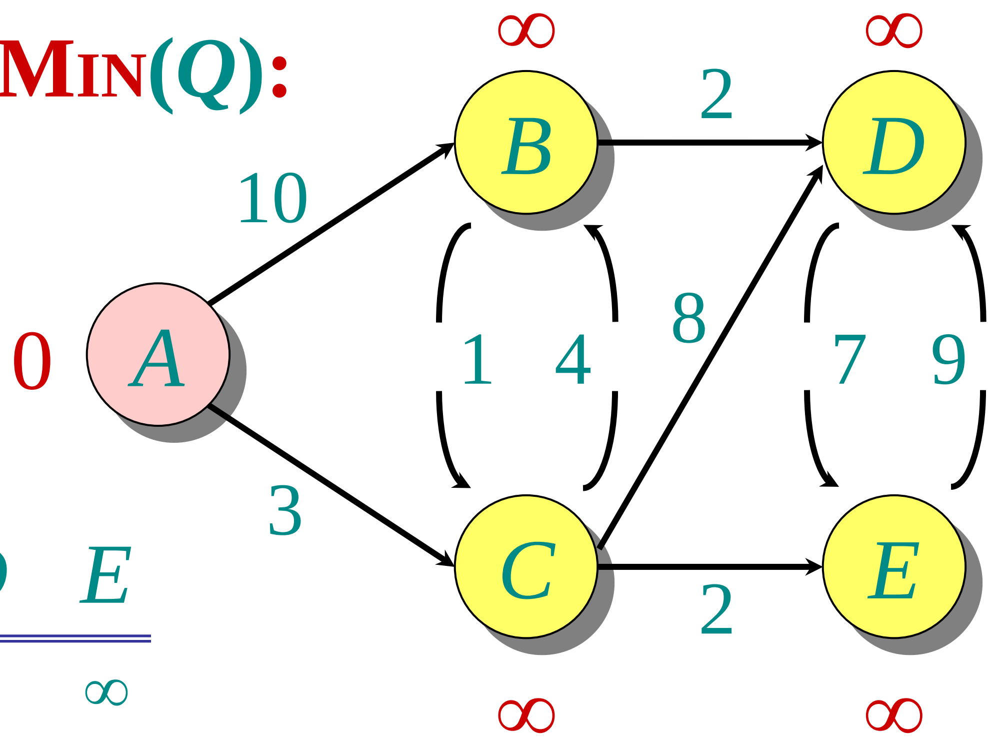


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Example of Dijkstra’s algorithm

“*A*” ← EXTRACT-MIN(*Q*):

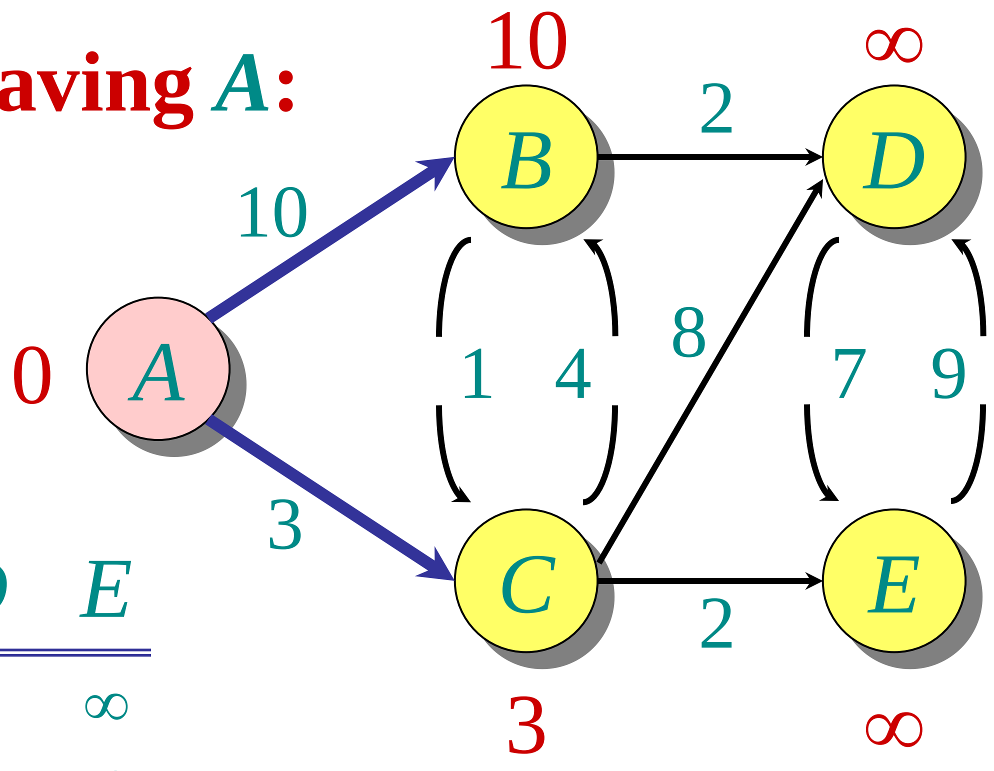


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞

S: { *A* }

Example of Dijkstra’s algorithm

Relax all edges leaving *A*:

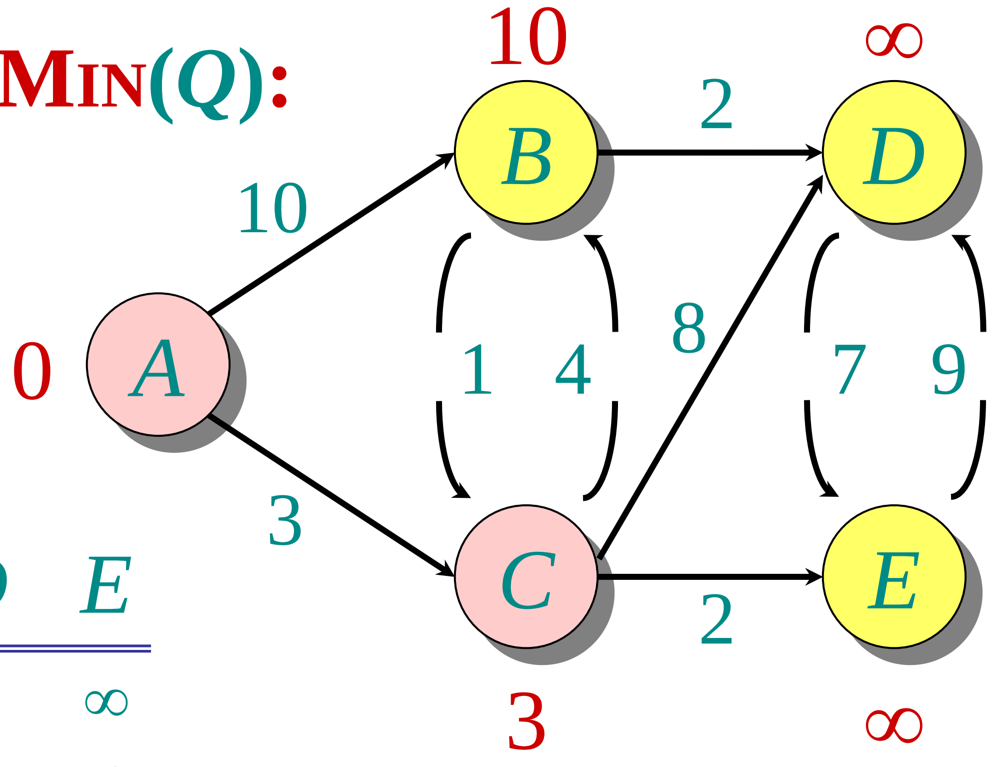


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A* }

Example of Dijkstra's algorithm

"C" ← EXTRACT-MIN(Q):

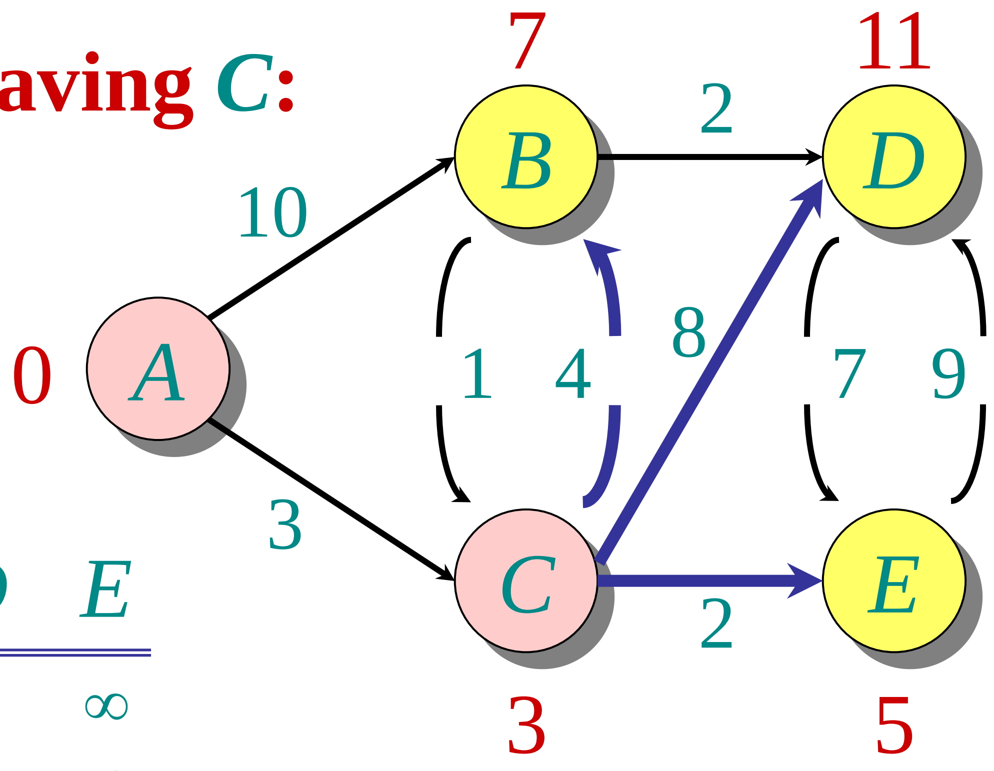


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

Example of Dijkstra's algorithm

Relax all edges leaving *C*:

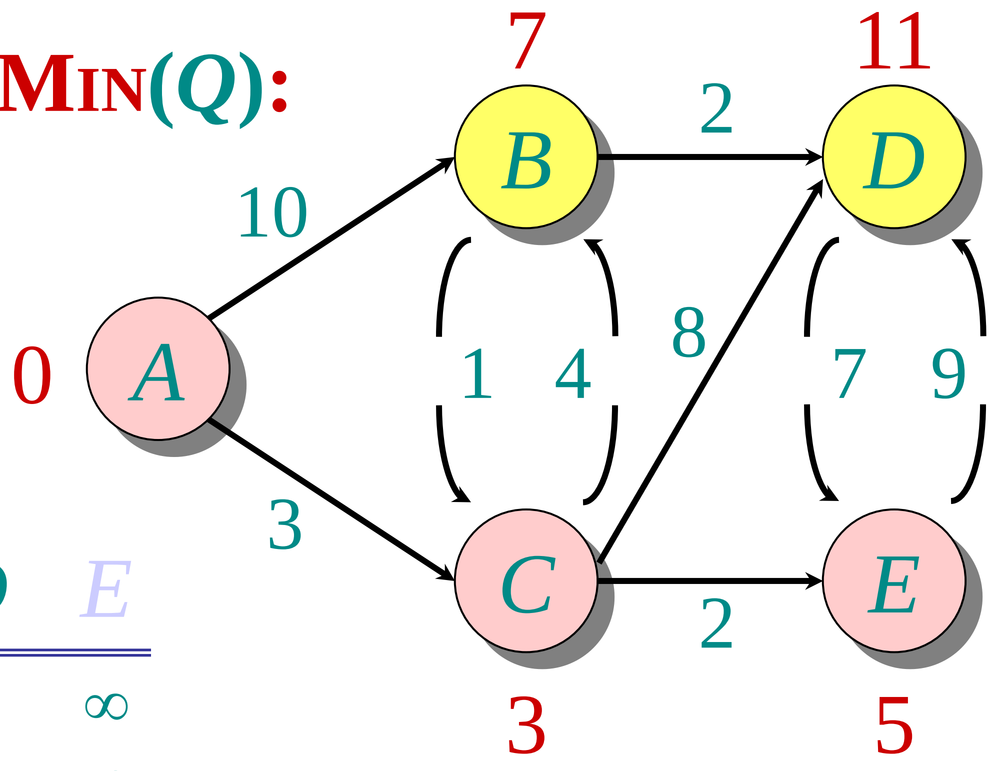


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }

Example of Dijkstra's algorithm

"E" ← EXTRACT-MIN(Q):

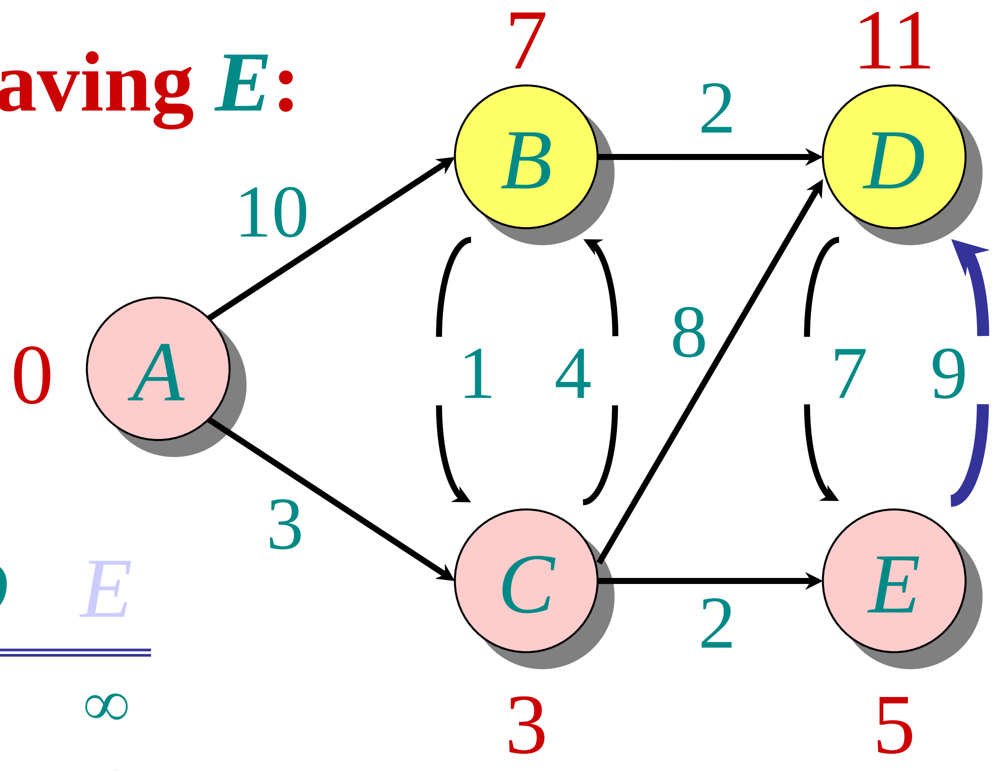


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

Example of Dijkstra's algorithm

Relax all edges leaving *E*:

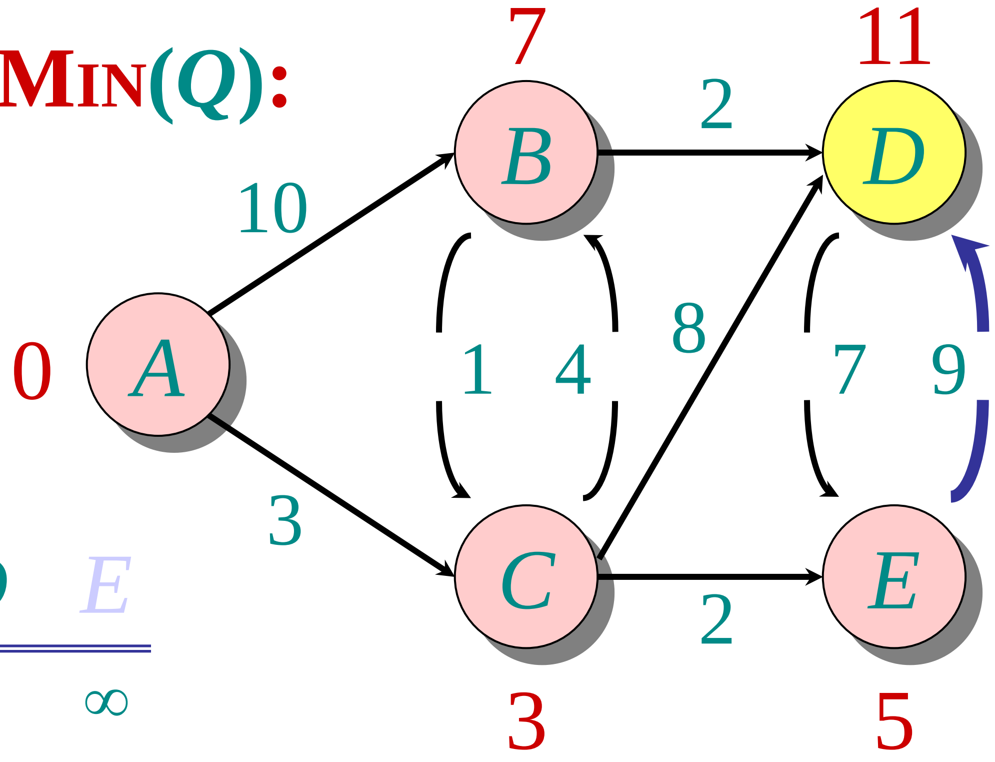


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E* }

Example of Dijkstra's algorithm

"B" ← EXTRACT-MIN(Q):

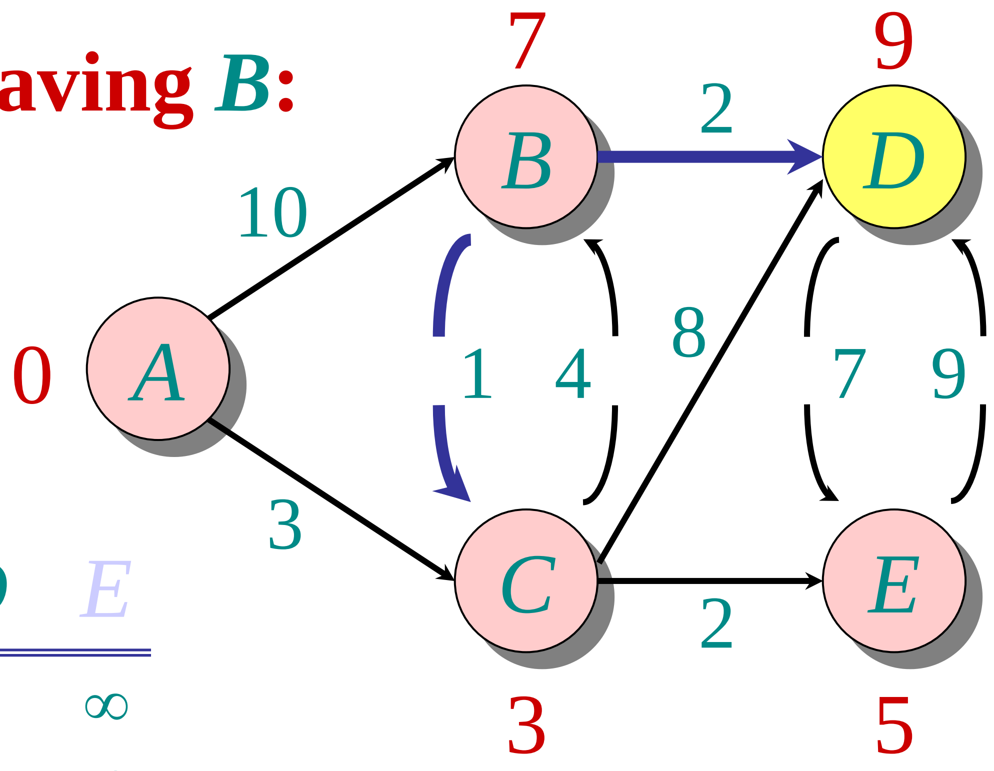


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Example of Dijkstra's algorithm

Relax all edges leaving *B*:

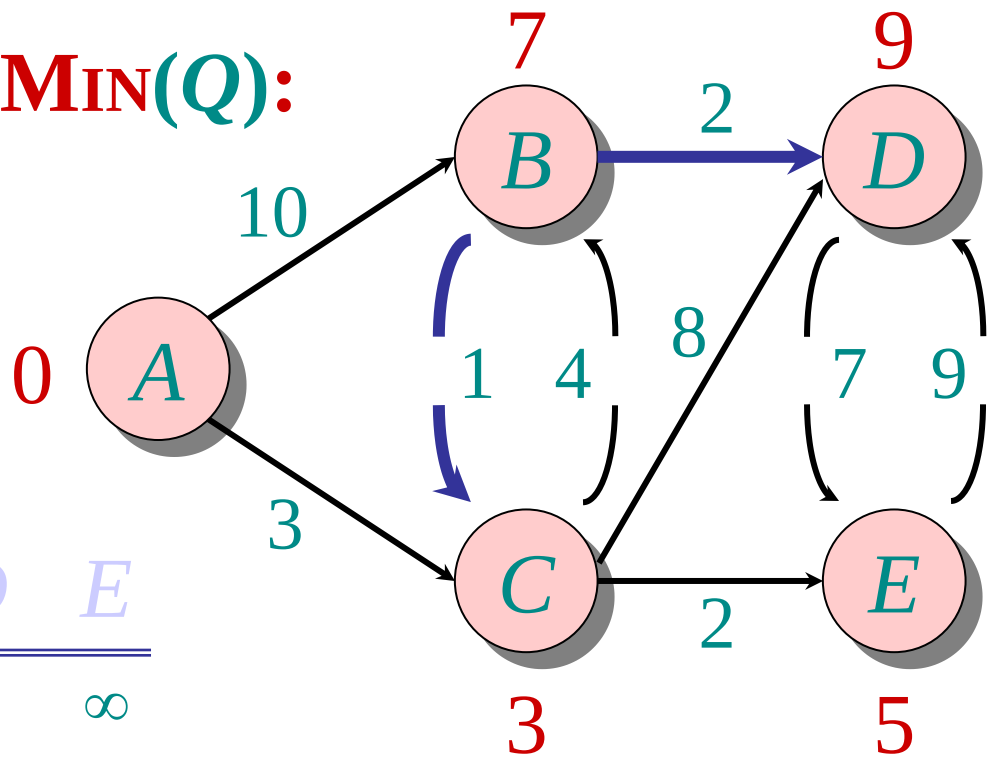


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B* }

Example of Dijkstra's algorithm

"*D*" ← **EXTRACT-MIN(*Q*):**



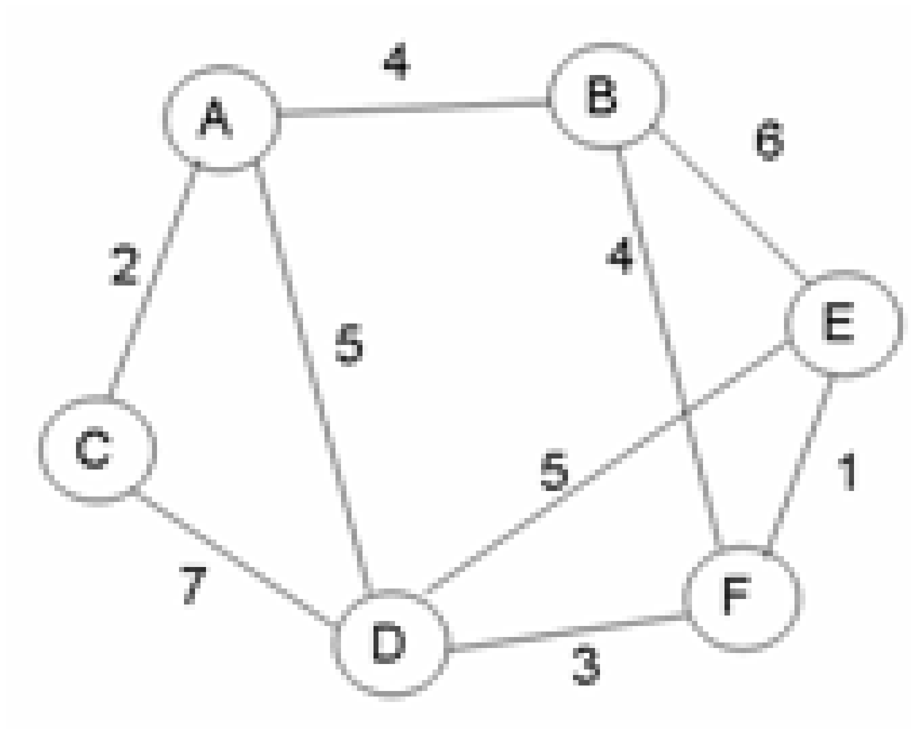
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B*, *D* }

Summary

- Given a weighted directed graph, we can find the shortest distance between two vertices by:
  - starting with a trivial path containing the initial vertex
  - growing this path by always going to the next vertex which has the shortest current path

