CMPSC 465
Data Structures and Algorithms
Fall 2024

Instructor: Debarati Das

October 16, 2024

Optimization Problem

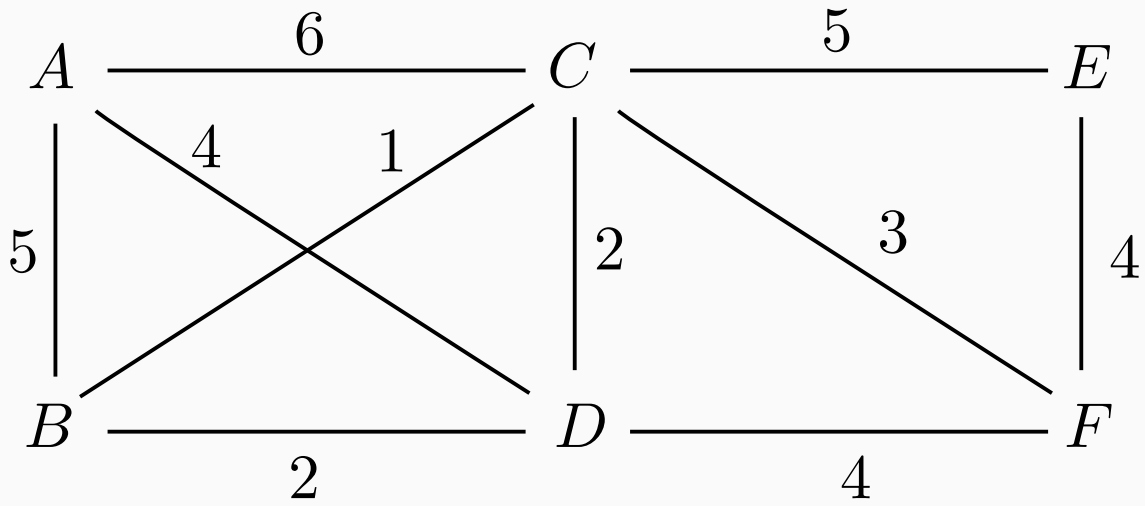
- Goal: find the *best* solution according to some objective function
- Design Techniques:
  1. Divide and conquer
  2. Greedy
  3. Dynamic Program

How to design a greedy algorithm?

- Build up a solution to an optimization problem at each step shortsightedly choosing the option that currently seems the best.
- Does it always work?
  may, may not
- Key to designing greedy algorithms: find structure that ensures you don't leave behind other options

Minimum Spanning Tree

Minimum Spanning Tree (MST)



The Minimum Spanning Tree Problem (MST)

Input: undirected graph $G = (V, E)$ with edge weights $w_e \in \mathbb{R}, \forall e \in E$.

Output: A **tree** $T = (V, E')$ s.t.

1. $E' \subseteq E$
2. E' minimizes $\text{weight}(T) = \sum_{e \in E'} w_e$

Greedy approach for MST

View the problem as one where a sequence of choices are made, and each choice leaves a single subproblem to solve

Greedy heuristic: add the lightest edge that doesn't induce a cycle

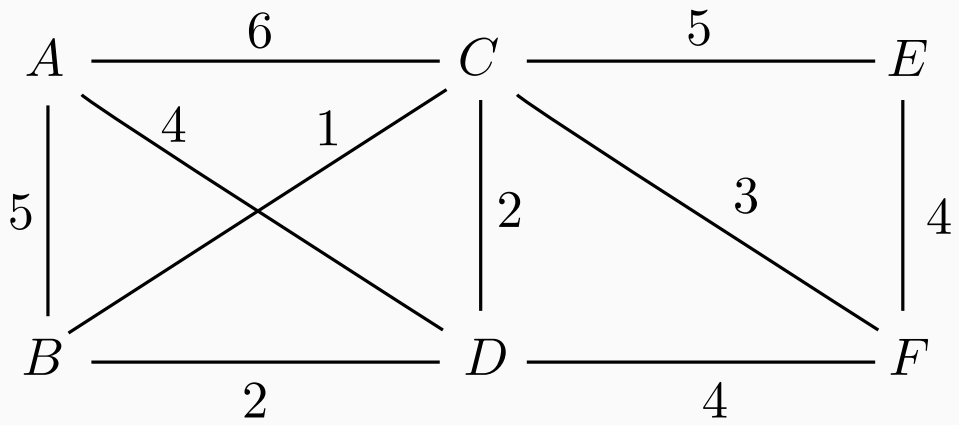
Example:

Greedy approach for MST

View the problem as one where a sequence of choices are made, and each choice leaves a single subproblem to solve

Greedy heuristic: add the lightest edge that doesn't induce a cycle

Example:

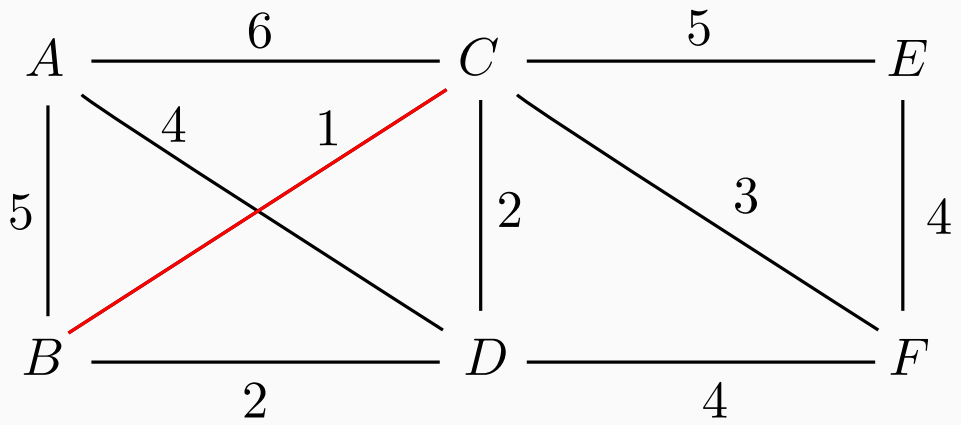


Greedy approach for MST

View the problem as one where a sequence of choices are made, and each choice leaves a single subproblem to solve

Greedy heuristic: add the lightest edge that doesn't induce a cycle

Example:

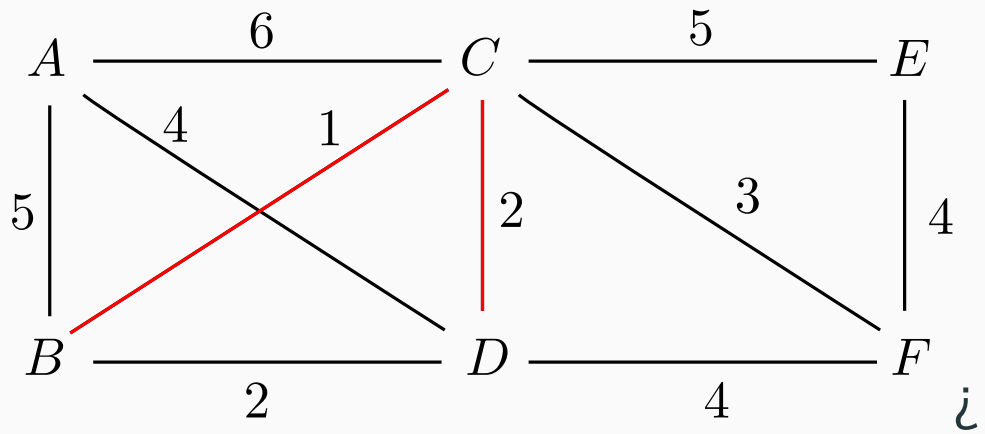


Greedy approach for MST

View the problem as one where a sequence of choices are made, and each choice leaves a single subproblem to solve

Greedy heuristic: add the lightest edge that doesn't induce a cycle

Example:

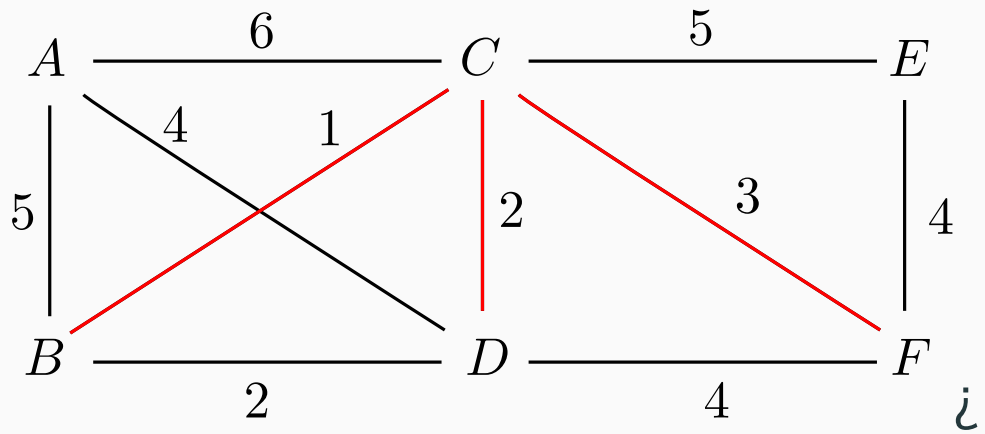


Greedy approach for MST

View the problem as one where a sequence of choices are made, and each choice leaves a single subproblem to solve

Greedy heuristic: add the lightest edge that doesn't induce a cycle

Example:

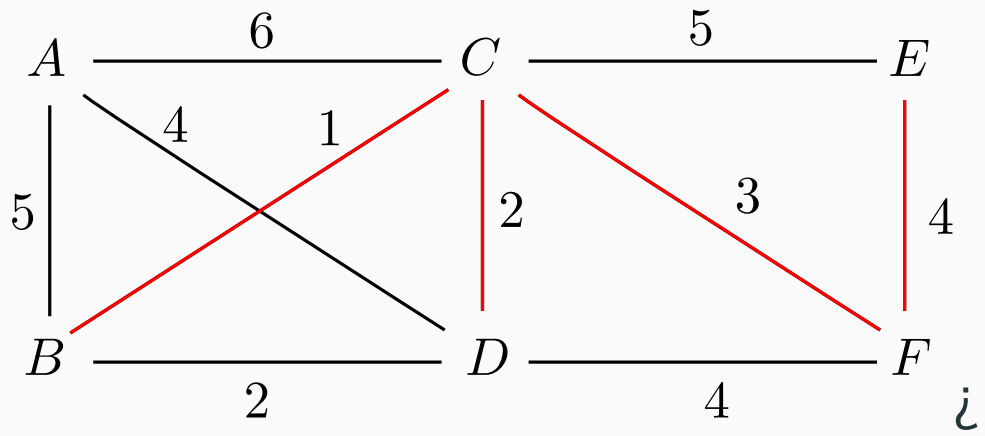


Greedy approach for MST

View the problem as one where a sequence of choices are made, and each choice leaves a single subproblem to solve

Greedy heuristic: add the lightest edge that doesn't induce a cycle

Example:

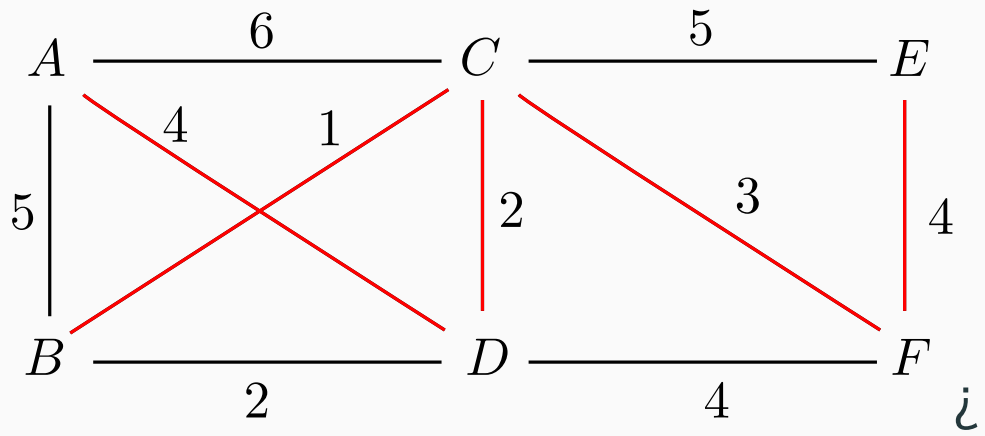


Greedy approach for MST

View the problem as one where a sequence of choices are made, and each choice leaves a single subproblem to solve

Greedy heuristic: add the lightest edge that doesn't induce a cycle

Example:

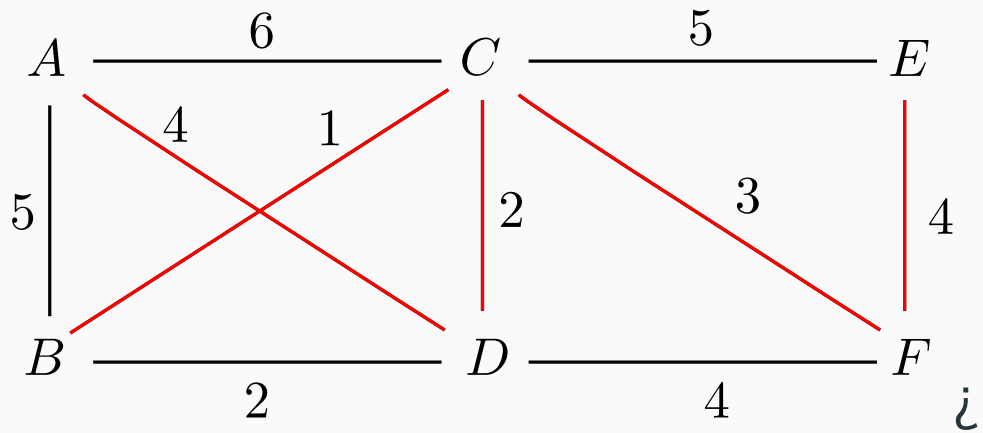


Greedy approach for MST

View the problem as one where a sequence of choices are made, and each choice leaves a single subproblem to solve

Greedy heuristic: add the lightest edge that doesn't induce a cycle

Example:



$\text{weight}(T) = 14$

Optimality?

We need to show greedy choice and optimal substructure

Optimality of greedy (I)

Some technical preliminaries

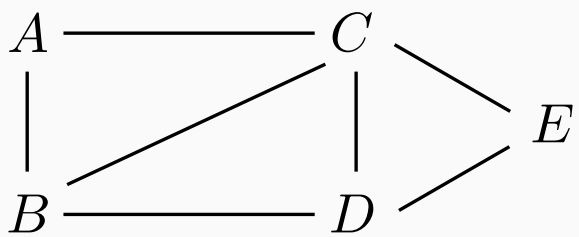
Definition

A **partition** of a set V is in the form (A, B) where $A \cup B = V$ and $A \cap B = \emptyset$

Definition

For an undirected graph $G = (V, E)$, a **cut** is a partition of V

Example:



$(\{A, B, C\}, \{D, E\})$ is a cut

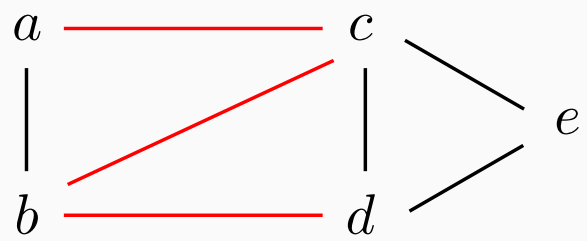
$(\{A, B\}, \{D, E\})$

Optimality of greedy (II)

Definition

A cut $(S, V - S)$ **respects** $A \subseteq E$ if $\forall (u, v) \in A$, either $u, v \in S$ or $u, v \in V - S$

Example:



A: red edges

cut $(\{b, d\}, \{a, c, e\})$ doesn't respect A

cut $(\{e\}, \{a, b, c, d\})$

respects A

Optimality of greedy (III)

Theorem (The cut property)

Let A be a subset of edges of some MST of $G = (V, E)$.

Let $(S, V - S)$ be a cut that respects A.

Let e be the lightest edge across the cut.

Then $A \cup \{e\}$ is part of some MST

Theorem (The cycle property)

Let C be a cycle in G.

Let f be the max weight edge in C.

Then the MST does not contain f.

How does this help?

- It shows the greedy choice property:

Kurskal's Algorithm

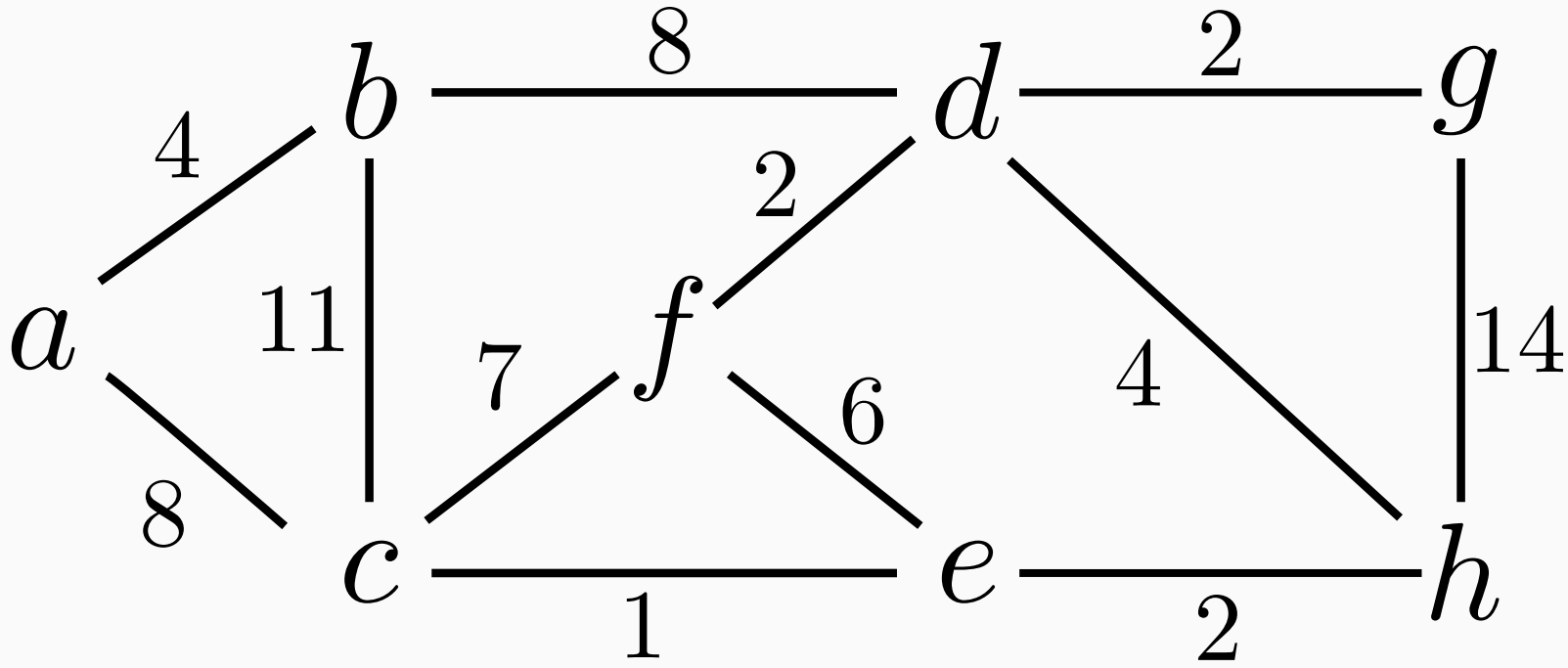
def KRUSKAL_MST*(undirected $G = (V, E)$, weights $w = (w_e)_{e \in E}$)*:

```
    Set A := { };
    for v ∈ V:
        make_set(v);
    Sort E in increasing order of edge weights;
    for (u, v) ∈ E:
        if find_set(u) ≠ find_set(v):
            A := A ∪ {(u, v)};
            union(u, v);
```

- $\text{make\_set}(v)$: put v into a set containing itself. $v \mapsto \{v\}$
- $\text{find\_set}(u)$: find which set u belongs to
- $\text{union}(u, v)$: merge the sets that u and v are in

Example

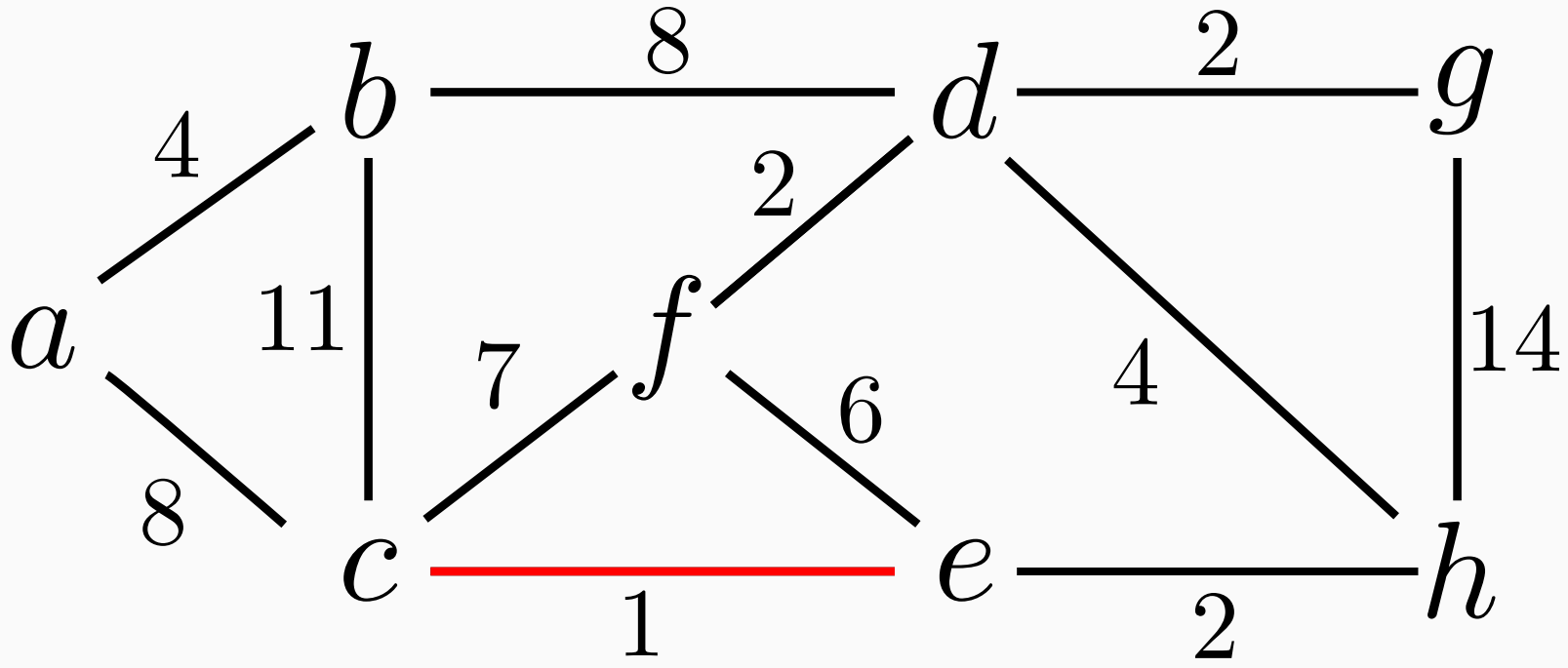
A: red edges



$$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}$$

Example

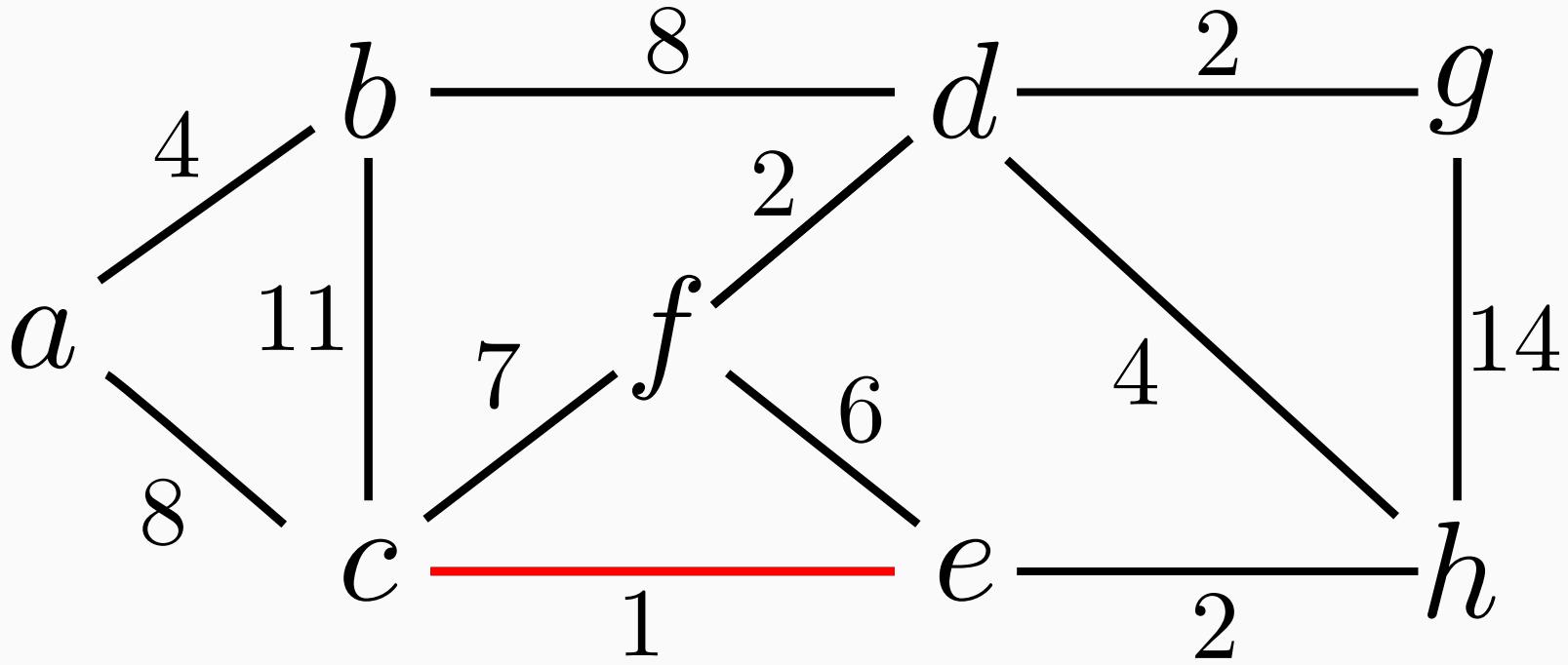
A: red edges



$$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}$$

Example

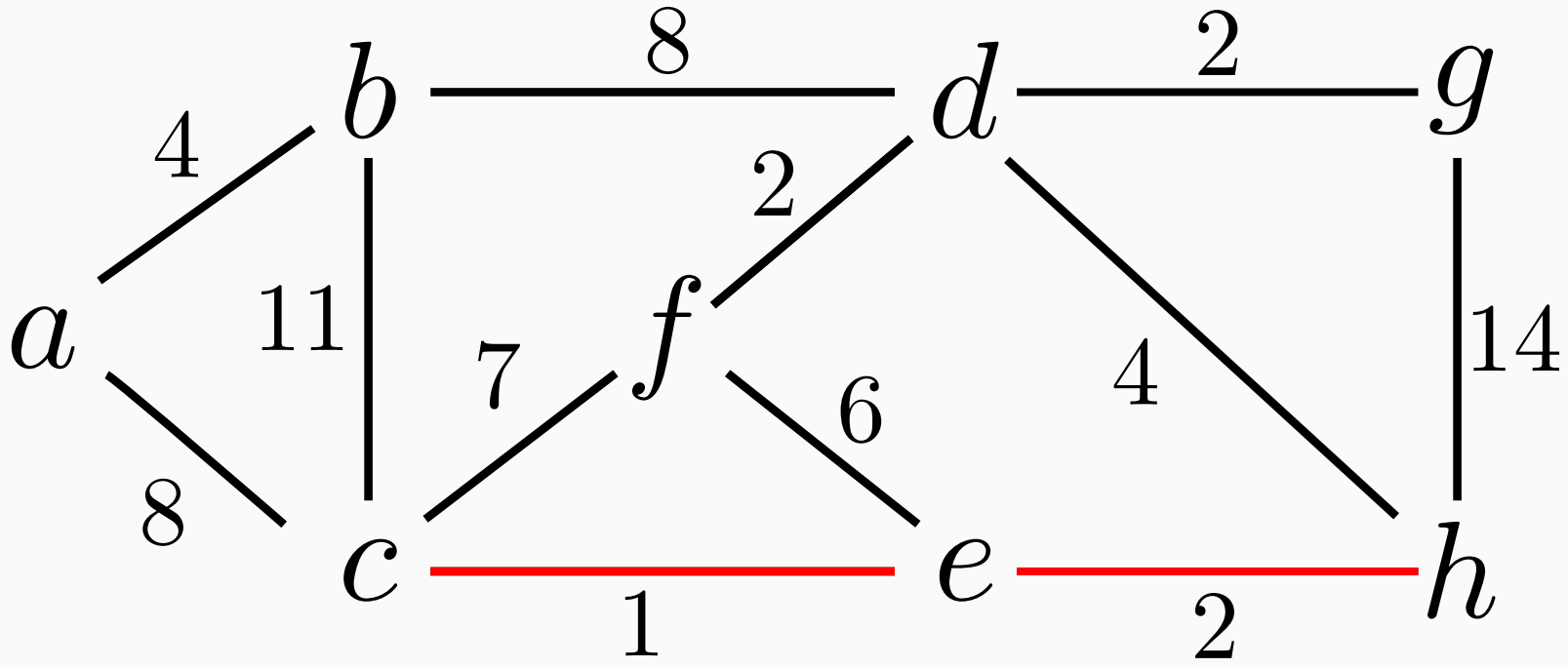
A: red edges



$$\{a\}, \{b\}, \{c, e\}, \{d\}, \{f\}, \{g\}, \{h\}$$

Example

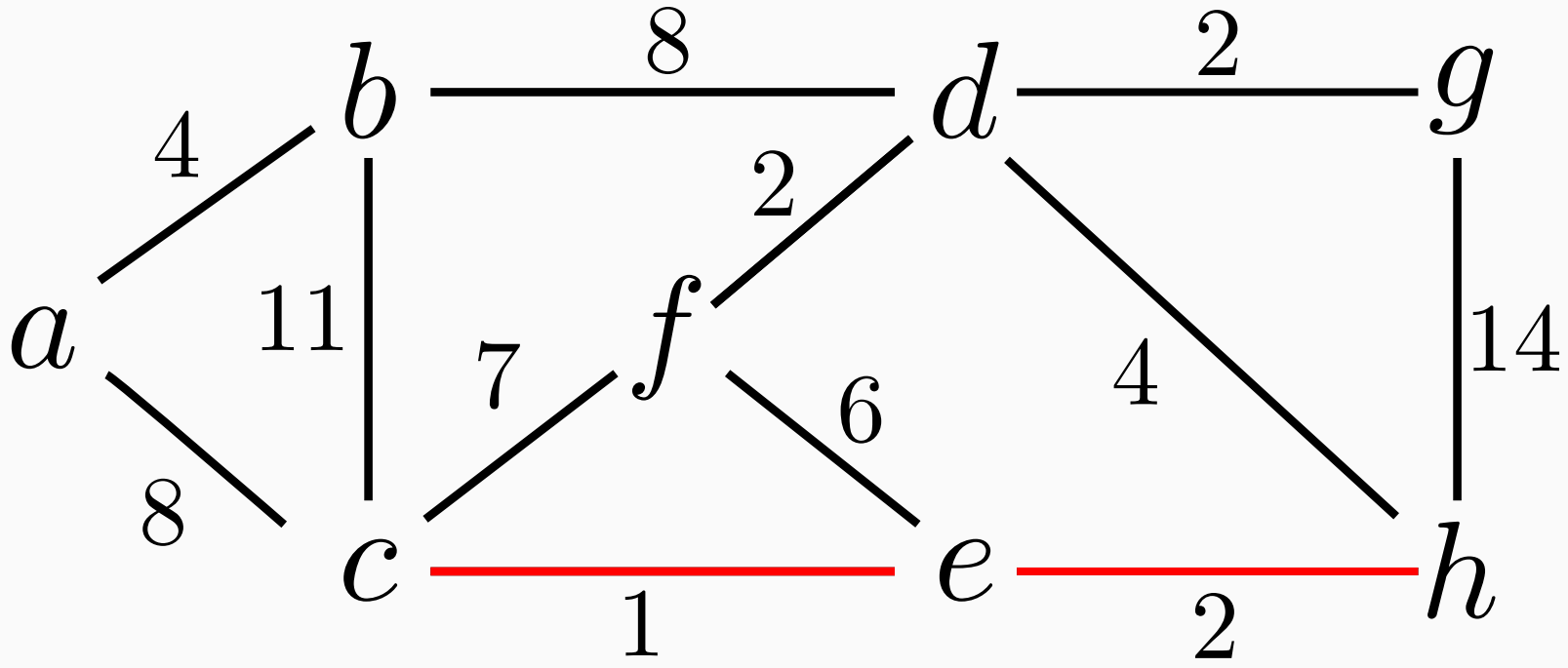
A: red edges



$$\{a\}, \{b\}, \{c, e\}, \{d\}, \{f\}, \{g\}, \{h\}$$

Example

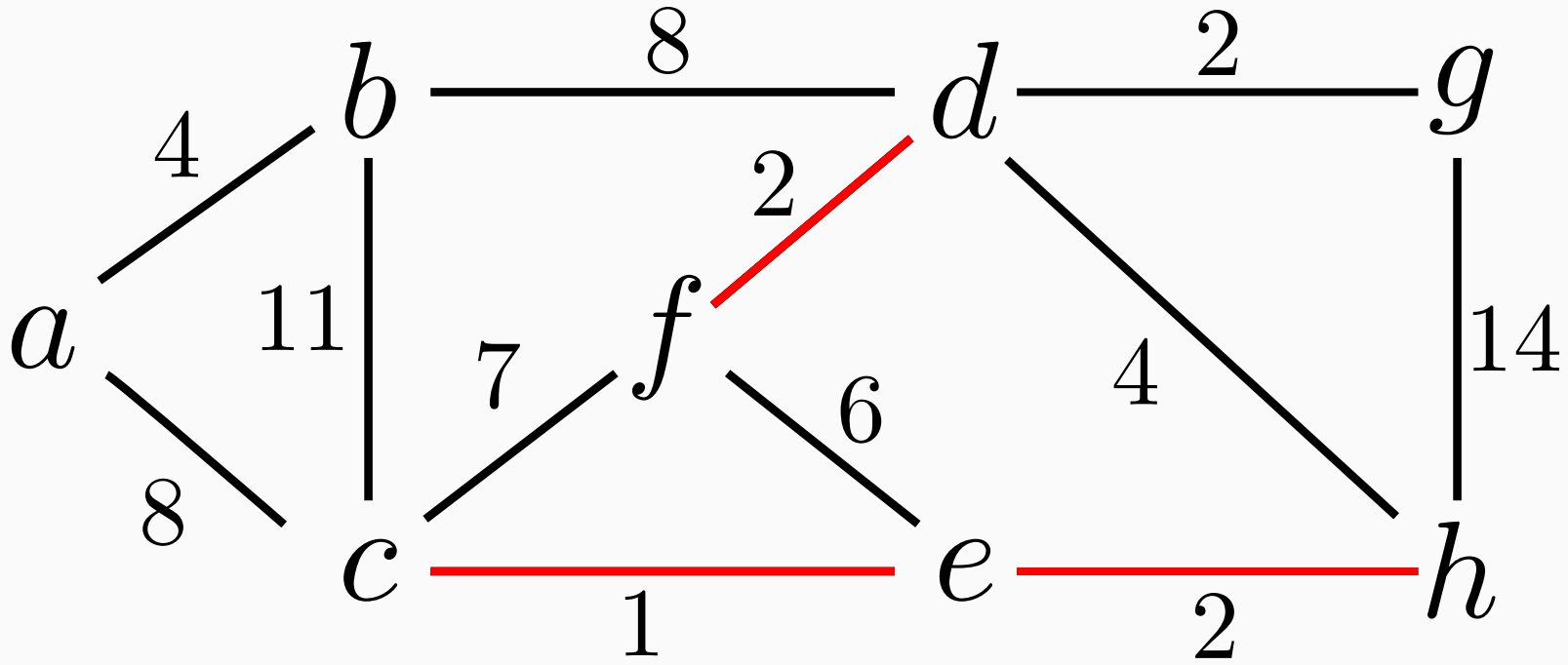
A: red edges



$$\{a\}, \{b\}, \{c, e, h\}, \{d\}, \{f\}, \{g\}$$

Example

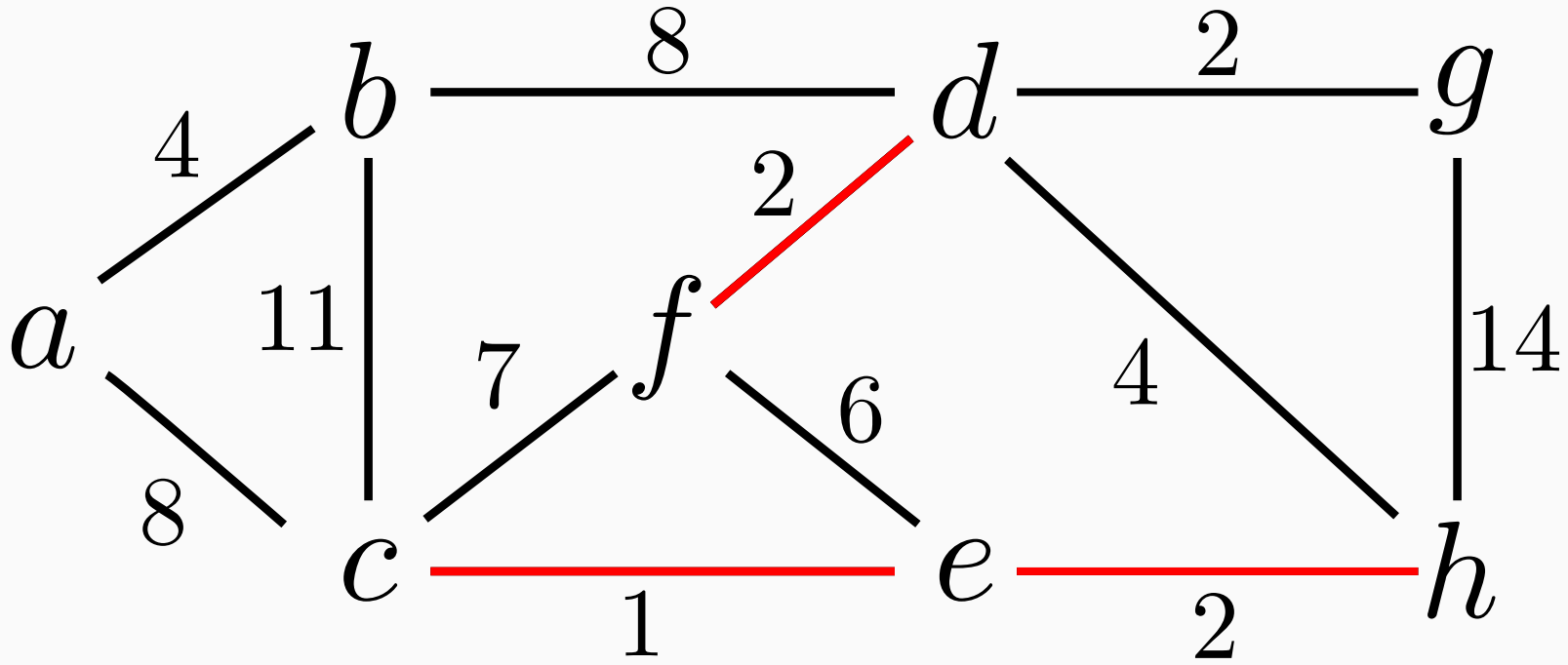
A: red edges



$$\{a\}, \{b\}, \{c, e, h\}, \{d\}, \{f\}, \{g\}$$

Example

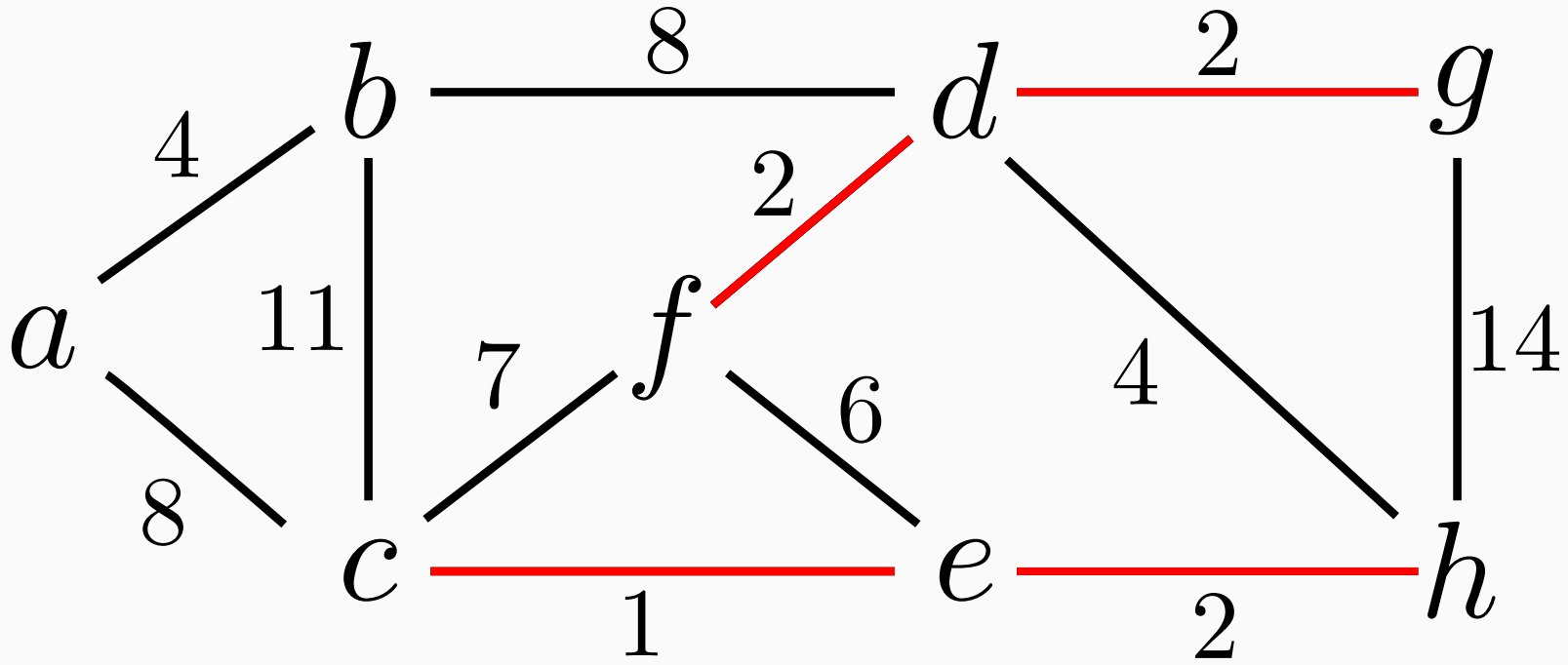
A: red edges



$$\{a\}, \{b\}, \{c, e, h\}, \{d, f\}, \{g\}$$

Example

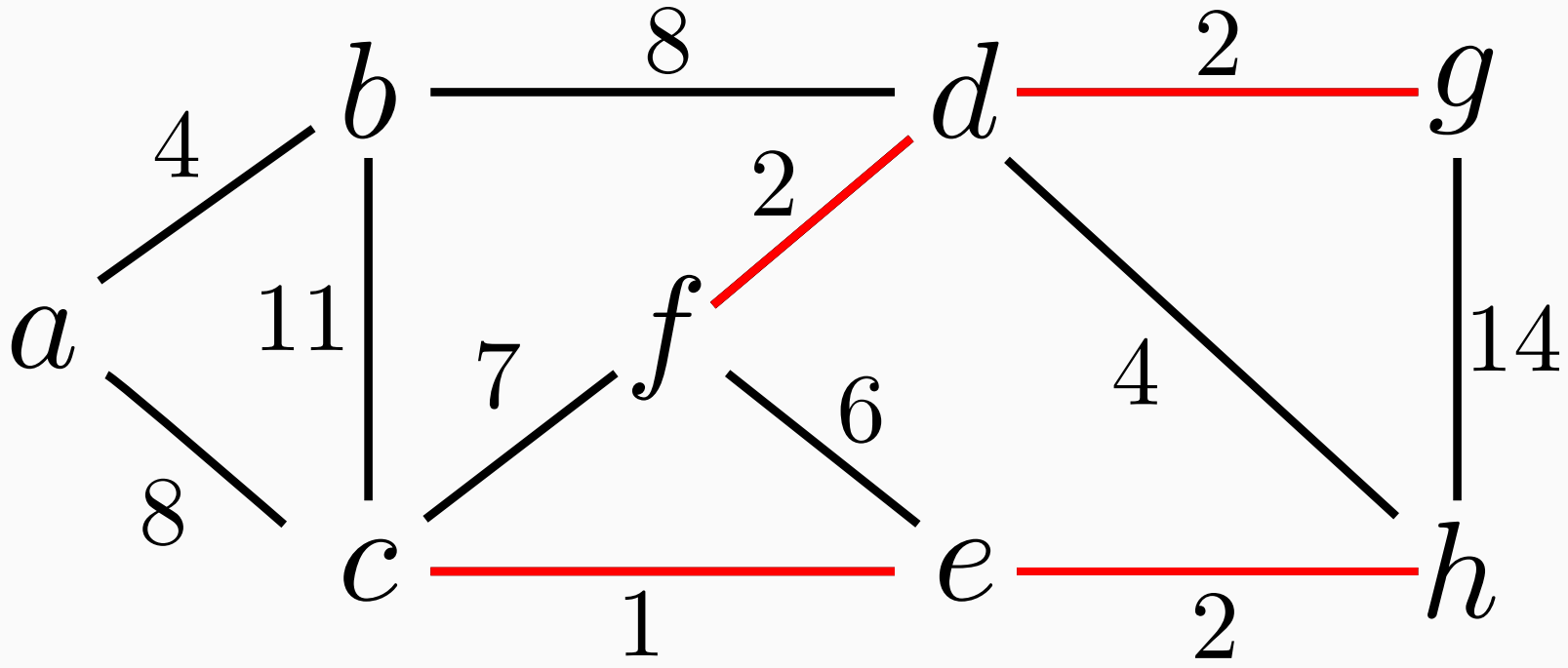
A: red edges



$$\{a\}, \{b\}, \{c, e, h\}, \{d, f\}, \{g\}$$

Example

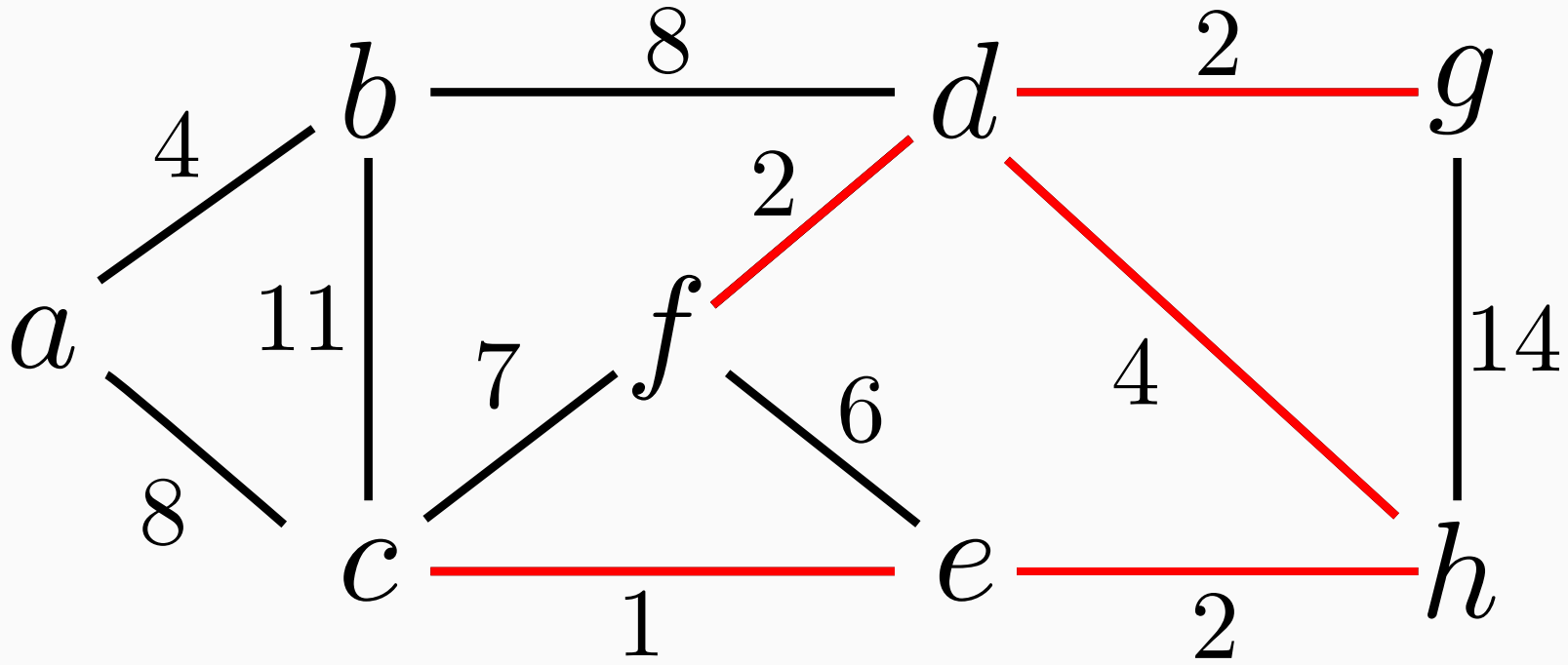
A: red edges



$$\{a\}, \{b\}, \{c, e, h\}, \{d, f, g\}$$

Example

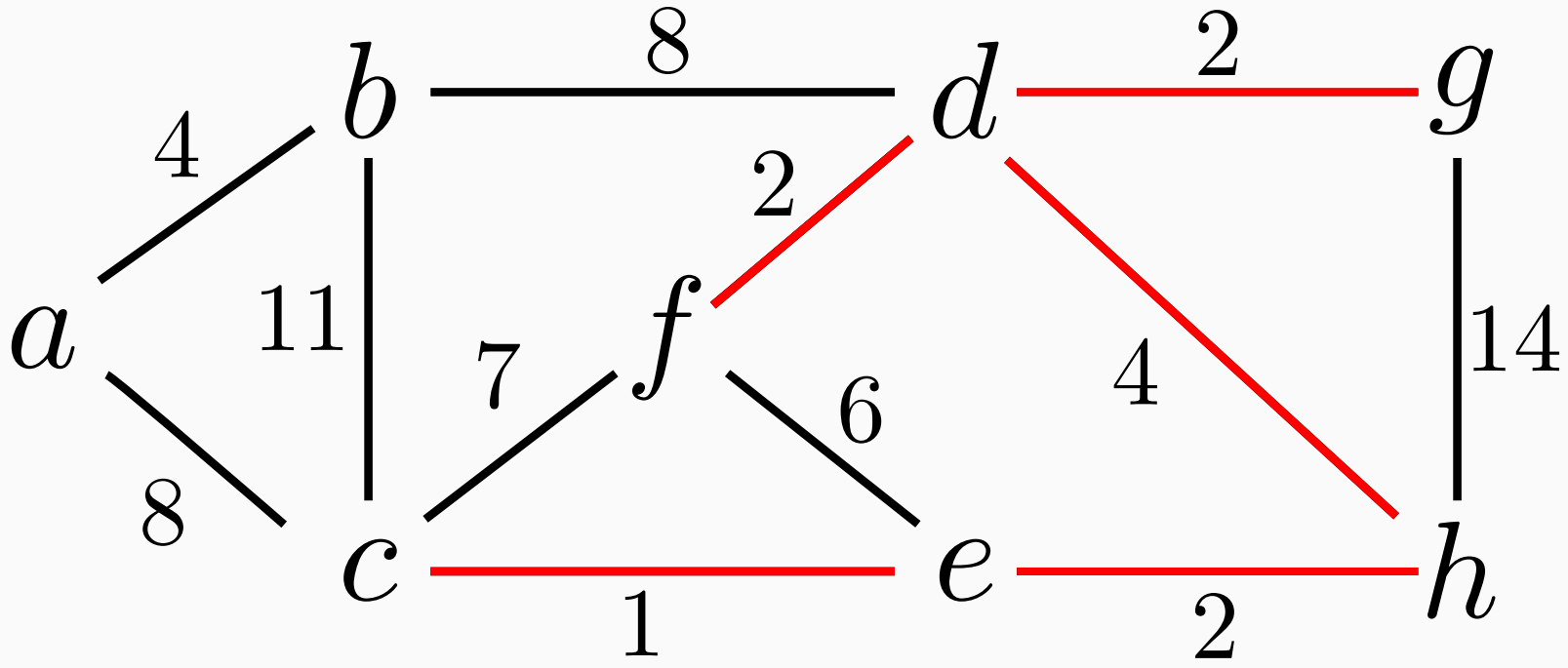
A: red edges



$$\{a\}, \{b\}, \{c, e, h\}, \{d, f, g\}$$

Example

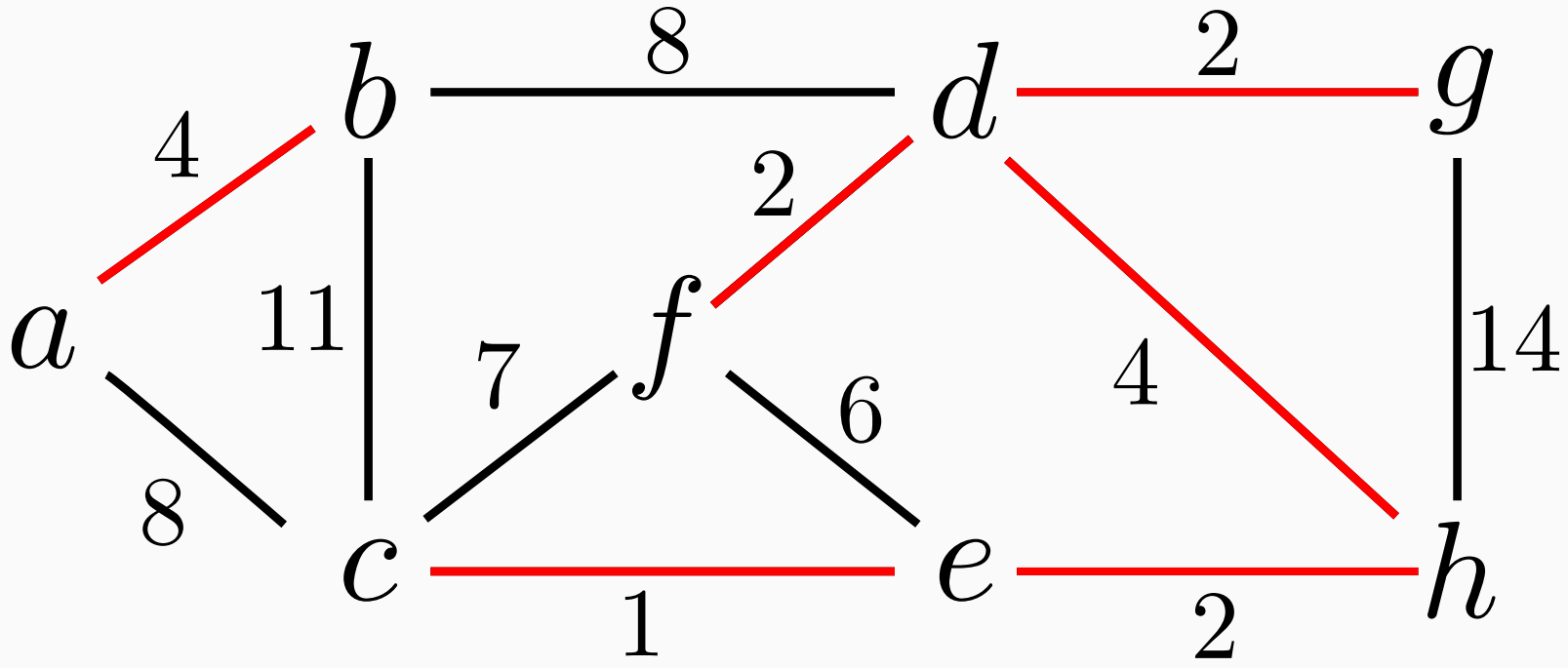
A: red edges



$$\{a\}, \{b\}, \{c, e, h, d, f, g\}$$

Example

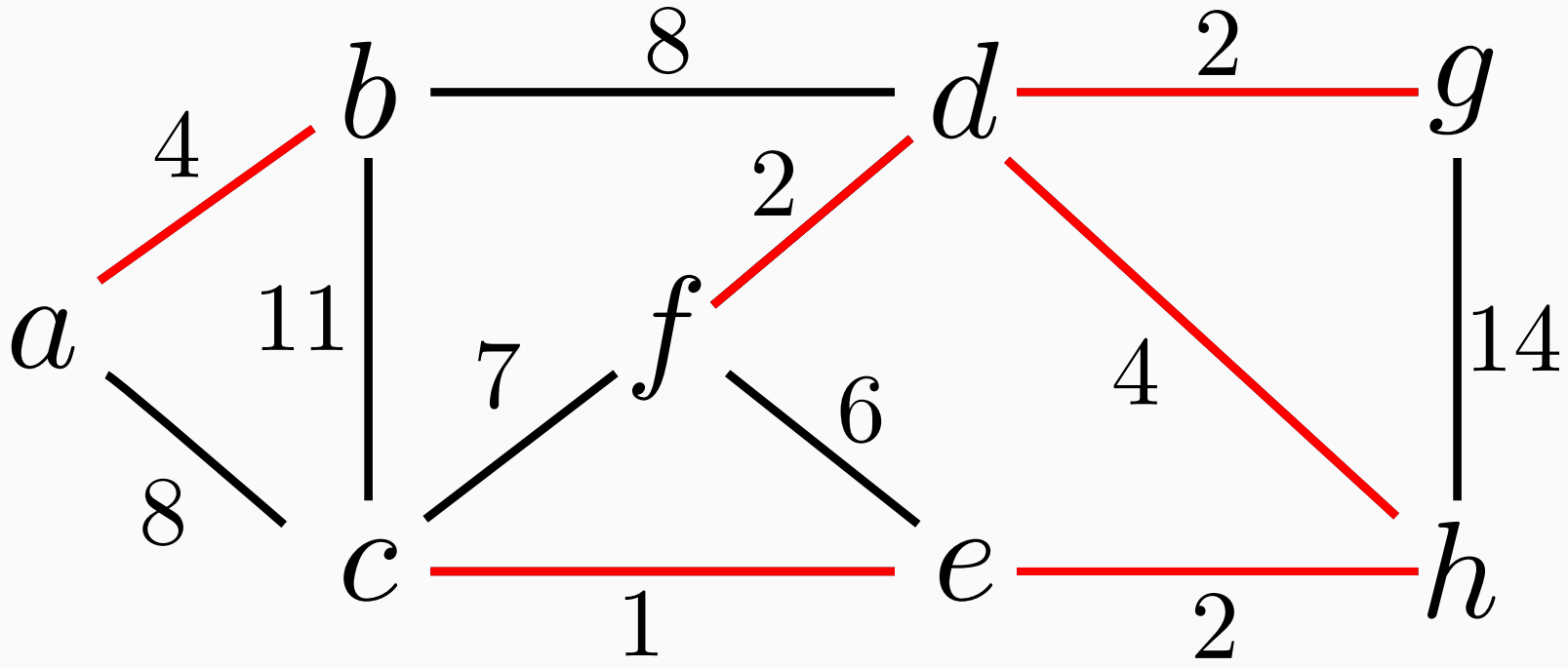
A: red edges



$$\{a\}, \{b\}, \{c, e, h, d, f, g\}$$

Example

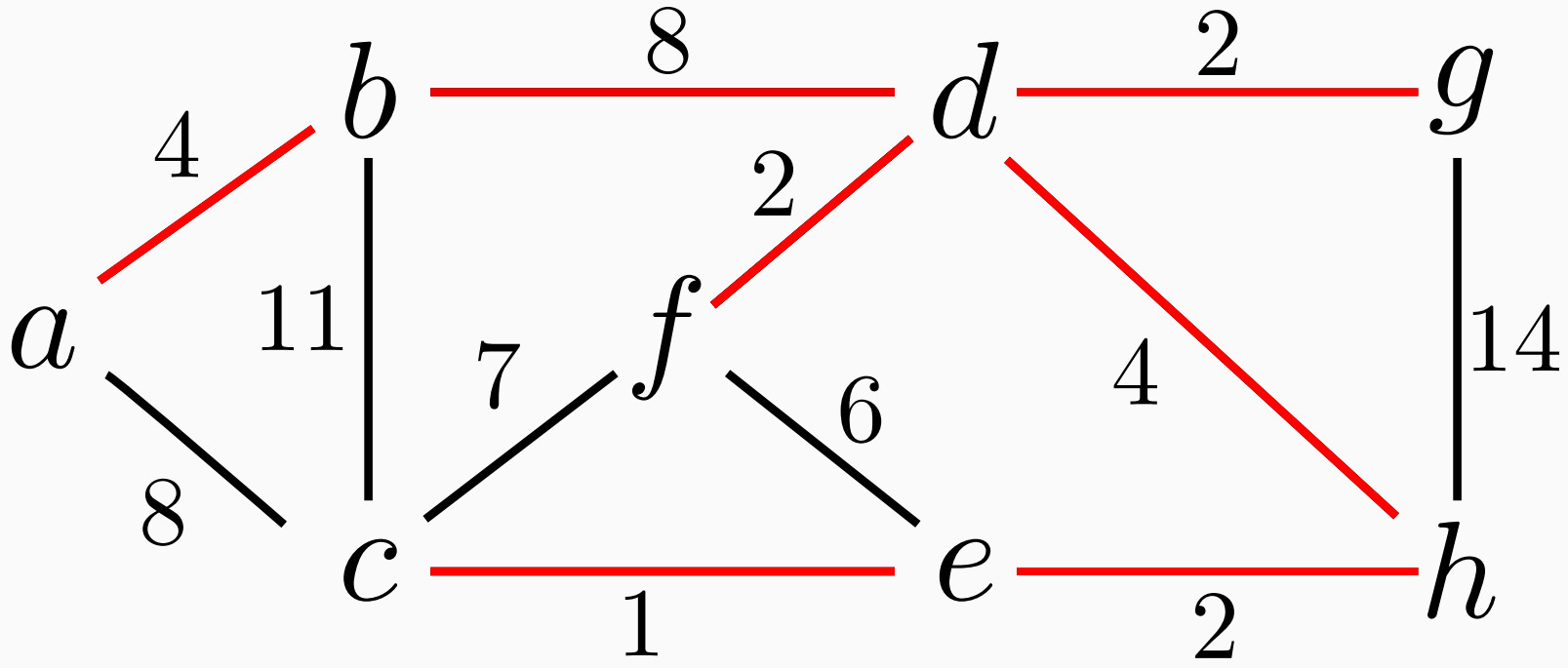
A: red edges



$$\{a, b\}, \{c, e, h, d, f, g\}$$

Example

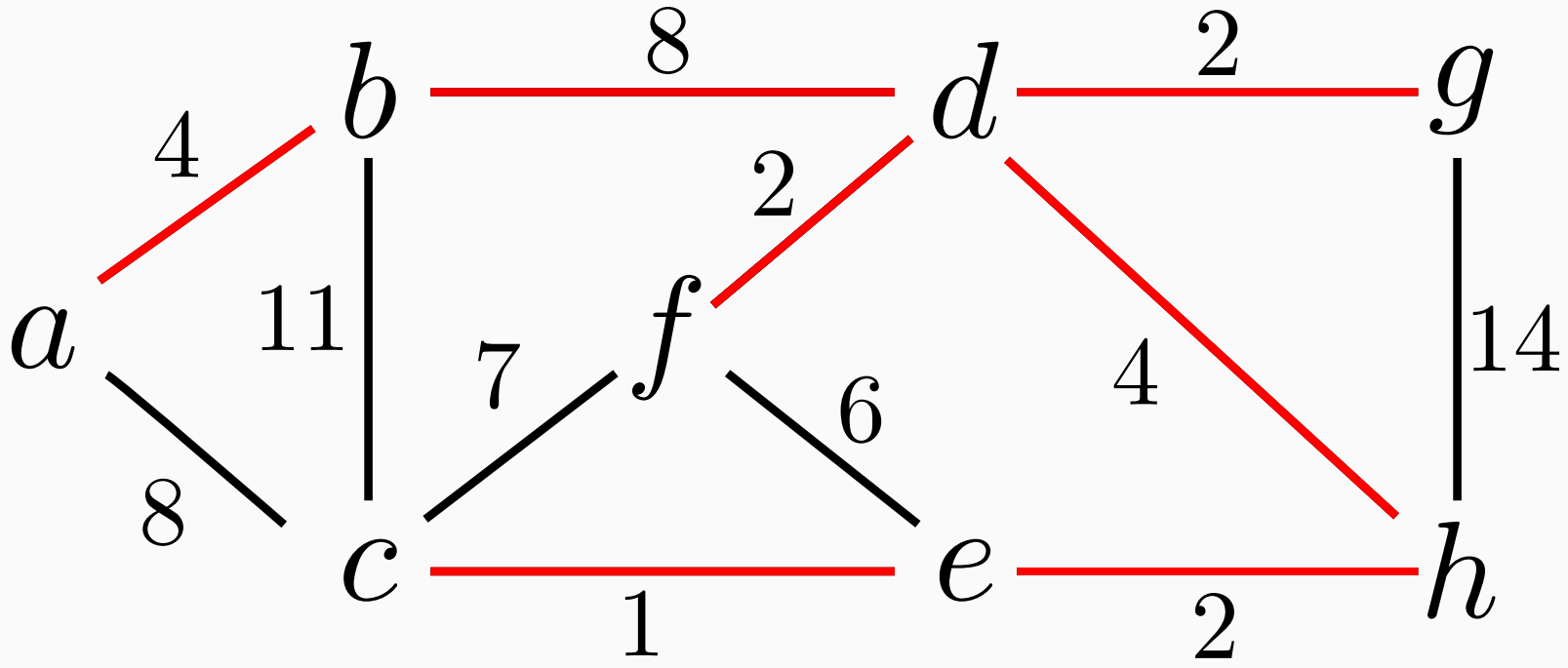
A: red edges



$$\{a, b\}, \{c, e, h, d, f, g\}$$

Example

A: red edges



$$\{a, b, c, e, h, d, f, g\}$$

Proof of correctness

We need to show two things:

- A is a spanning tree
  - it has no cycles
  - if there exists $v \in V$ that is not connected by A

    then there exists $e \in E$ s.t. $A \cup \{e\}$ contains no cycle

    then KRUSKAL_MST will add the lightest such edge

Proof of correctness

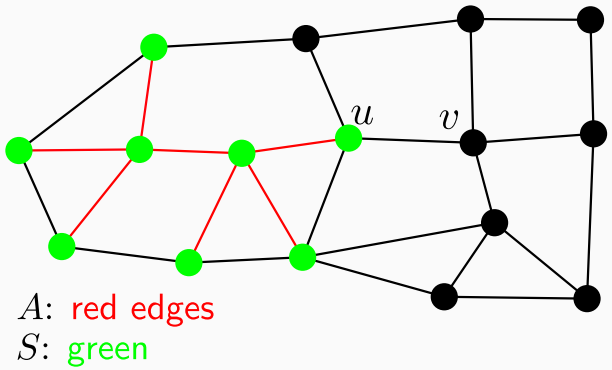
- A has the minimum weight.

Theorem (The cut property)

Let A be a subset of edges of some MST of $G = (V, E)$. Let $(S, V - S)$ be a cut that respects A. Let e be the lightest edge across the cut. Then $A \cup \{e\}$ is part of some MST

Theorem (The cycle property)

Let C be a cycle in G. Let f be the max weight edge in C. Then the MST does not contain f.



Case 1: if KRUSKAL_MST adds $e = (u, v)$.
Let S be the vertices reachable from u in A (the partial solution so far)
Then $A \cup \{e\}$ is part of some MST □

Case 2: if KRUSKAL_MST discards $e = (u, v)$. Adding e creates a cycle. By cycle property e is not in the MST.

Proof of the cut property theorem (I)

To prove the cut property, we need a lemma

Lemma

Any connected and undirected graph $G = (V, E)$ is a tree if and only if $|E| = |V| - 1$

Proof.

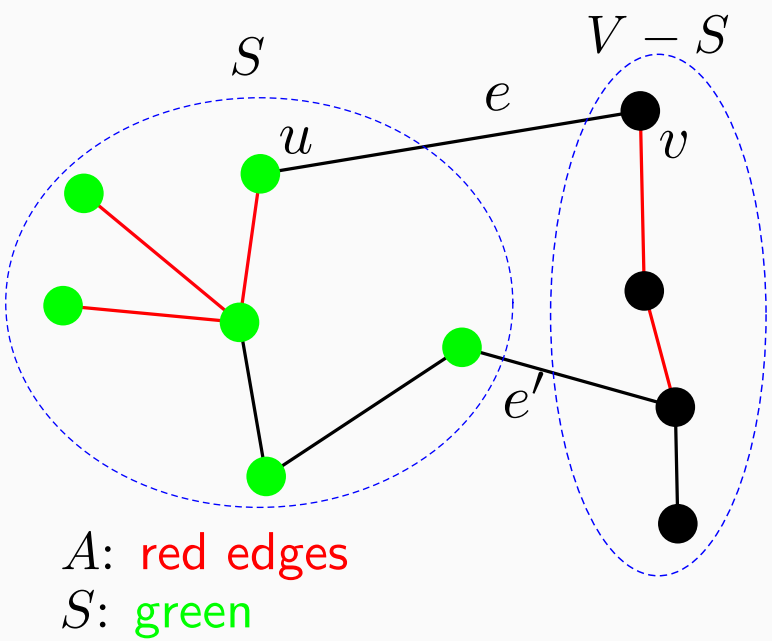
See textbook page 129 □

Proof of the theorem (II)

By assumption, A is a subset of edges of some MST T

If e is part of T, then there is nothing to prove

If $e \notin T$, we will construct a new MST T' with $e \in T'$:



Add e to T. This will create a cycle, which must have an edge $e' \in T$ across the cut

By definition of e, $w_e \leq w_{e'}$

Let $T' = T \cup \{e\} - \{e'\}$.

We didn't change the edge count.

Lemma $\implies$ T' is a tree

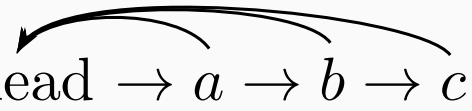
$$\begin{aligned} \text{weight}(T') &= \text{weight}(T) + w_e - w_{e'} \\ &\leq \text{weight}(T) + w_{e'} - w_{e'} \\ &= \text{weight}(T) \end{aligned}$$