Practice Problem



BELLMAN-FORD ALGORITHM

```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

Initialize d[] which will converge to shortest-path value

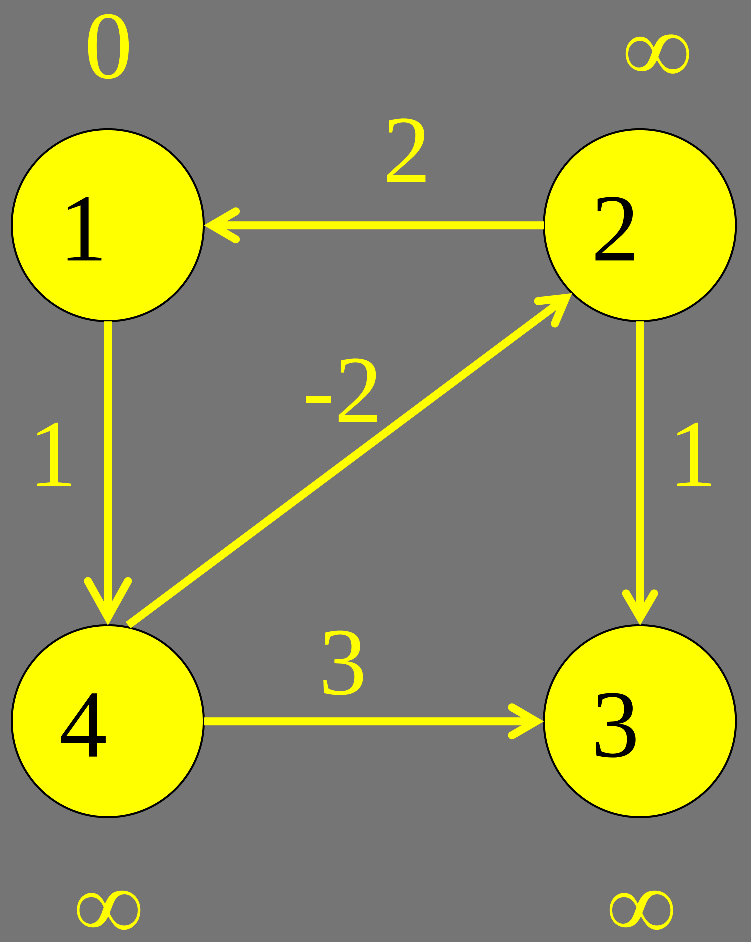
Relaxation: Make |V|-1 passes, relaxing each edge

Test for solution: have we converged yet? Ie, ∃ negative cycle?

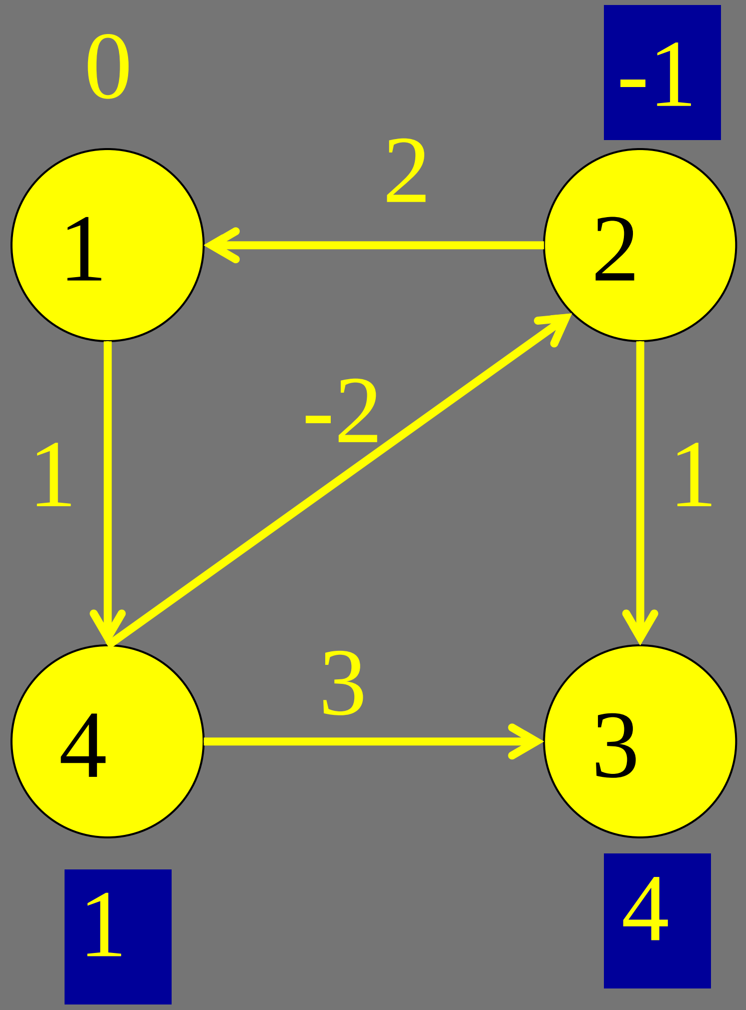
Can have negative-weight edges. Will "detect" reachable negative-weight cycles.

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

BELLMAN-FORD EXAMPLE I



BELLMAN-FORD EXAMPLE I



$k = 1$

$d^1(1) = 0$

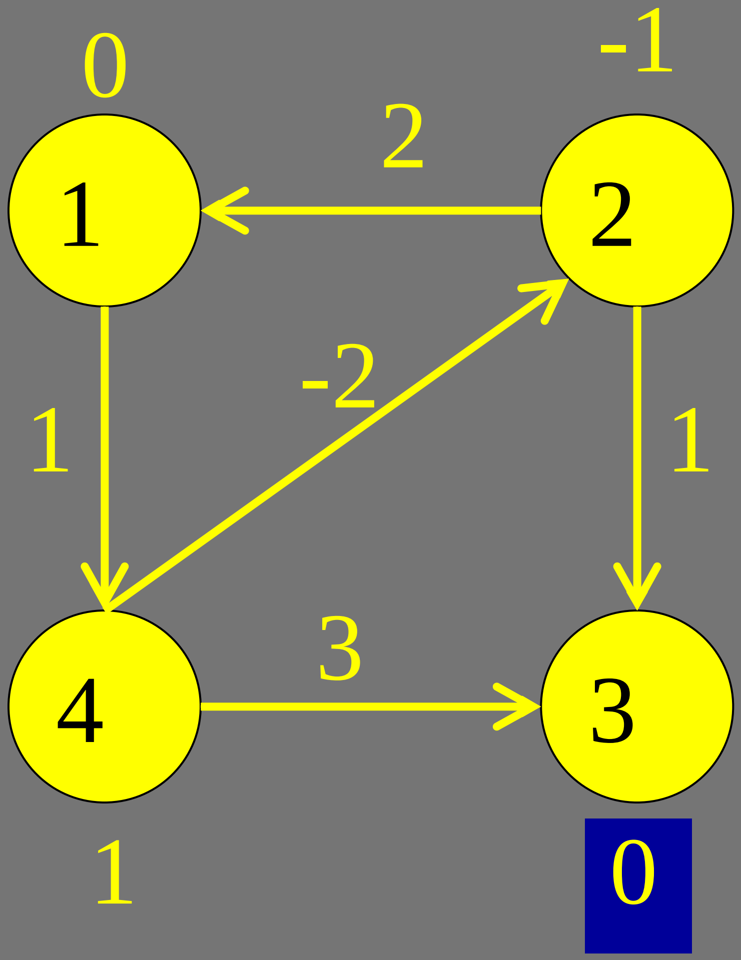
$d^1(2) = -1$

$d^1(3) = 4$

$d^1(4) = 1$

$d^k(i)$ denotes $d(i)$ after iteration k

BELLMAN-FORD EXAMPLE I



$k = 2$

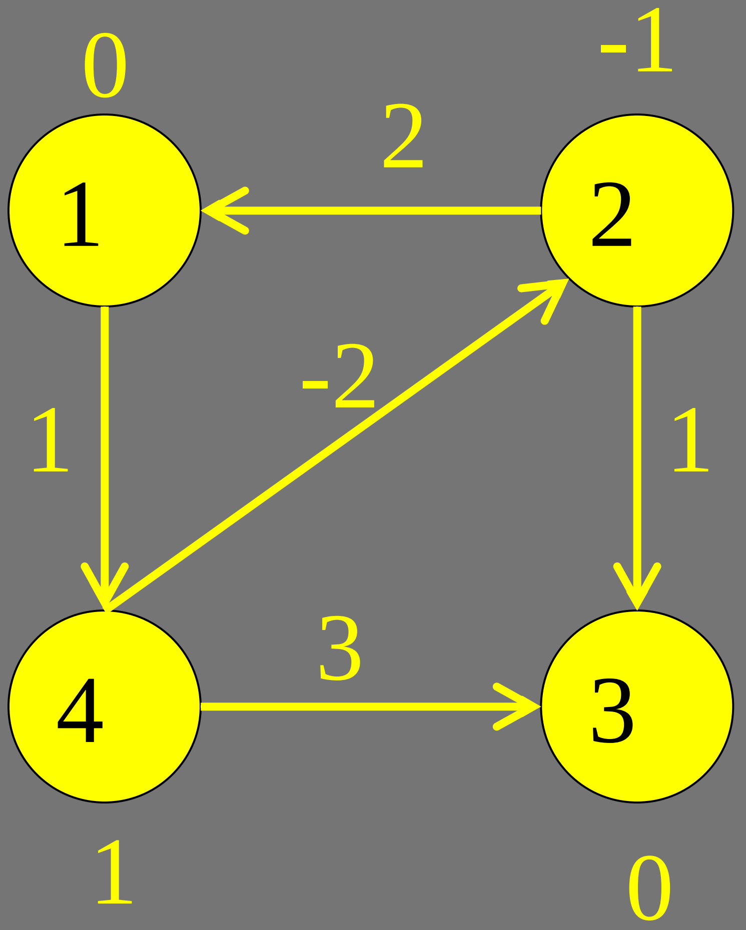
$d^2(1)=0$

$d^2(2)=-1$

$d^2(3)=0$

$d^2(4)=1$

BELLMAN-FORD EXAMPLE I



$k = 3$

$d^3(1)=0$

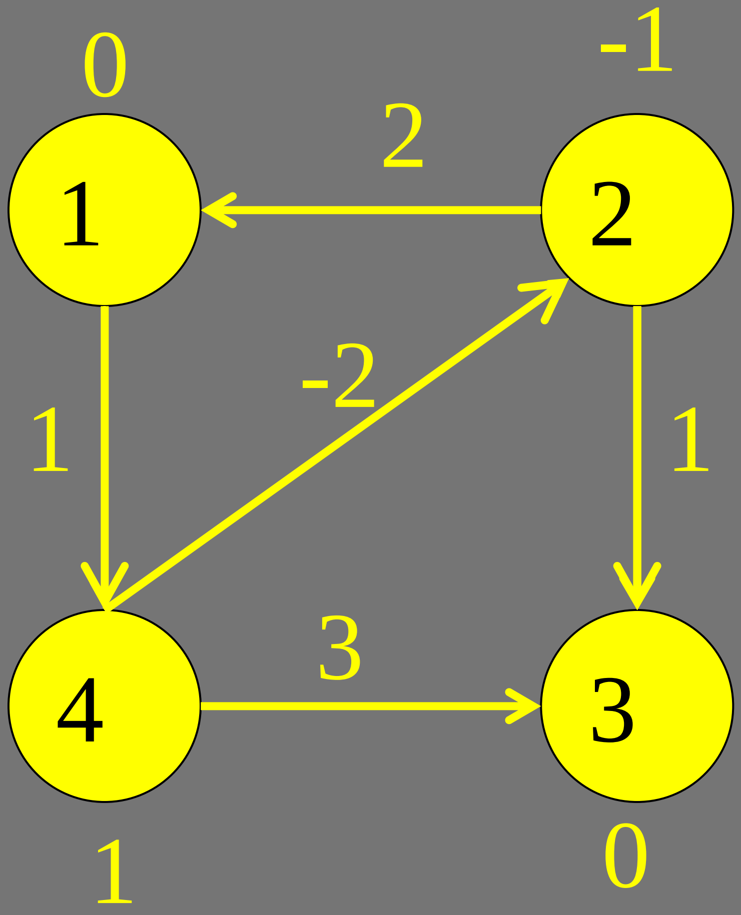
$d^3(2)=-1$

$d^3(3)=0$

$d^3(4)=1$

$d^3(i)= d^2(i)$

SHORTEST PATH TREE

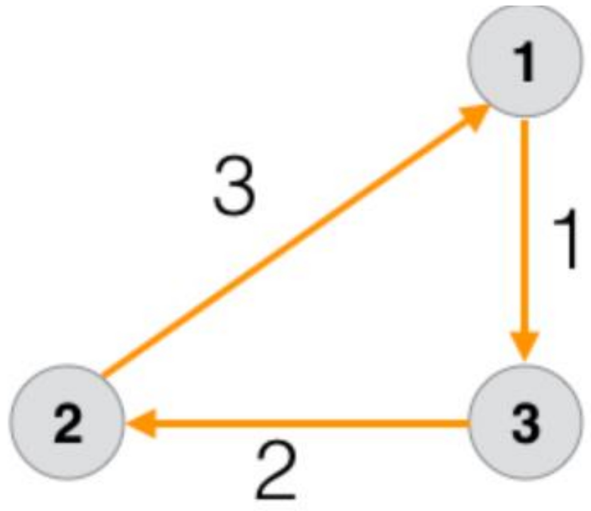


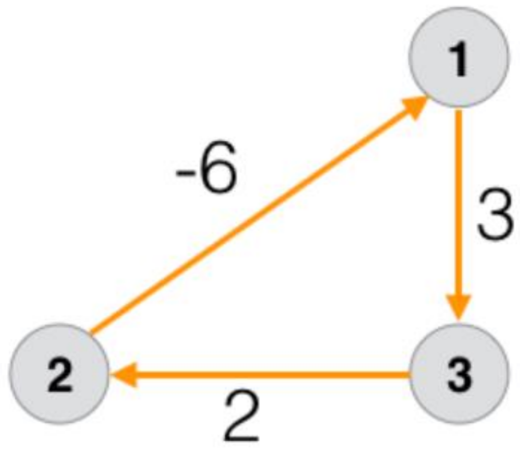
$d(j) \le d(i) + c_{ij}$
and
$d(j) = d(i) + c_{ij}$
for all predecessor arcs

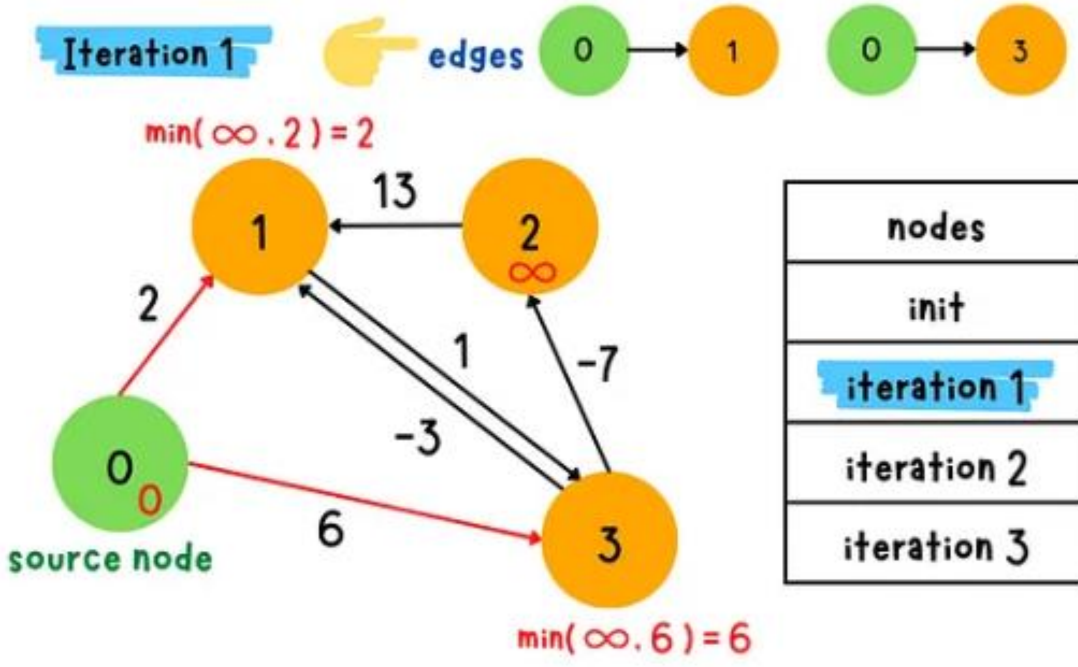
If Bellman-Ford has not converged after V(G) - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

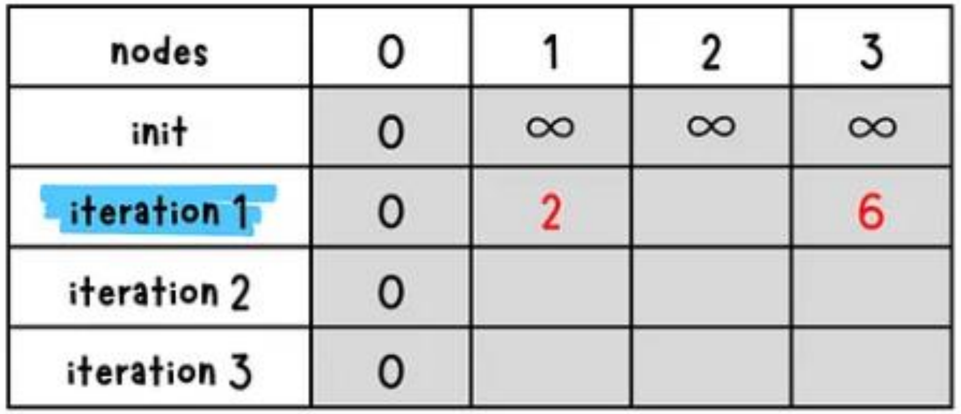
WHAT IS A NEGATIVE CYCLE

A negative cycle in a weighted graph is a cycle whose total weight is negative.









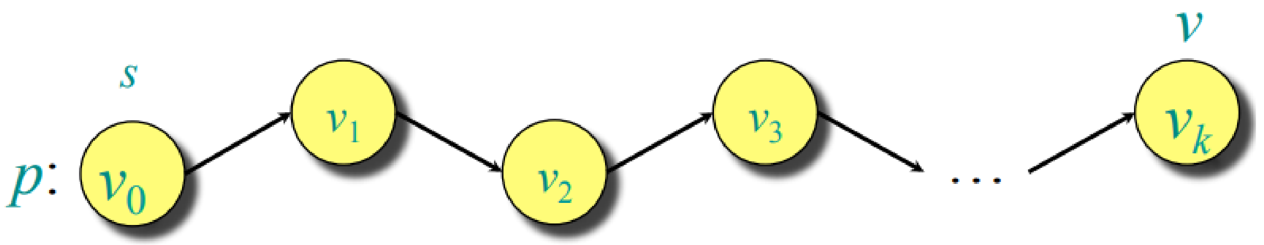
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2		6
iteration 2	0			
iteration 3	0			

iteration 1–1

Correctness

Theorem. If $G = (V, E)$ contains no negative-weight cycles, then after the Bellman-Ford algorithm executes, $d[v] = \delta(s, v)$ for all $v \in V$.

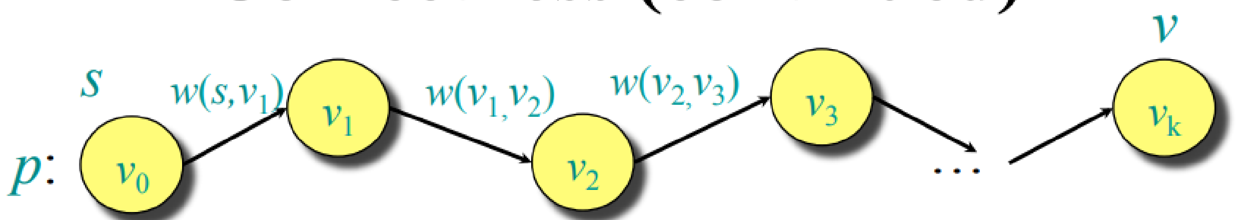
Proof. Let $v \in V$ be any vertex, and consider a shortest path p from s to v with the minimum number of edges.



Since p is a shortest path, we have

$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i) \quad \text{for every } i.$$

Correctness (continued)



Let p be the shortest path from s to a vertex v. Lets re-label the vertices along p so $s=v_0, v_1, \dots v_k=v$
Note that (a portion of) p is also the shortest path to each v_i.

Initially, $d[v_0] = 0 = \delta(s, v_0)$, and $d[s]$ is unchanged by subsequent relaxations (note $\delta(s, s) \geq 0$ (why ?)).

- After 1 pass through E, we have $d[v_1]=\delta(s, v_1) = w(s,v_1)$.
- After 2 passes through E, we have
  $$d[v_2] = d[v_1]+w(v_1, v_2) = w(s,v_1)+w(v_1, v_2) = \delta(s, v_2).$$
  ...
- After k passes through E, we have $d[v_k] = \delta(s, v_k)$.

Since G contains no negative-weight cycles, p is simple. Longest simple path has $\leq |V| - 1$ edges. $\square$

```
INITIALIZE($V, E, s$)
1   for $v \in V$
2       $d[v] \leftarrow \infty$
3       $\pi[v] \leftarrow$ NULL
4   $d[s] \leftarrow 0$
5   $\pi[s] \leftarrow s$
```

```
RELAX($u, v$)
1   if $d[v] > d[u] + w(u, v)$
2       $d[v] \leftarrow d[u] + w(u, v)$
3       $\pi[v] \leftarrow u$
```