T is MST $\implies$ $\text{weight}(T') = \text{weight}(T)$

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

$\{a, b, c\}$ 

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

$\{a, b, c\}$  find_set(b):

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

$\{a, b, c\}$  $\text{find\_set}(b)$: $O(1)$

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

$\{a, b, c\}$  $\text{head} \rightarrow a \rightarrow b \rightarrow c$

$\text{find\_set}(b)$: $O(1)$

$\text{make\_set}(v)$:

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

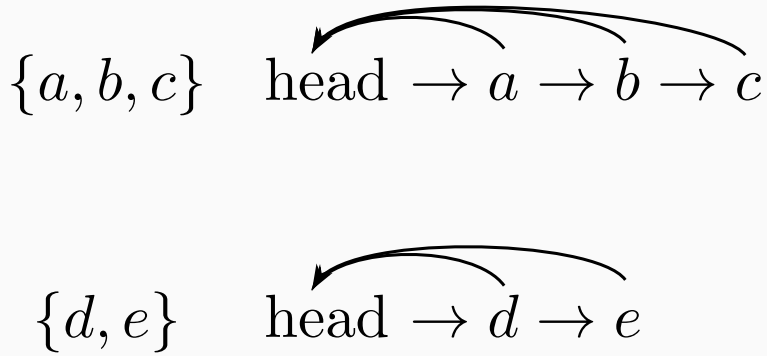
$\text{find\_set}(b)$: $O(1)$

$\text{make\_set}(v)$: $O(1)$

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:



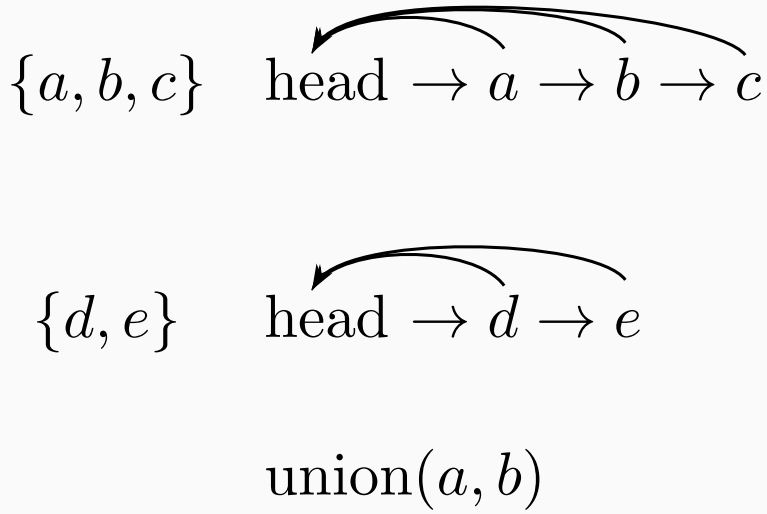
$\text{find\_set}(b)$: $O(1)$

$\text{make\_set}(v)$: $O(1)$

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:



find_set(b): $O(1)$

make_set(v): $O(1)$

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

$\{a, b, c\}$ $\text{head} \rightarrow a \rightarrow b \rightarrow c$ 

$\text{find\_set}(b)$: $O(1)$

$\text{make\_set}(v)$: $O(1)$

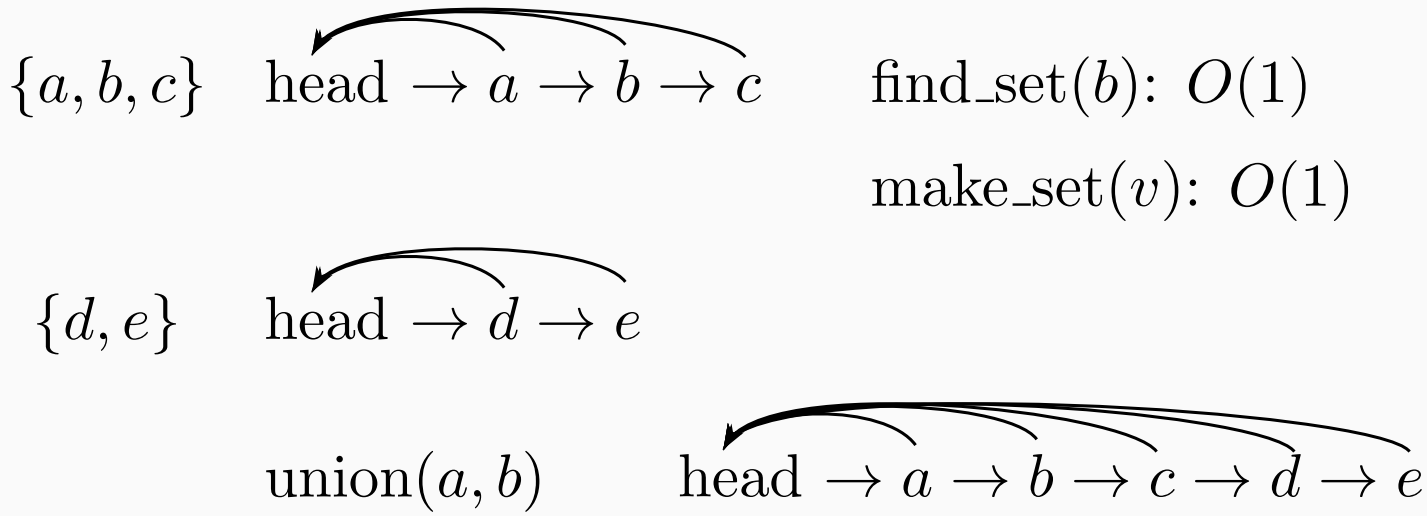
$\{d, e\}$ $\text{head} \rightarrow d \rightarrow e$ 

$\text{union}(a, b)$ $\text{head} \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$ 

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:



Cost of union: O(length of the shorter list)

Using an array to implement it:

vertex	1	2	3	4	5
head	1	1	1	4	4

$\xrightarrow{\text{union}}$

1	2	3	4	5
1	1	1	1	1

Running time of Kruskal's algorithm (II)

```
1 def KRUSKAL_MST(undirected G = (V, E), weights w = (w_e)_{e∈E}):
2     Set A := { };
3     for v ∈ V:
4         make_set(v) ;                                          // O(|V|)
5     Sort E in increasing order of edge weights ;               // O(|E| log |V|)
6     for (u, v) ∈ E:
7         if find_set(u) ≠ find_set(v):
8             A := A ∪ {(u, v)};
9             union(u, v);
```

Worst-case cost for union: $O(|V|)$.

What about the cost for lines 6-9?

Consider a single $v \in V$. Once it's touched in some union operation, the size of the set at least doubles. Since the maximum size of a set can be $|V|$, each v is touched at most $O(\log |V|)$ times, at most $|V|$ vertices are involved in union operations, so the total cost of lines 6-9: $O(|V| \log |V|)$

Total cost of the algorithm: $O(|E| \log |V|)$

Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

Directed tree disjoint set:

Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

Directed tree disjoint set:

$$\{a\}$$

Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

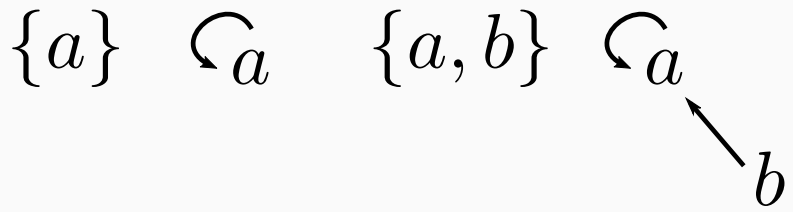
Directed tree disjoint set:

$$\{a\} \quad \circlearrowleft a \quad \{a, b\}$$

Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

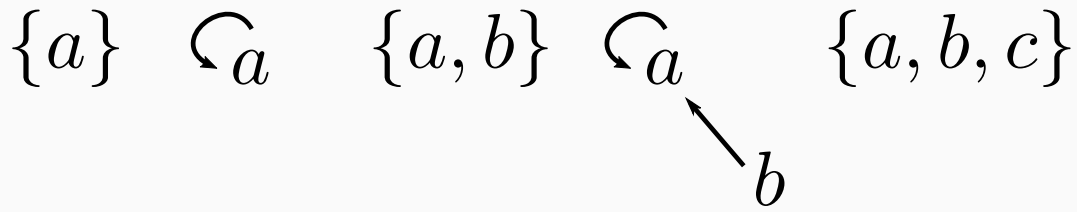
Directed tree disjoint set:



Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

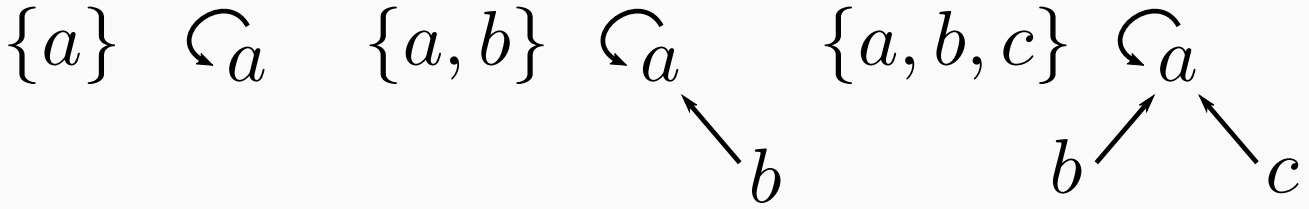
Directed tree disjoint set:



Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

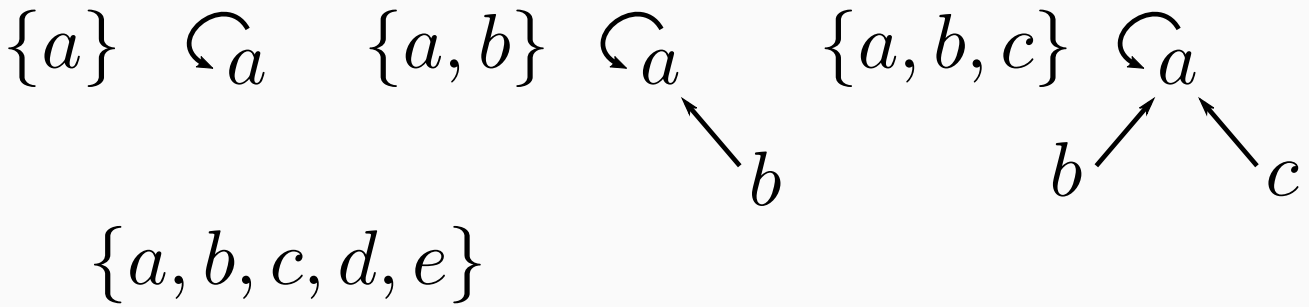
Directed tree disjoint set:



Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

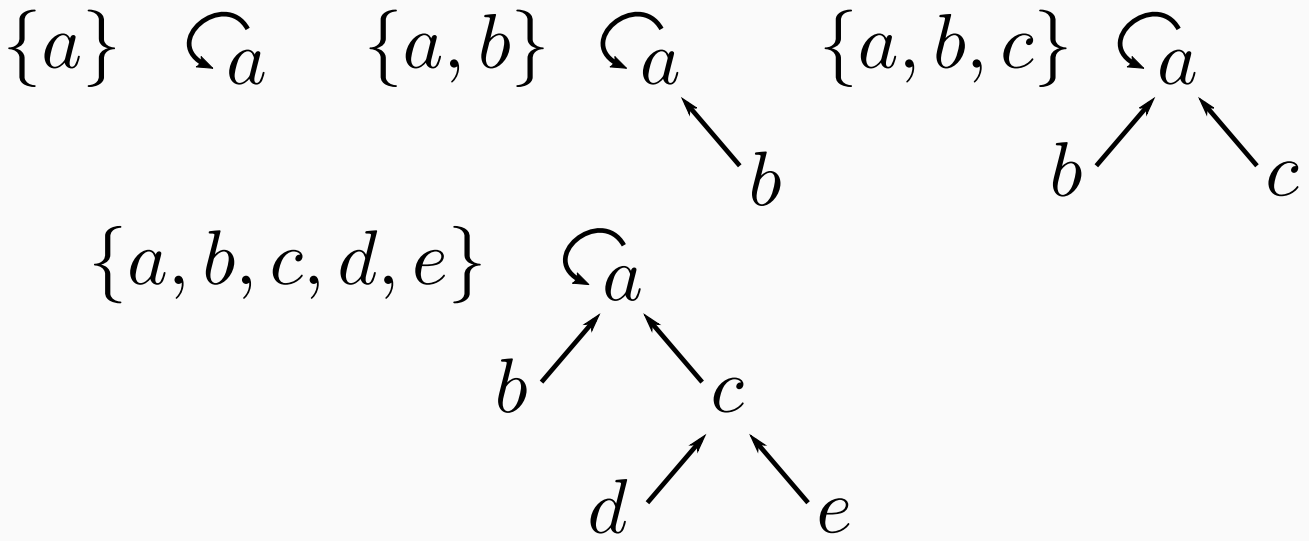
Directed tree disjoint set:



Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

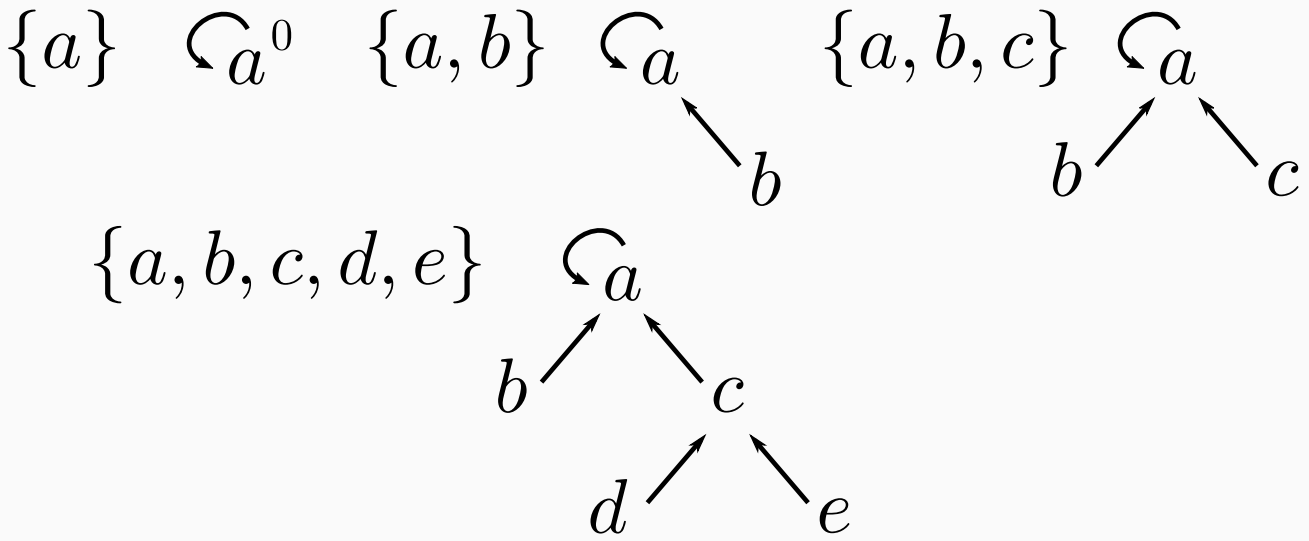
Directed tree disjoint set:



Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

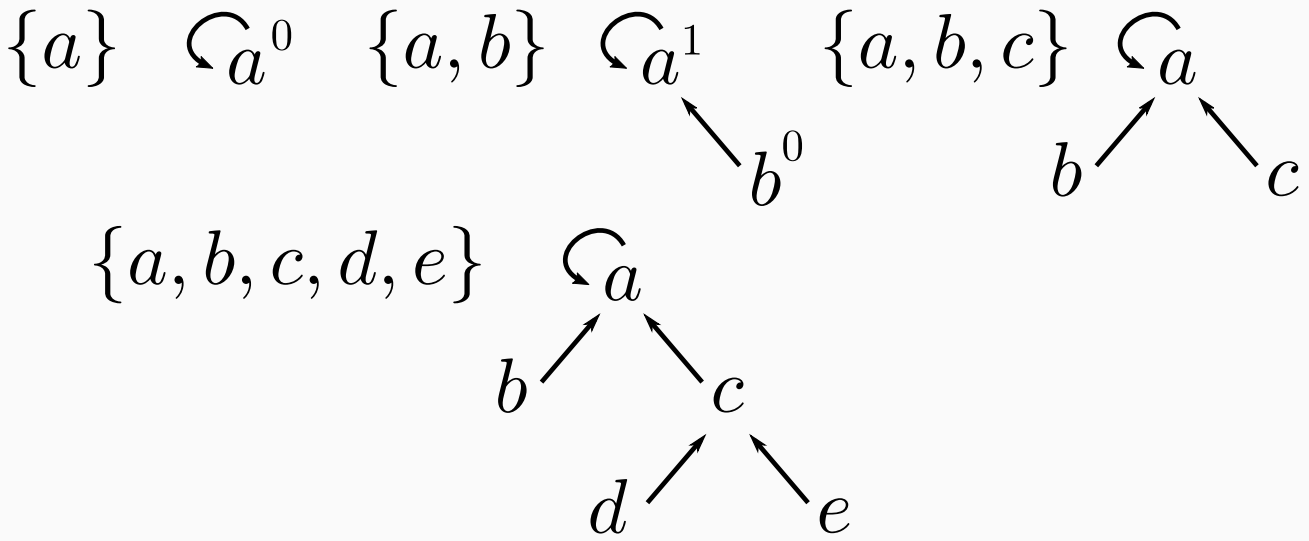
Directed tree disjoint set:



Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

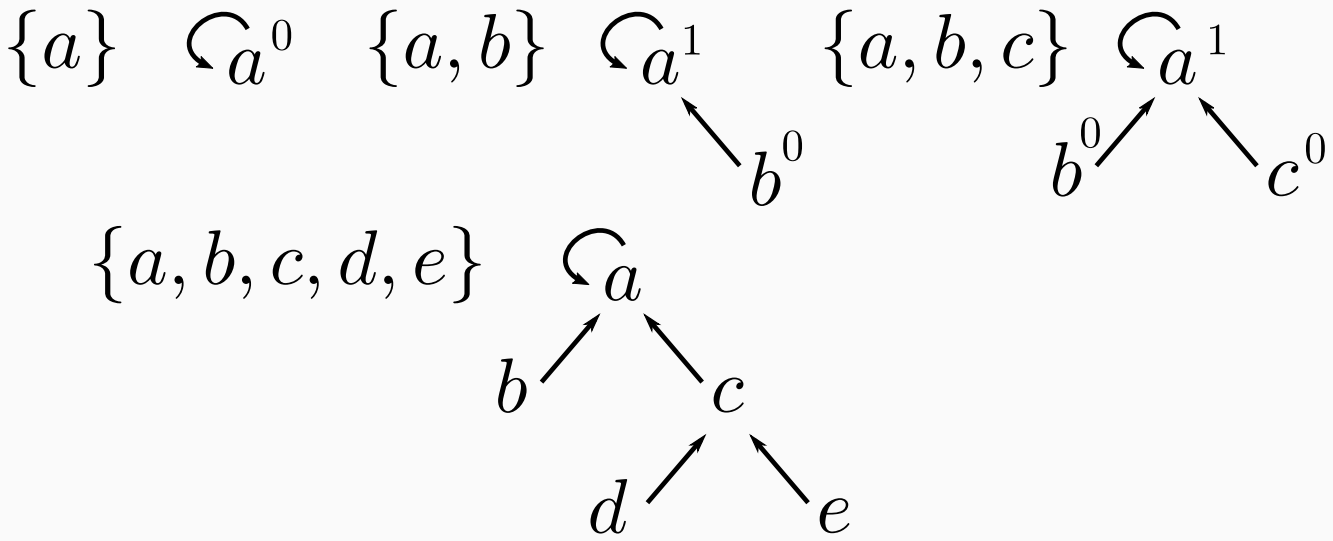
Directed tree disjoint set:



Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

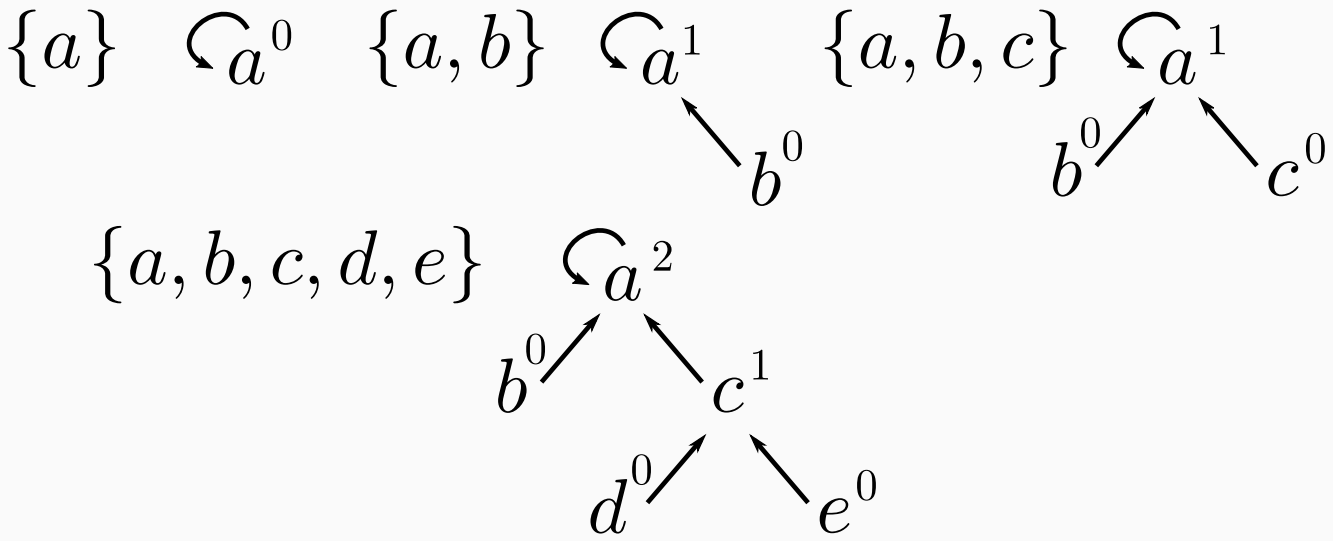
Directed tree disjoint set:



Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

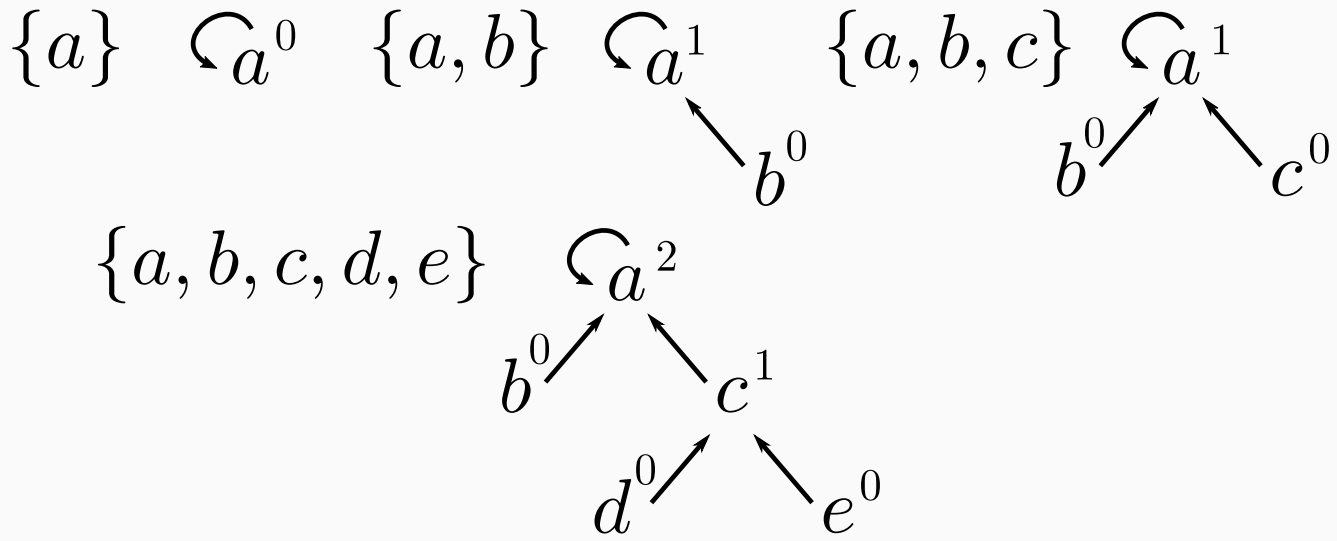
Directed tree disjoint set:



Alternative data structure

The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

Directed tree disjoint set:



Definition

$\pi(x)$: parent of x

root node: x s.t. $\pi(x) = x$

$\mathrm{rank}(x)$: number of the edges in the longest simple path from x to a leaf

Operations of direct tree disjoint set (I)

- $\text{make\_set}(v)$

  def $\text{make\_set}(v)$**:**

  $\pi(v) := v$;

  $\text{rank}(v) = 0$;

  Cost: $O(1)$

- $\text{find\_set}(v)$

  def $\text{find\_set}(v)$**:**

  while $v \neq \pi(v)$**:**

  $v := \pi(v)$;

  return v;

  Cost: O(depth of the node in the tree)

- what about union?

Operations of direct tree disjoint set (II)

- union:

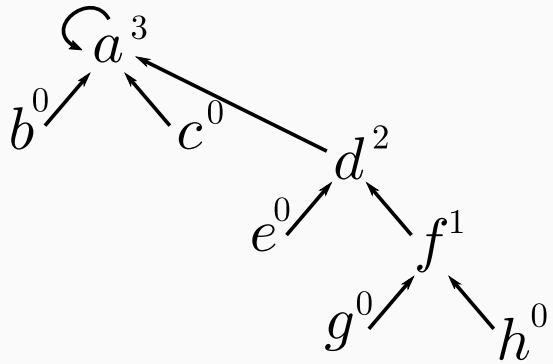
Operations of direct tree disjoint set (II)

- union:

Operations of direct tree disjoint set (II)

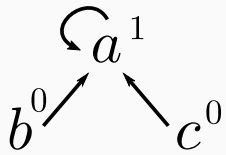
- union:

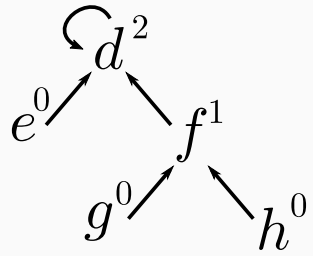
Option 1



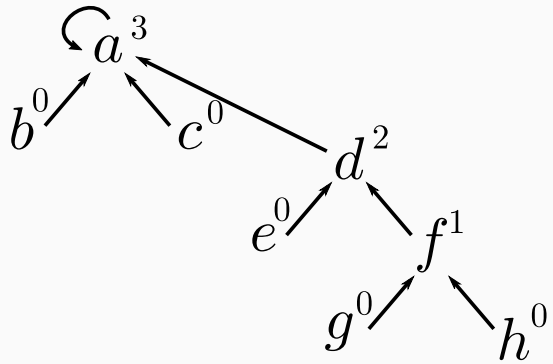
Operations of direct tree disjoint set (II)

- union:

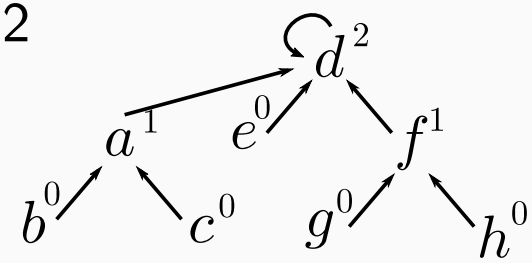




Option 1

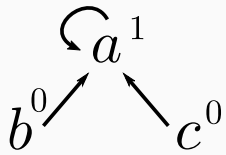


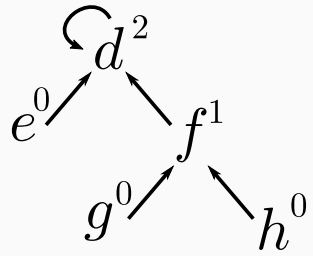
Option 2



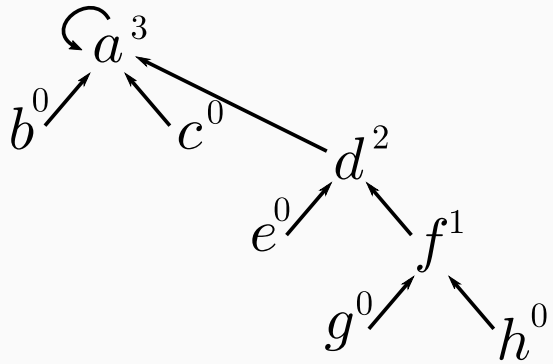
Operations of direct tree disjoint set (II)

- union:

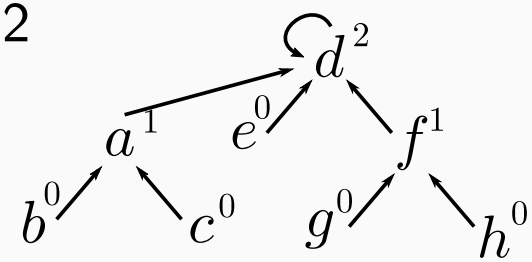




Option 1



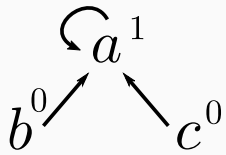
Option 2

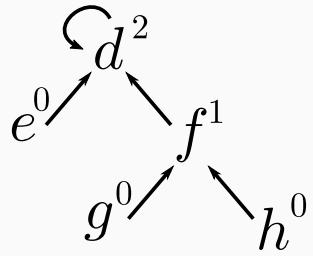


better!

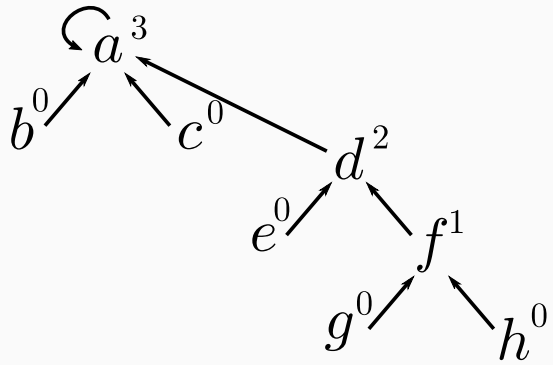
Operations of direct tree disjoint set (II)

- union:

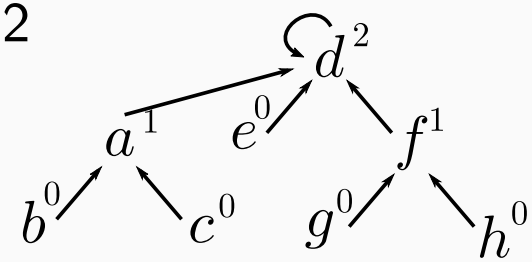




Option 1



Option 2



better!

Basic idea: attach the smaller ranked tree to a larger one

Operations of direct tree disjoint set (II)

```
def union(x, y):
    r_x := find_set(x), r_y := find_set(y);
    if rank(r_x) > rank(r_y):
        π(r_y) := r_x;
    else:
        π(r_x) := r_y;
        if rank(r_x) == rank(r_y):
            rank(r_y) := rank(r_y) + 1;
```

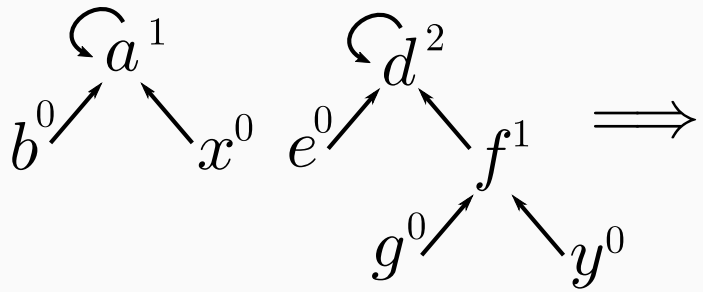
Cost: dominated by find_set

Operations of direct tree disjoint set (II)

def union(x, y)**:**

    r_x := find_set(x), r_y := find_set(y);
    if rank(r_x) > rank(r_y)**:**
        $\pi(r_y) := r_x$;
    else:
        $\pi(r_x) := r_y$;
        if rank(r_x) == rank(r_y)**:**
            rank(r_y) := rank(r_y) + 1;

Cost: dominated by find_set

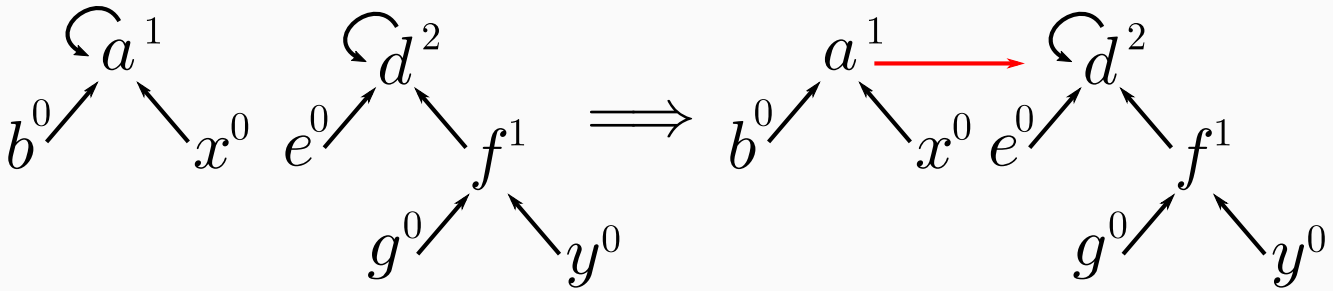


Operations of direct tree disjoint set (II)

def union(x, y)**:**

    $r_x := \text{find\_set}(x)$, $r_y := \text{find\_set}(y)$;
    if $\text{rank}(r_x) > \text{rank}(r_y)$**:**
        $\pi(r_y) := r_x$;
    else:
        $\pi(r_x) := r_y$;
        if $\text{rank}(r_x) == \text{rank}(r_y)$**:**
            $\text{rank}(r_y) := \text{rank}(r_y) + 1$;

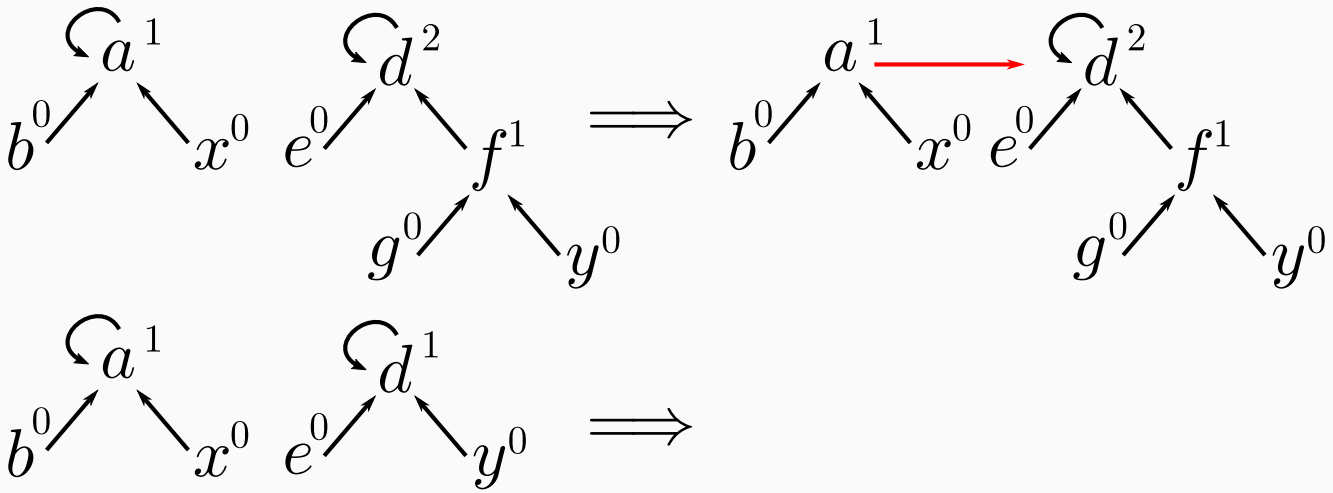
Cost: dominated by find_set



Operations of direct tree disjoint set (II)

def union(x, y):
    $r_x := \text{find\_set}(x)$, $r_y := \text{find\_set}(y)$;
    if $\text{rank}(r_x) > \text{rank}(r_y)$:
        $\pi(r_y) := r_x$;
    else:
        $\pi(r_x) := r_y$;
        if $\text{rank}(r_x) == \text{rank}(r_y)$:
            $\text{rank}(r_y) := \text{rank}(r_y) + 1$;

Cost: dominated by find_set

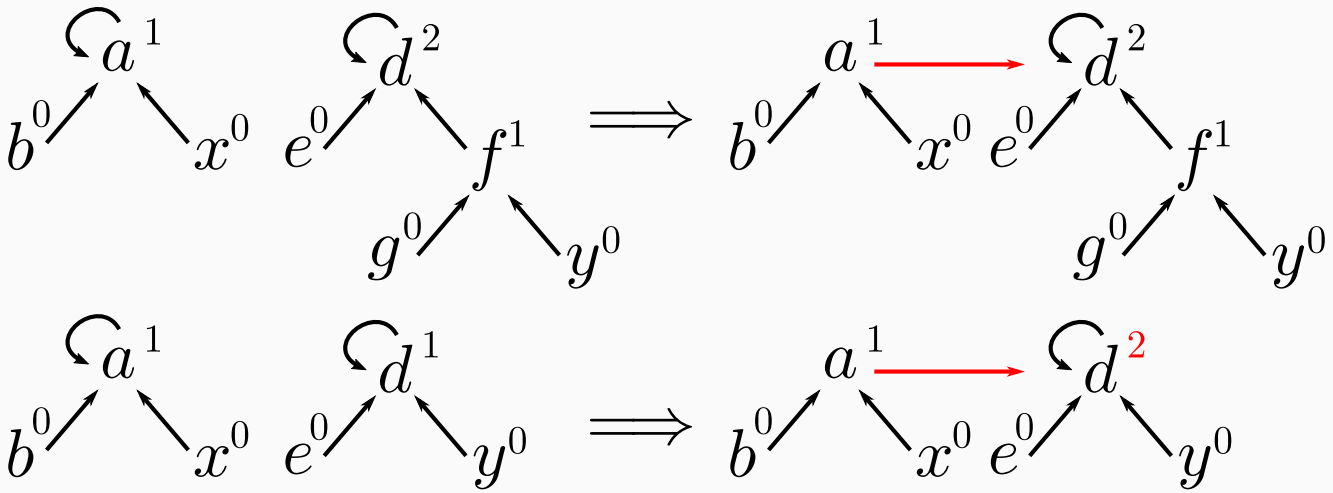


Operations of direct tree disjoint set (II)

def union(x, y)**:**

- $r_x := \text{find\_set}(x)$, $r_y := \text{find\_set}(y)$;
- **if** $\text{rank}(r_x) > \text{rank}(r_y)$**:**
  - $\pi(r_y) := r_x$;
- **else:**
  - $\pi(r_x) := r_y$;
  - **if** $\text{rank}(r_x) == \text{rank}(r_y)$**:**
    - $\text{rank}(r_y) := \text{rank}(r_y) + 1$;

Cost: dominated by find_set



Cost of find_set using directed tree disjoint set

Observation

Root note with rank k is formed by the merge of two rank $k-1$ trees

Lemma

Any root node of rank k has at least 2^k nodes in it

Proof.

By induction: base case has $k = 0$ and $2^0 = 1$.

Assume the statement is true for $k - 1$.

By observation: after merging, the number of nodes is $\geq 2^{k-1} + 2^{k-1} = 2^k$ □

By the lemma, if we have $|V|$ nodes, the maximum rank is $\log |V|$. So

- the cost of $\mathrm{find\_set}$: $O(\log |V|)$
- the cost of union: $O(\log |V|)$

Total running time of Kruskal using directed tree disjoint set

```
1 def KRUSKAL_MST(undirected G = (V, E), weights w = (w_e)_{e∈E}):
2     Set A := { };
3     for v ∈ V:
4         make_set(v) ;                                   // O(|V|)
5     Sort E in increasing order of edge weights ;        // O(|E| log |V|)
6     for (u, v) ∈ E:
7         if find_set(u) ≠ find_set(v):
8             A := A ∪ {(u, v)};
9             union(u, v);
```

Lines 6-9:

$O(|E| \log |V|)$

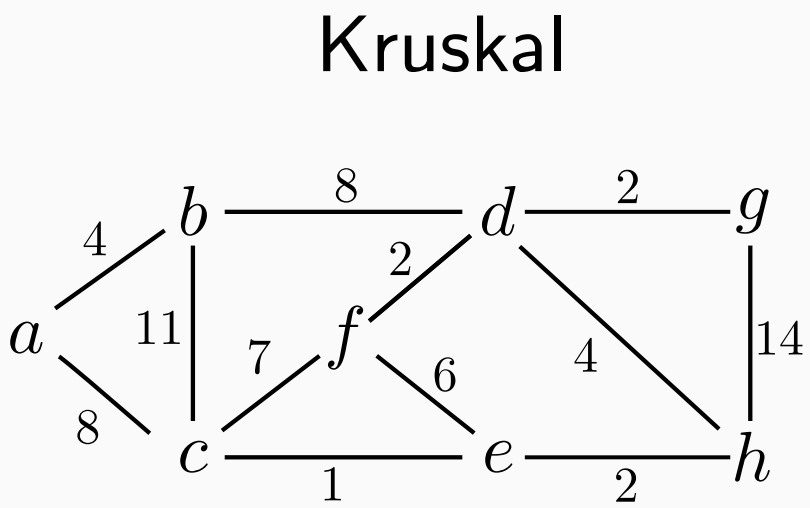
Total cost: $O(|E| \log |V|)$

Prim's algorithm

Intuition: iteratively grows the tree

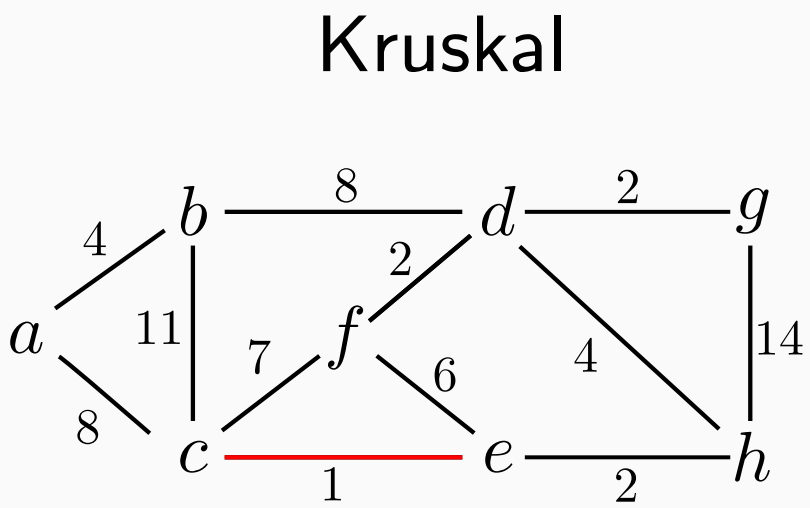
Prim's algorithm

Intuition: iteratively grows the tree



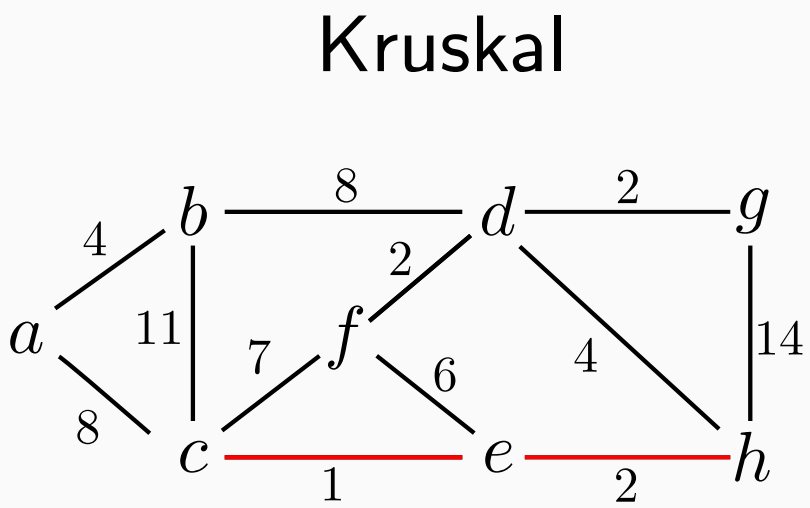
Prim's algorithm

Intuition: iteratively grows the tree



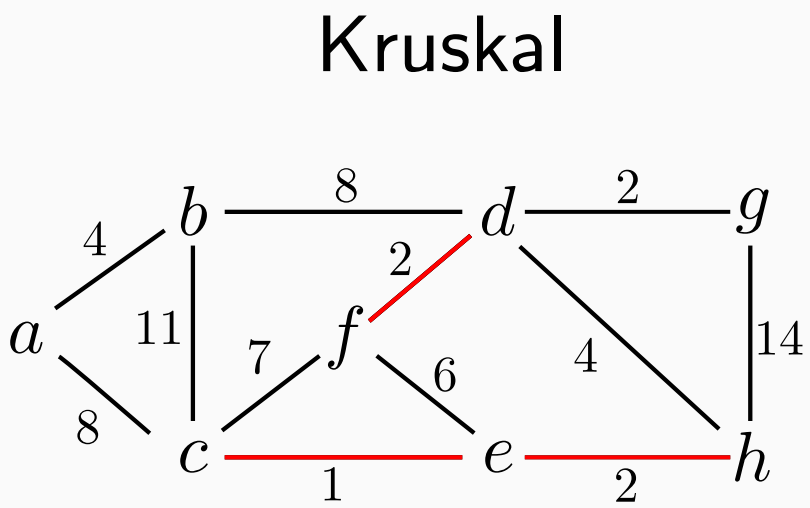
Prim's algorithm

Intuition: iteratively grows the tree



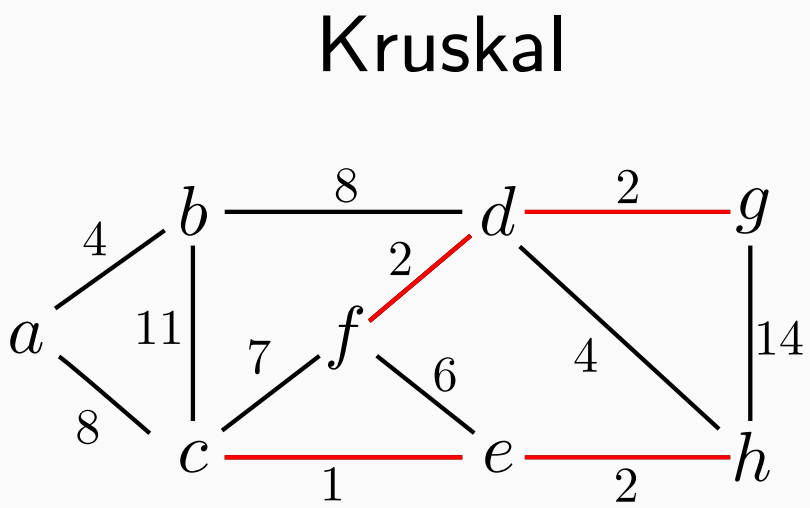
Prim's algorithm

Intuition: iteratively grows the tree



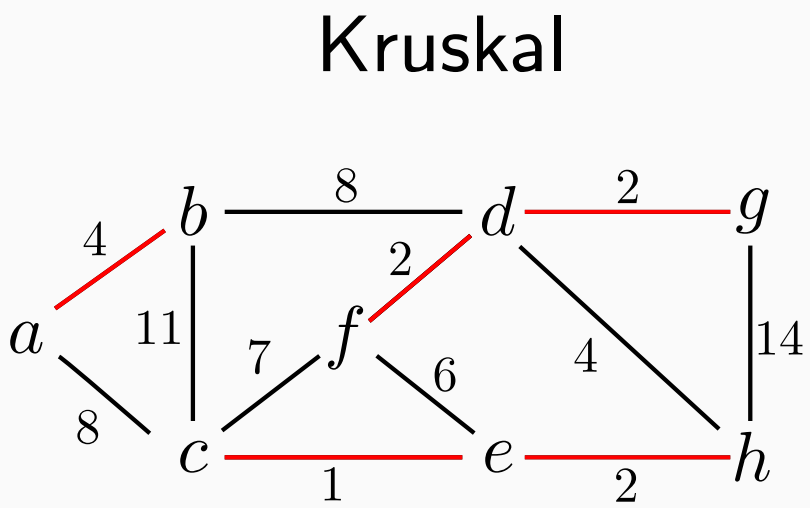
Prim's algorithm

Intuition: iteratively grows the tree



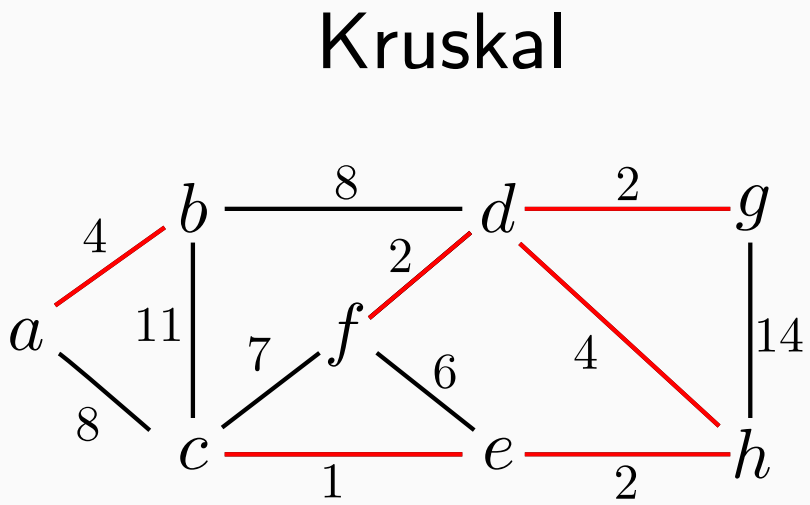
Prim's algorithm

Intuition: iteratively grows the tree



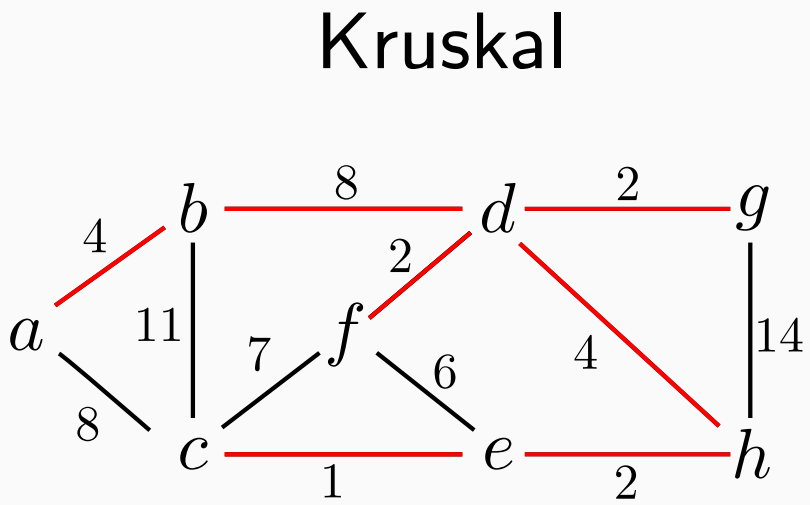
Prim's algorithm

Intuition: iteratively grows the tree



Prim's algorithm

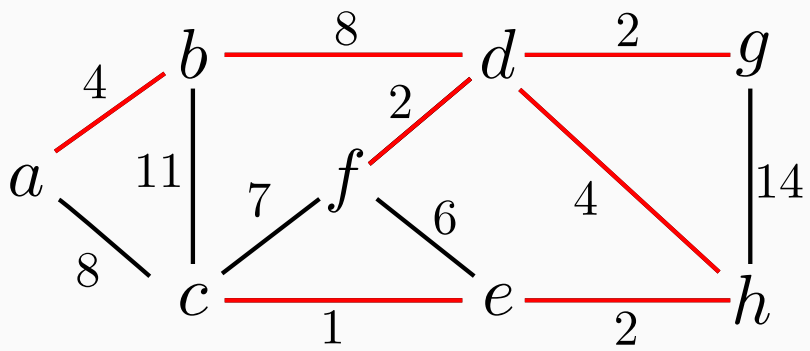
Intuition: iteratively grows the tree



Prim's algorithm

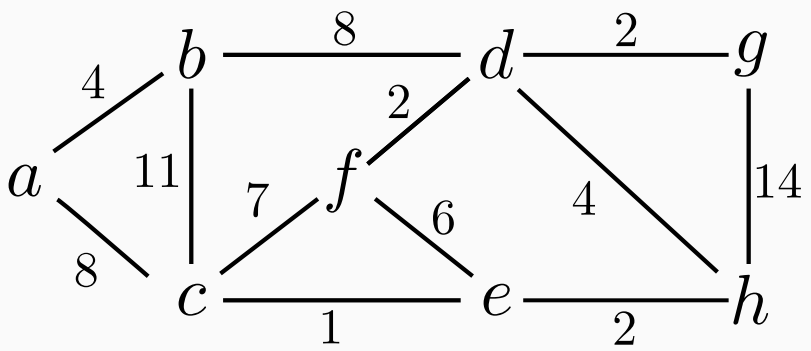
Intuition: iteratively grows the tree

Kruskal



Prim

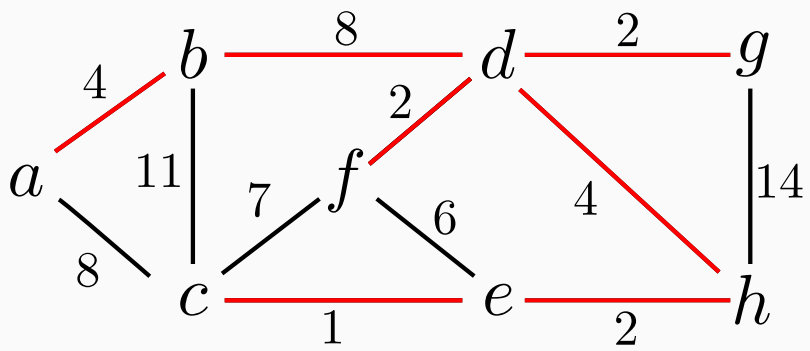
starting with c



Prim's algorithm

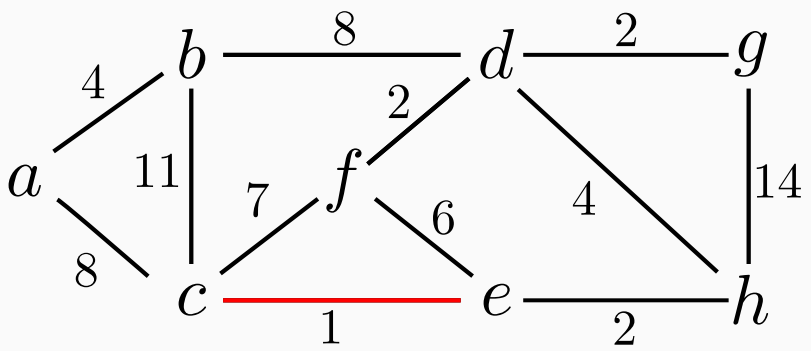
Intuition: iteratively grows the tree

Kruskal



Prim

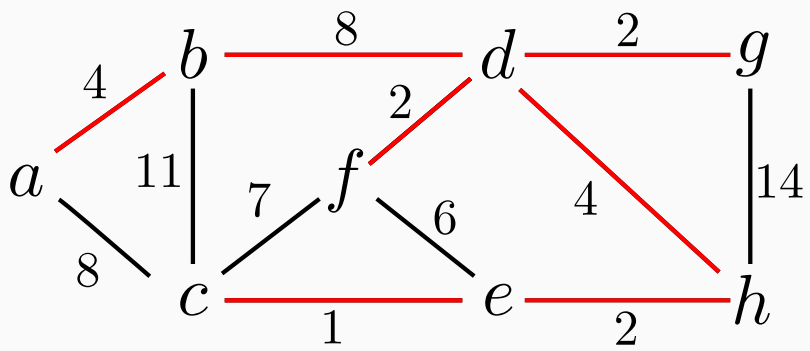
starting with c



Prim's algorithm

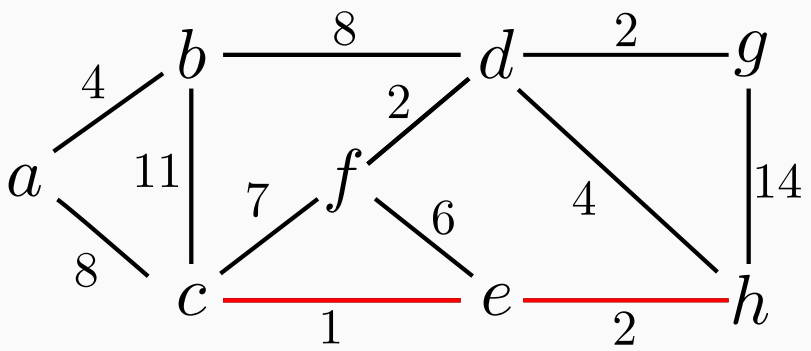
Intuition: iteratively grows the tree

Kruskal



Prim

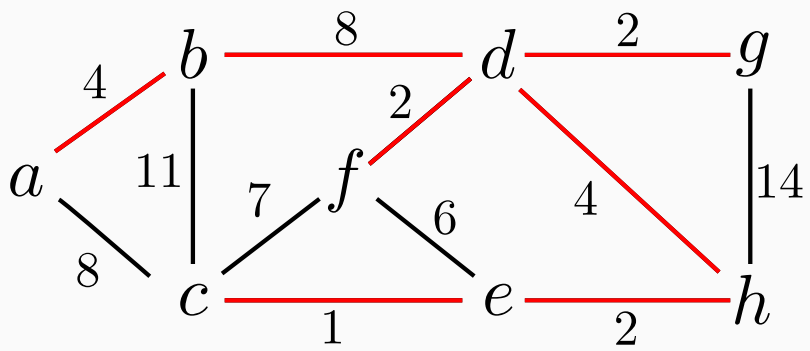
starting with c



Prim's algorithm

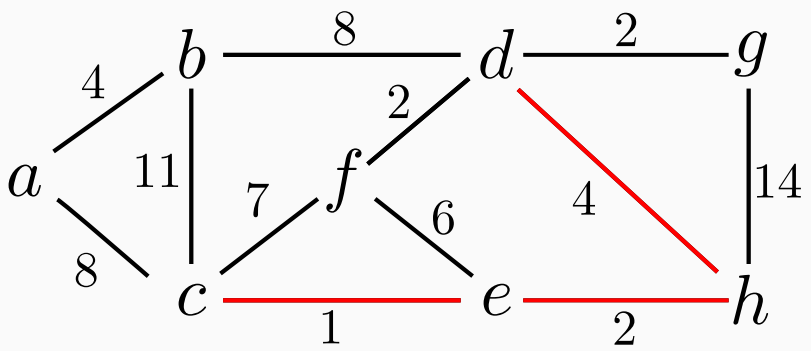
Intuition: iteratively grows the tree

Kruskal



Prim

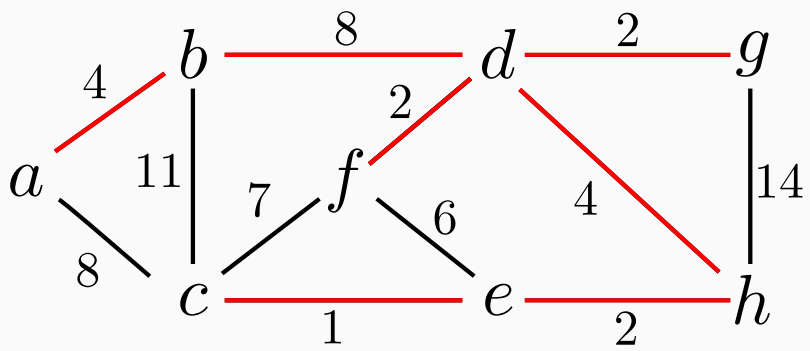
starting with c



Prim's algorithm

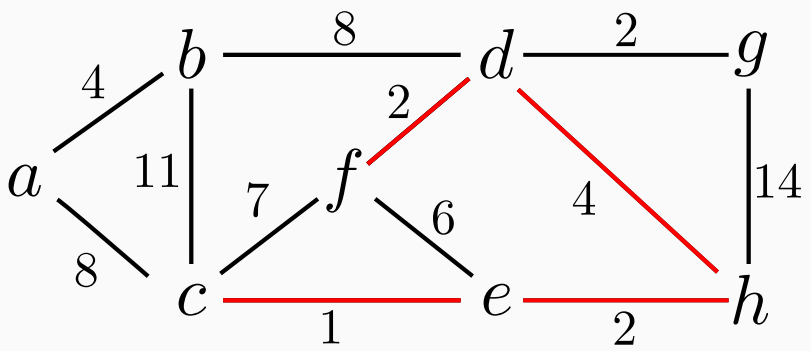
Intuition: iteratively grows the tree

Kruskal



Prim

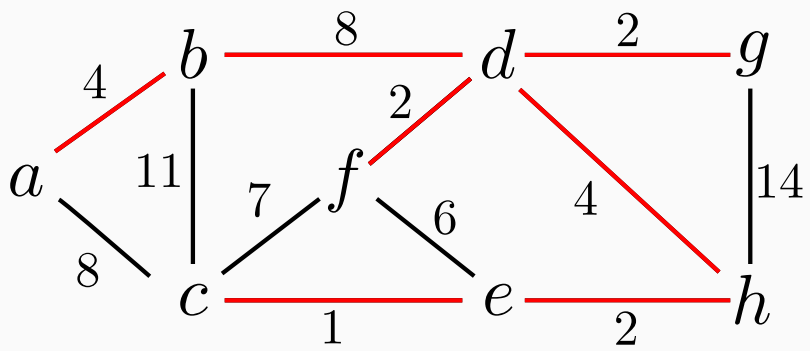
starting with c



Prim's algorithm

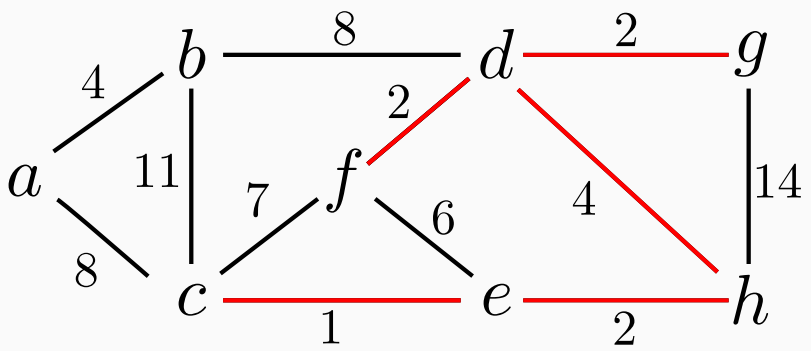
Intuition: iteratively grows the tree

Kruskal



Prim

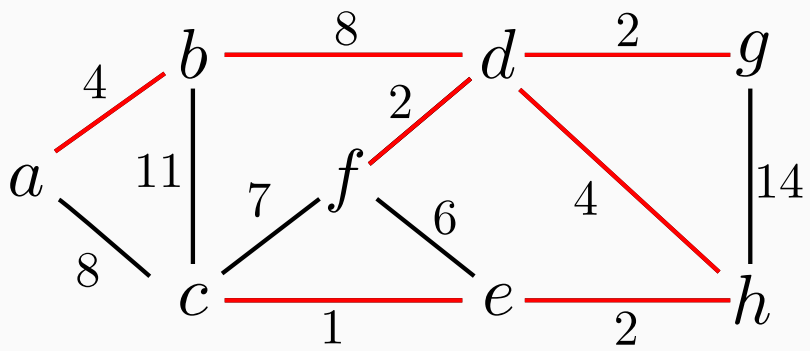
starting with c



Prim's algorithm

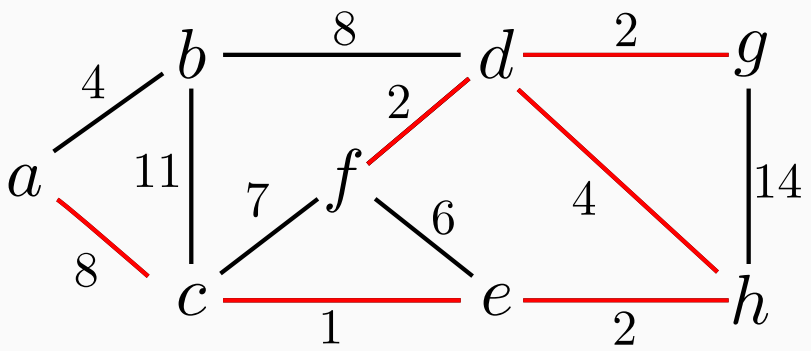
Intuition: iteratively grows the tree

Kruskal



Prim

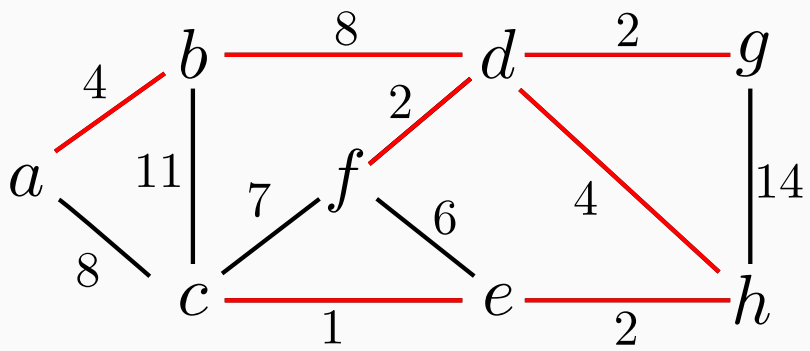
starting with c



Prim's algorithm

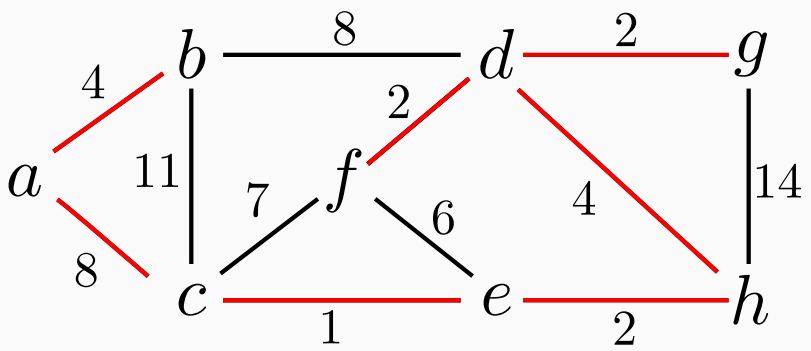
Intuition: iteratively grows the tree

Kruskal



Prim

starting with c



Prim's algorithm: pseudocode

Let S be the set included in the tree so far

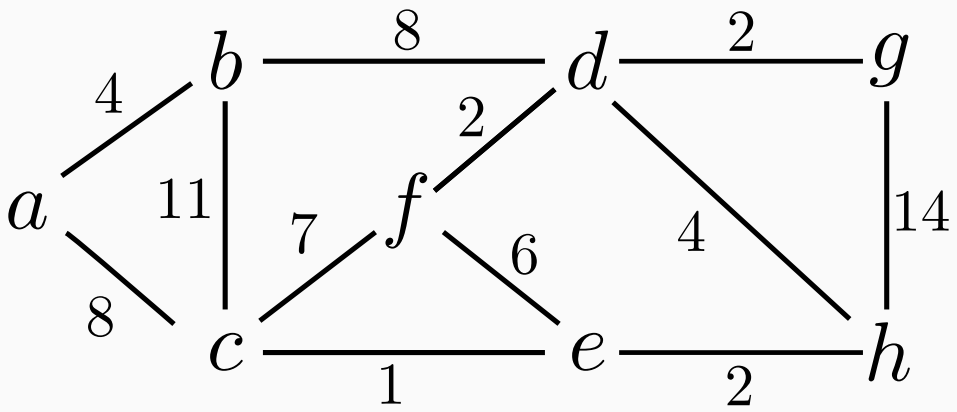
$\text{cost}(v) := \min_{e=(u,v) \text{ s.t. } u \in S} w_e$

and $\text{prev}(\cdot)$ is used to keep track of the tree

```
def PRIM_MST(undirected G = (V, E), weights w = (w_e)_{e∈E}):
    for v ∈ V:
        cost(v) := ∞;
        prev(v) := nil;
    Pick any initial vertex u_0;
    cost(u_0) := 0;
    H := make_queue(V) ;                          // keys are cost(v)
    while H is not empty:
        v = delete_min(H);
        for e := (v, z) ∈ E:
            if cost(z) > w_e:
                cost(z) := w_e;
                prev(z) := v;
```

Prim's algorithm: a running example

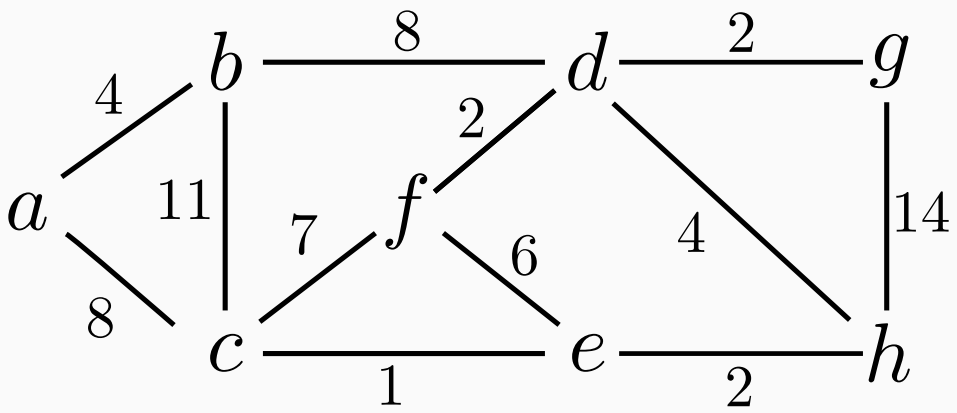
Starting with f



Set S	a	b	c	d	e	f	g	h
$\{\}$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	0/nil	∞/nil	∞/nil

Prim's algorithm: a running example

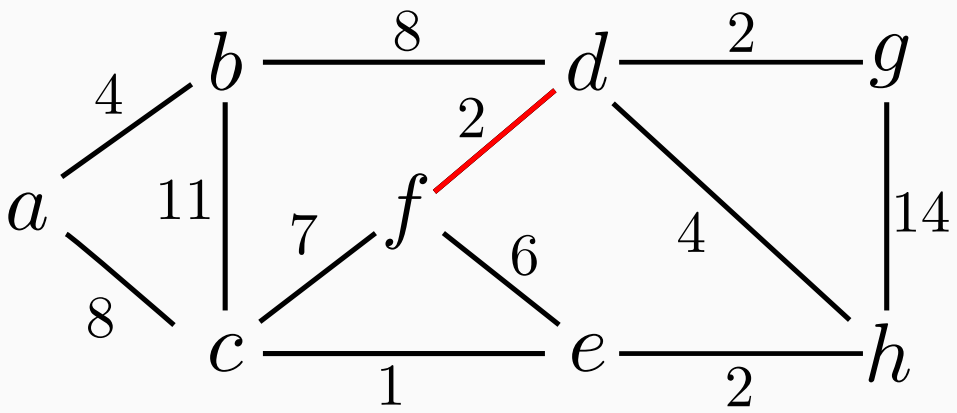
Starting with f



Set S	a	b	c	d	e	f	g	h
$\{\}$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	0/nil	∞/nil	∞/nil
f	∞/nil	∞/nil	7/f	2/f	6/f		∞/nil	∞/nil

Prim's algorithm: a running example

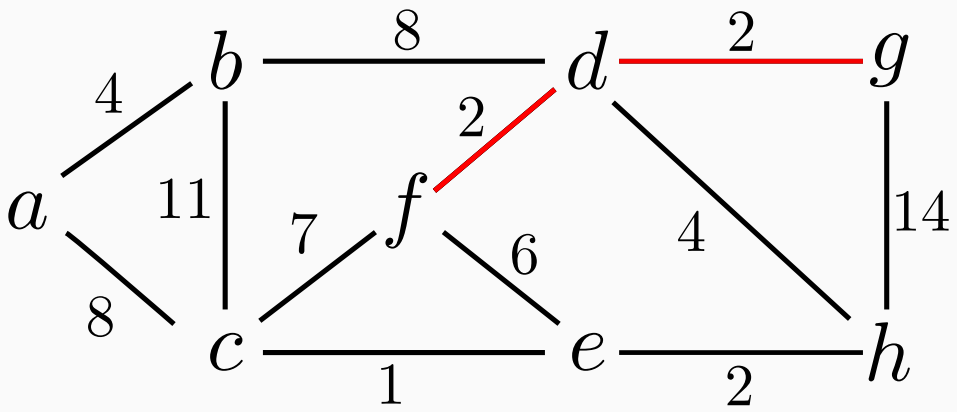
Starting with f



Set S	a	b	c	d	e	f	g	h
$\{\}$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	0/nil	∞/nil	∞/nil
f	∞/nil	∞/nil	7/f	2/f	6/f		∞/nil	∞/nil
f, d	∞/nil	8/d	7/f		6/f		2/d	4/d