```
BELLMAN-FORD($V, E, s$)
1   INITIALIZE($V, E, s$)
2   for $i = 1 : |V| - 1$
3       for each edge $(u, v) \in E$
4           RELAX($u, v$)
5   for each edge $(u, v) \in E$
6       if $d[v] > d[u] + w(u, v)$
7           return FALSE
8   return TRUE
```

Bellman-Ford Correctness 1: Assume we have a graph G with no negative cycles reachable by s. Then after running Bellman-Ford, we have d[v] = δ(s, v) for all v ∈ V reachable from s.

Proof: If $v \in V$ is reachable from s then there must exist an acyclic shortest path $p = \langle s, v_1, ..., v_j \rangle$ where $v_j = v$. Now, since p is acyclic, p can contain no more than V vertices and therefore no more than $|V| - 1$ edges. Each time we run through the loop on line 3 of BELLMAN-FORD, we relax every edge. Therefore, we relax (s, v_1) on the first iteration, (v_1, v_2) on the second iteration, etc (we also relax (v_1, v_2) on the first iteration, of course, but we cannot guarantee that relaxation comes after the relaxation of (s, v_1)). By the time we have reached the $|V| - 1$ iteration, we must have relaxed every edge in p in order. Therefore, by the path relaxation property, we have $d[v] = \delta(s, v)$ when we have done $|V| - 1$ iterations of the loop on line 3.

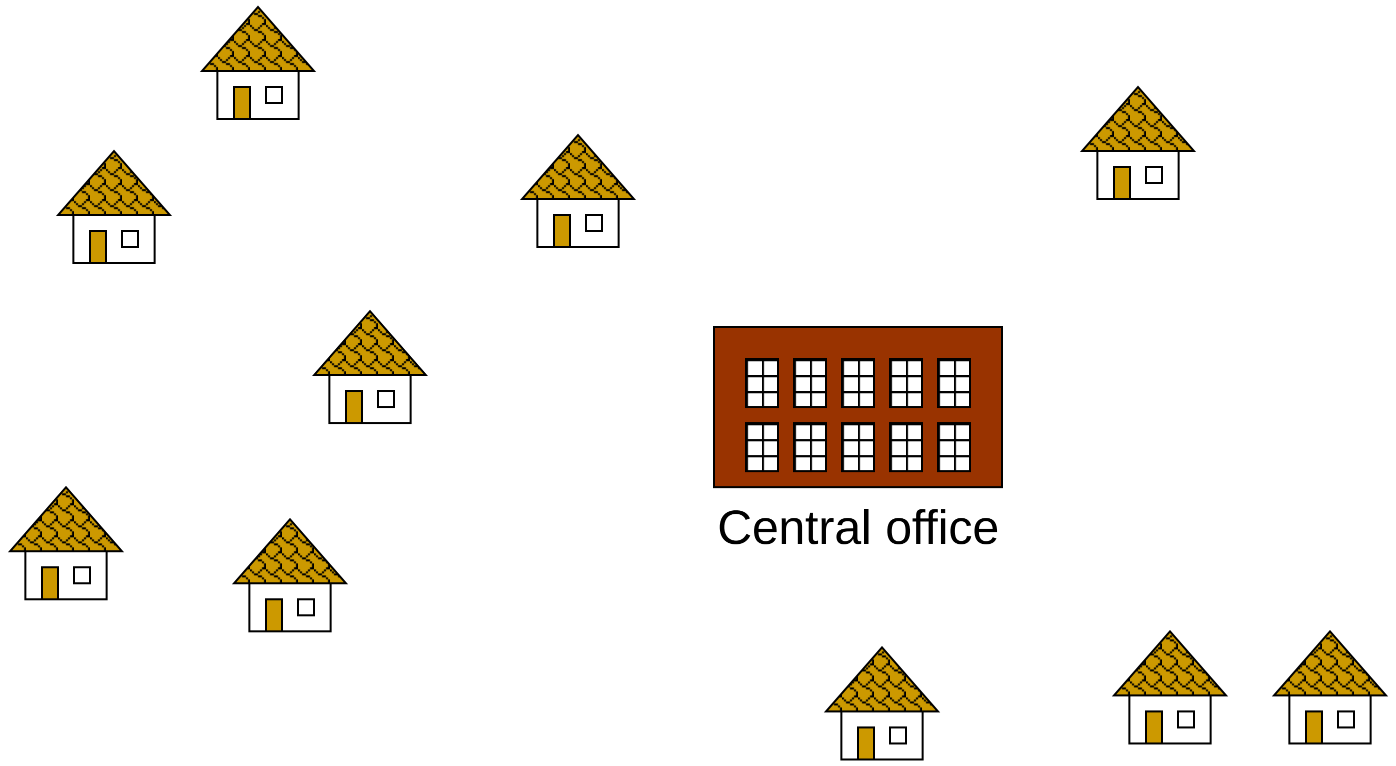
DAG SHORTEST PATHS

- Bellman-Ford takes O(VE) time.
- For finding shortest paths in a DAG, we can do much better by using a topological sort.
- If we process vertices in topological order, we are guaranteed to never relax a vertex unless the adjacent edge is already finalized. Thus: just one pass. O(V+E)

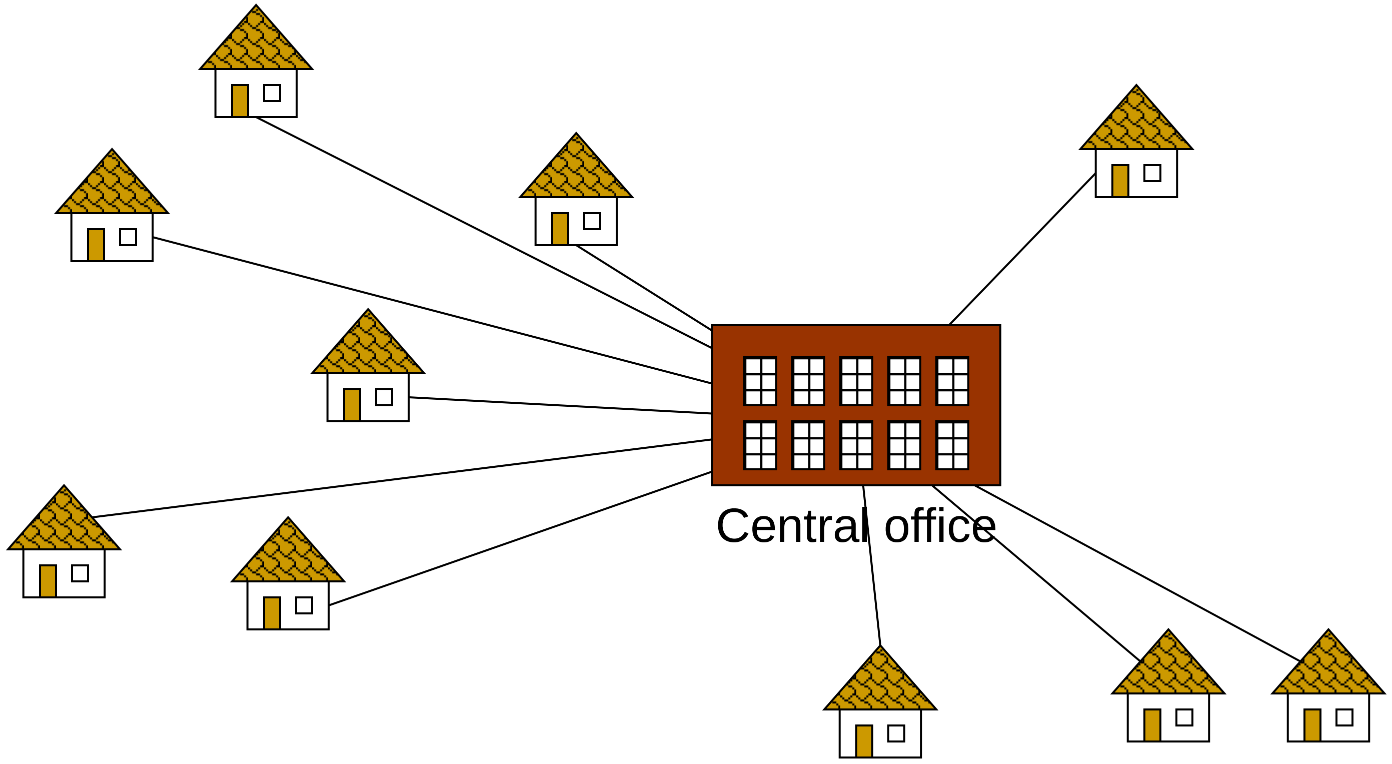
```
DAG-Shortest-Paths(G, w, s)
1.  topologically sort the vertices of G
2.  INITIALIZE-SINGLE-SOURCE(G, s)
3.  for each vertex u, taken in topologically sorted order
4.      do for each vertex v ∈ Adj[u]
5.          do Relax(u, v, w)
```

MINIMUM SPANNING TREE (MST)

PROBLEM: LAYING TELEPHONE WIRE

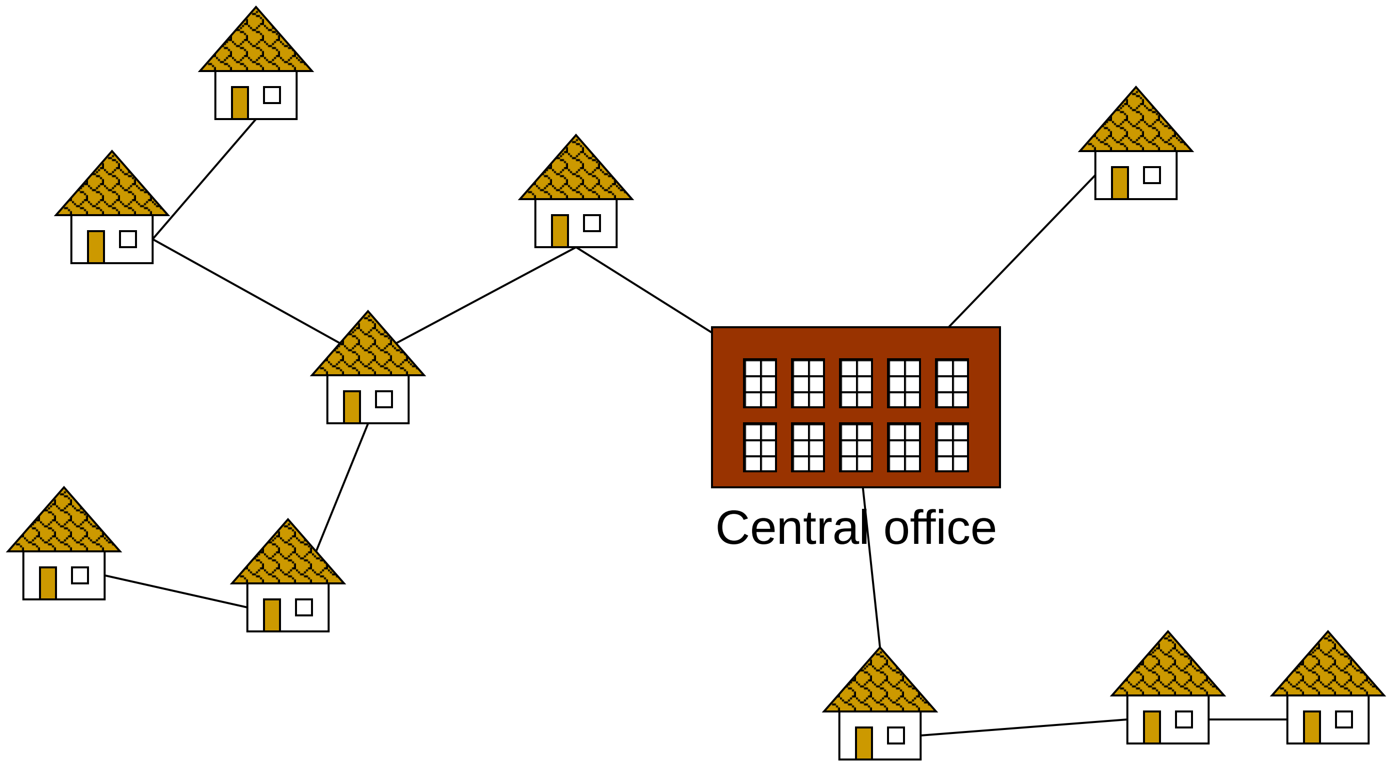


WIRING: NAÏVE APPROACH



Expensive!

WIRING: BETTER APPROACH



Minimize the total length of wire connecting the customers

MINIMUM SPANNING TREE (MST)

A **minimum spanning tree** is a subgraph of an undirected weighted graph ***G***, such that

- it is a tree (i.e., it is acyclic)
- it covers all the vertices ***V***
  - contains $|V| - 1$ edges
- the total cost associated with tree edges is the minimum among all possible spanning trees
- not necessarily unique

MINIMUM SPANNING TREES (MST)

Punchline: a MST of a graph connects all the vertices together while minimizing the number of edges used (and their weights).

Minimum spanning trees

Given a connected, undirected graph G = (V, E), a minimum spanning tree is a subgraph G' = (V' , E') such that...

- V = V' (G' is spanning)
- There exists a path from any vertex to any other one
- The sum of the edge weights in E' is minimized

In order for a graph to have a MST, the graph must...

- ...be connected – there is a path from a vertex to any other vertex. (Note: this means $|V| \leq |E|$).
- ...be undirected.

APPLICATIONS OF MST

- Any time you want to visit all vertices in a graph at minimum cost (e.g., wire routing on printed circuit boards, sewer pipe layout, road planning…)
- Internet content distribution
  - $$$, also a hot research topic
  - Idea: publisher produces web pages, content distribution network replicates web pages to many locations so consumers can access at higher speed
  - MST may not be good enough!
    - content distribution on minimum cost tree may take a long time!
- Provides a heuristic for traveling salesman problems. The optimum traveling salesman tour is at most twice the length of the minimum spanning tree (*why??*)

MINIMUM SPANNING TREES: APPLICATIONS

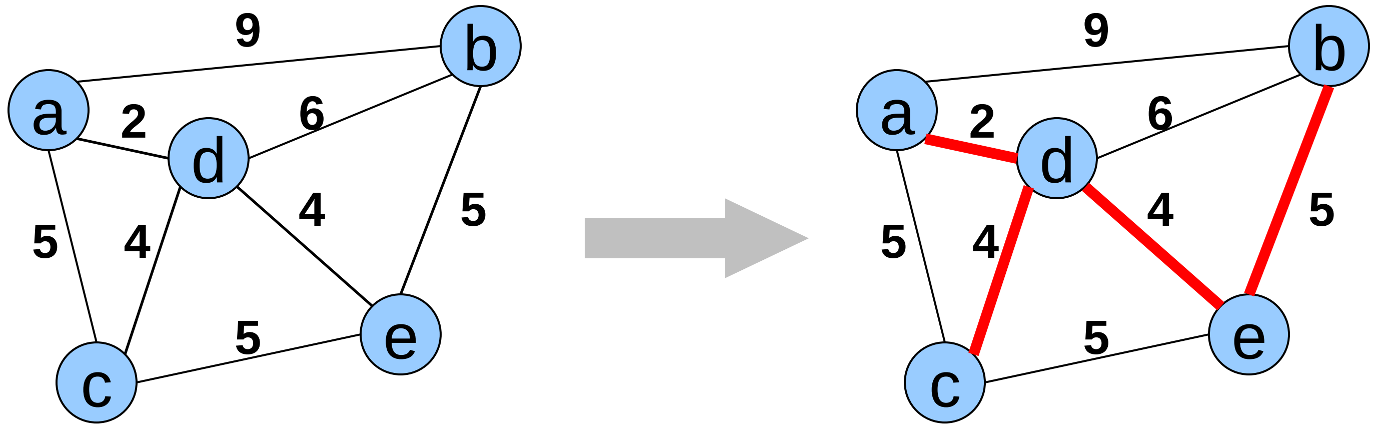
- **Pipeline Network Construction**: Suppose a water supply company wants to build a pipeline network to connect a group of cities, with the primary goal of minimizing the total length of pipelines used. Each city must have access to water, and the company must consider the distances between cities when deciding where to lay the pipelines.

- **Airline Route Optimization**: An airline company wants to establish new flight routes between several airports, aiming to minimize the total distance flown while ensuring that passengers can travel between any pair of airports, either directly or through connecting flights.

Other applications:

☑ Implement efficient multiple constant multiplication
☑ Minimizing number of packets transmitted across a network
☑ Machine learning (e.g. real-time face verification)
☑ Graphics (e.g. image segmentation)

HOW CAN WE GENERATE A MST?



MINIMUM SPANNING TREES: PROPERTIES

- ☑ A valid MST cannot contain a cycle
- ☑ If we add or remove an edge from an MST, it’s no longer a valid MST for that graph. Adding an edge introduces a cycle; removing an edge means vertices are no longer connected
- ☑ If there are |V| vertices, the MST contains exactly |V| − 1 edges.
- ☑ An MST is always a tree. I If every edge has a unique weight, there exists a unique MST

PennState

PRIM'S ALGORITHM

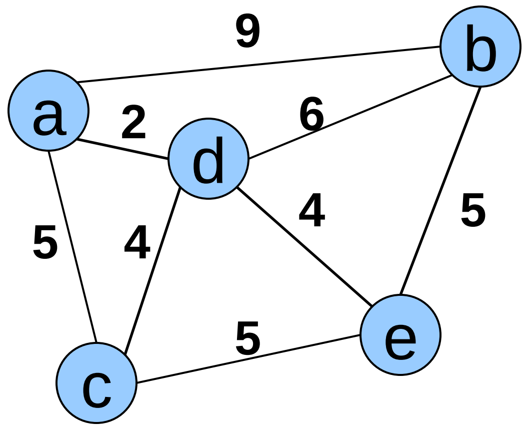
Initialization

a. Pick a vertex r to be the root

b. Set $D(r) = 0$, $parent(r) = null$

c. For all vertices $v \in V$, $v \neq r$, set $D(v) = \infty$

d. Insert all vertices into priority queue P, using distances as the keys



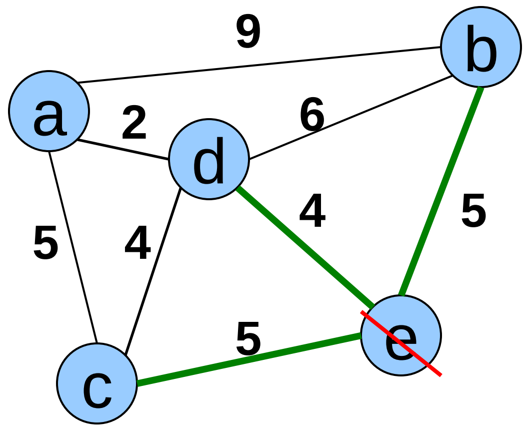
e	a	b	c	d
0	∞	∞	∞	∞

Vertex	Parent
e	-

PRIM'S ALGORITHM

```
While $P$ is not empty:

   1. Select the next vertex $u$ to add to the tree
           $u = P.deleteMin()$

   2. Update the weight of each vertex $w$ adjacent to
      $u$ which is not in the tree (i.e., $w \in P$)
           If $weight(u,w) < D(w)$,
              a. $parent(w) = u$
              b. $D(w) = weight(u,w)$
              c. Update the priority queue to reflect
                 new distance for $w$
```

PRIM'S ALGORITHM



e	d	b	c	a
0	∞	∞	∞	∞

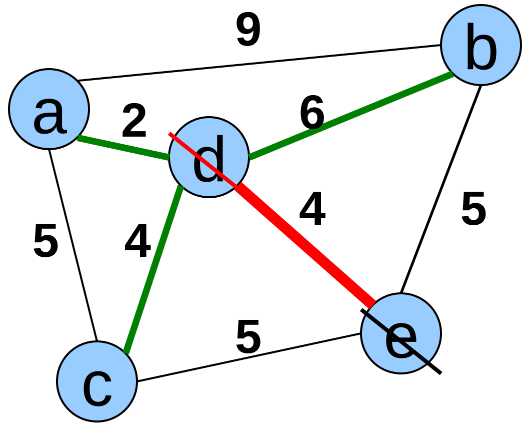
Vertex	Parent
e	-
b	-
c	-
d	-

d	b	c	a
4	5	5	∞

Vertex	Parent
e	-
b	e
c	e
d	e

The MST initially consists of the vertex **e,** and we update the distances and parent for its adjacent vertices

PRIM'S ALGORITHM



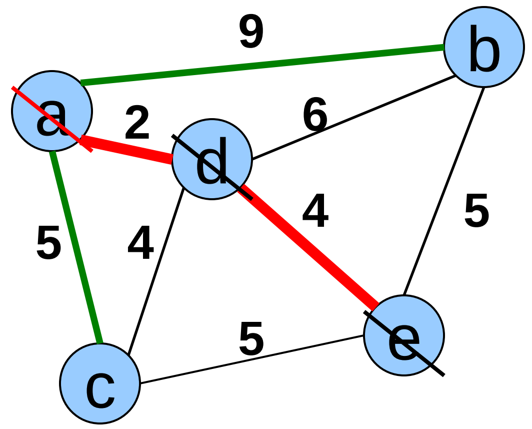
d	b	c	a
4	**5**	**5**	∞

Vertex	Parent
e	-
b	e
c	e
d	e

a	c	b
2	**4**	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

PRIM'S ALGORITHM



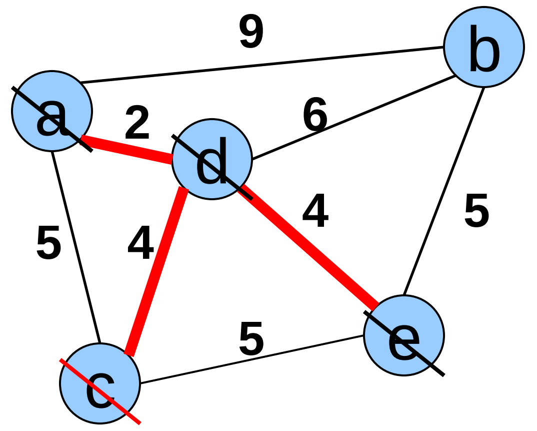
a	c	b
2	4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

c	b
4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

PRIM'S ALGORITHM



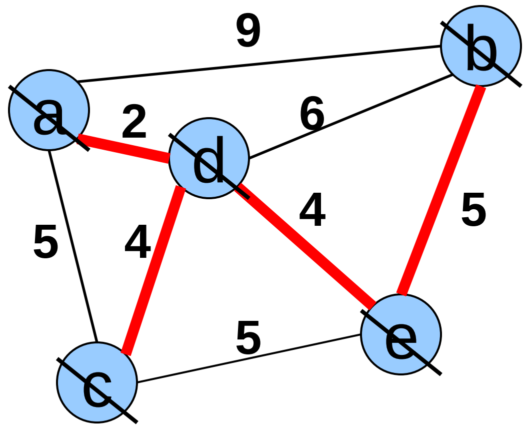
c	b
4	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

PRIM'S ALGORITHM



b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

The final minimum spanning tree

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

RUNNING TIME OF PRIM'S ALGORITHM (WITHOUT HEAPS)

Initialization of priority queue (array): O($|V|$)

Update loop: $|V|$ calls

- Choosing vertex with minimum cost edge: O($|V|$)
- Updating distance values of unconnected vertices: each edge is considered only **once** during entire execution, for a **total** of O($|E|$) updates

Overall cost without heaps: $\boxed{O(|E| + |V|^2)}$

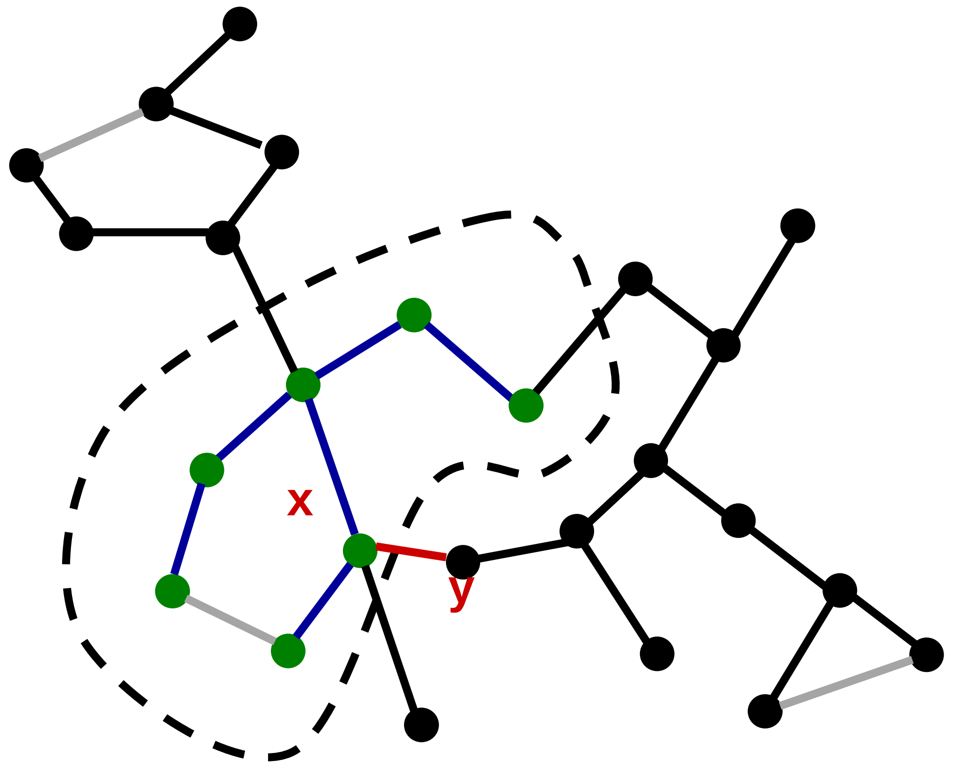
PRIM'S ALGORITHM INVARIANT

- At each step, we add the edge (***u***,***v***) s.t. the weight of (***u***,***v***) is **minimum** among all edges where ***u*** is in the tree and ***v*** is not in the tree
- Each step maintains a minimum spanning tree of the vertices that have been included thus far
- When all vertices have been included, we have a MST for the graph!

CORRECTNESS OF PRIM'S

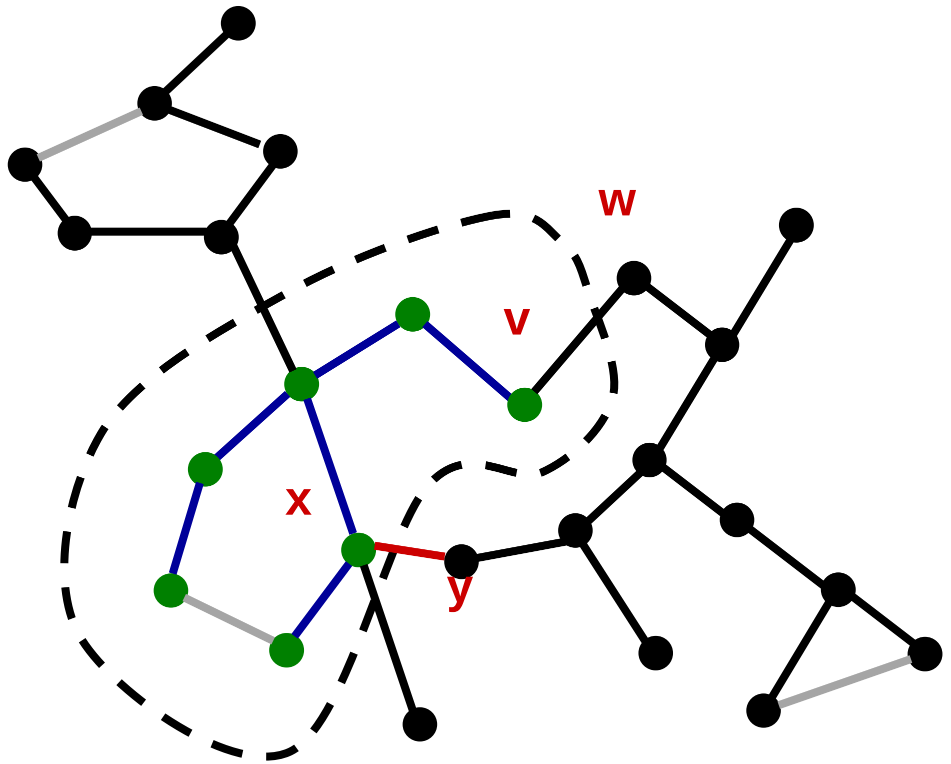
- This algorithm adds *n-1* edges without creating a cycle, so clearly it creates a spanning tree of any connected graph (*you should be able to prove this*).
- But is this a *minimum* spanning tree?
- Suppose it wasn't.
- There must be point at which it fails, and in particular there must a single edge whose insertion first prevented the spanning tree from being a minimum spanning tree.

CORRECTNESS OF PRIM'S



- Let ***G*** be a connected, undirected graph
- Let **S** be the set of edges chosen by Prim's algorithm *before* choosing an errorful edge ***(x,y)***
- Let **V'** be the vertices incident with edges in **S**
- Let **T** be a MST of ***G*** containing all edges in **S**, but not ***(x,y)***.

CORRECTNESS OF PRIM'S



- Edge ***(x,y)*** is not in **T**, so there must be a path in **T** from ***x*** to ***y*** since **T** is connected.
- Inserting edge ***(x,y)*** into **T** will create a cycle
- There is exactly one edge on this cycle with exactly one vertex in **V'**, call this edge **(*v,w*)**

CORRECTNESS OF PRIM'S

- Since Prim's chose **(x,y)** over **(*v*,*w*)**, w(**v,w**) >= w(**x,y**).
- We could form a new spanning tree **T'** by swapping **(*x*,*y*)** for **(*v*,*w*)** in **T** (*prove this is a spanning tree*).
- w(**T'**) is clearly no greater than w(**T**)
- But that means **T'** is a MST
- And yet it contains all the edges in **S**, and also **(*x*,*y*)**

...Contradiction

Recap: Prim's algorithm works similarly to Dijkstra's – we start with a single node, and "grow" our MST.

…. instead of "growing" our MST, we...

- ❑ Initially place each node into its own MST of size 1 – so, we start with |V| MSTs in total.

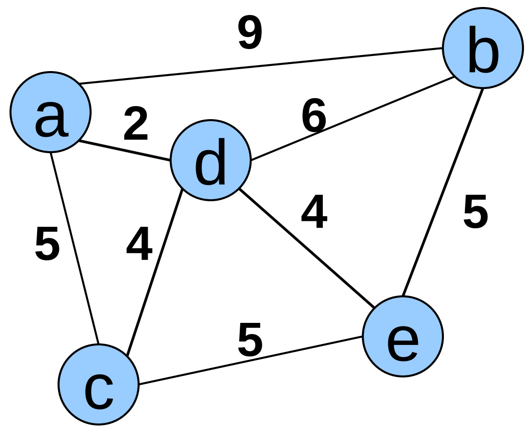
- ❑ Steadily combine together different MSTs until we have just one left

- ❑ How? Loop through every single edge, see if we can use it to join two different MSTs together.

This algorithm is called Kruskal's algorithm

ANOTHER APPROACH

- Create a forest of trees from the vertices
- Repeatedly merge trees by adding "**safe edges**" until only one tree remains
- A "safe edge" is an edge of minimum weight which does not create a cycle



forest: {a}, {b}, {c}, {d}, {e}

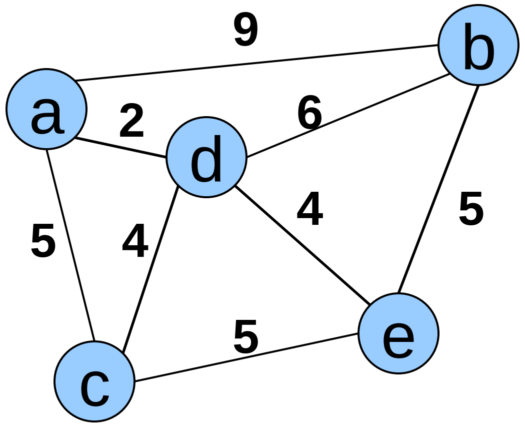
KRUSKAL'S ALGORITHM

Initialization

a. Create a set for each vertex $\boldsymbol{v} \in \boldsymbol{V}$

b. Initialize the set of "safe edges" $\boldsymbol{A}$ comprising the MST to the empty set

c. Sort edges by increasing weight



F **=** {a}, {b}, {c}, {d}, {e}

A **=** ∅

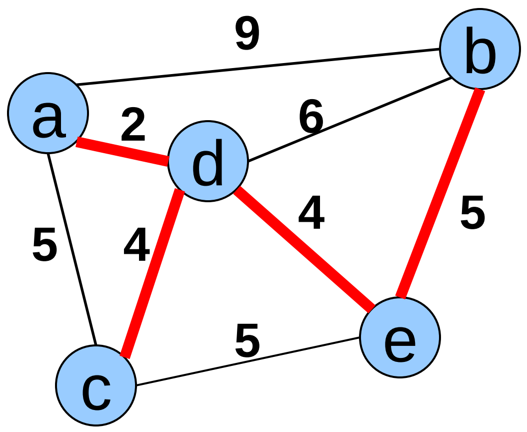
E **=** {(a,d), (c,d), (d,e), (a,c), (b,e), (c,e), (b,d), (a,b)}

KRUSKAL'S ALGORITHM