Prim's algorithm: a running example

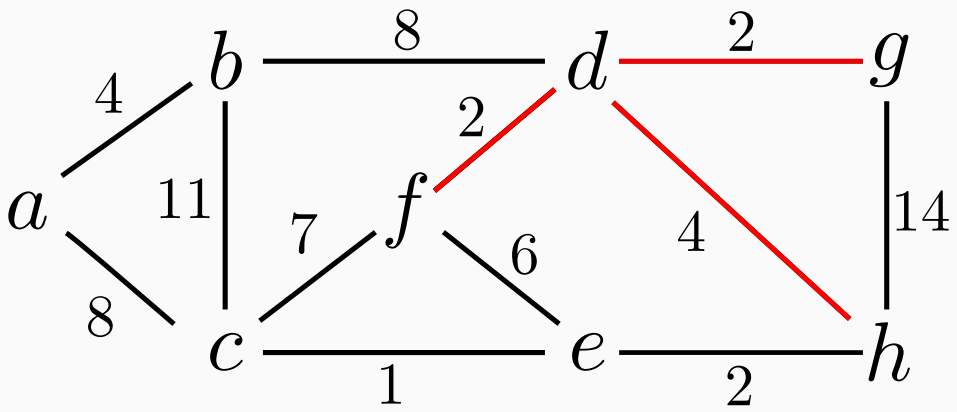
Starting with f



Set S	a	b	c	d	e	f	g	h
$\{\}$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	0/nil	∞/nil	∞/nil
f	∞/nil	∞/nil	7/f	2/f	6/f		∞/nil	∞/nil
f, d	∞/nil	8/d	7/f		6/f		2/d	4/d
f, d, g	∞/nil	8/d	7/f		6/f			4/d

Prim's algorithm: a running example

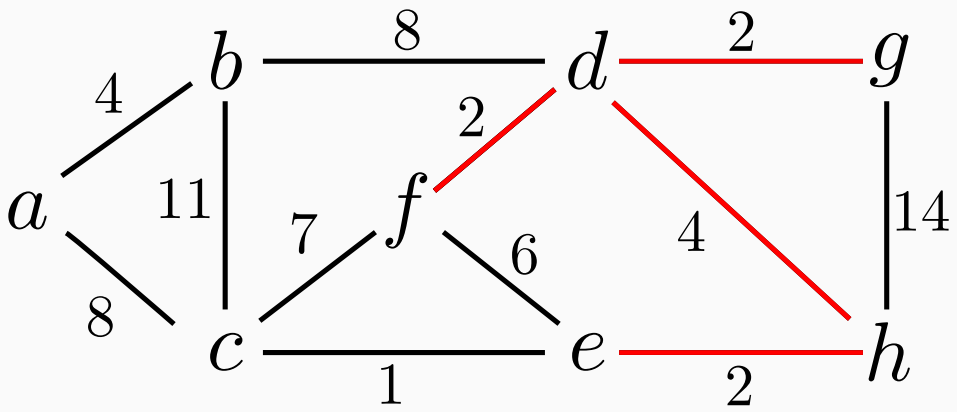
Starting with f



Set S	a	b	c	d	e	f	g	h
$\{\}$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	0/nil	∞/nil	∞/nil
f	∞/nil	∞/nil	7/f	2/f	6/f		∞/nil	∞/nil
f, d	∞/nil	8/d	7/f		6/f		2/d	4/d
f, d, g	∞/nil	8/d	7/f		6/f			4/d
f, d, g, h	∞/nil	8/d	7/f		2/h			

Prim's algorithm: a running example

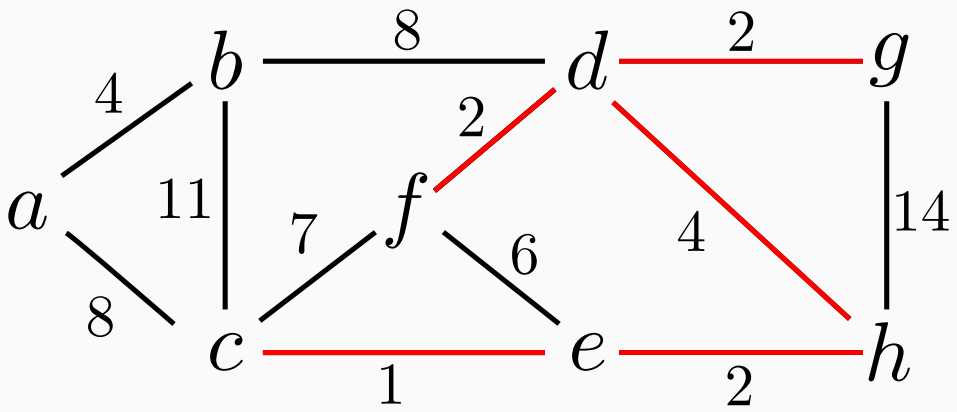
Starting with f



Set S	a	b	c	d	e	f	g	h
$\{\}$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	0/nil	∞/nil	∞/nil
f	∞/nil	∞/nil	7/f	2/f	6/f		∞/nil	∞/nil
f, d	∞/nil	8/d	7/f		6/f		2/d	4/d
f, d, g	∞/nil	8/d	7/f		6/f			4/d
f, d, g, h	∞/nil	8/d	7/f		2/h			
f, d, g, h, e	∞/nil	8/d	1/e					

Prim's algorithm: a running example

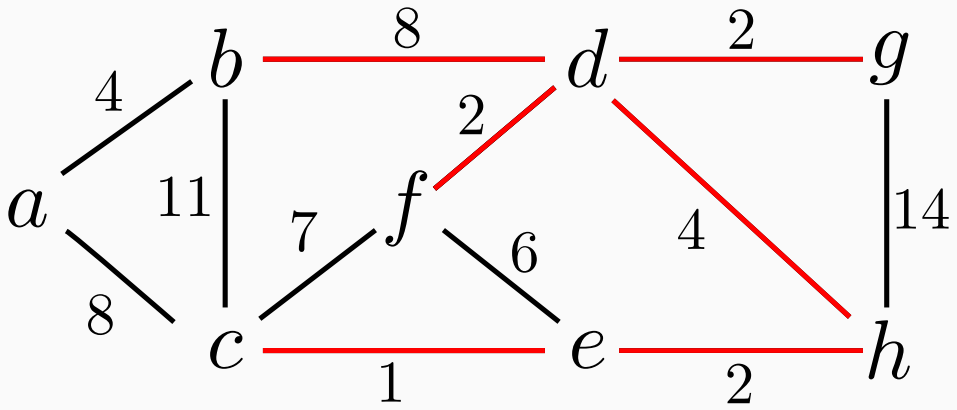
Starting with f



Set S	a	b	c	d	e	f	g	h
$\{\}$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	0/nil	∞/nil	∞/nil
f	∞/nil	∞/nil	7/f	2/f	6/f		∞/nil	∞/nil
f, d	∞/nil	8/d	7/f		6/f		2/d	4/d
f, d, g	∞/nil	8/d	7/f		6/f			4/d
f, d, g, h	∞/nil	8/d	7/f		2/h			
f, d, g, h, e	∞/nil	8/d	1/e					
f, d, g, h, e, c	8/c	8/d						

Prim's algorithm: a running example

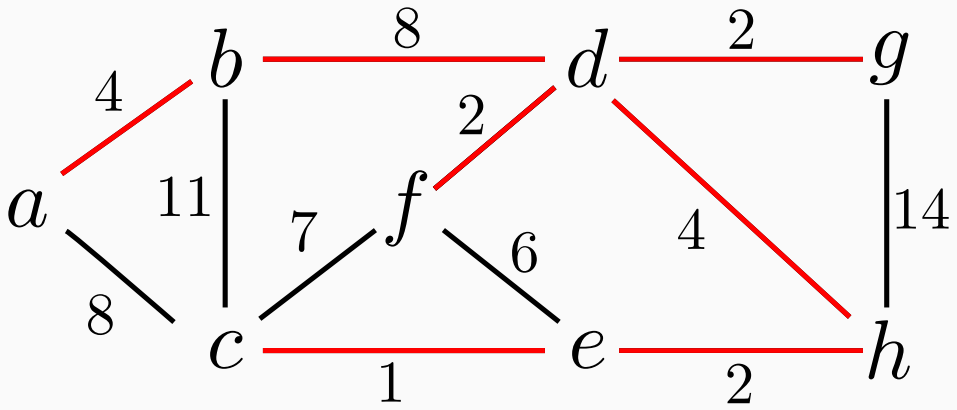
Starting with f



Set S	a	b	c	d	e	f	g	h
$\{\}$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	0/nil	∞/nil	∞/nil
f	∞/nil	∞/nil	7/f	2/f	6/f		∞/nil	∞/nil
f, d	∞/nil	8/d	7/f		6/f		2/d	4/d
f, d, g	∞/nil	8/d	7/f		6/f			4/d
f, d, g, h	∞/nil	8/d	7/f		2/h			
f, d, g, h, e	∞/nil	8/d	1/e					
f, d, g, h, e, c	8/c	8/d						
f, d, g, h, e, c, b	4/b							

Prim's algorithm: a running example

Starting with f



Set S	a	b	c	d	e	f	g	h
$\{\}$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	0/nil	∞/nil	∞/nil
f	∞/nil	∞/nil	7/f	2/f	6/f		∞/nil	∞/nil
f, d	∞/nil	8/d	7/f		6/f		2/d	4/d
f, d, g	∞/nil	8/d	7/f		6/f			4/d
f, d, g, h	∞/nil	8/d	7/f		2/h			
f, d, g, h, e	∞/nil	8/d	1/e					
f, d, g, h, e, c	8/c	8/d						
f, d, g, h, e, c, b	4/b							
f, d, g, h, e, c, b, a								

Huffman Encoding (Textbook Section 5.2)

Huffman Encoding

An encoding scheme used in, e.g., MP3 encoding

Data: a string S of symbols over an alphabet Γ

Goal: find a binary encoding e of Γ resulting in minimum encoded length of S

Denote the encoded string by S_e

Example: ASCII encoding

a	01100001
b	01100010
$\vdots$	$\vdots$

Different encodings

Consider $\Gamma = \{a, b, c\}$ Stats on S: a appears 45 times, b 16 times, and c twice

- Fixed-length encoding

$$e_1: \begin{array}{l} a \to 00 \\ b \to 01 \\ c \to 10 \end{array} \qquad |S_{e_1}| = 45 \times 2 + 16 \times 2 + 2 \times 2 = 126$$

- Variable-length encoding

$$e_2: \begin{array}{l} a \to 0 \\ b \to 10 \\ c \to 11 \end{array} \qquad |S_{e_2}| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

- Be careful!

$$e_2: \begin{array}{l} a \to 0 \\ b \to 1 \\ c \to 01 \end{array}$$

Decoding will lead to ambiguity

Prefix-free encoding

Consider the bad encoding e_2 : $a \rightarrow 0$, $b \rightarrow 1$, $c \rightarrow 01$

How to decode 010110?

ababba?, *ccba*?, *abcba*?, or . . . ?

To avoid ambiguity, we need the encoding to be prefix-free

Definition

An encoding is **prefix-free** if no codeword is a prefix of any other codewords

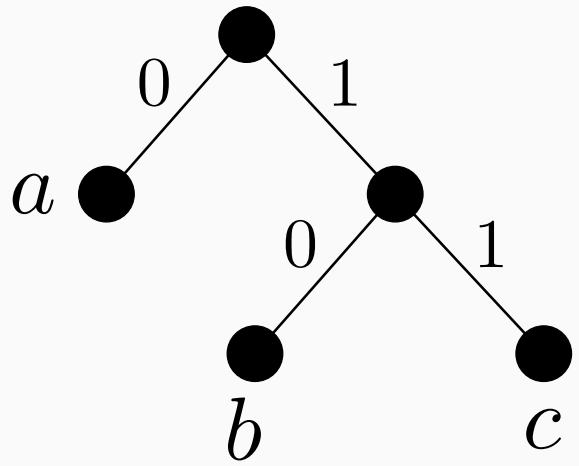
Tree representation of a prefix-free encoding (I)

Definition

A **full binary tree** is a binary tree where each node is either a leaf or it has two children

We use a full binary tree to represent a prefix-free encoding

- leaves are corresponding to symbols in Γ
- label edge to the left child with 0
- label edge to the right child with 1



To obtain the encoding, read edge labels from root to a symbol

$a \to 0$, $b \to 10$, $c \to 11$,

Depth of a leaf $\equiv$ length of its codeword

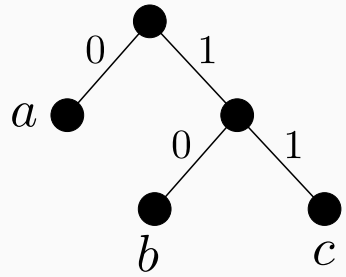
It guarantees to be prefix-free

Tree representation of a prefix-free encoding (II)

Let e be an encoding represented by a tree

For string S, let f_v be the symbol count in S for each $v \in \Gamma$

$$|S_e| = \sum_{v \in \Gamma} f_v \cdot \text{depth}(v)$$



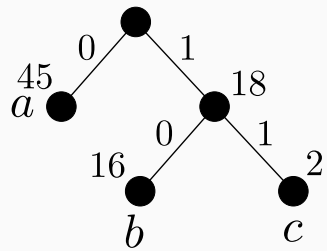
$$|S_e| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

A useful re-write: label internal nodes with counts of descendants

For all non-root node v, define

$\text{cost}(v) :=$ sum of leaf node counts descending from v

$$|S_e| = \sum_{v \in T - \{\text{root}\}} \text{cost}(v)$$



$$|S_e| = 45 + 16 + 2 + 18 = 81$$

Constructing the prefix-free encoding tree

Idea: put more frequent symbols at smaller depth

Greedy approach: continually merge least frequent symbols/nodes until you have a full binary tree encoding all symbols

Constructing the prefix-free encoding tree – examples

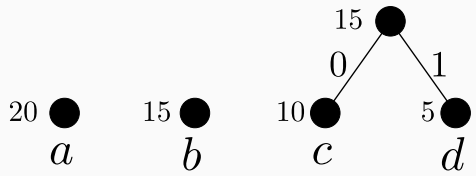
- $a : 20, b : 15, c : 10, d : 5$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$

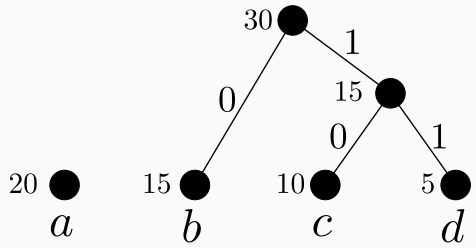
Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



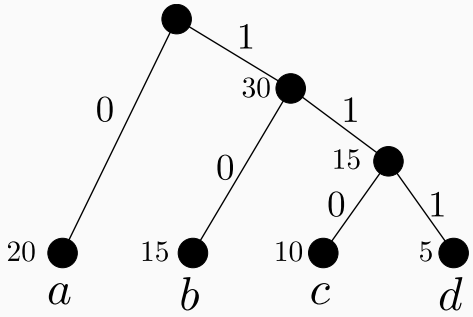
Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



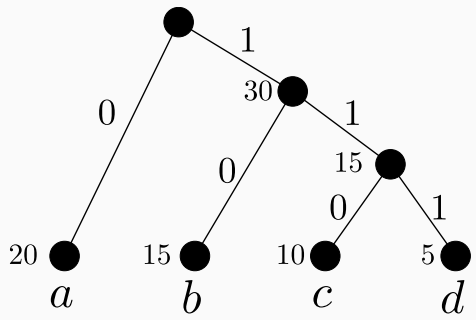
Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



$a \rightarrow 0$

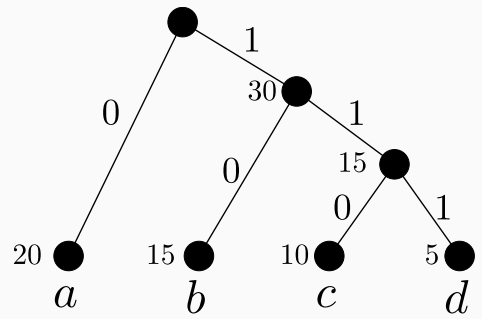
$b \rightarrow 10$

$c \rightarrow 110$

$d \rightarrow 111$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



$a \rightarrow 0$

$b \rightarrow 10$

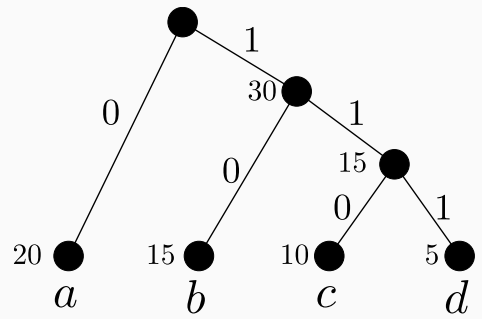
$c \rightarrow 110$

$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



$a \rightarrow 0$

$b \rightarrow 10$

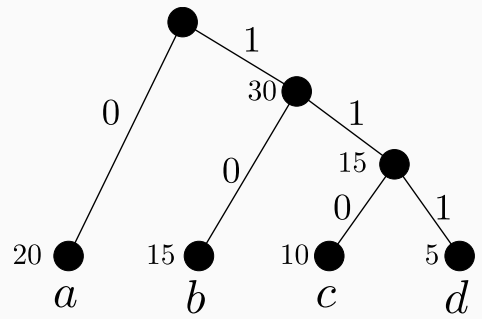
$c \rightarrow 110$

$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



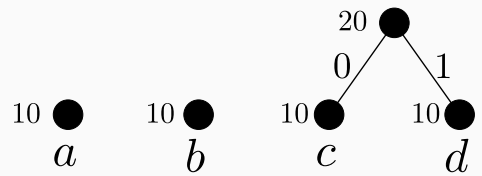
$a \rightarrow 0$

$b \rightarrow 10$

$c \rightarrow 110$

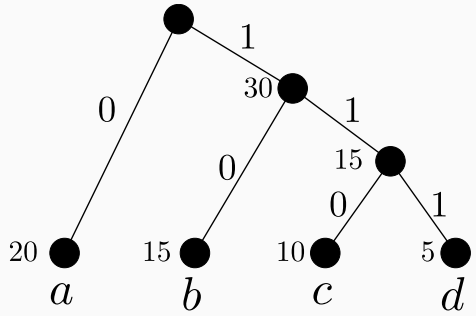
$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



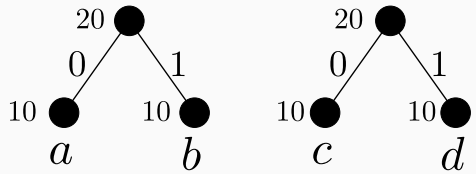
$a \rightarrow 0$

$b \rightarrow 10$

$c \rightarrow 110$

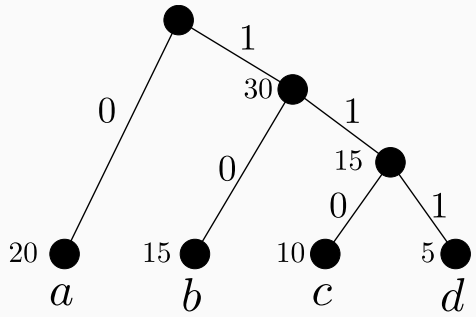
$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



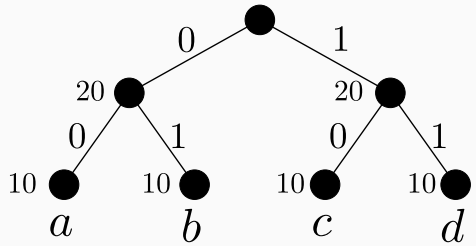
$a \rightarrow 0$

$b \rightarrow 10$

$c \rightarrow 110$

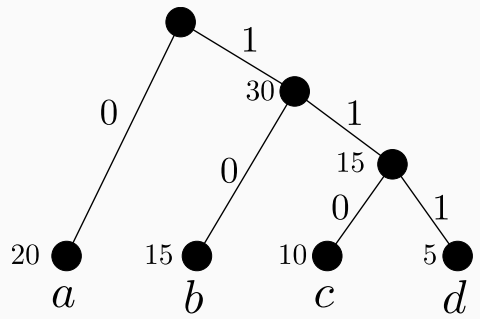
$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



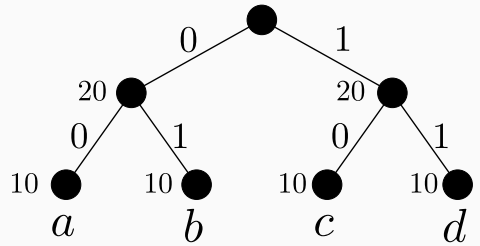
$a \rightarrow 0$

$b \rightarrow 10$

$c \rightarrow 110$

$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$



$a \rightarrow 00$

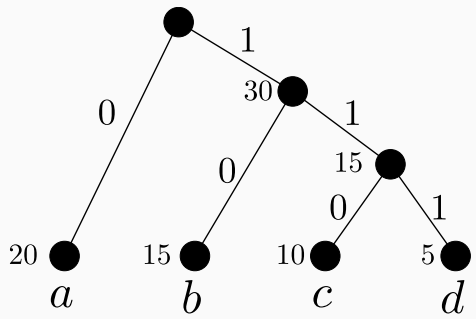
$b \rightarrow 01$

$c \rightarrow 10$

$d \rightarrow 11$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



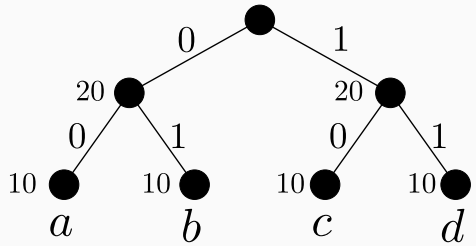
$a \rightarrow 0$

$b \rightarrow 10$

$c \rightarrow 110$

$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$



$a \rightarrow 00$

$b \rightarrow 01$

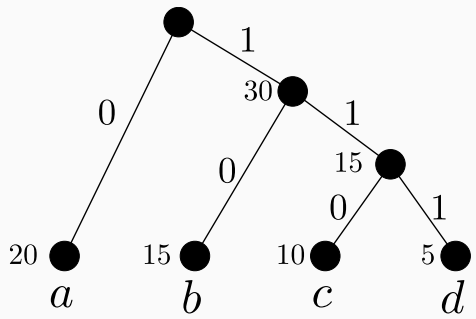
$c \rightarrow 10$

$d \rightarrow 11$

- $a : 10, b : 10, c : 5, d : 5$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



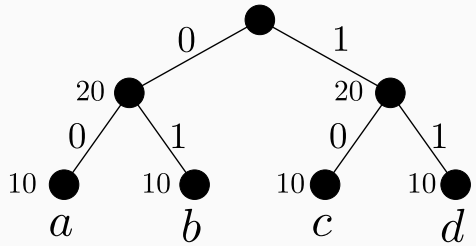
$a \rightarrow 0$

$b \rightarrow 10$

$c \rightarrow 110$

$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$



$a \rightarrow 00$

$b \rightarrow 01$

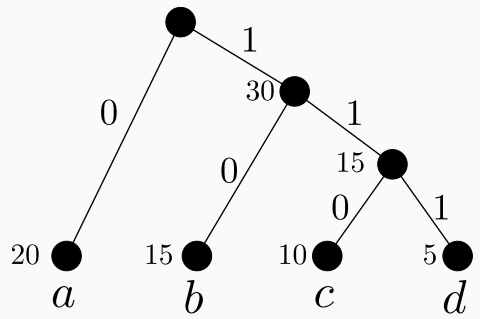
$c \rightarrow 10$

$d \rightarrow 11$

- $a : 10, b : 10, c : 5, d : 5$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



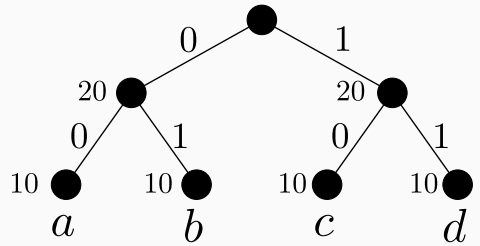
$a \rightarrow 0$

$b \rightarrow 10$

$c \rightarrow 110$

$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$



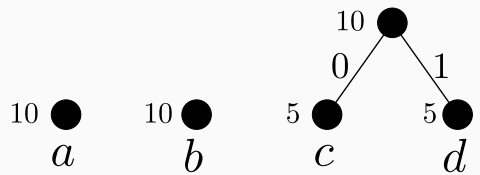
$a \rightarrow 00$

$b \rightarrow 01$

$c \rightarrow 10$

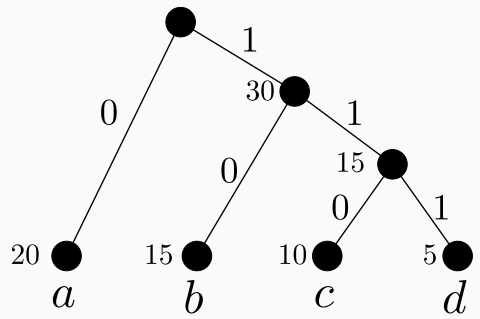
$d \rightarrow 11$

- $a : 10, b : 10, c : 5, d : 5$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



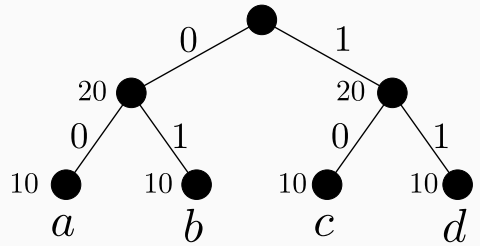
$a \rightarrow 0$

$b \rightarrow 10$

$c \rightarrow 110$

$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$



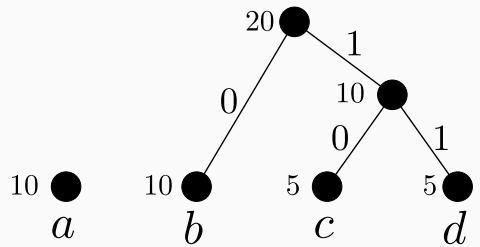
$a \rightarrow 00$

$b \rightarrow 01$

$c \rightarrow 10$

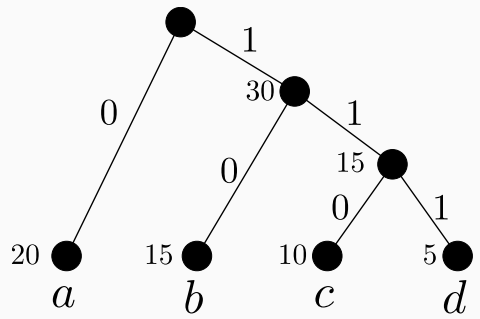
$d \rightarrow 11$

- $a : 10, b : 10, c : 5, d : 5$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



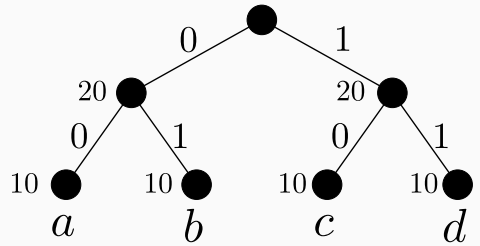
$a \rightarrow 0$

$b \rightarrow 10$

$c \rightarrow 110$

$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$



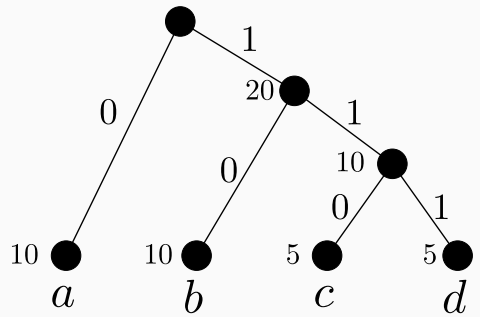
$a \rightarrow 00$

$b \rightarrow 01$

$c \rightarrow 10$

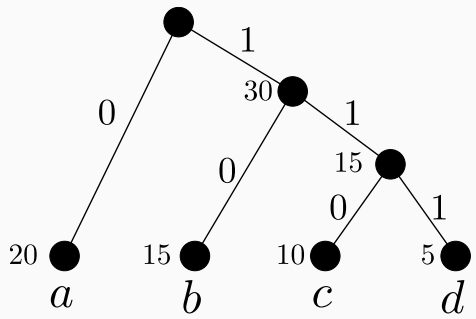
$d \rightarrow 11$

- $a : 10, b : 10, c : 5, d : 5$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



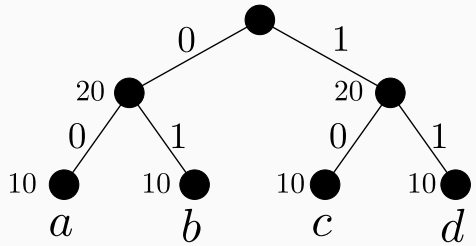
$a \rightarrow 0$

$b \rightarrow 10$

$c \rightarrow 110$

$d \rightarrow 111$

- $a : 10, b : 10, c : 10, d : 10$



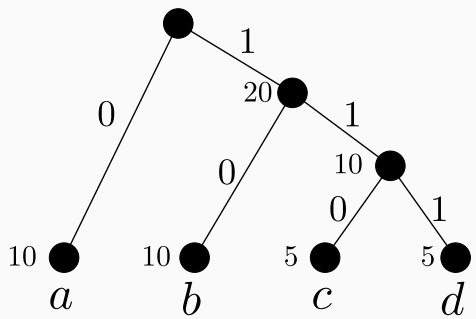
$a \rightarrow 00$

$b \rightarrow 01$

$c \rightarrow 10$

$d \rightarrow 11$

- $a : 10, b : 10, c : 5, d : 5$



$a \rightarrow 0$

$b \rightarrow 10$

$c \rightarrow 110$

$d \rightarrow 111$

Proof of optimality (I)

Proof sketch

- Greedy choice property:

Claim

Every optimal solution has two lowest frequent symbols as leaves connected to an internal node of greatest depth

Proof. (exchange argument).