```
For each edge $(u,v) \in E$ in increasing order
while more than one set remains:
    If $u$ and $v$, belong to different sets $U$ and $V$
        a. add edge $(u,v)$ to the safe edge set
            $A = A \cup \{(u,v)\}$
        b. merge the sets $U$ and $V$
            $F = F - U - V + (U \cup V)$

Return $A$
```

- Running time bounded by sorting (or findMin)
- O($|E|\log|E|$), or equivalently, O($|E|\log|V|$)

KRUSKAL'S ALGORITHM



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

<u>Forest</u>	<u>***A***</u>
{a}, {b}, {c}, {d}, {e}	∅
{a,d}, {b}, {c}, {e}	{(a,d)}
{a,d,c}, {b}, {e}	{(a,d), (c,d)}
{a,d,c,e}, {b}	{(a,d), (c,d), (d,e)}
{a,d,c,e,b}	{(a,d), (c,d), (d,e), (b,e)}

Kruskal’s Algorithm Invariant

- After each iteration, every tree in the forest is a MST of the vertices it connects

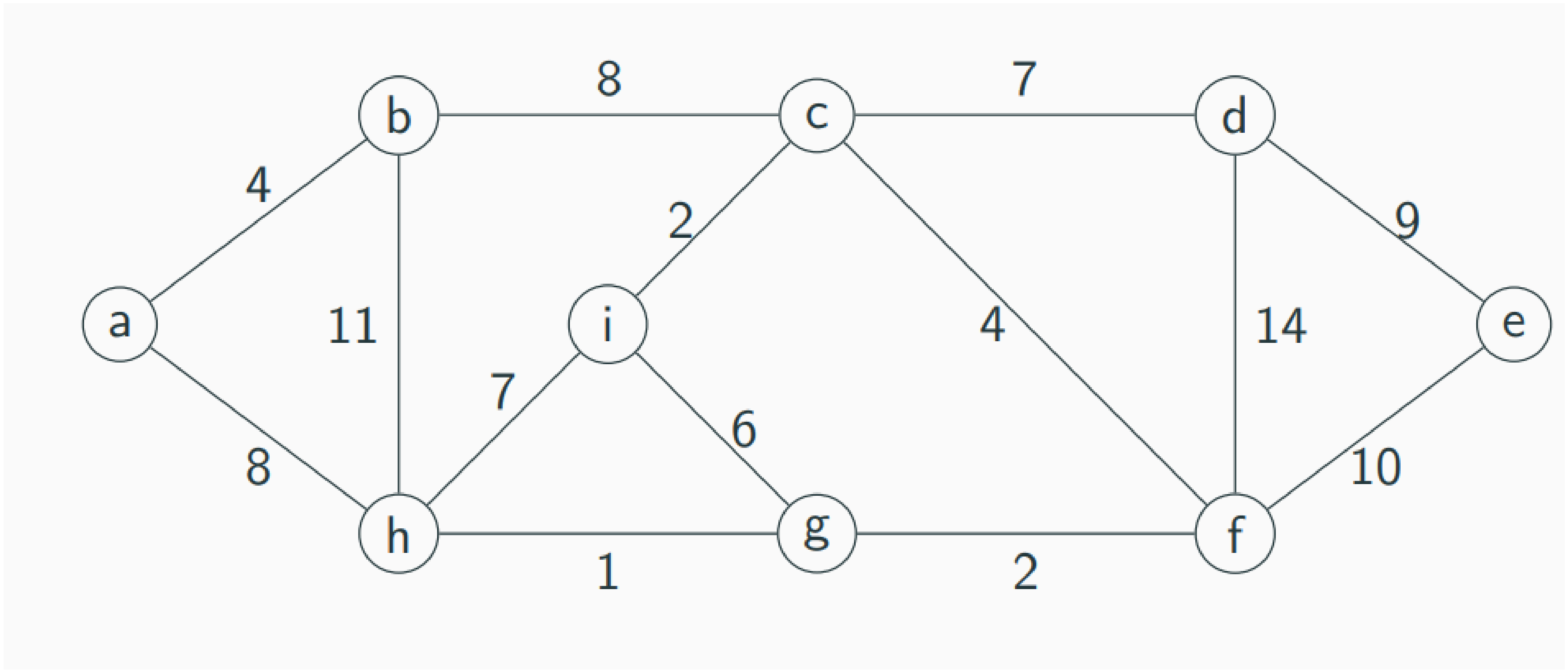
- Algorithm terminates when all vertices are connected into one tree

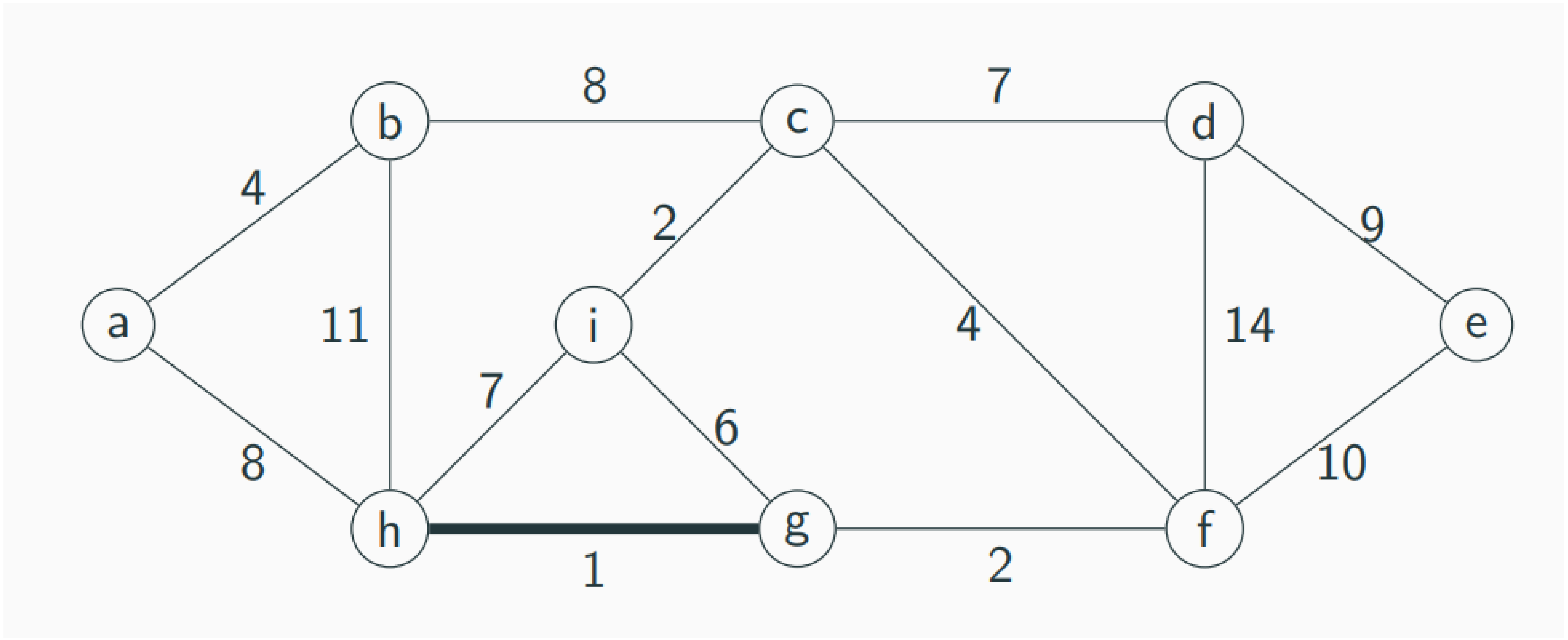
PSEUDOCODE FOR KRUSKAL'S ALGORITHM

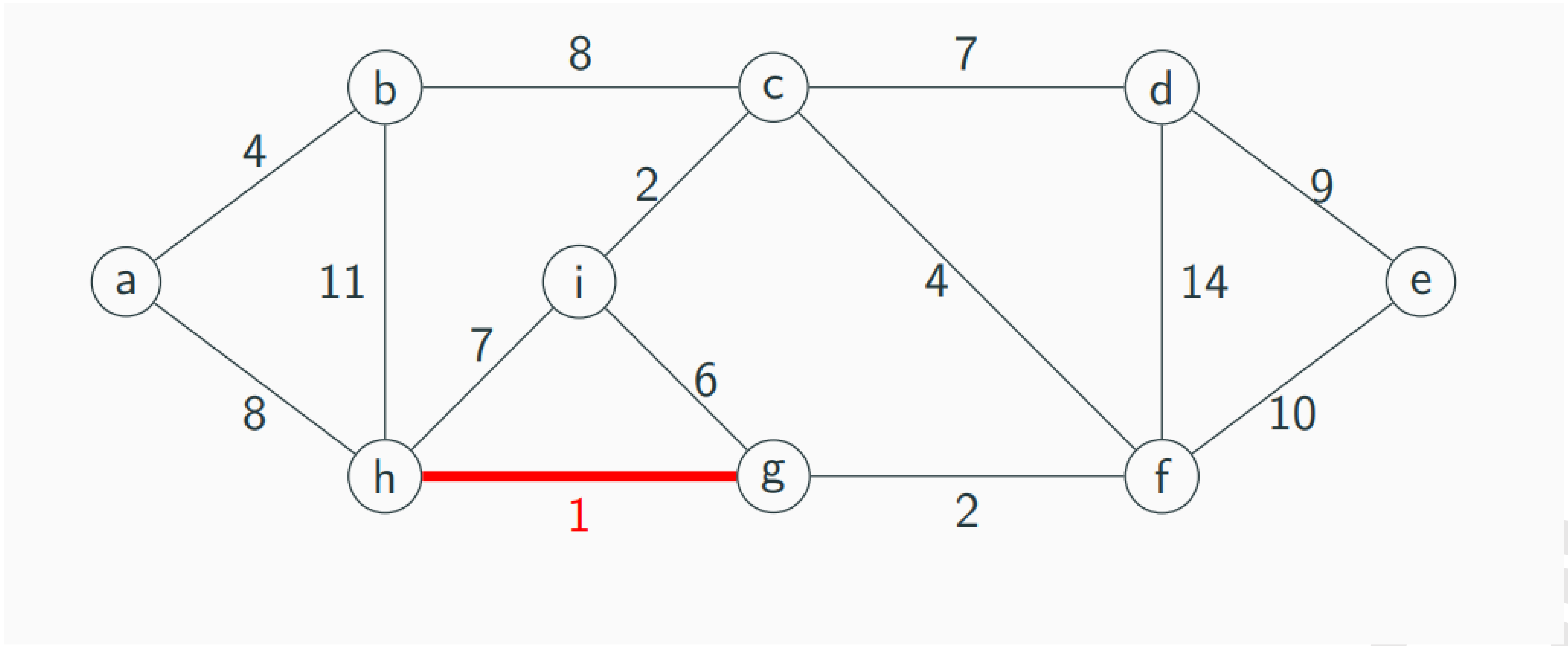
```
def kruskal():
    mst = new SomeSet<Edge>()

    for (v : vertices):
        makeMST(v)

    sort edges in ascending order by their weight

    for (edge : edges):
        if findMST(edge.src) != findMST(edge.dst):
            union(edge.src, edge.dst)
            mst.add(edge)