Let we have a tree T with two lowest frequent symbols not as deep as possible.
Then at least one has a smaller depth.
Switch it with one of the deepest nodes that is more frequent.
This improves the encoding length. Thus T is not optimal □

Proof of optimality (I)

Proof sketch

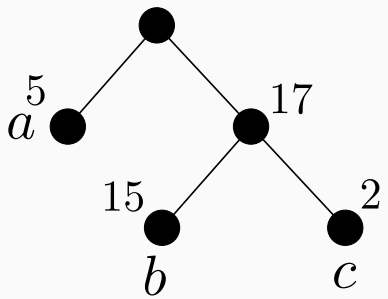
- Greedy choice property:

Claim

Every optimal solution has two lowest frequent symbols as leaves connected to an internal node of greatest depth

Proof. (exchange argument).

Let we have a tree T with two lowest frequent symbols not as deep as possible.
Then at least one has a smaller depth.
Switch it with one of the deepest nodes that is more frequent.
This improves the encoding length. Thus T is not optimal □



Proof of optimality (I)

Proof sketch

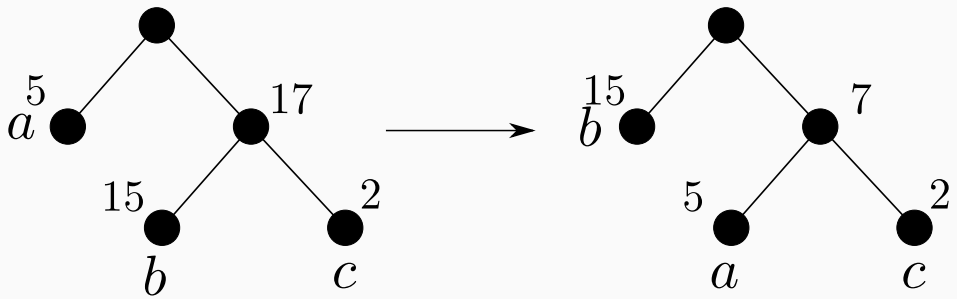
- Greedy choice property:

Claim

Every optimal solution has two lowest frequent symbols as leaves connected to an internal node of greatest depth

Proof. (exchange argument).

Let we have a tree T with two lowest frequent symbols not as deep as possible.
Then at least one has a smaller depth.
Switch it with one of the deepest nodes that is more frequent.
This improves the encoding length. Thus T is not optimal □

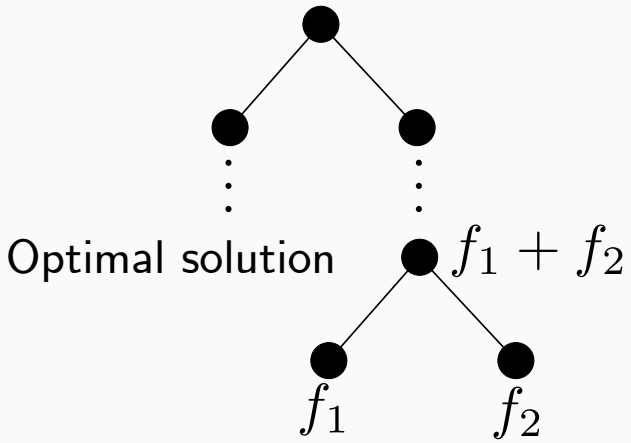


Proof of optimality (II)

- Optimal substructure Consider a global optimal solution for $(f_1, f_2, \ldots, f_n)$ with f_1, f_2 the least frequencies.
  Our subproblem is: find an optimal solution for $(f_1 + f_2, f_3, \ldots, f_n)$

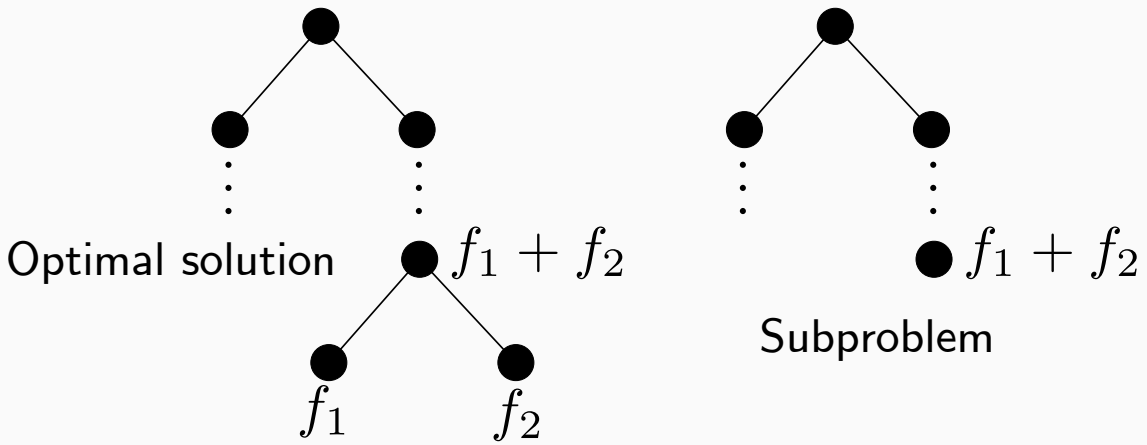
Proof of optimality (II)

- Optimal substructure Consider a global optimal solution for $(f_1, f_2, \ldots, f_n)$ with f_1, f_2 the least frequencies.
  Our subproblem is: find an optimal solution for $(f_1 + f_2, f_3, \ldots, f_n)$



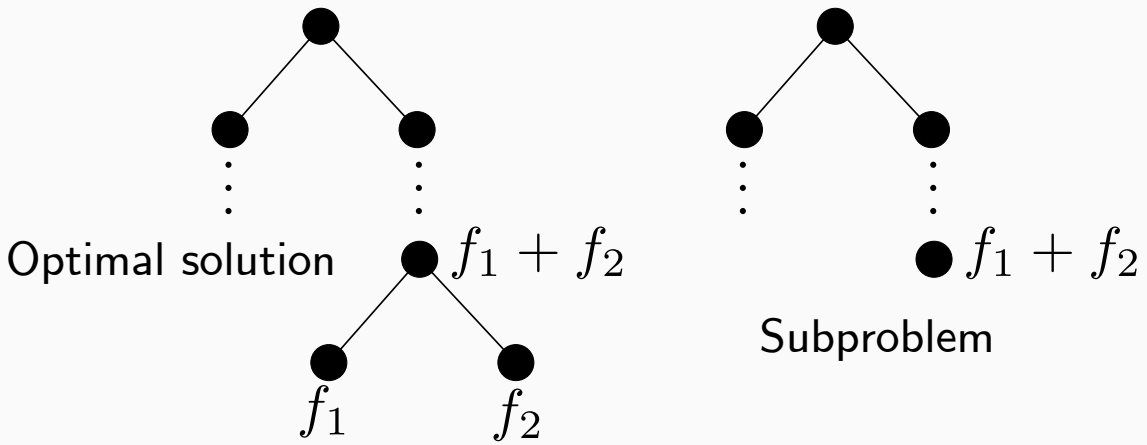
Proof of optimality (II)

- Optimal substructure Consider a global optimal solution for $(f_1, f_2, \ldots, f_n)$ with f_1, f_2 the least frequencies.
  Our subproblem is: find an optimal solution for $(f_1 + f_2, f_3, \ldots, f_n)$



Proof of optimality (II)

- Optimal substructure Consider a global optimal solution for $(f_1, f_2, \ldots, f_n)$ with f_1, f_2 the least frequencies.
  Our subproblem is: find an optimal solution for $(f_1 + f_2, f_3, \ldots, f_n)$



Thus, the greedy solution will lead to the global optimal solution

Pseudocode

```
1  def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
2      T: empty tree;
3      H: priority queue ordered by f;
4      for i := 1 in n:
5          insert(H, i);
6      for k := n + 1 in 2n − 1:
7          i := extract_min(H);
8          j := extract_min(H);
9          Creat a node k in T with children i and j;
10         f[k] := f[i] + f[j];
11         insert(H, k);
```

Binary heap: insert $O(\log n)$, extract_min: $O(\log n)$

Lines 4-5: $O(n \log n)$;

Lines 6-11: $O(n \log n)$

Total cost: $O(n \log n)$

More about the pseudocode

Question: why $2n - 1$ in line 6?

Answer: if a full binary tree has n leaves, then it has $2n - 1$ total nodes

More examples

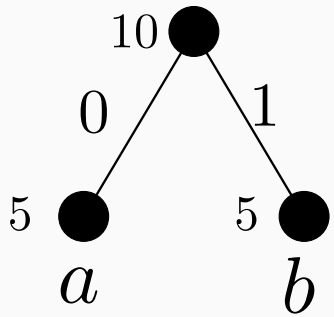
$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$

More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$

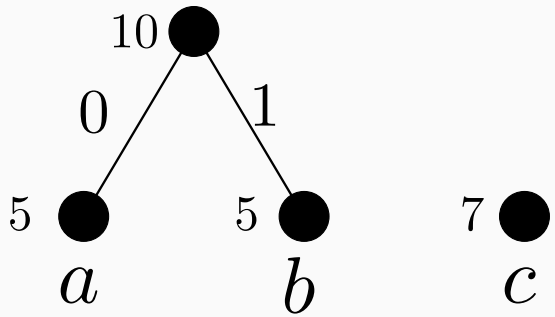
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



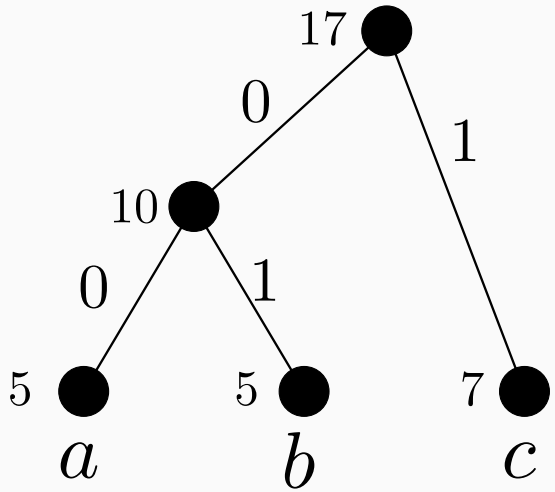
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



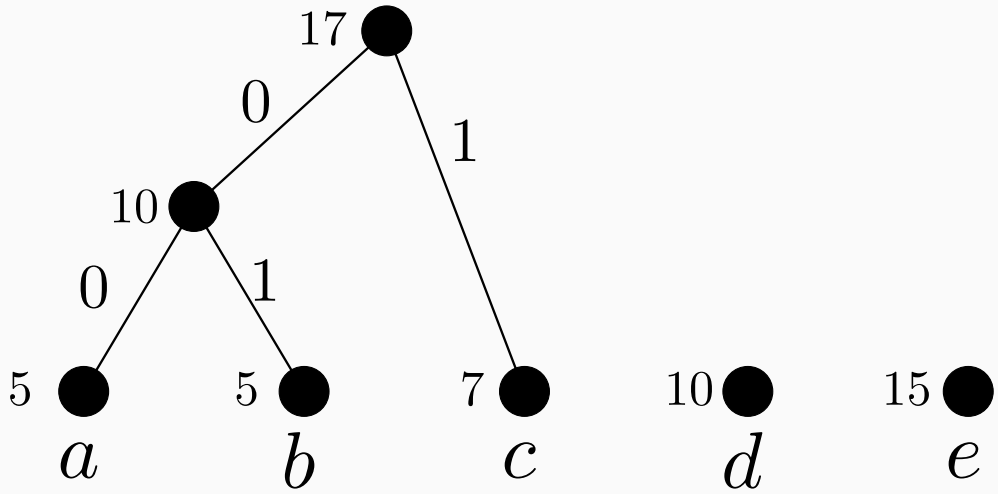
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



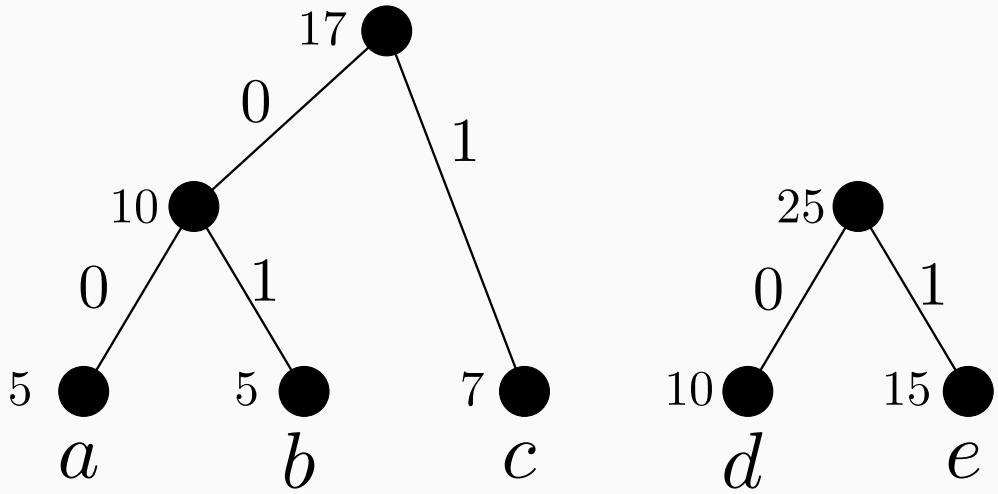
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



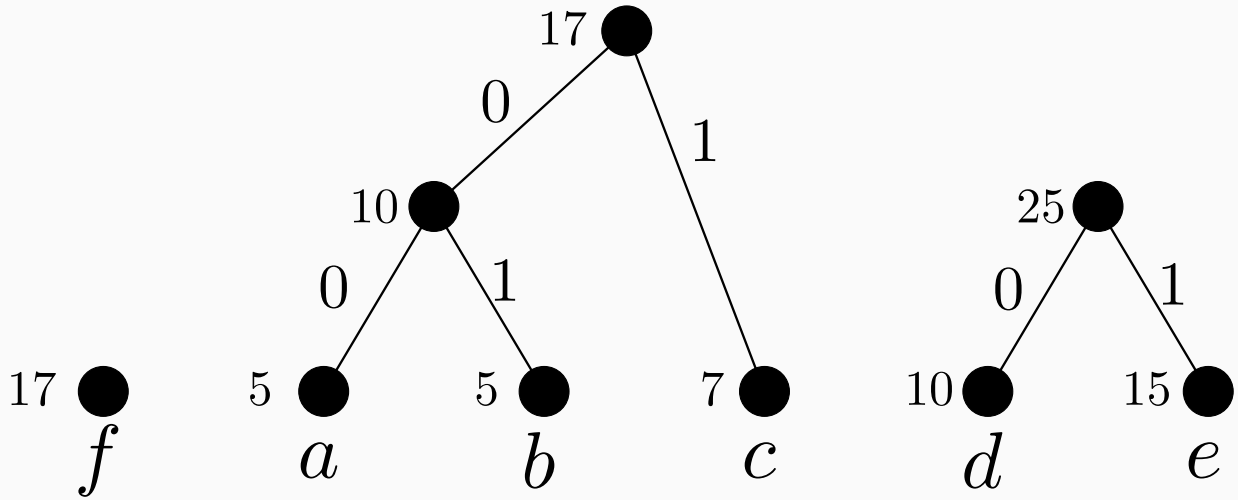
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



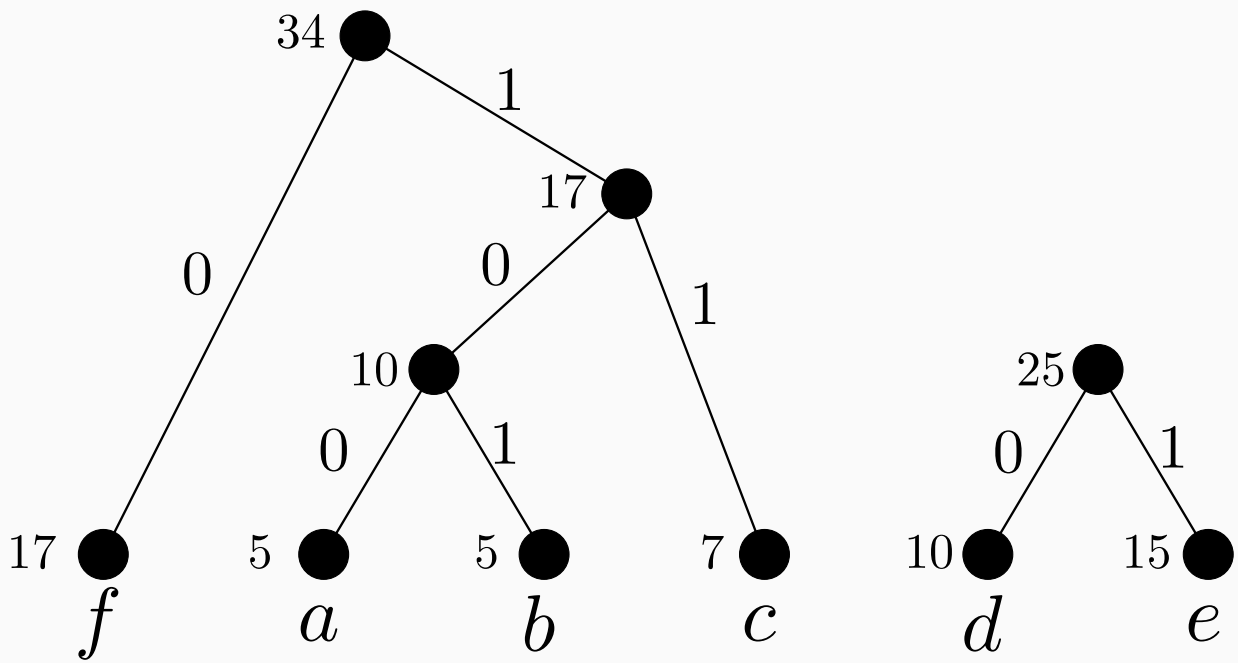
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



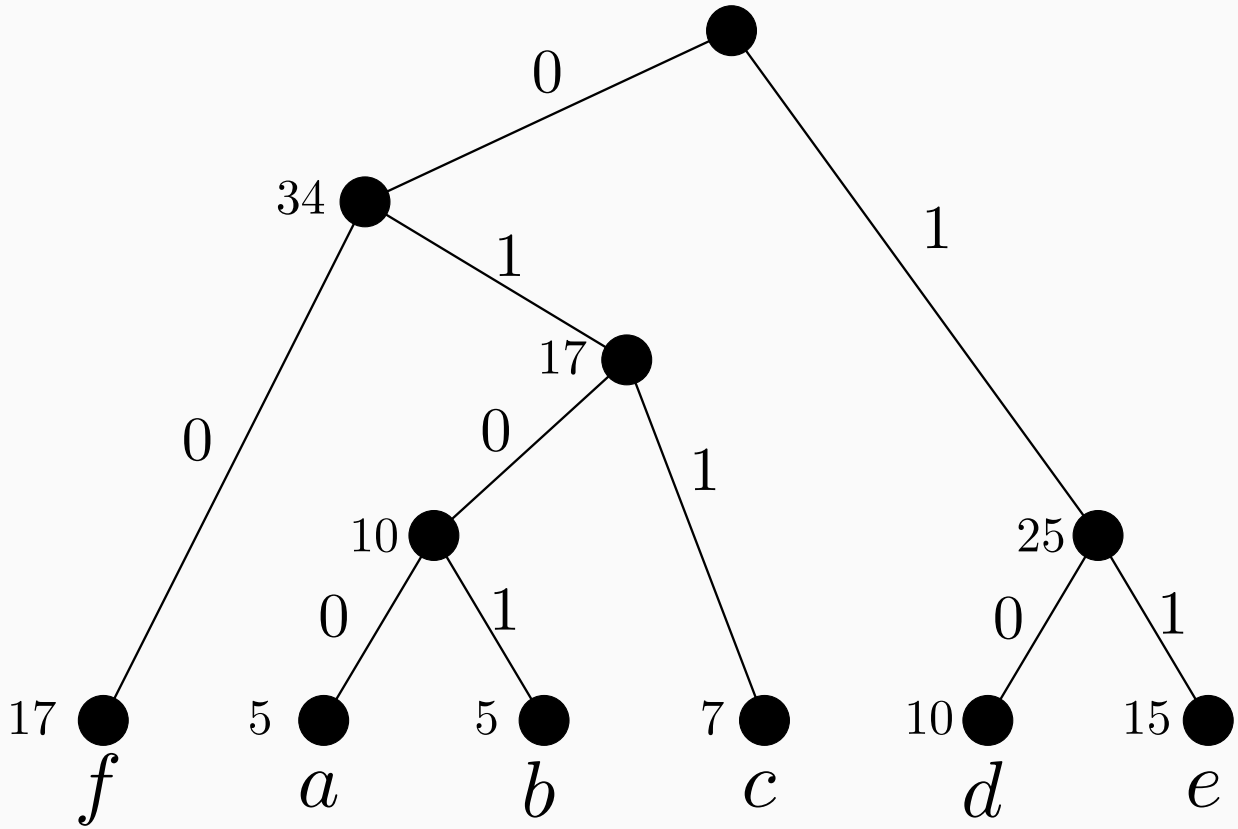
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



More examples

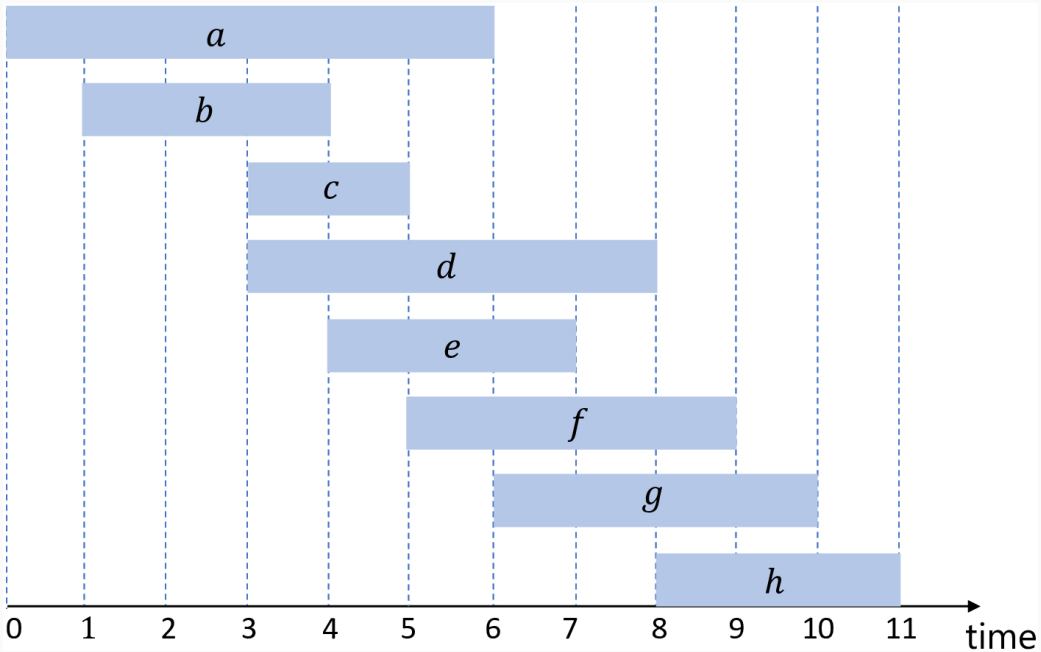
$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



Interval Scheduling Problem (Kleinberg, Tardos 4.1)

Interval Scheduling Problem

- Job j starts at time s_j and ends at time f_j.
- Two jobs i and j are compatible if they do not overlap i.e., either. $s_j > f_i$ (j starts after i ends) or $s_i > f_j$ (i starts after j ends)
- Objective: maximum subset of mutually compatible jobs.

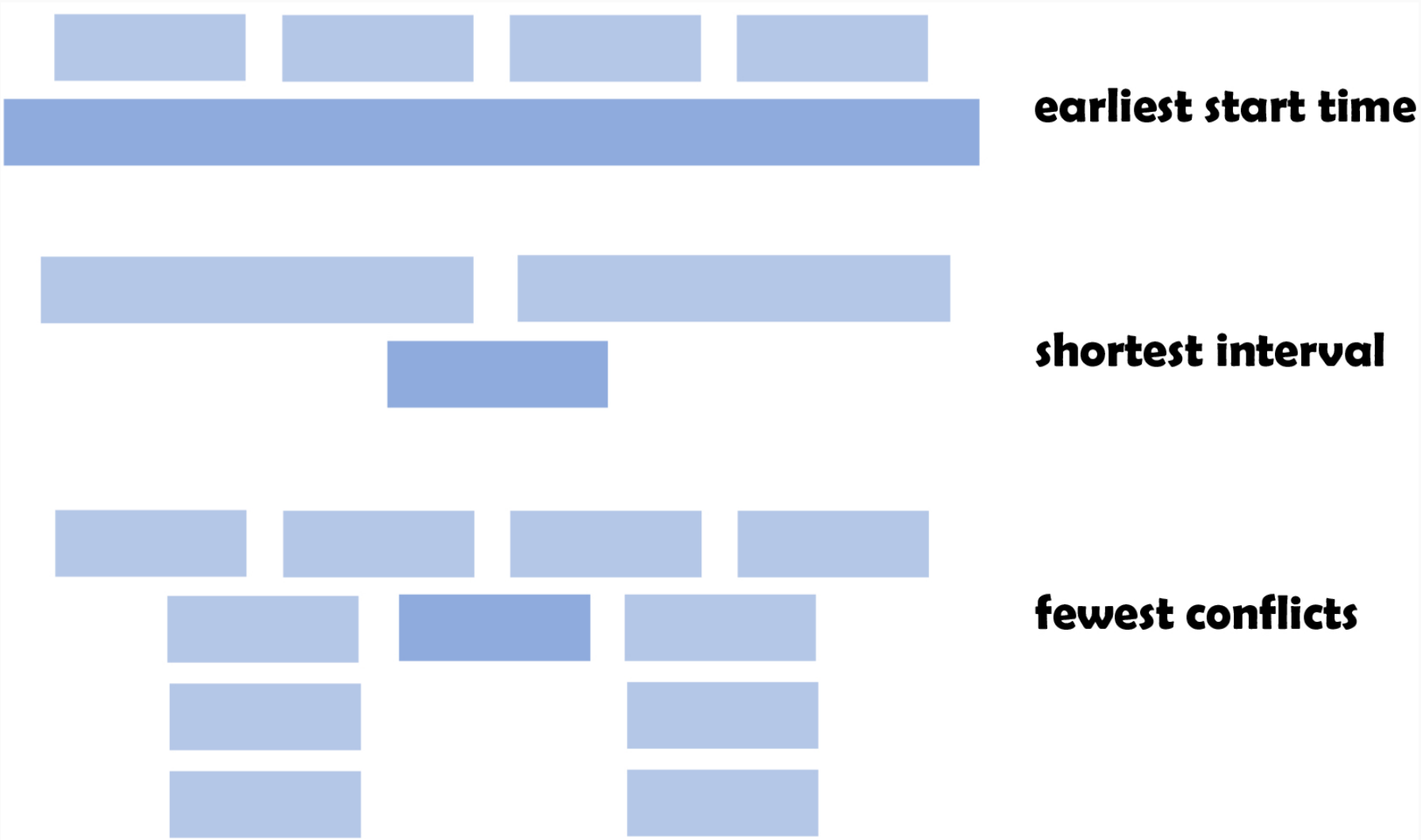


Different Greedy Strategies

Consider some ordering on the input jobs using a greedy strategy. For the next job in this order decide if it is compatible with all previously selected jobs and add it accordingly.

- Earliest Start Time: schedule ascending order of s_j.
- Earliest Finish Time: schedule ascending order of f_j.
- Shortest Interval: schedule ascending order of $(f_j - s_j)$.
- Fewest Conflicts: For each job j let c_j be the number of jobs that conflict with j. Schedule ascending order of c_j.