    return mst
```

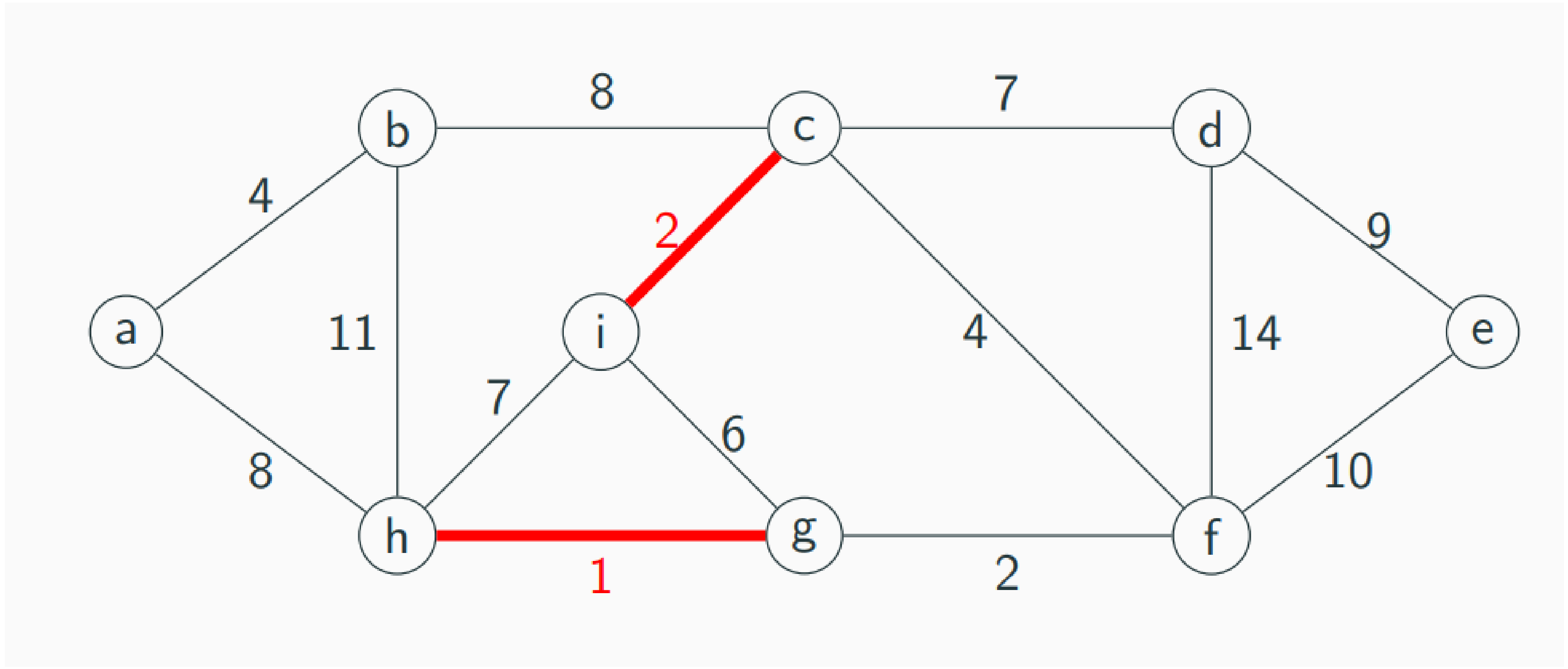
- `makeMST(v)`: stores v as a MST containing just one node
- `findMST(v)`: finds the MST that vertex is a part of
- `union(u, v)`: combines the two MSTs of the two given vertices, using the edge (u, v)

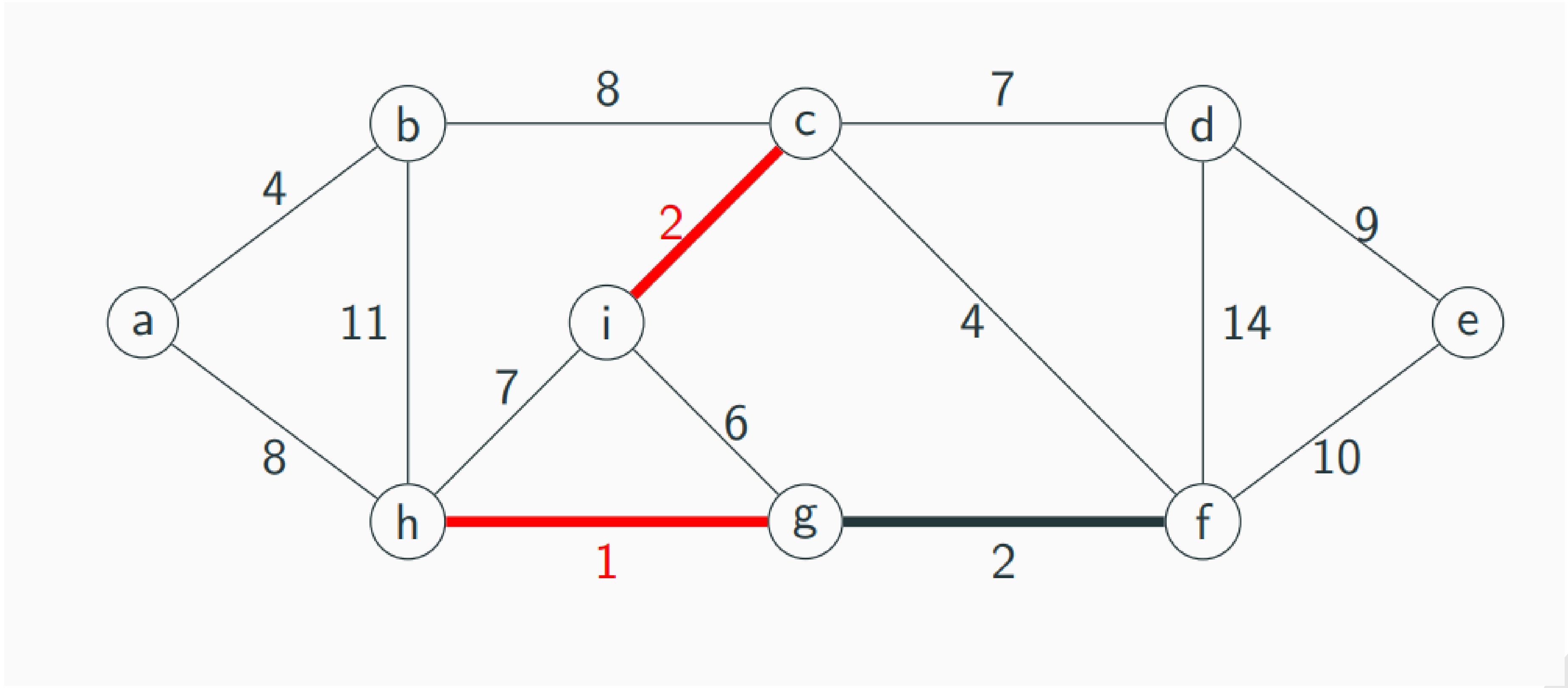
KRUSKAL'S ALGORITHM: EXAMPLE WITH A WEIGHTED GRAPH

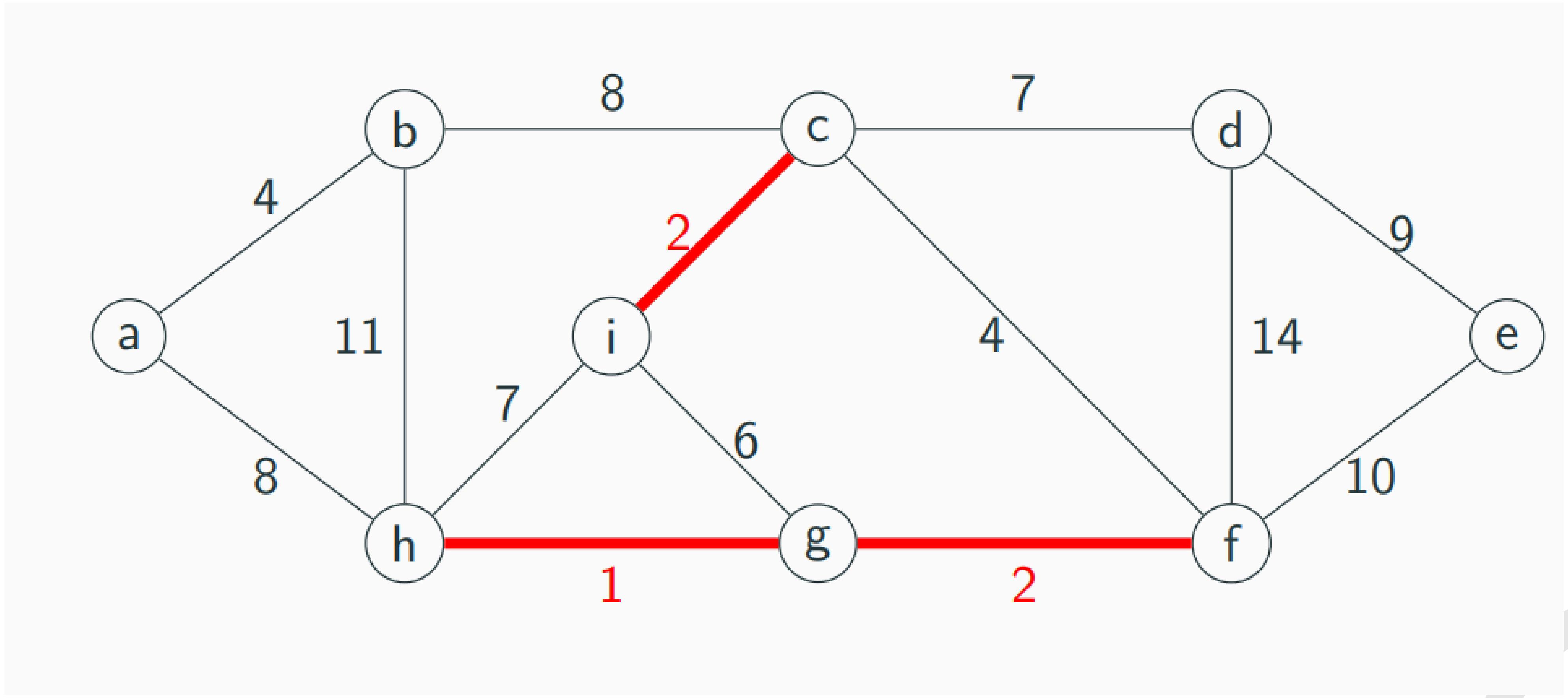



a
b
c
d
e
f
g
h
i
4
8
8
7
11
2
9
14
4
7
6
10
1
2

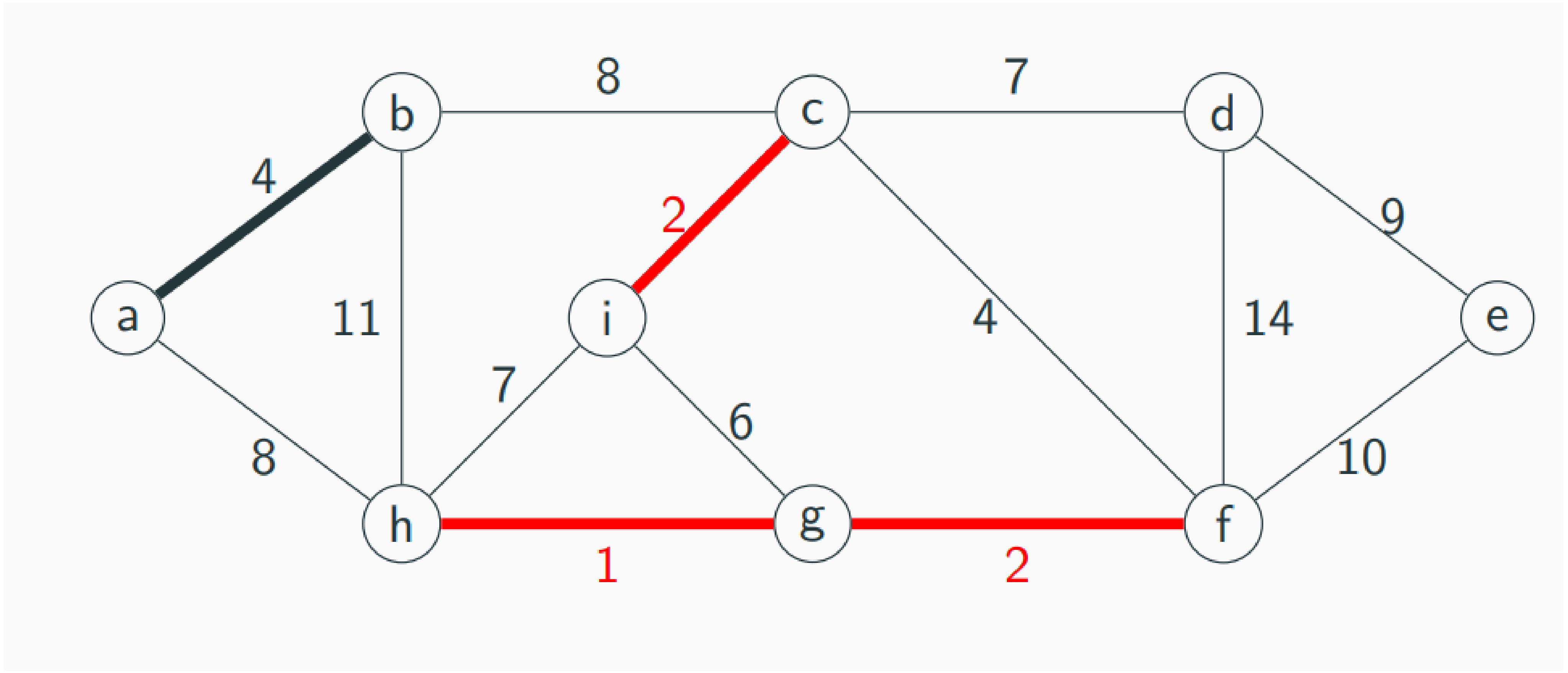

a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
7
6
1
2
4
14
9
10

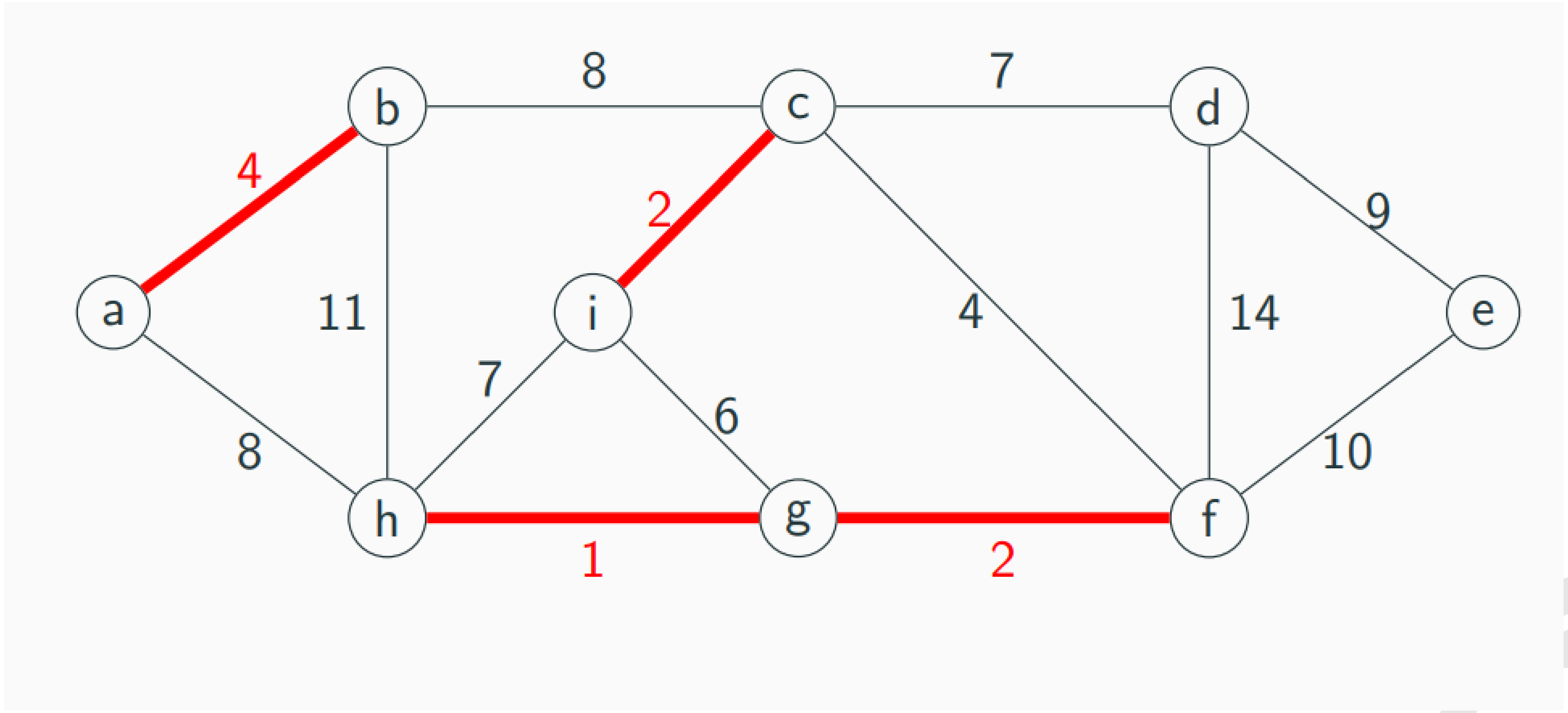

a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
7
6
1
2
4
14
9
10


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

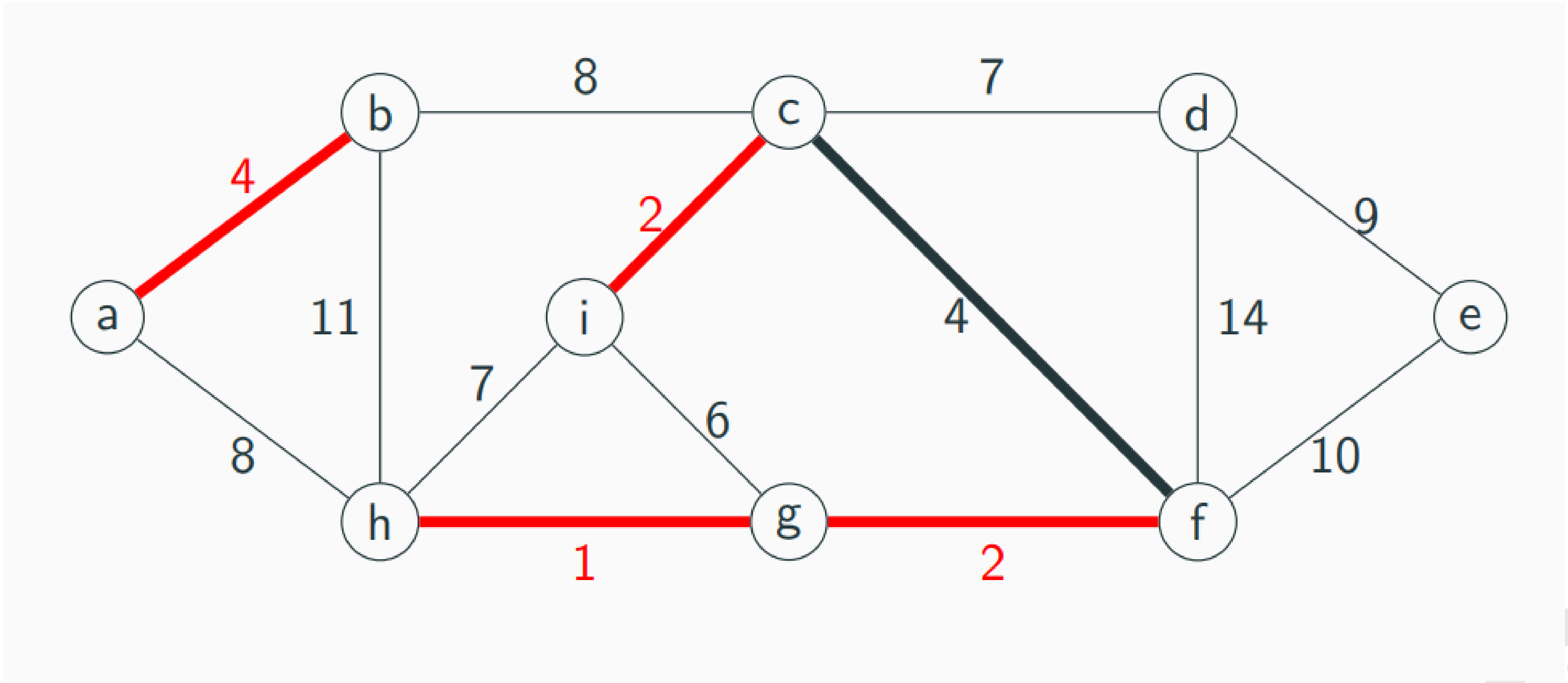


PennState

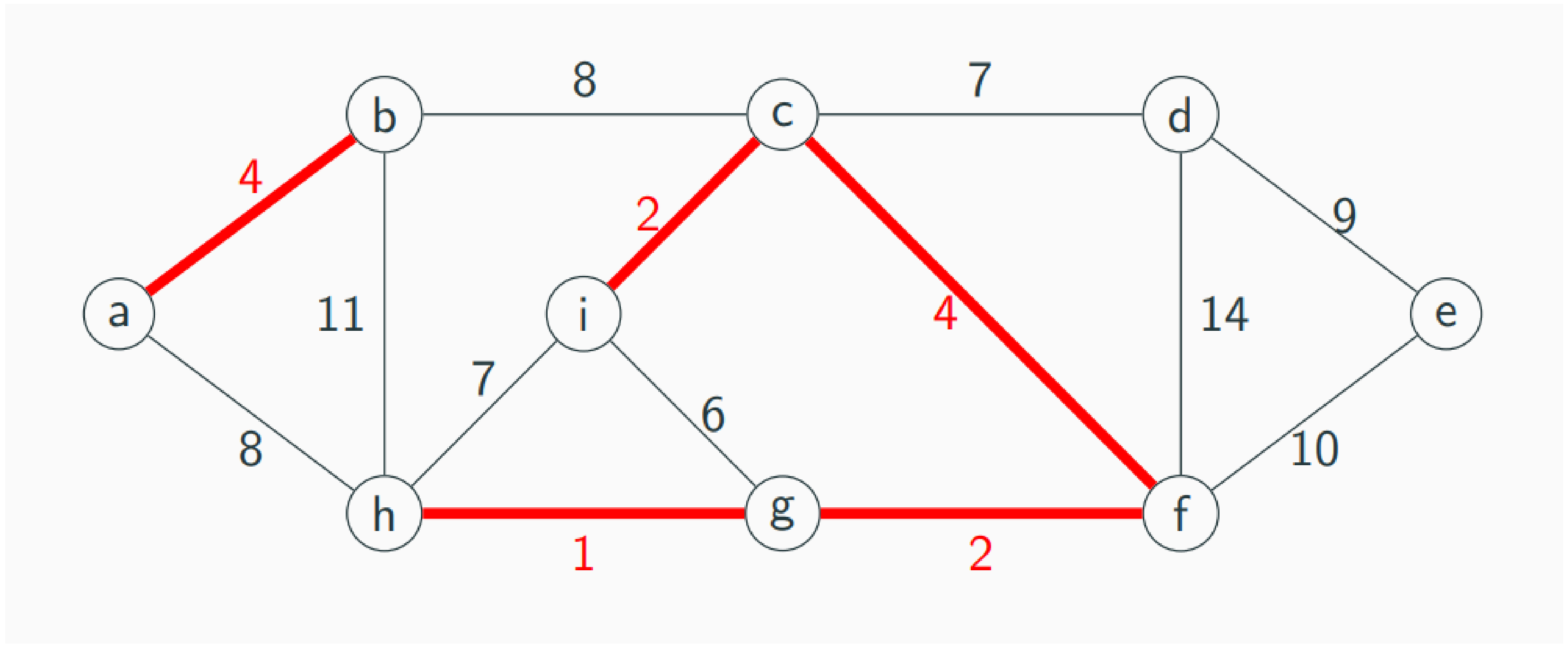

a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
8
10
1
2

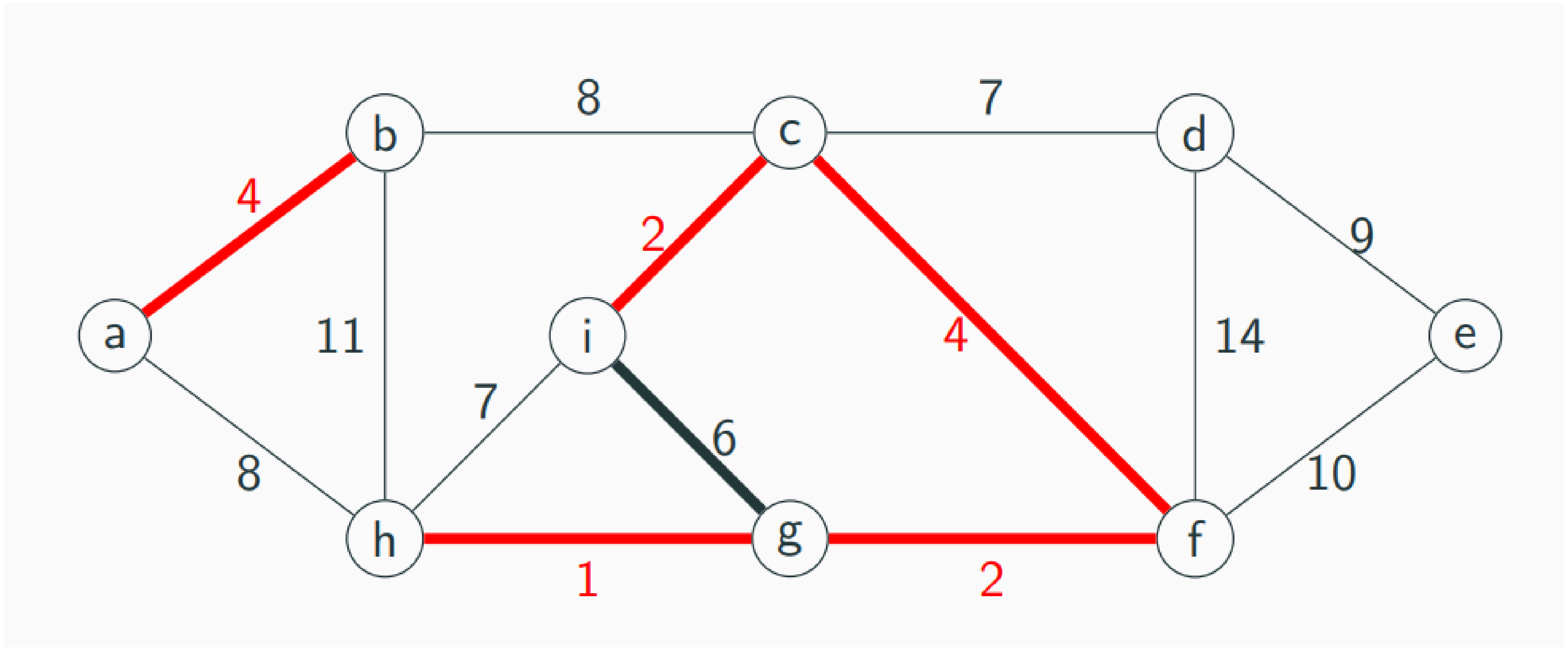


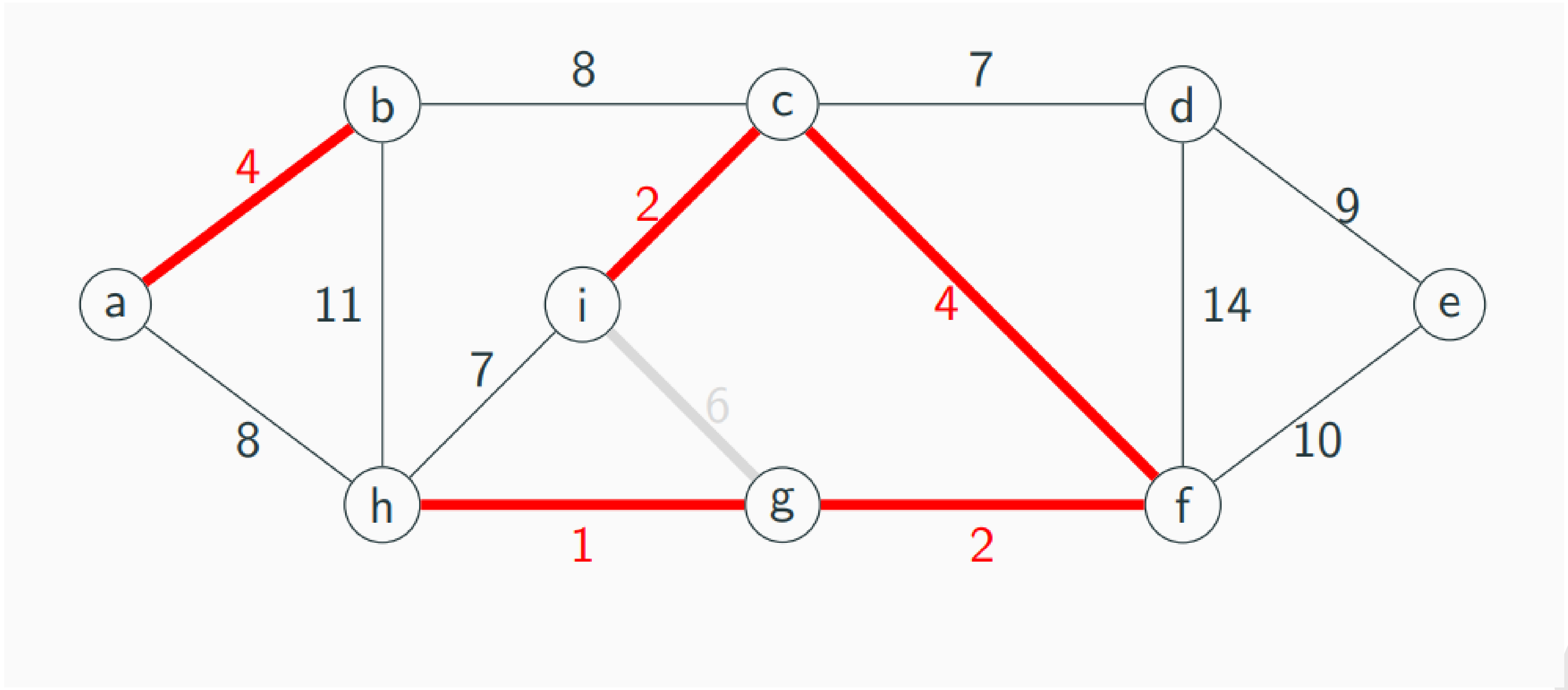
PennState



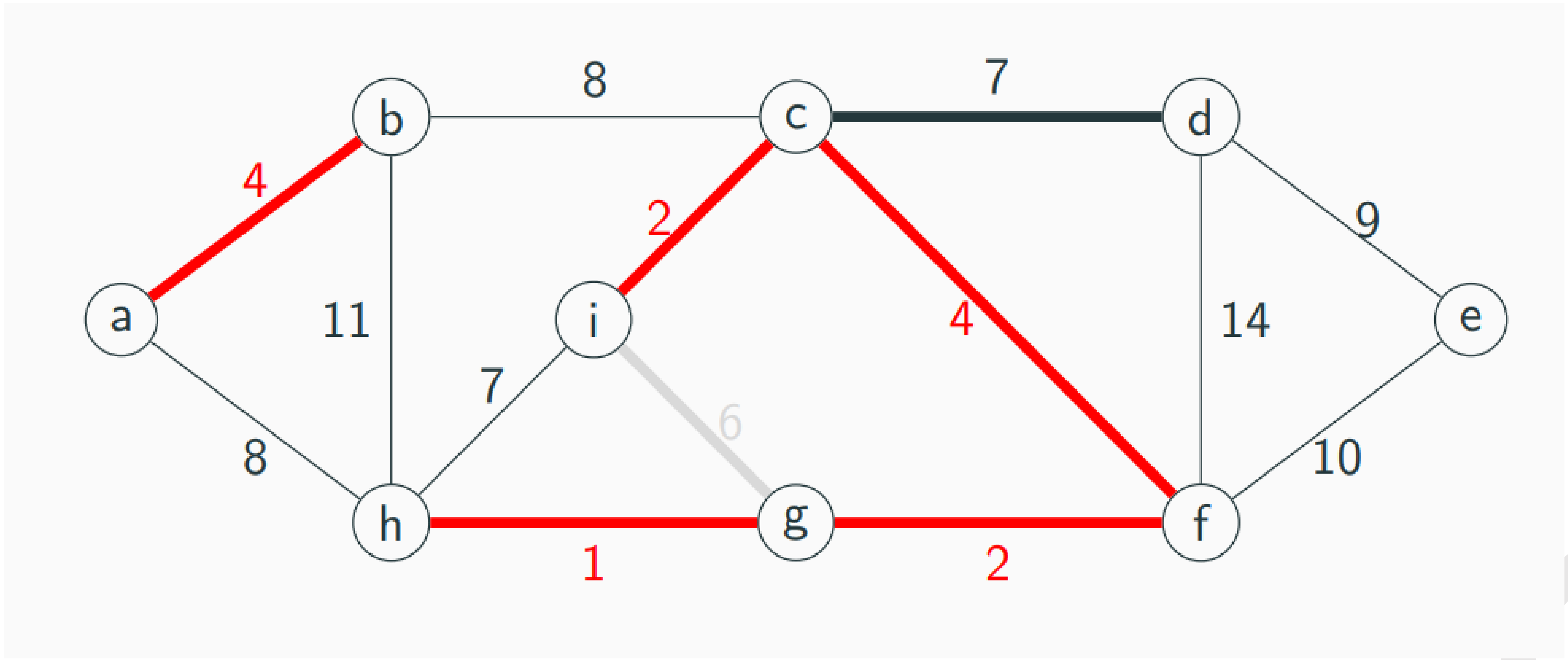
PennState

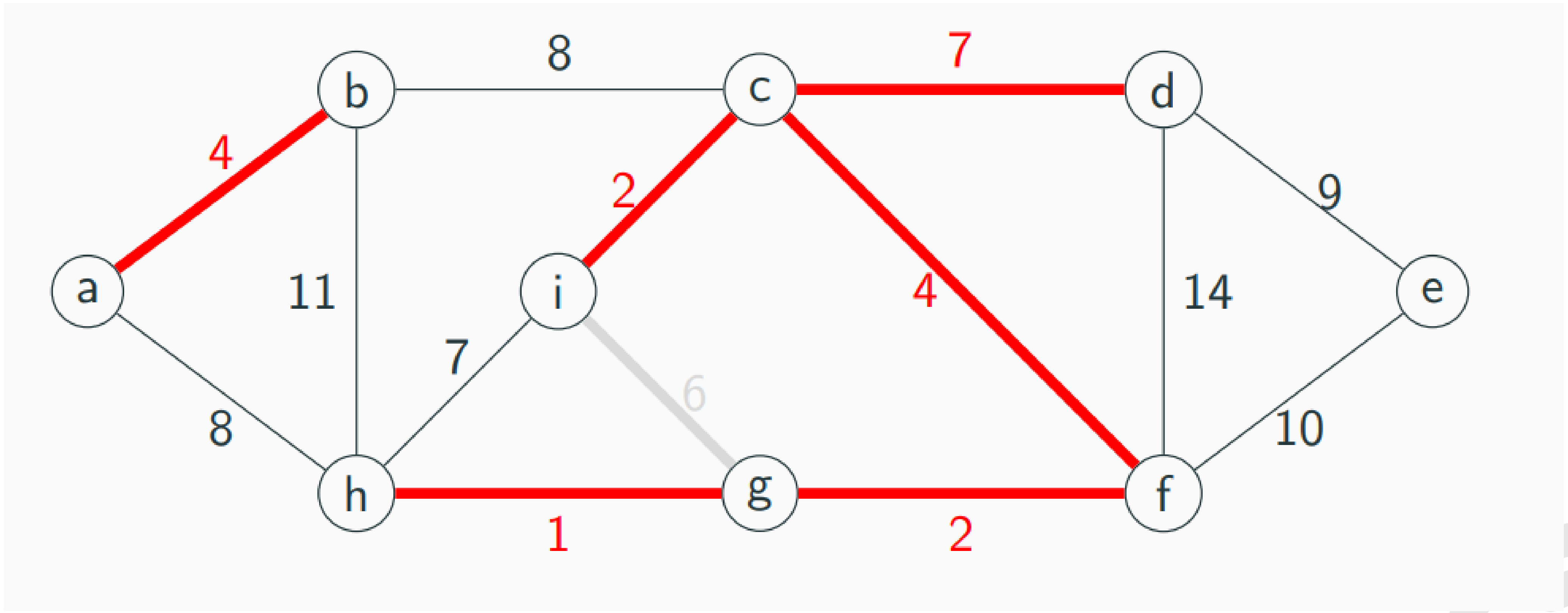

b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
4
14
7
6
8
10
1
2

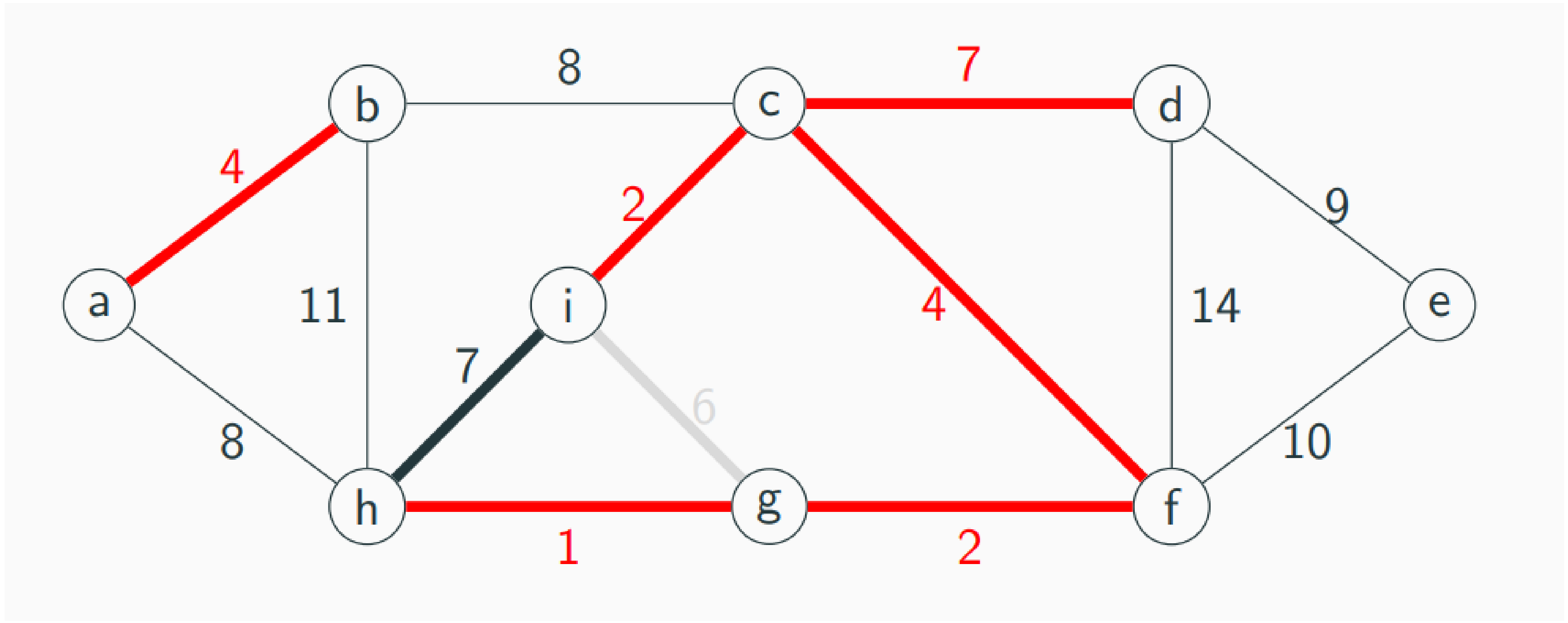

a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
6
7
4
14
9
10
1
2

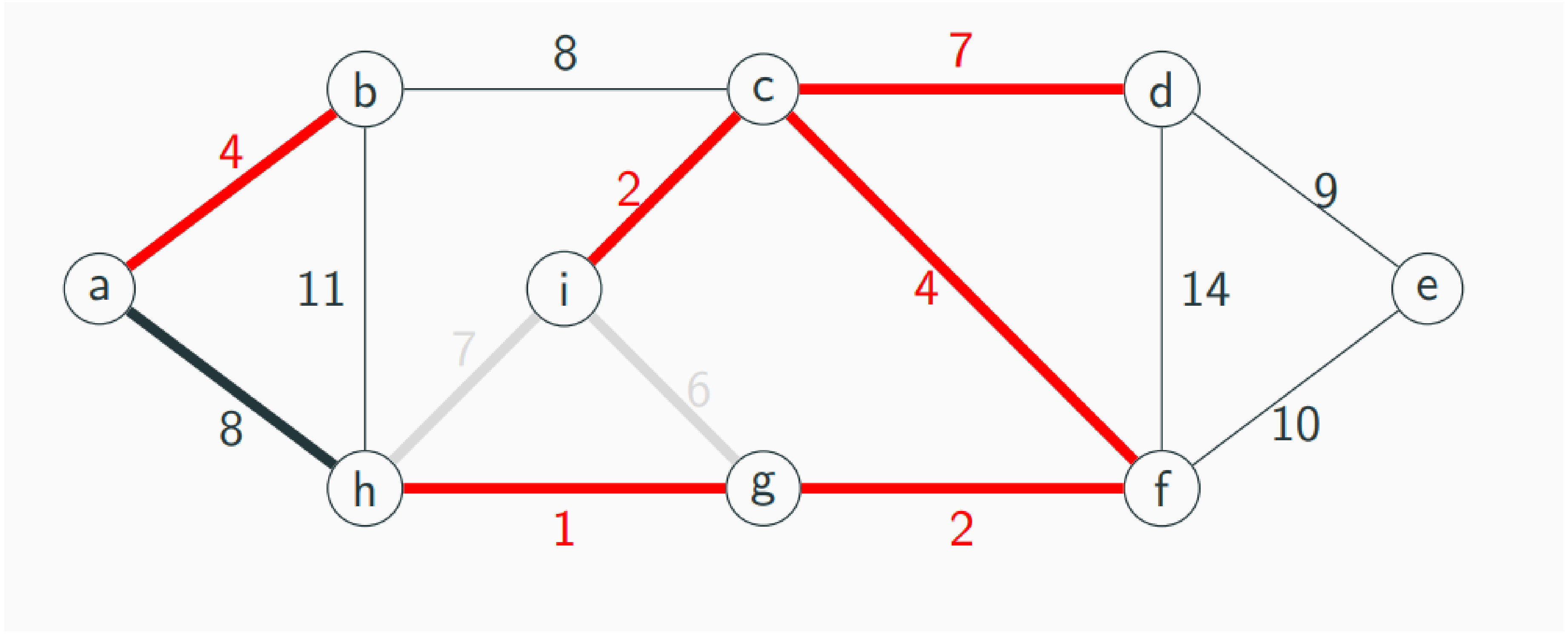


a
b
c
d
e
f
g
h
i
4
8
7
8
11
2
4
14
9
7
6
10
1
2

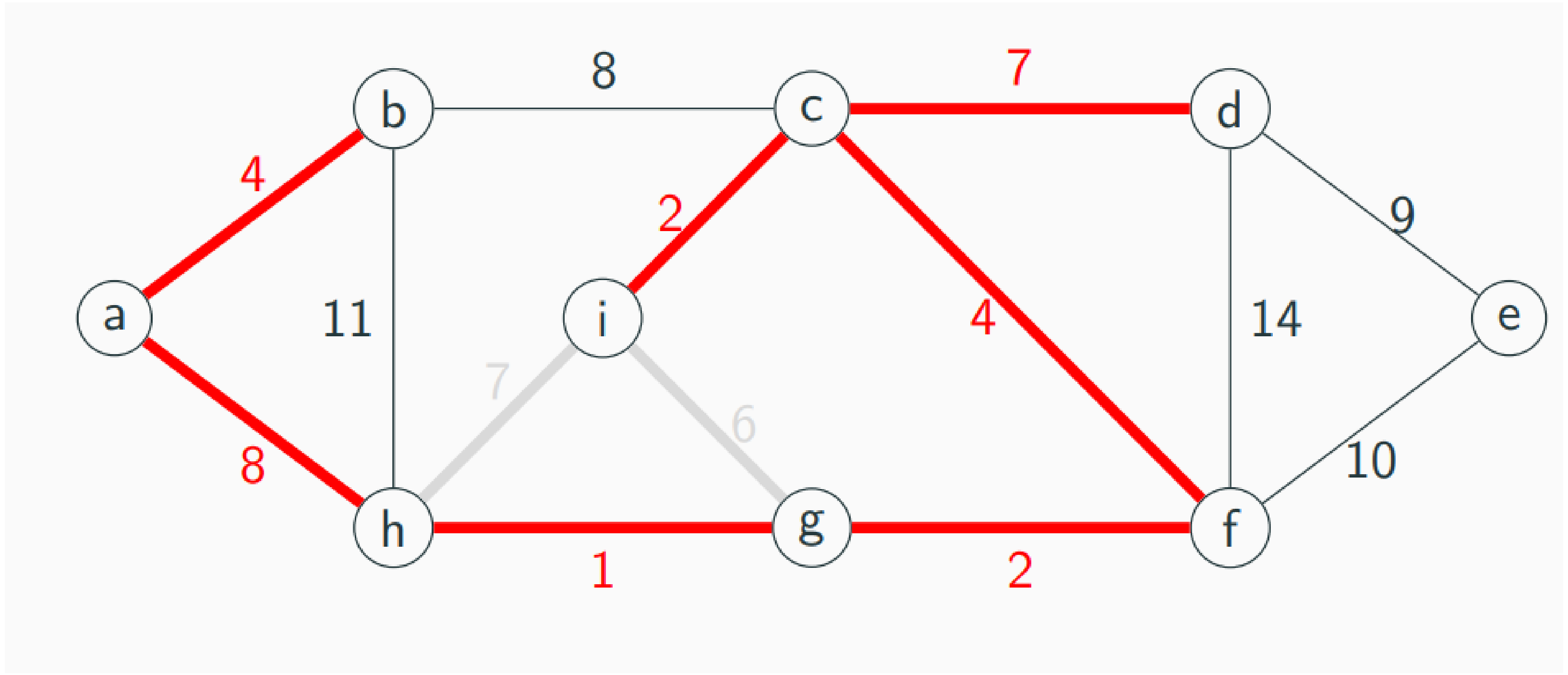

a
b
c
d
e
f
g
h
i
4
8
7
11
2
4
14
9
7
6
8
1
2
10


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

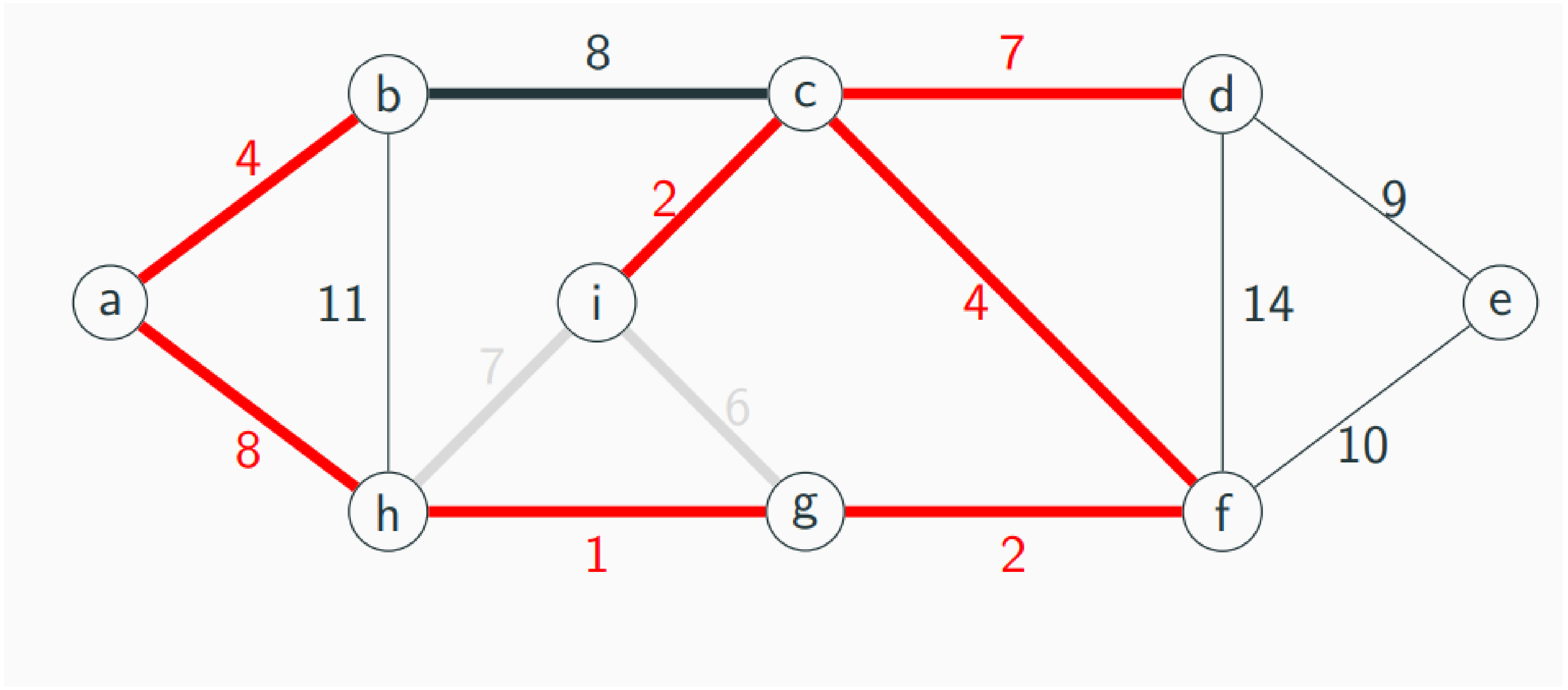

a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
8
1
2
10



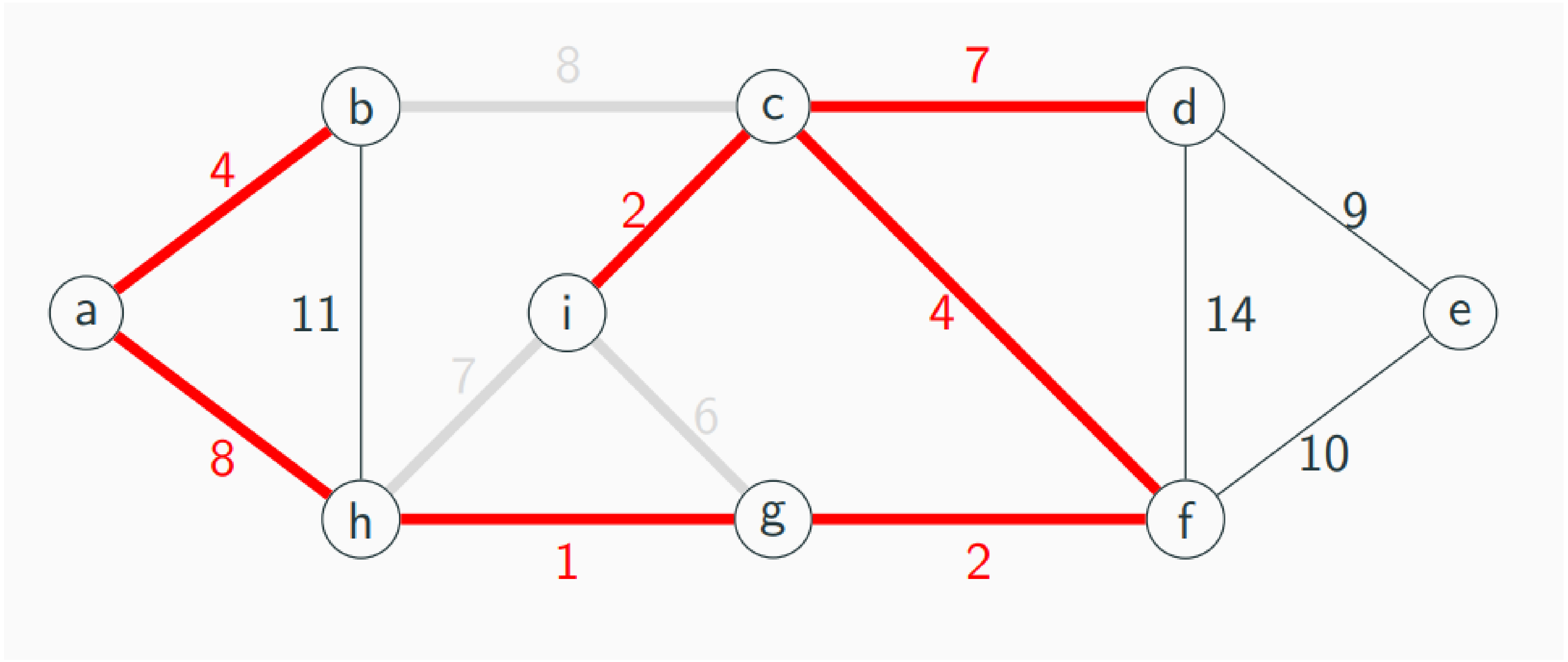
a
b
c
d
e
f
g
h
i
4
8
7
11
2
9
14
4
7
6
8
1
2
10


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

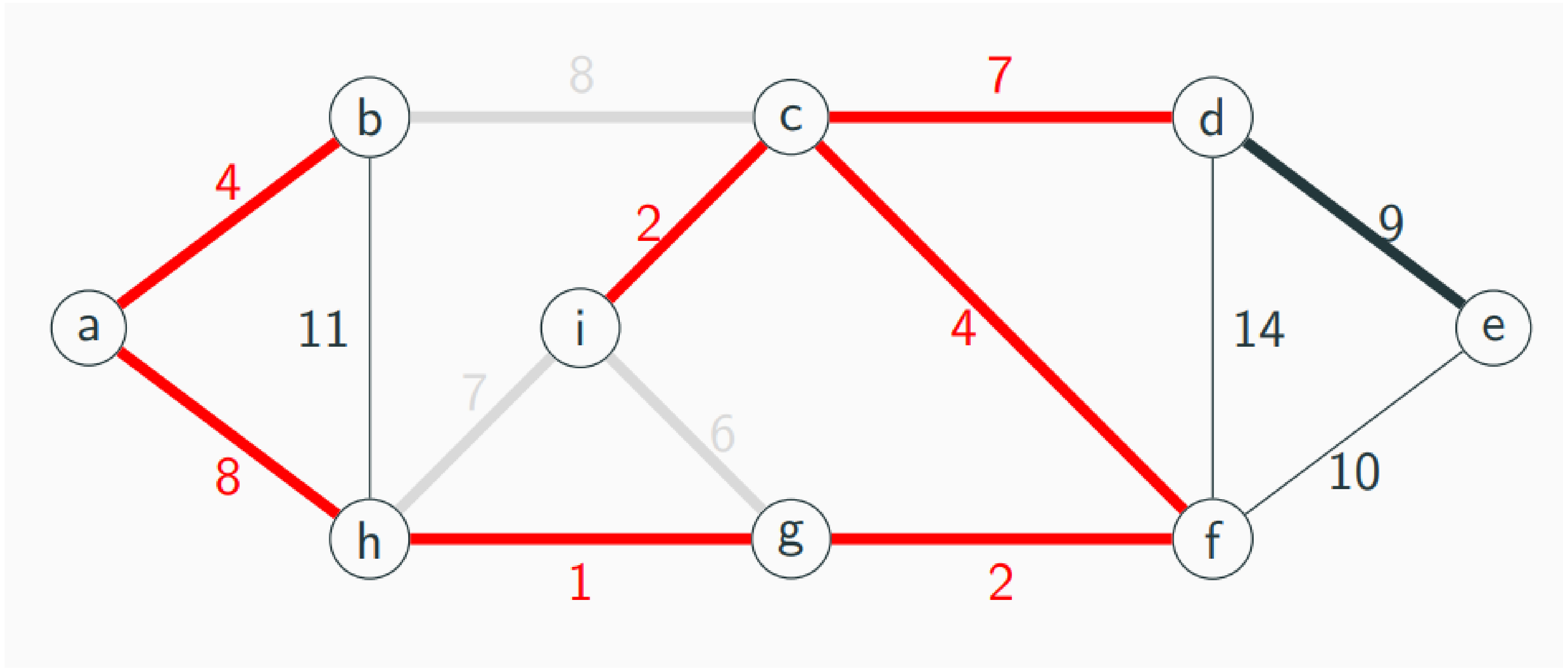
KRUSKAL'S ALGORITHM: EXAMPLE WITH A WEIGHTED GRAPH



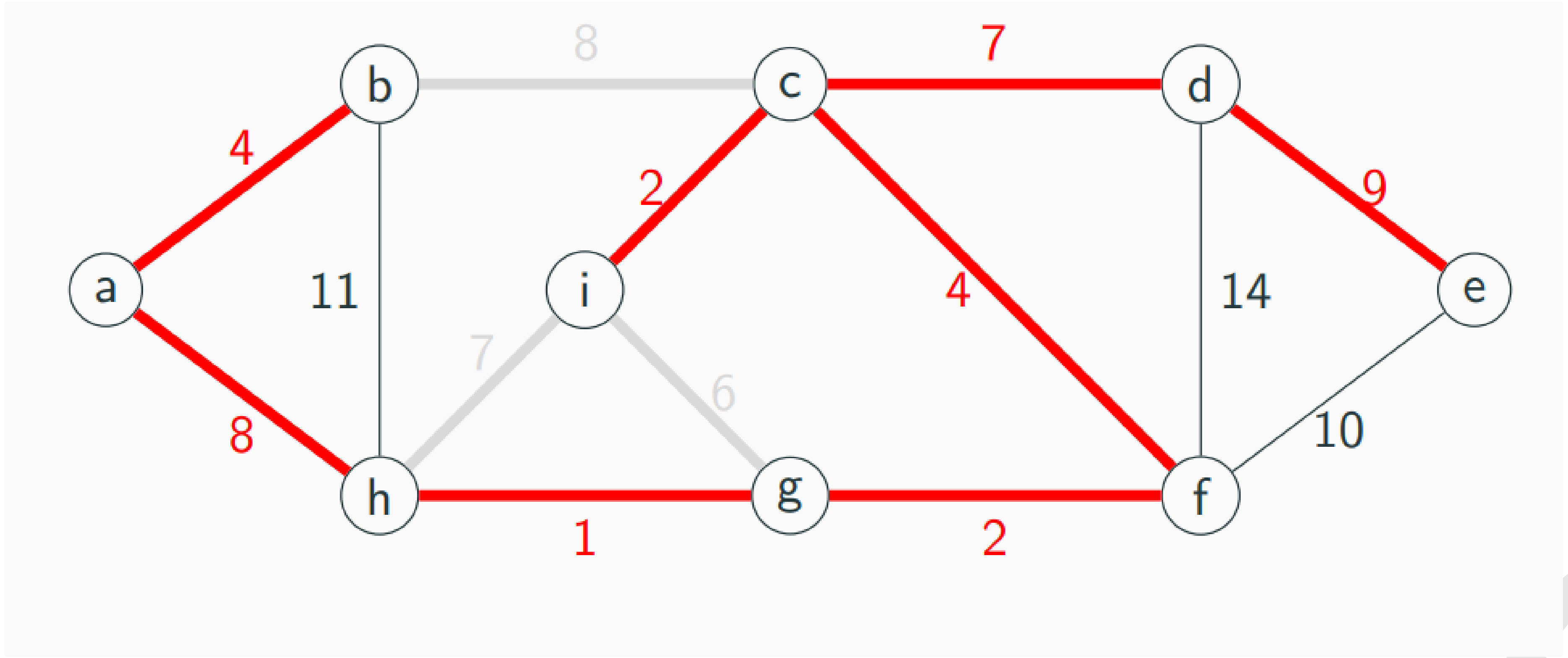
KRUSKAL'S ALGORITHM: EXAMPLE WITH A WEIGHTED GRAPH



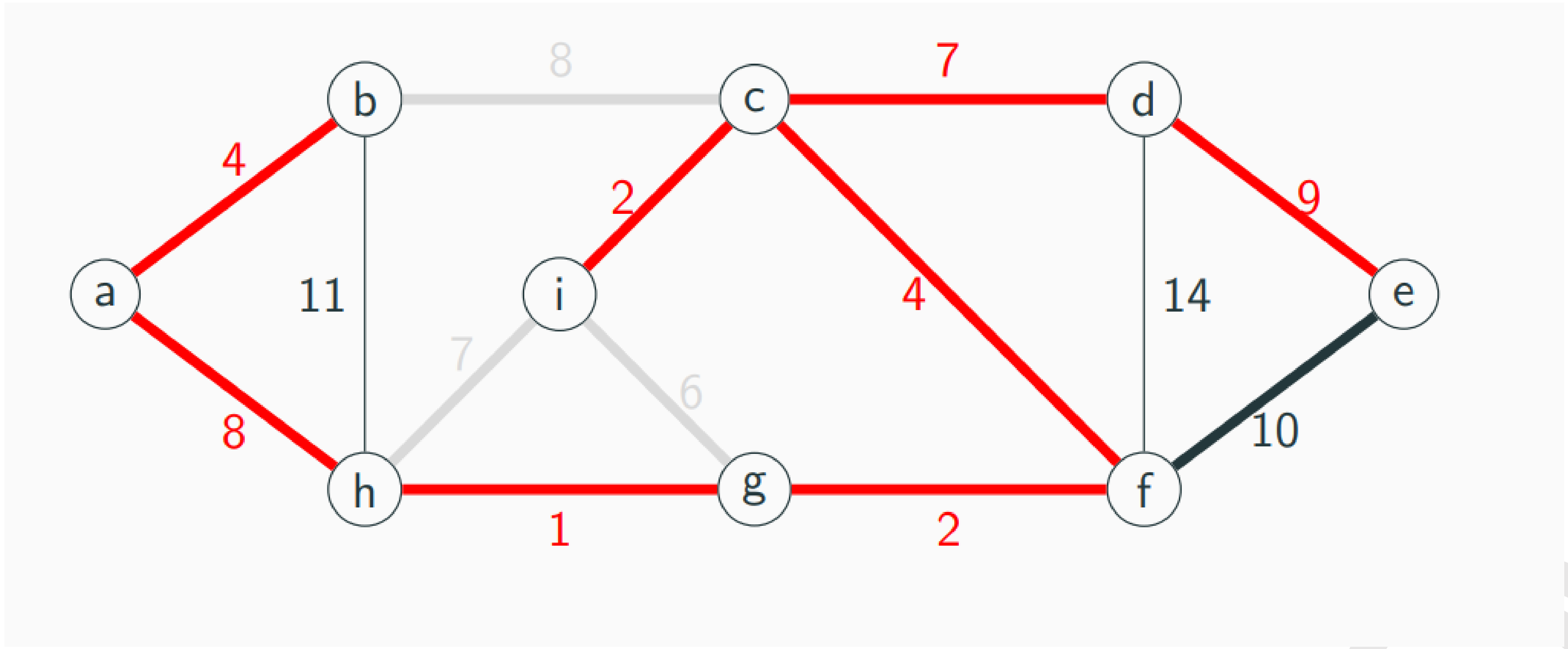
KRUSKAL'S ALGORITHM: EXAMPLE WITH A WEIGHTED GRAPH



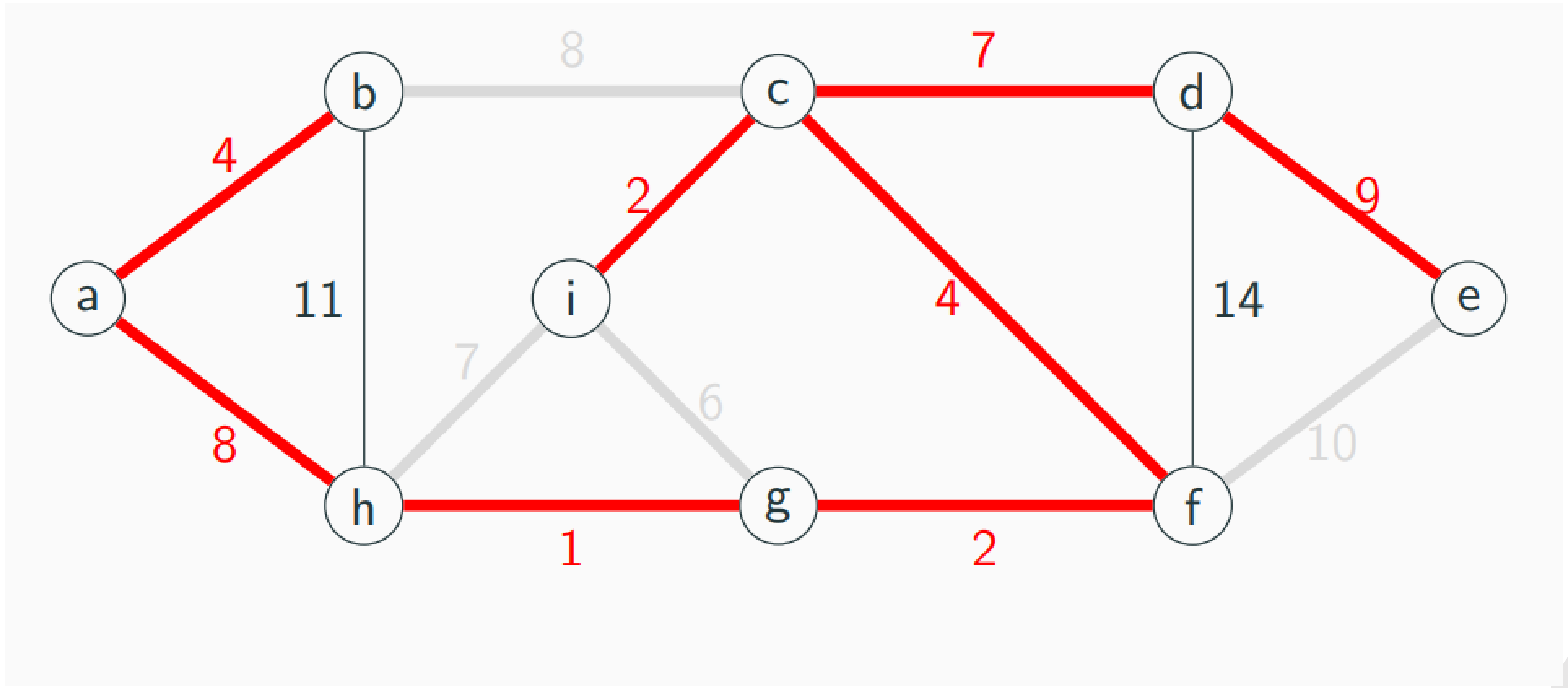
KRUSKAL'S ALGORITHM: EXAMPLE WITH A WEIGHTED GRAPH



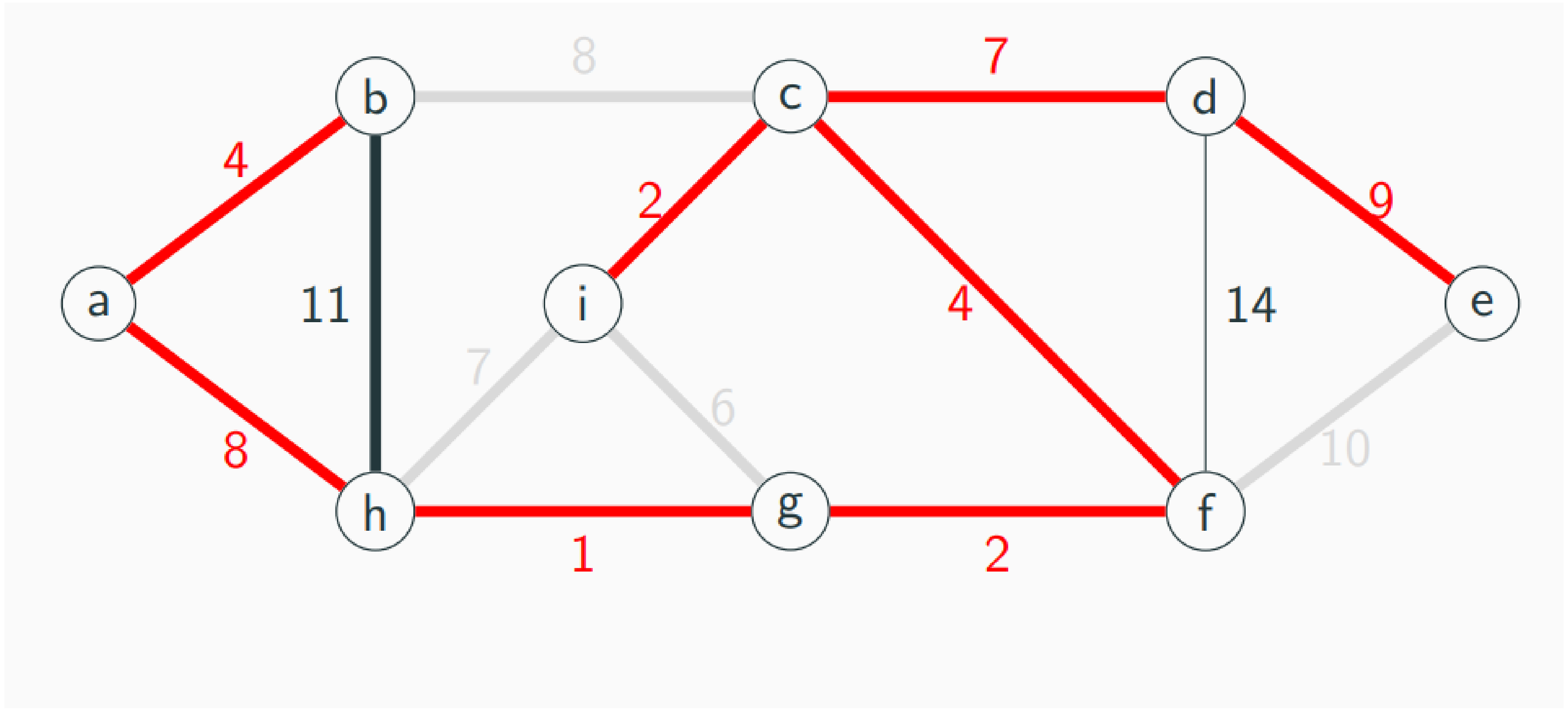
KRUSKAL'S ALGORITHM: EXAMPLE WITH A WEIGHTED GRAPH



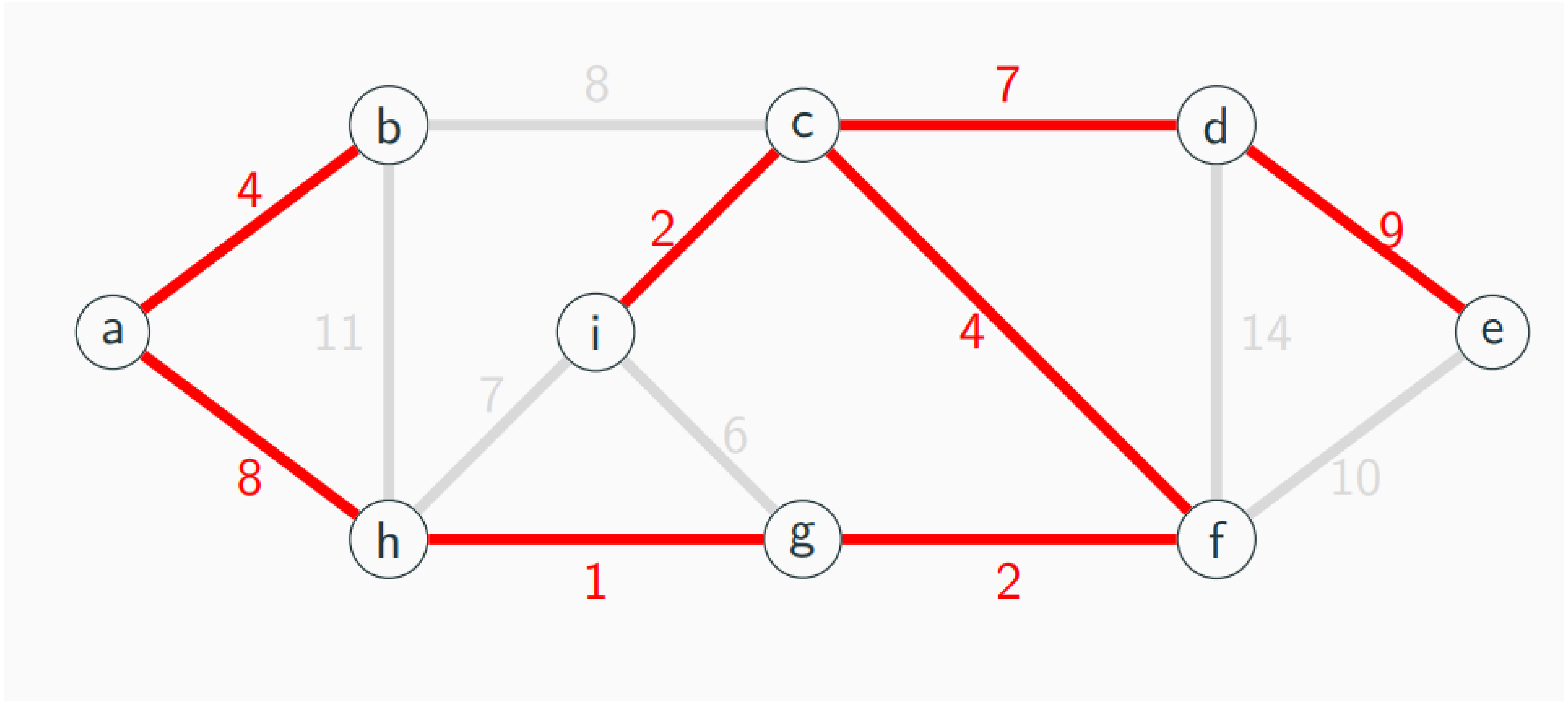
KRUSKAL'S ALGORITHM: EXAMPLE WITH A WEIGHTED GRAPH



KRUSKAL'S ALGORITHM: EXAMPLE WITH A WEIGHTED GRAPH

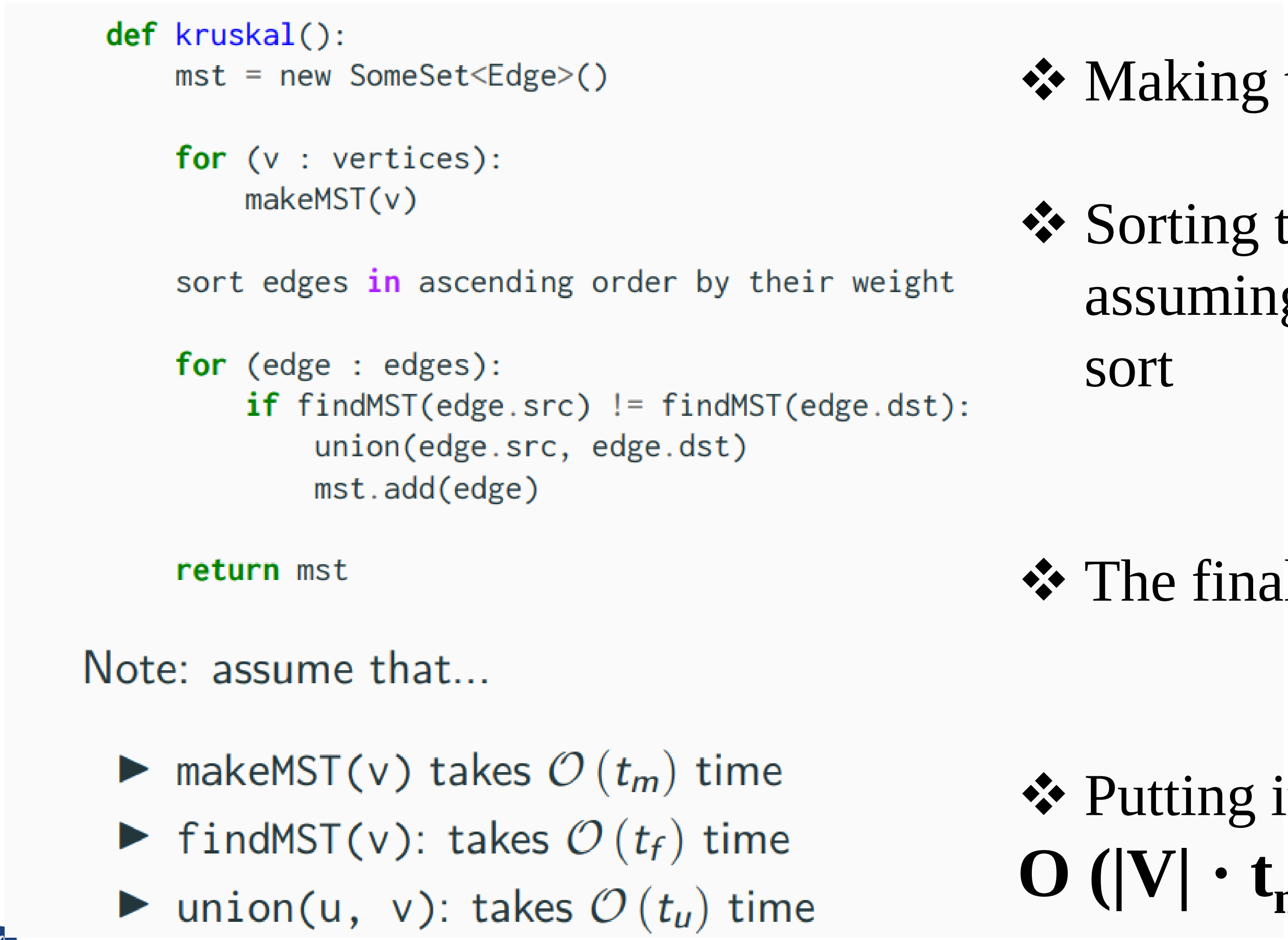


KRUSKAL'S ALGORITHM: EXAMPLE WITH A WEIGHTED GRAPH



PennState

KRUSKAL'S ALGORITHM: WHAT IS THE WORST-CASE RUNTIME?



```
def kruskal():
    mst = new SomeSet<Edge>()

    for (v : vertices):
        makeMST(v)

    sort edges in ascending order by their weight

    for (edge : edges):
        if findMST(edge.src) != findMST(edge.dst):
            union(edge.src, edge.dst)
            mst.add(edge)

    return mst
```

Note: assume that...

- makeMST(v) takes $\mathcal{O}(t_m)$ time
- findMST(v): takes $\mathcal{O}(t_f)$ time
- union(u, v): takes $\mathcal{O}(t_u)$ time

❖ Making the |V| MSTs takes O ($|V| \cdot t_m$) time

❖ Sorting the edges takes O ($|E| \cdot \log(|E|)$) time, assuming we use a general-purpose comparison sort

❖ The final loop takes **O ($|E| \cdot t_f + |V| \cdot t_u$)** time

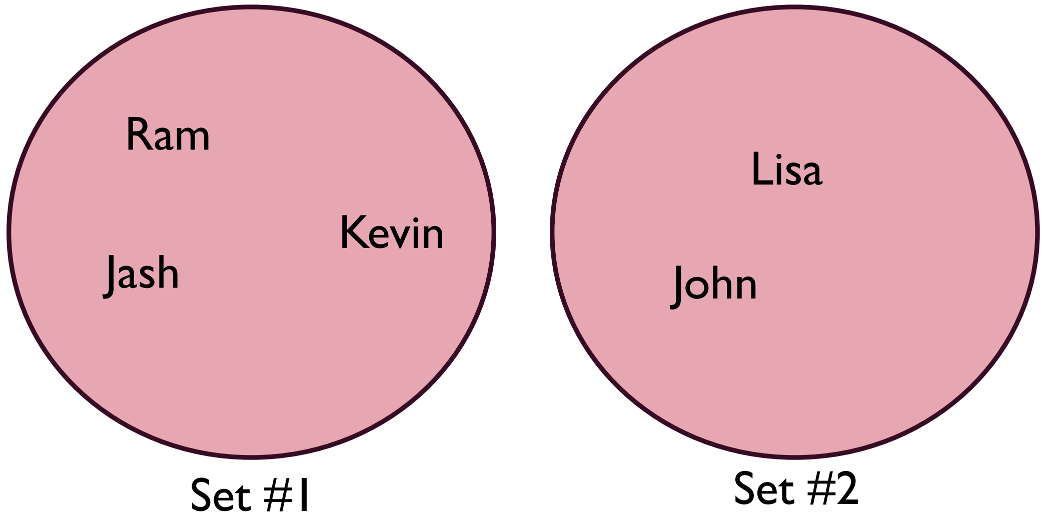
❖ Putting it all together:

O ($|V| \cdot t_m + |E| \cdot \log(|E|) + |E| \cdot t_f + |V| \cdot t_u$)

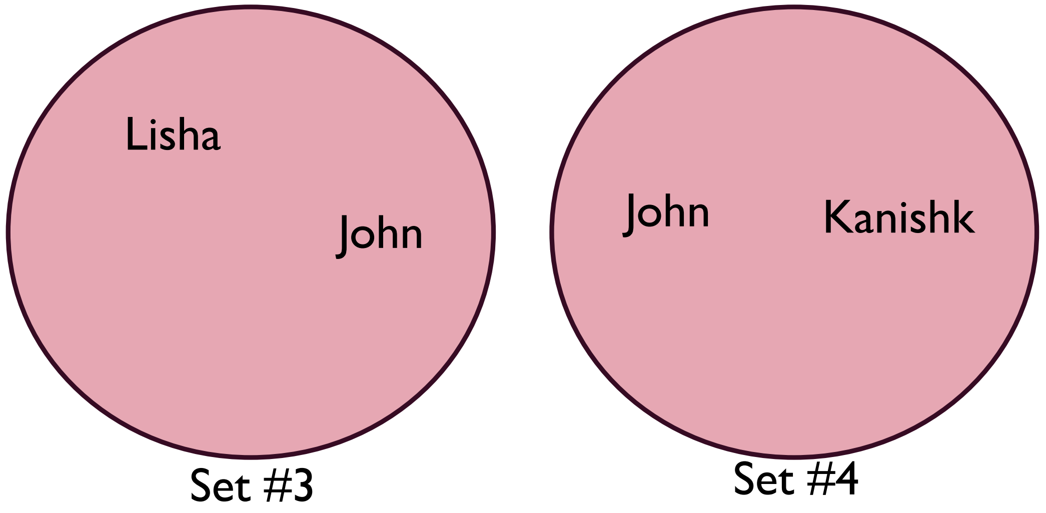
But, what exactly is t_m, t_f , and t_u?

How exactly do we implement makeMST(v), findMST(v), and union(u, v)? We can do so using a new ADT called the DisjointSet ADT!

DISJOINT SETS IN MATHEMATICS



These two sets are disjoint sets



These two sets are not disjoint sets

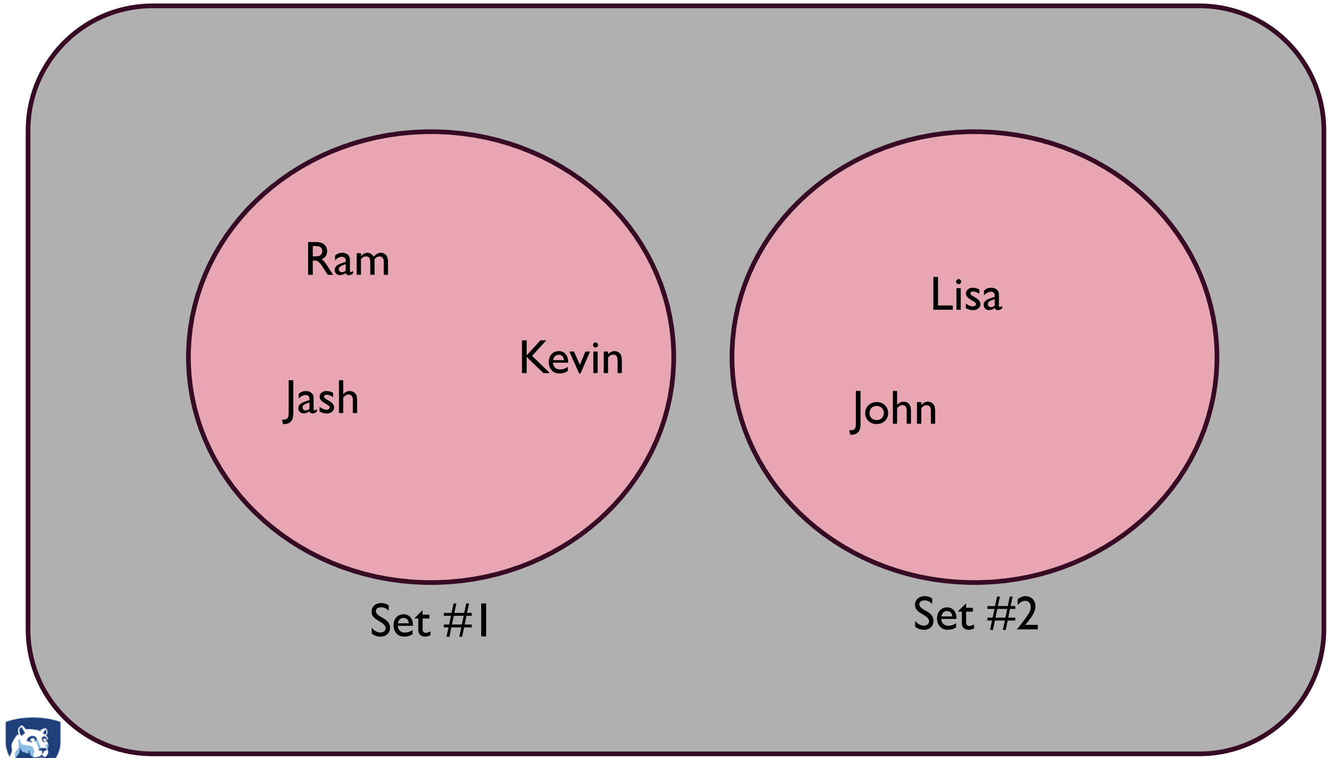
“In mathematics, two **sets** are said to be **disjoint sets** if they have no element in common.” - Wikipedia

- disjoint = not overlapping

new ADT!

DISJOINT SETS IN COMPUTER SCIENCE

- In computer science, disjointsets can refer to this ADT/data structure that keeps track of the multiple "mini" sets that are disjoint



This overall grey blob thing is the actual disjoint sets, and it's keeping track of any number of mini-sets, which are all disjoint (the mini sets have no overlapping values).

Note: this might feel really different than ADTs we've run into before. The ADTs we've seen before (dictionaries, lists, sets, etc.) just store values directly. But the Disjoint Set ADT is particularly interested in letting you group your values into sets and keep track of which particular set your values are in.

DISJOINTSETS ADT METHODS

- Just 3 methods

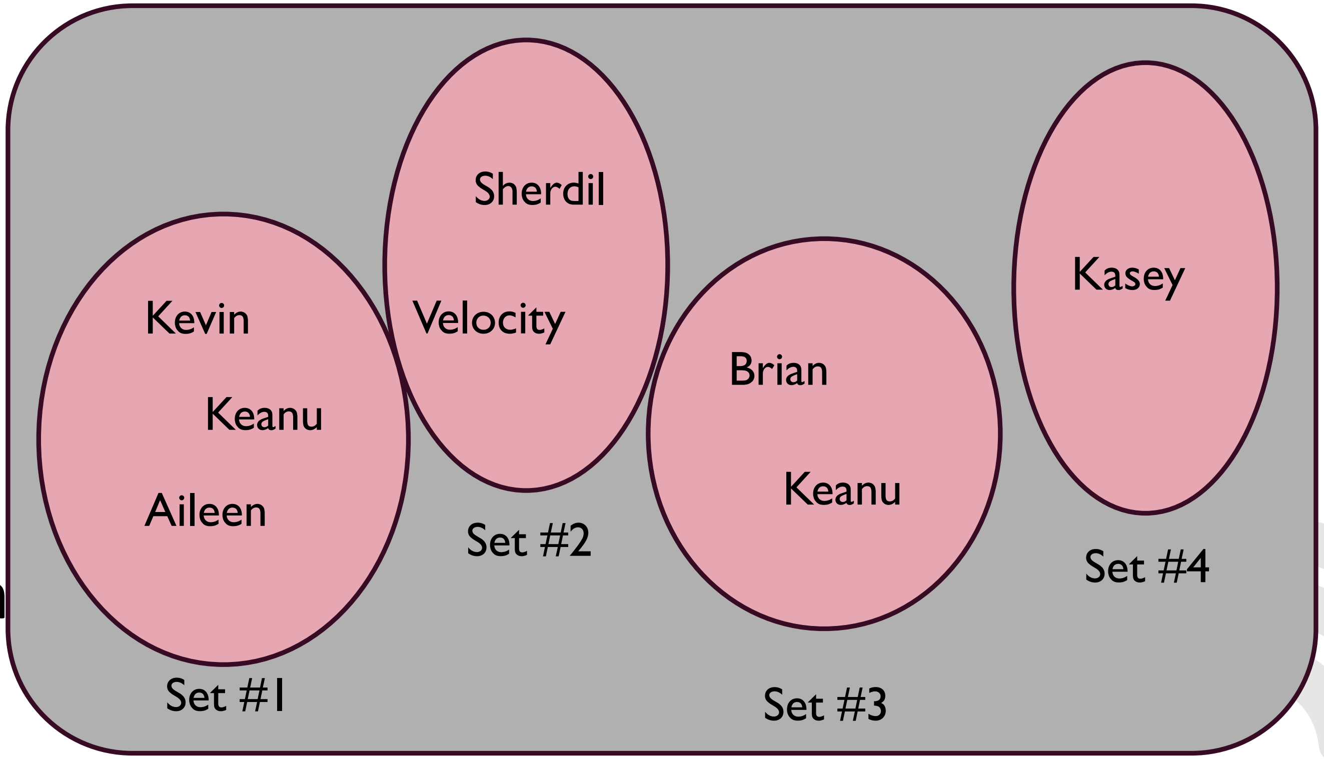
- - findSet(value)
- - union(valueA, valueB)
- - makeSet(value)

FINDSET(VALUE)

- **findSet(value)** returns some ID for which particular set the value is in. For Disjoint Sets, we often call this the **representative** (as it's a value that represents the whole set).

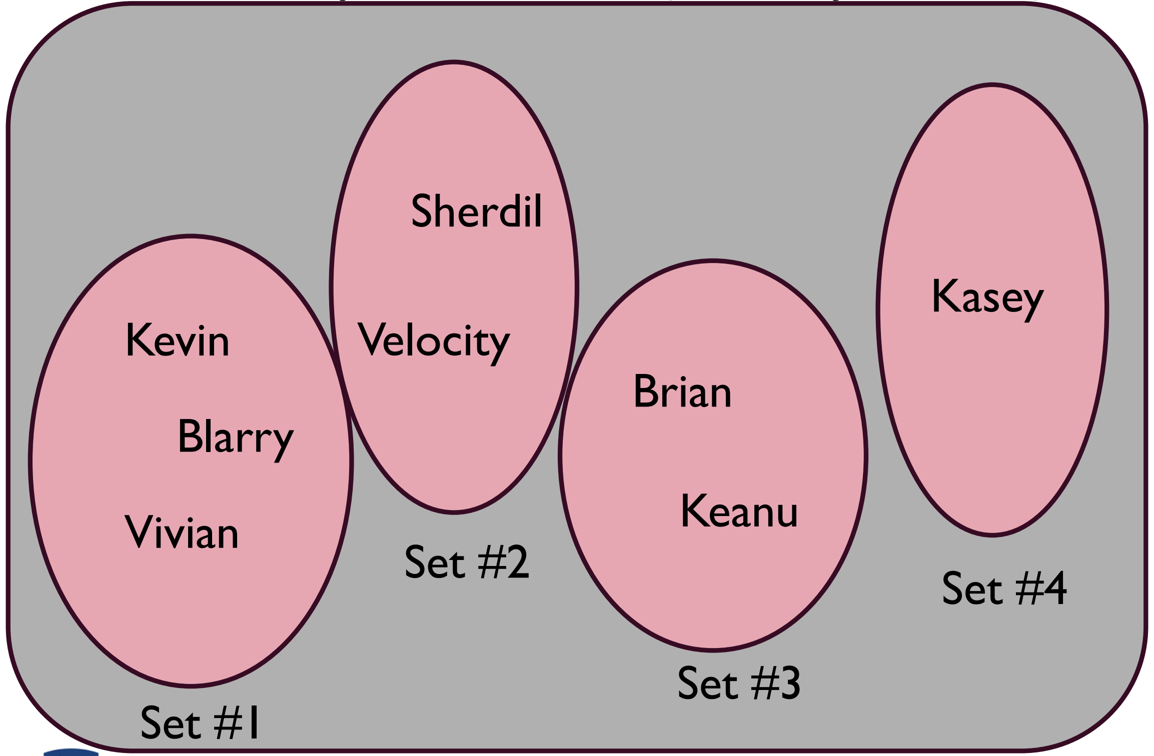
- Examples:
- findSet(Brian) 3
- findSet(Sherdil) 2
- findSet(Velocity) 2
- findSet(Kevin) == findSet(Aileen

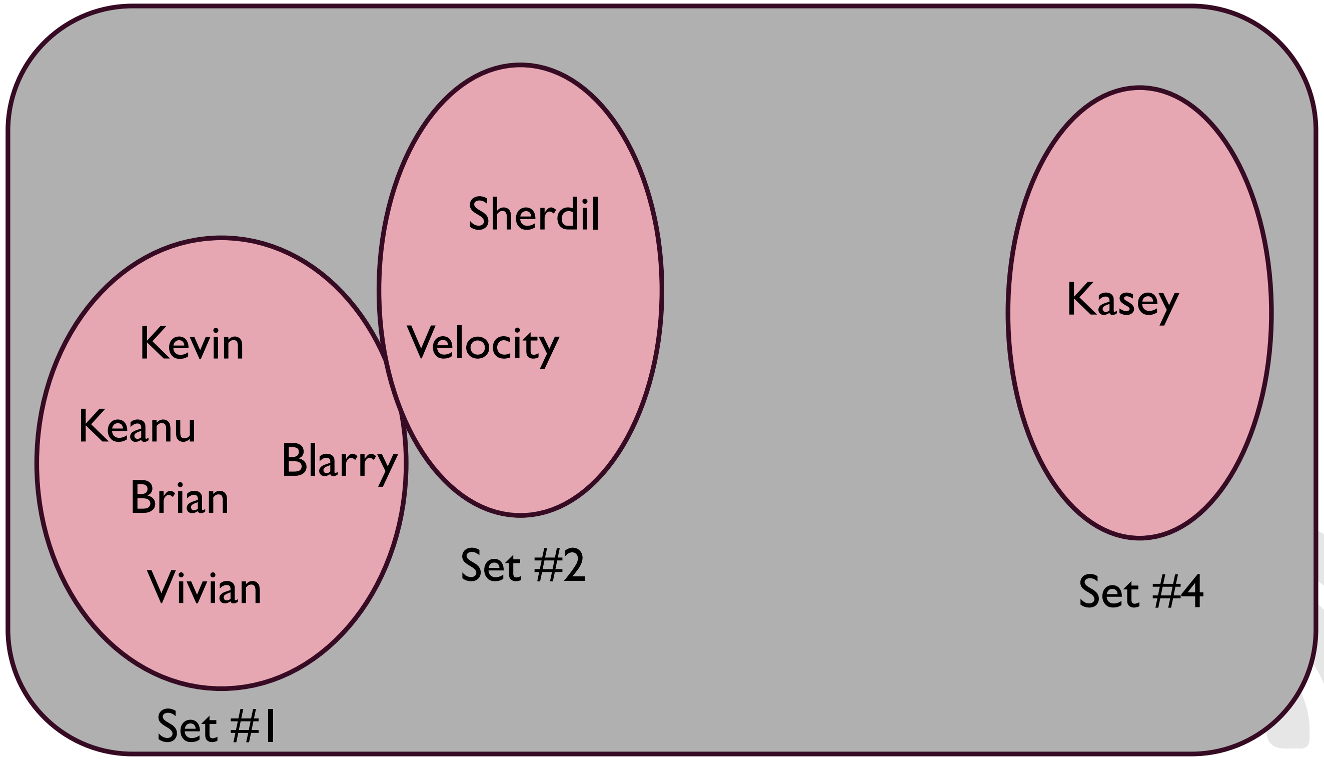
true



UNION(VALUEA,VALUEB)

- **union(valueA, valueB)** merges the set that A is in with the set that B is in. (basically add the two sets together into one)
- Example: union(Blarry,Brian)

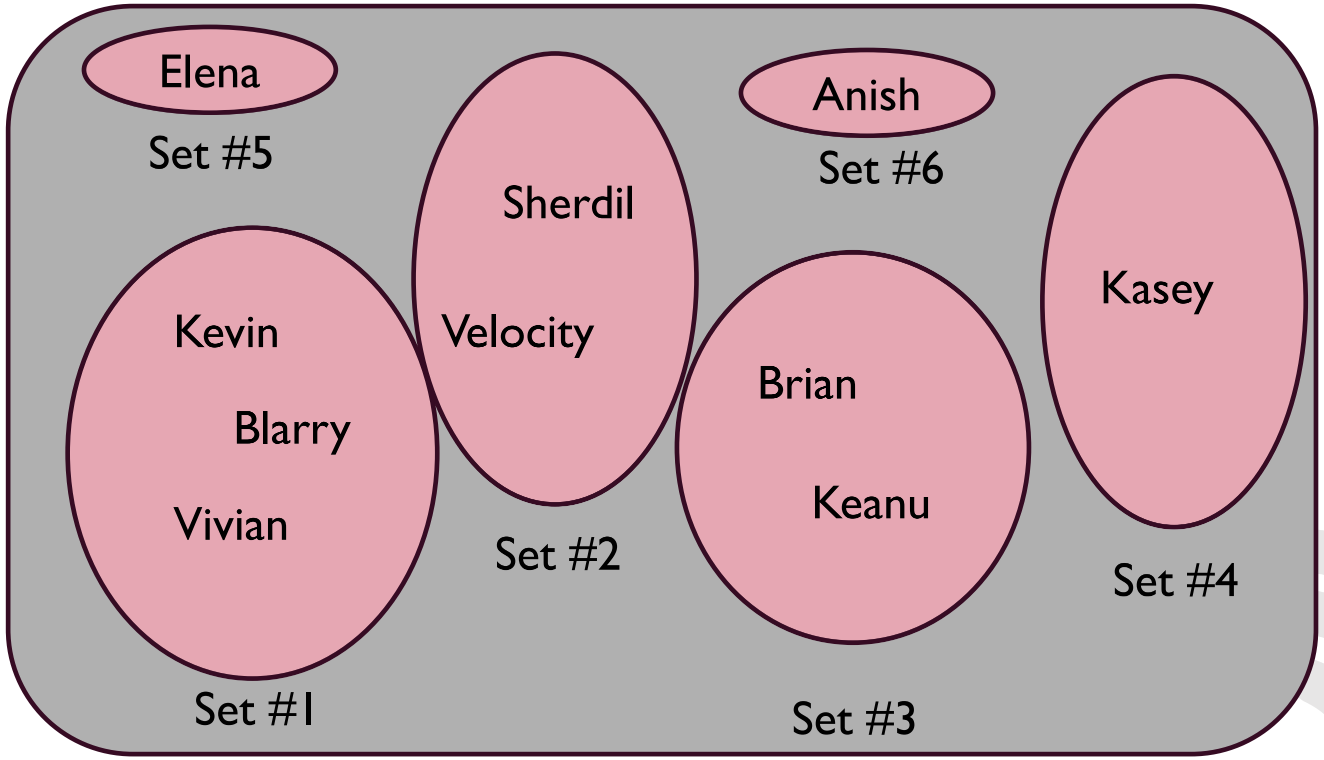




MAKESET(VALUE)

- **makeSet(value)** makes a new mini set that just has the value parameter in it.

- Examples:
- makeSet(Elena)
- makeSet(Anish)



DISJOINT SETS ADT SUMMARY

Disjoint-Sets ADT

state

Set of Sets
- **Mini sets are disjoint:** Elements must be unique across mini sets
- No required order
- Each set has id/representative

behavior

makeSet(value) – creates a new set within the disjoint set where the only member is the value. Picks id/representative for set

findSet(value) – looks up the set containing the value, returns id/representative/ of that set

union(x, y) – looks up set containing x and set containing y, combines two sets into one. All of the values of one set are added to the other, and the now empty set goes away.

PennState

KRUSKAL'S ALGORITHM IMPLEMENTATION

```
KruskalMST(Graph G)
   initialize each vertex to be an
independent component
   sort the edges by weight
   foreach(edge (u, v) in sorted order){
      if(u and v are in different
components){
           update u and v to be in the
same component
         add (u,v) to the MST
      }
   }
```

```
KruskalMST(Graph G)
   foreach (V : G.vertices) {
      makeSet(v);
   }
   sort the edges by weight
   foreach(edge (u, v) in sorted
order){
      if(findSet(v) is not the
same as findSet(u))
      {
         union(u, v)
         add (u, v) to the MST
      }
   }
```

ARE WE DOING THIS AGAIN? (CONTINUED)

- Disjoint Sets help us **manage groups of distinct values**.

- This is a common idea in graphs, where we want to keep track of different connected components of a graph.

- In Kruskal’s, if each connected-so-far-island of the graph is its own mini set in our disjoint set, we can easily check that we don’t introduce cycles. If we’re considering a new edge, we just check that the two vertices of that edge are in different mini sets by calling findSet.

CORRECTNESS OF KRUSKAL'S

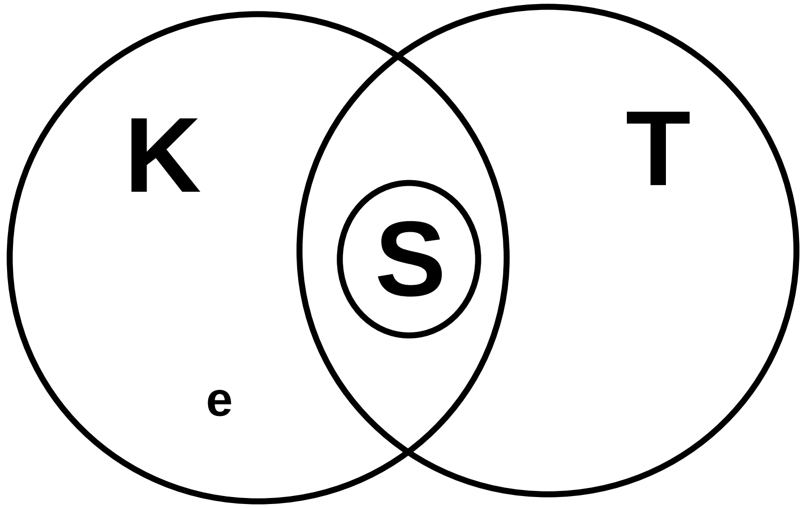
- This algorithm adds *n-1* edges without creating a cycle, so clearly it creates a spanning tree of any connected graph (*you should be able to prove this*).

But is this a *minimum* spanning tree?

Suppose it wasn't.

- There must be point at which it fails, and in particular there must a single edge whose insertion first prevented the spanning tree from being a minimum spanning tree.

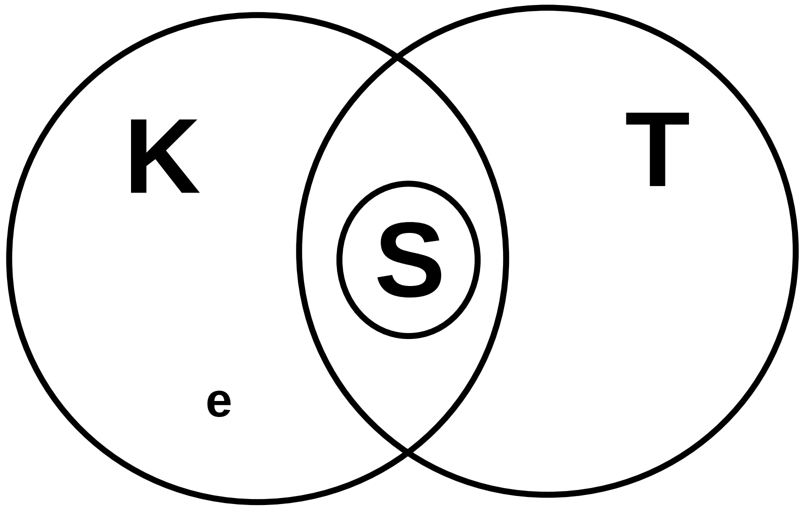
CORRECTNESS OF KRUSKAL'S



- Let **e** be this first errorful edge.
- Let **K** be the Kruskal spanning tree
- Let **S** be the set of edges chosen by Kruskal's algorithm *before* choosing **e**
- Let **T** be a MST containing all edges in **S**, but not **e**.

CORRECTNESS OF KRUSKAL'S

Lemma: w(**e'**) >= w(**e**) for all edges **e'** in **T - S**



Proof (*by contradiction*):

- Assume there exists some edge **e'** in **T - S**, w(**e'**) < w(**e**)
- Kruskal's must have considered **e'** before **e**
  - However, since **e'** is not in **K** (*why??*), it must have been discarded because it caused a cycle with some of the other edges in **S**.
  - But **e' + S** is a subgraph of **T**, which means it cannot form a cycle **...Contradiction**

CORRECTNESS OF KRUSKAL'S

- Inserting edge **e** into **T** will create a cycle
- There must be an edge on this cycle which is not in **K** (*why??*). Call this edge **e'**
- **e'** must be in **T - S,** so (by our lemma) w(**e'**) >= w(**e**)
- We could form a new spanning tree **T'** by swapping **e** for **e'** in **T** (*prove this is a spanning tree*).
- w(**T'**) is clearly no greater than w(**T)**
- But that means **T'** is a MST
- And yet it contains all the edges in **S**, and also **e**

…Contradiction

REVISITING: ALGORITHM DESIGN PARADIGMS

- Algorithm Design: No "single silver bullet" for solving problems
- Some Design Paradigms:
  - Divide and Conquer
  - Greedy Algorithms
  - Dynamic Programming
  - …..
  - …..

- Does it work?
- Is it fast?
- Can I do it better?

GREEDY ALGORITHM

- Definition: Iteratively make “myopic” decisions, hope everything works out at the end!

Make choices one-at-a-time.

- Example: **Dijkstra**

[A one-pass algorithm,

and we made a choice at each node]

Never look back.

Hope for the best!

Divide and conquer: Difficult to formulate the correct algorithm to recursively solve the problem.

Greedy algorithms: Easy to propose. Running time: Easier; sort things and do one pass.

Divide and conquer: Correctness was usually mechanical

Greedy algorithms: Proving correctness can be tricky. There are often greedy algorithms that seem correct but do not work.

OPTIMIZATION PROBLEMS

- For most optimization problems you want to find, not just ***a*** solution, but the ***best*** solution.
- A ***greedy algorithm*** sometimes works well for optimization problems. It works in phases. At each phase:
  - **You take the best you can get right now**, without regard for future consequences.
  - **You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum.**

ACTIVITY SELECTION

…………

Evening hike

Historical Place Visit

Restaurant

Underwater Exploration

…………

You can only do one activity at a time, and you want to maximize the number of activities that you do. What to choose?

…………

Amusement Park

…………

Shopping Mall

Social Activity

…………

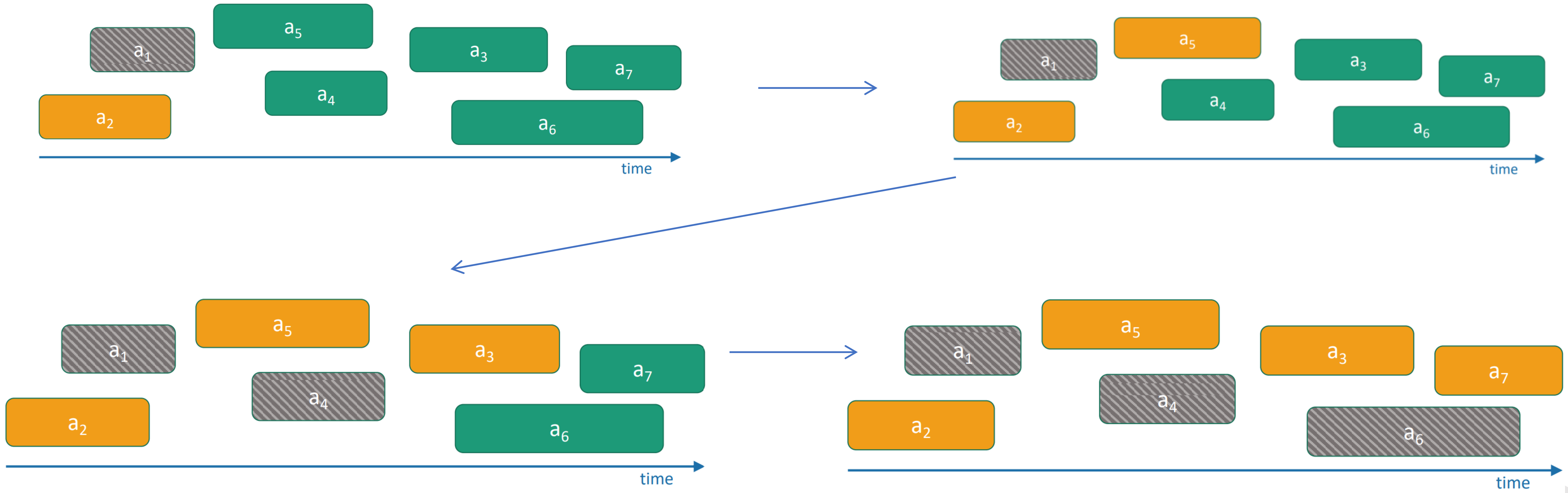
Input:
Activities $a_1, a_2, \ldots\ldots\ldots\ldots a_n$

Start times $s_1, s_2, \ldots, s_n$
Finish times $f_1, f_2, \ldots, f_n$

Output:
A way to maximize the number of activities you can do today

ACTIVITY SELECTION

- Pick activity you can add with the smallest finish time.
- Repeat



Running time:
O(n) if the activities are already sorted by finish time.
O(n log(n)) if you have to sort them first.

ACTIVITY SELECTION

1) Sort the activities according to their finishing time
2) Select the first activity from the sorted array and print it.
3) Do the following for the remaining activities in the sorted array.

……

If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it.

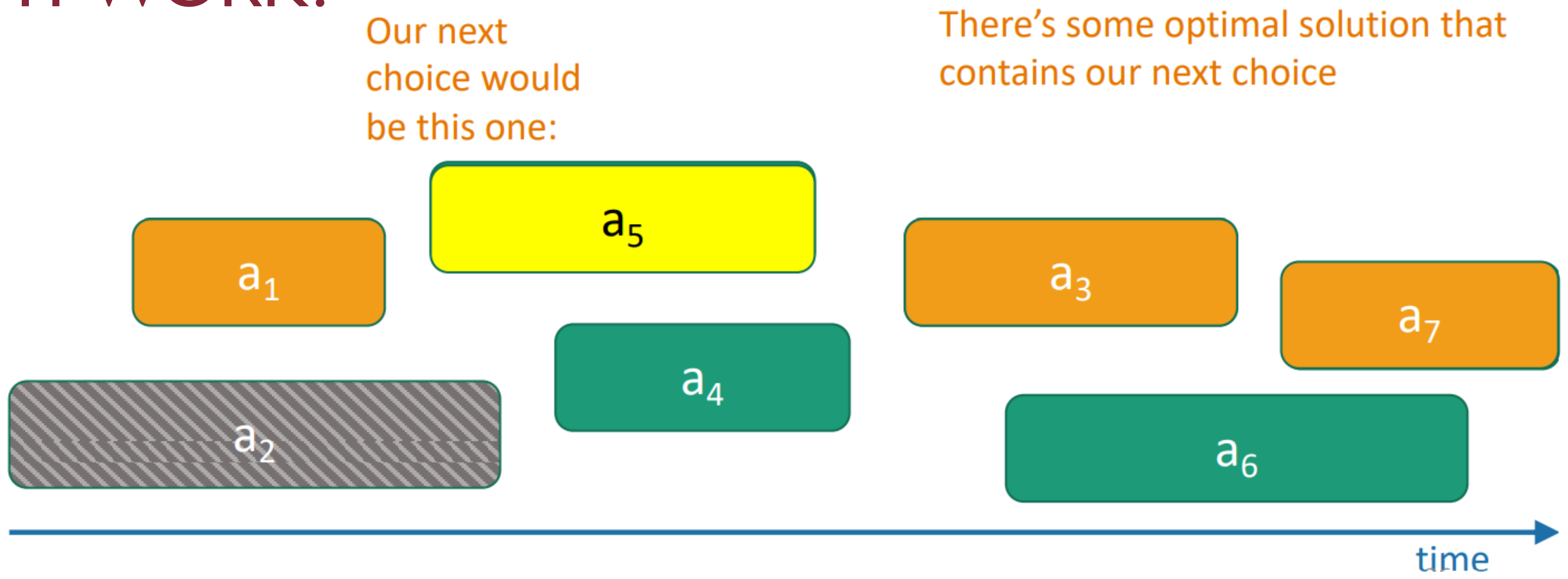
Well, what makes it greedy!!

At each step in the algorithm, make a choice.
I can increase my activity set by one,
And leave lots of room for future choices, Let’s do that and hope for the best!!!

Hope that at the end of the day, this results in a globally optimal solution.

PennState

WHY DOES IT WORK?


Our next choice would be this one:
There's some optimal solution that contains our next choice
a_5
a_1
a_3
a_7
a_4
a_2
a_6
time

Whenever we make a choice, we don't rule out an optimal solution

Different strategy: Longest first/ Shortest first/ Earliest start first
Earliest finish first/ ..
It leaves as much resource (time) as possible for other tasks

TAKE AWAY

- Greedy algorithms aim for global optimality by iteratively making a locally optimal decision.

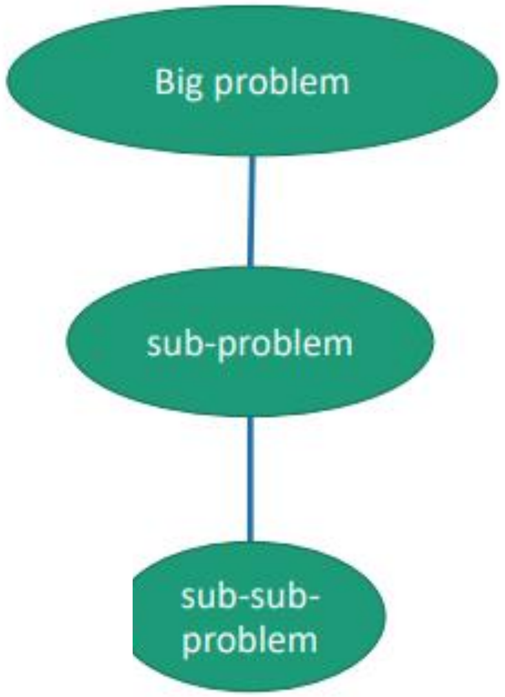
- To show correctness, typically need to show
  - The algorithm produces a legal answer, and
  - The algorithm produces an optimal answer.
- **Often use "greedy stays ahead" to show optimality**.

- Optimal sub-structure in greedy algorithms: Our greedy activity selection algorithm exploited a natural sub-problem structure:
- S[i-1] = number of activities you can do after the end of activity i-1

GREEDY APPROACH

- Like Dijkstra’s algorithm, both Prim’s and Kruskal’s algorithms are **greedy algorithms**
- The greedy approach works for the MST problem; however, **it does not work for many other problems**!

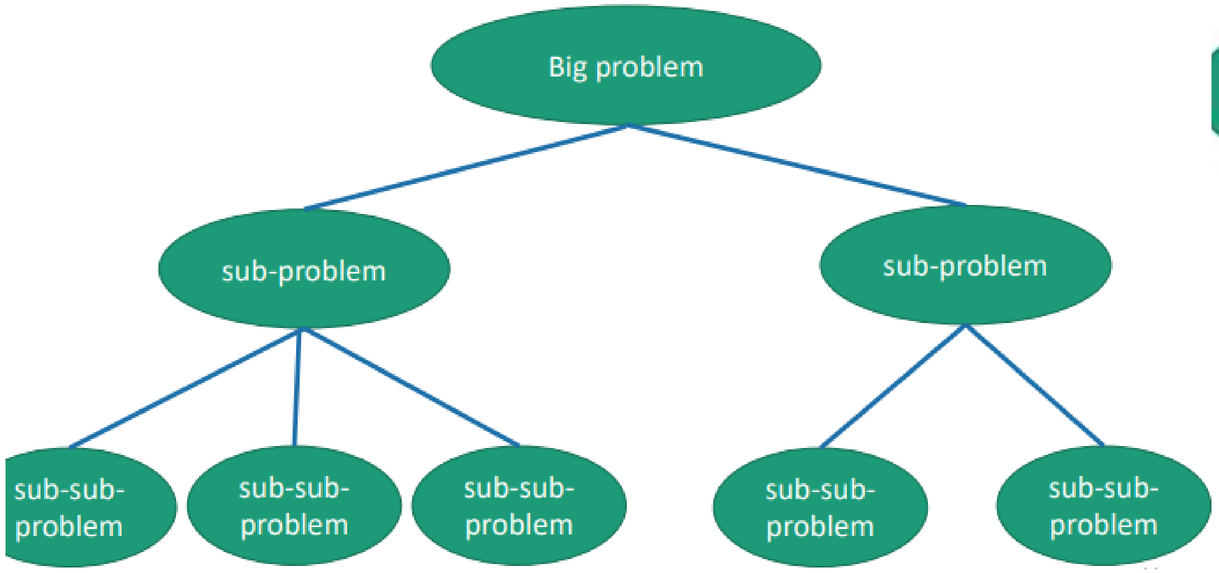
Greedy Algorithm



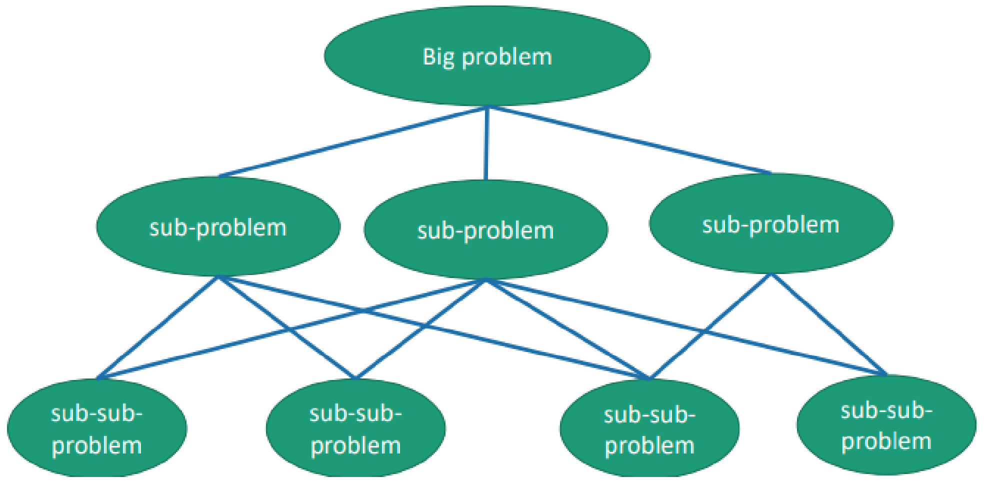
Not only is there optimal sub-structure:

- optimal solutions to a problem are made up from optimal solutions of sub-problems
- but each problem depends on only one sub-problem

Divide and Conquer



Dynamic Programming


Big problem
sub-problem
sub-problem
sub-problem
sub-sub-problem
sub-sub-problem
sub-sub-problem
sub-sub-problem