Counterexamples



Greedy algorithm

Earliest finish time: ascending order of f_j.

```
1 def GREEDY(s_1, ..., s_n, f_1, ..., f_n):
2     Sort jobs by finish times so that f_1 ≤ f_2 ≤ ··· ≤ f_n
3     A := ∅ ;                          // set of selected jobs so far
4     for j = 1 to n:
5         if job j compatible with A:
6             A := A ∪ {x};
7     return A;
```

Implementation: how to check whether job j is compatible with A?

- j^*: job last added to A (store it).
- Job j is compatible with A if $s_j > f_{j^*}$.

Implementation and Run Time Analysis

```
1 def GREEDY(s_1, ..., s_n, f_1, ..., f_n):
2     Sort jobs by finish times so that f_1 ≤ f_2 ≤ ··· ≤ f_n
3     A := ∅ ;                          // Queue of selected jobs so far
4     j* = 0;
5     for j = 1 to n:
6         if s_j > f_{j*}:
7             enqueque(j onto A);
8             j* = j
9     return A;
```

- Sorting at line 2 takes time $O(n \log n)$
- Initialization at line 3,4 takes time $O(1)$
- The for loop at line 5 repeats n times. Each iteration takes $O(1)$ time.

$O(n \log n)$ time

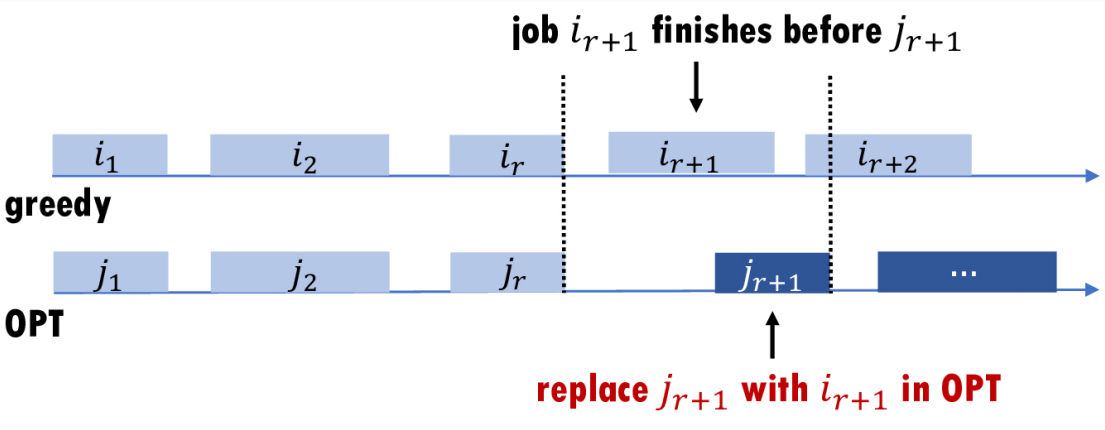
Correctness Analysis

Theorem

Greedy algorithm's solution is optimal.

Proof (by contradiction): Suppose greedy not optimal.

- Let $i_1, i_2, \ldots, i_k$ be the jobs selected by the greedy algorithm.
- Let $j_1, j_2, \ldots, j_\ell$ be the jobs selected by the optimal algorithm.
- Look for the largest r such that: $i_1 = j_1,\ i_2 = j_2, \ldots, i_r = j_r$.
- If $r < k$,



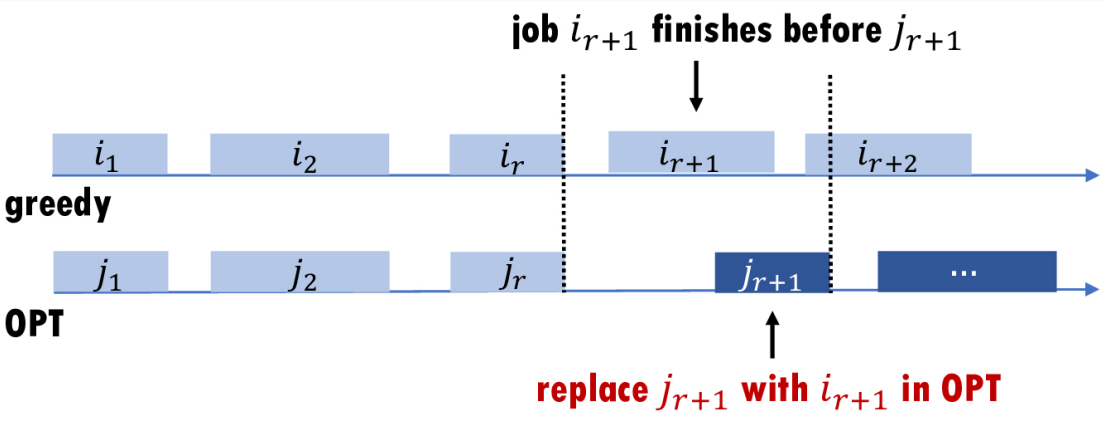
Correctness Analysis

Theorem

Greedy algorithm's solution is optimal.

Proof (by contradiction): Suppose greedy not optimal.

- Let $i_1, i_2, \ldots, i_k$ be the jobs selected by the greedy algorithm.
- Let $j_1, j_2, \ldots, j_\ell$ be the jobs selected by the optimal algorithm.
- Look for the largest r such that: $i_1 = j_1,\ i_2 = j_2, \ldots, i_r = j_r$.
- If $r < k$, we get a contradiction by replacing j_{r+1} with i_{r+1} as we get an optimal solution with larger r.



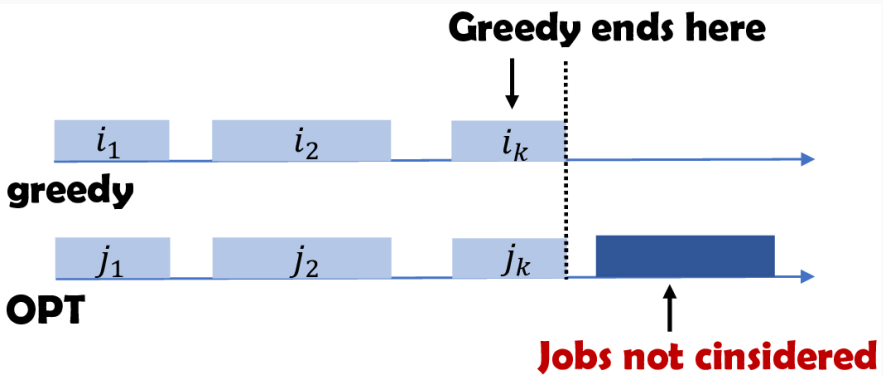
Correctness Analysis

Theorem

Greedy algorithm's solution is optimal.

Proof (by contradiction): Suppose greedy not optimal.

- Let $i_1, i_2, \ldots, i_k$ be the jobs selected by the greedy algorithm.
- Let $j_1, j_2, \ldots, j_\ell$ be the jobs selected by the optimal algorithm.
- Look for the largest r such that: $i_1 = j_1$, $i_2 = j_2, \ldots, i_r = j_r$.
- If $r < k$, we get a contradiction by replacing j_{r+1} with i_{r+1} as we get an optimal solution with larger r.
- If $r = k$ but $m > k$,



Correctness Analysis

Theorem

Greedy algorithm's solution is optimal.

Proof (by contradiction): Suppose greedy not optimal.

- Let $i_1, i_2, \ldots, i_k$ be the jobs selected by the greedy algorithm.
- Let $j_1, j_2, \ldots, j_\ell$ be the jobs selected by the optimal algorithm.
- Look for the largest r such that: $i_1 = j_1$, $i_2 = j_2, \ldots, i_r = j_r$.
- If $r < k$, we get a contradiction by replacing j_{r+1} with i_{r+1} as we get an optimal solution with larger r.
- If $r = k$ but $m > k$, we get a contradiction because greedy algorithm stopped before all jobs were considered.

Horn formulas (Textbook Section 5.3)

Horn formulas — Prelude

Consider the following puzzle

- If Alice has a dog, then Bob has a cat
- If Charlie and Bob both have pets of the same species, then Alice has a cat
- Charlie and Alice don't share a pet of the same species

Question: what pets do they have?

Boolean formulas

Basics of boolean formulas

- **Variables:** possibilities
  Knowledge about variables is represented by a special type of boolean formulas

  Goal; find a *consistent* explanation of the knowledge
- **Boolean variable:** $x = 1$ (true) or $x = 0$ (false)
- **Literal:** x (positive literal), $\bar{x}$ (negative literal)
- **Clause:** a clause consists of literals connected by $\wedge$ (AND), $\vee$ (OR), $\implies$ (implies)

  Examples: $x \wedge \bar{y}$, $(x \wedge y) \implies z$

Horn formulas

In a Horn formula, there are only two types of clauses (**Horn clauses**):

- **Implication:** $(x_1 \wedge x_2 \wedge \cdots \wedge x_n) \implies y$

  LHS: AND of any number of positive literals

  RHS: single positive literal
  - $(x \wedge \bar{y}) \implies z$ ✗
  - $(x \vee y) \implies z$ ✗
  - $\implies z$ ✓
- **Pure negative clauses** $\bar{x}_1 \vee \bar{x}_2 \vee \cdots \vee \bar{x}_n$

  OR of any number of negative literals

Horn formula example

Consider the puzzle:

- If Alice has a dog, then Bob has a cat
- If Charlie and Bob both have pets of the same species, then Alice has a cat
- Charlie and Alice don't share a pet of the same species

Define variables:

- a: Alice has a dog
- b: Bob has a dog
- c: Charlie has a dog
- x: Alice has a cat
- y: Bob has a cat
- z: Charlie has a cat

Modelled by a set of Horn clauses:

$a \implies y$

$(b \wedge c) \implies x$

$(y \wedge z) \implies x$

$\bar{a} \vee \bar{c}$

$\bar{x} \vee \bar{z}$

Question: satisfying assignment?

Greedy approach for Horn formulas

Problem (Horn Satisfiability)

Given a set of Horn clauses, determine whether or not there is a consistent explanation,

i.e., an assignment of 0/1 to variables that satisfy all clauses

Example: $(x \wedge y) \implies z$, $\bar{x} \vee \bar{w}$

can be satisfied by
$x = 0, y = 0, z = 0, w = 0$

Greedy heuristic: start with all 0. Only set a variable to 1 if you need to, i.e., when an implication says you need to

Recall: $p \implies q \iff \bar{p} \vee q$

Pseudocode

def GREEDY_HORN*(set of Horn clauses)*:

- Set all variables to 0;
- **while** *there exists an "* $\Longrightarrow$ *" that is not satisfied*:
  - Set its RHS to 1;
- **if** *all pure negative clauses are 1*:
  - **return** the assignment;
- **else:**
  - **return** "unsatisfiable";

Example: $\Longrightarrow x$, $x \Longrightarrow y$, $(\bar{x} \vee \bar{y})$

x	y	
0	0	$\Longrightarrow x$ ✗
1	0	$x \Longrightarrow y$ ✗
1	1	$\Longrightarrow x$ ✓, $x \Longrightarrow y$ ✓, $(\bar{x} \vee \bar{y})$ ✗

Unsatisfiable

Correctness and running time

Correctness: If GREEDY_HORN finds an assignment, then the problem has a satisfying assignment

If it returns "unsatisfiable", is it really unsatisfiable?

Theorem

The variables set to 1 by GREEDY_HORN *must be 1 in* ***any*** *satisfying assignment*

Exercise: Prove this by induction

How does this theorem help?

If all the pure negative clauses cannot be satisfied after the while loop, then there's no such assignment satisfying them

Running time: Let n be the size of the Horn formula, i.e., the number of occurrences of literals.

Total running time: $O(n^2)$.

Can be improved to $O(n)$ (**exercise**)

Set Cover (Textbook Section 5.4)

The set cover problem

Problem (Set Cover)

Input:

- *a set B*
- *subsets $S_1, \ldots, S_m \subseteq B$*

Output: *a collection of subsets $S_{i_1}, \ldots, S_{i_k}$ s.t. $\bigcup_{j=1}^{k} S_{i_j} = B$*

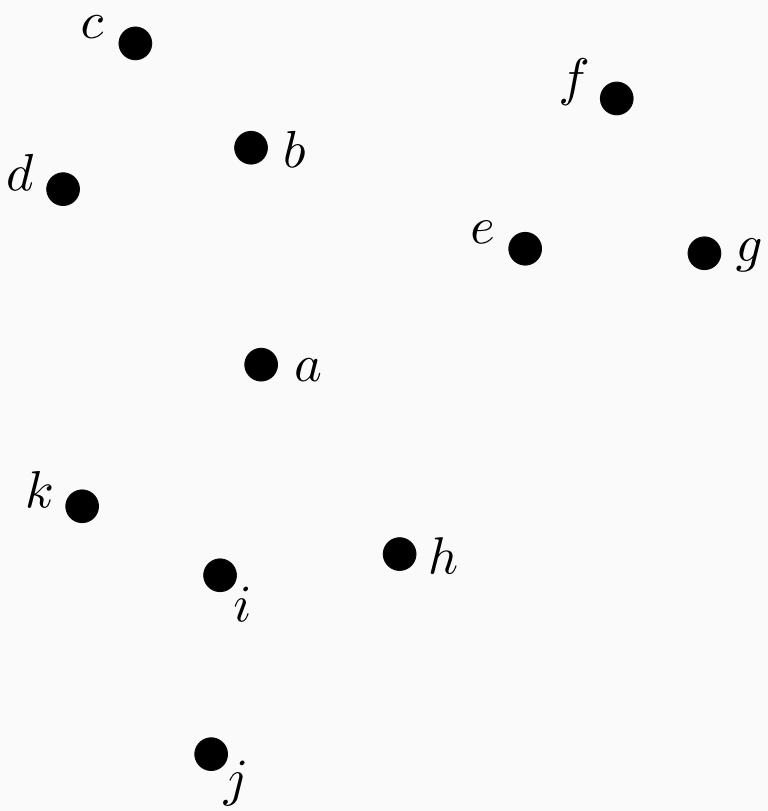
Goal: *minimize the number of selected subsets*

Set cover: example

Example: Each post office can serve 30 miles. Where to build post offices in centre county?

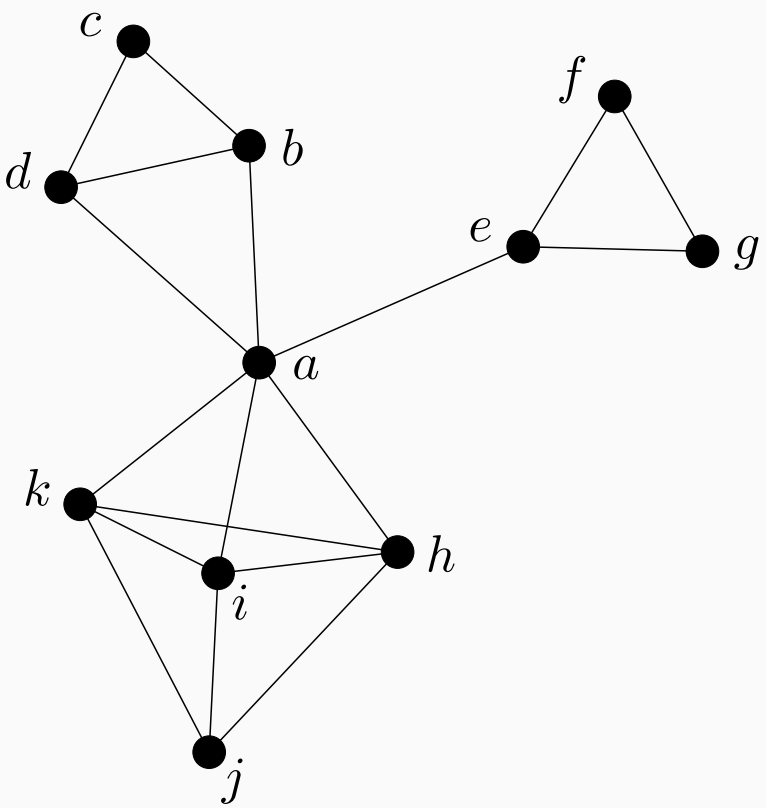
Set cover: example

Example: Each post office can serve 30 miles. Where to build post offices in centre county?



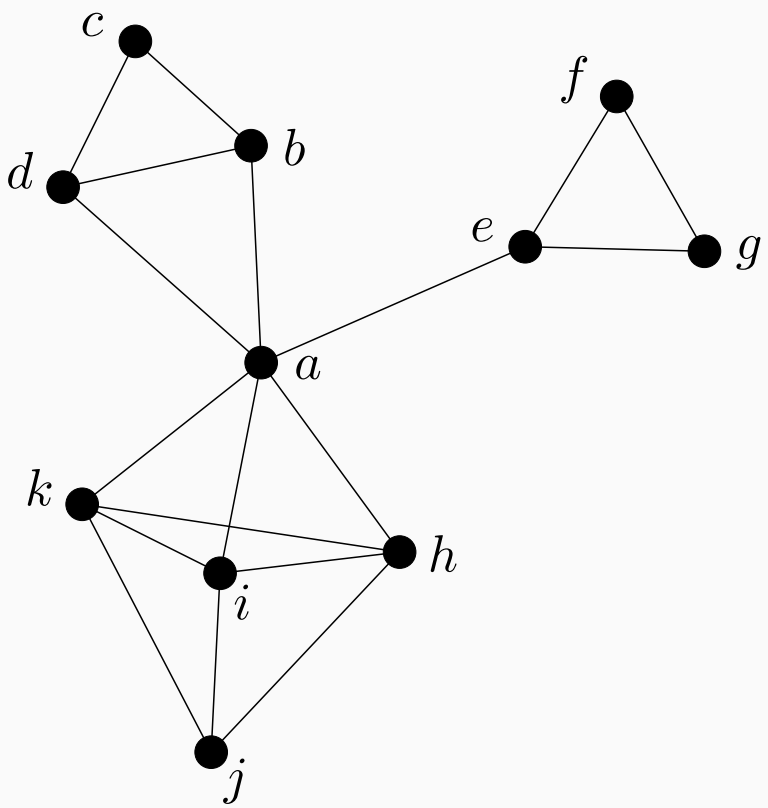
Set cover: example

Example: Each post office can serve 30 miles. Where to build post offices in centre county?



Set cover: example

Example: Each post office can serve 30 miles. Where to build post offices in centre county?



Draw an edge if two towns are within 30 miles

$$B = \{a, b, \ldots, k\}$$

$$S_a = \{a, b, d, e, h, i, k\}$$

$$S_b = \{b, c, a, d\}$$

$$\vdots$$

$$S_k = \{k, a, h, i, j\}$$

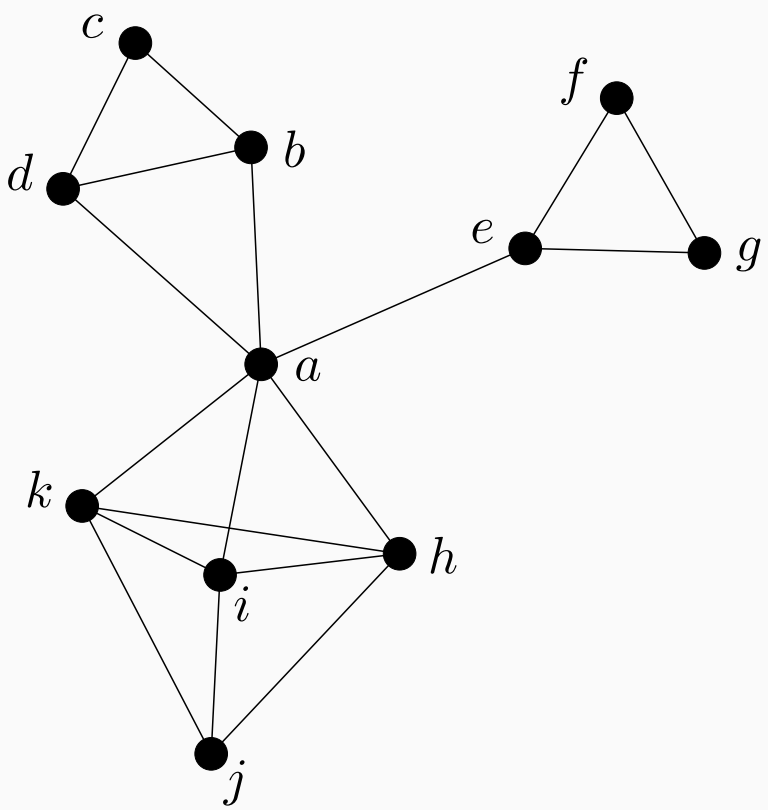
S_x: the towns within 30 miles of x

Set cover: greedy heuristic

Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered

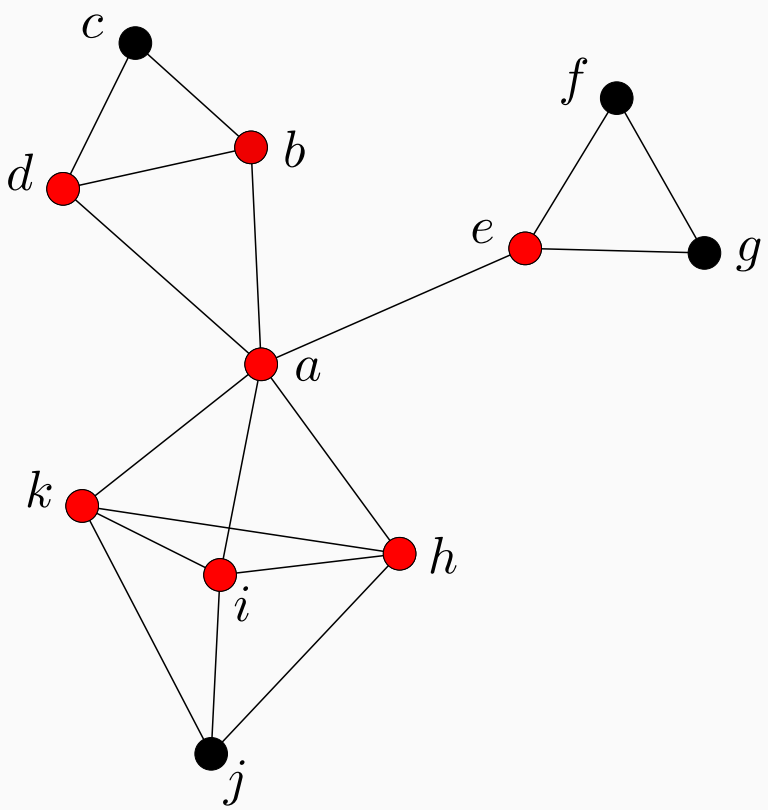
Set cover: greedy heuristic

Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



Set cover: greedy heuristic

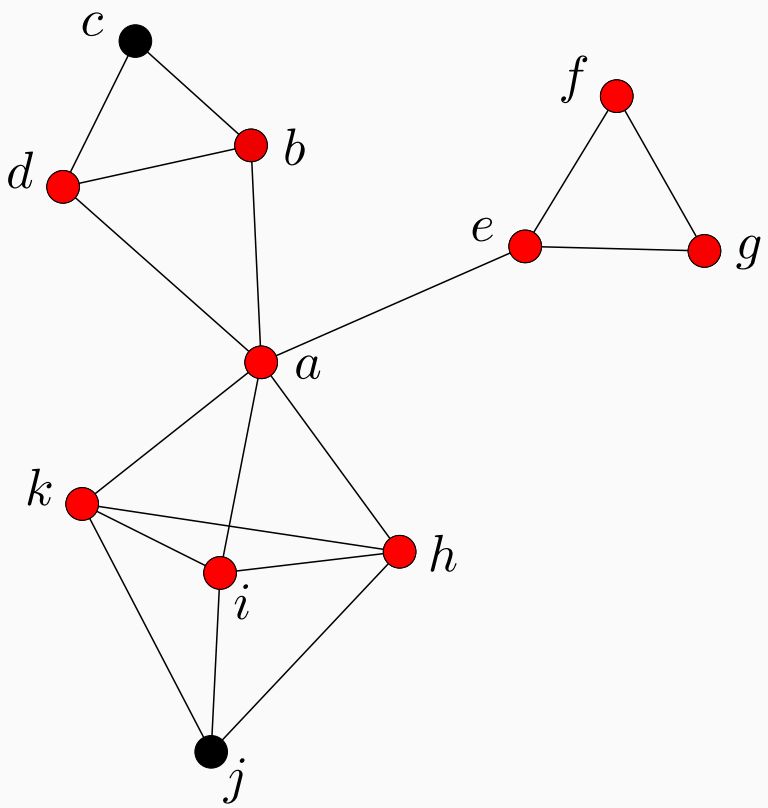
Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



$S_a = \{a, b, d, e, h, i, k\}$

Set cover: greedy heuristic

Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered

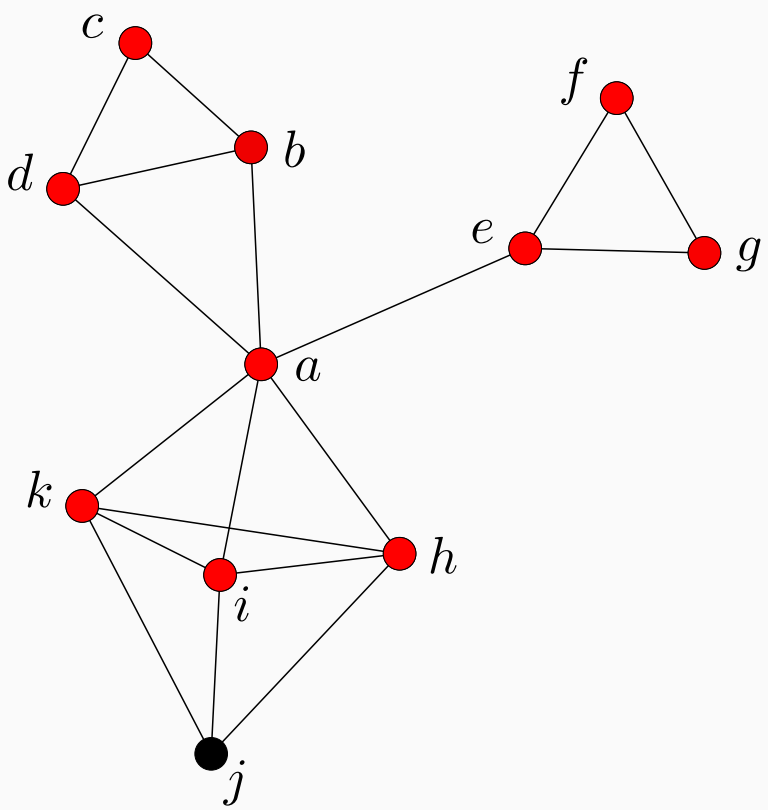


$S_a = \{a, b, d, e, h, i, k\}$

$S_f = \{f, g, e\}$

Set cover: greedy heuristic

Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



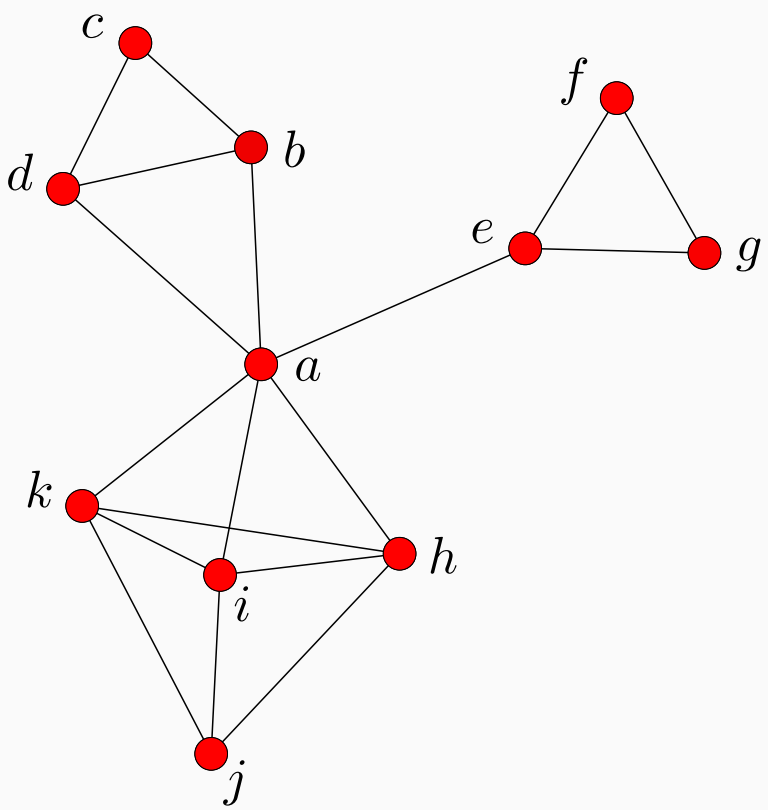
$$S_a = \{a, b, d, e, h, i, k\}$$

$$S_f = \{f, g, e\}$$

$$S_c = \{c, b, d\}$$

Set cover: greedy heuristic

Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



$$S_a = \{a, b, d, e, h, i, k\}$$

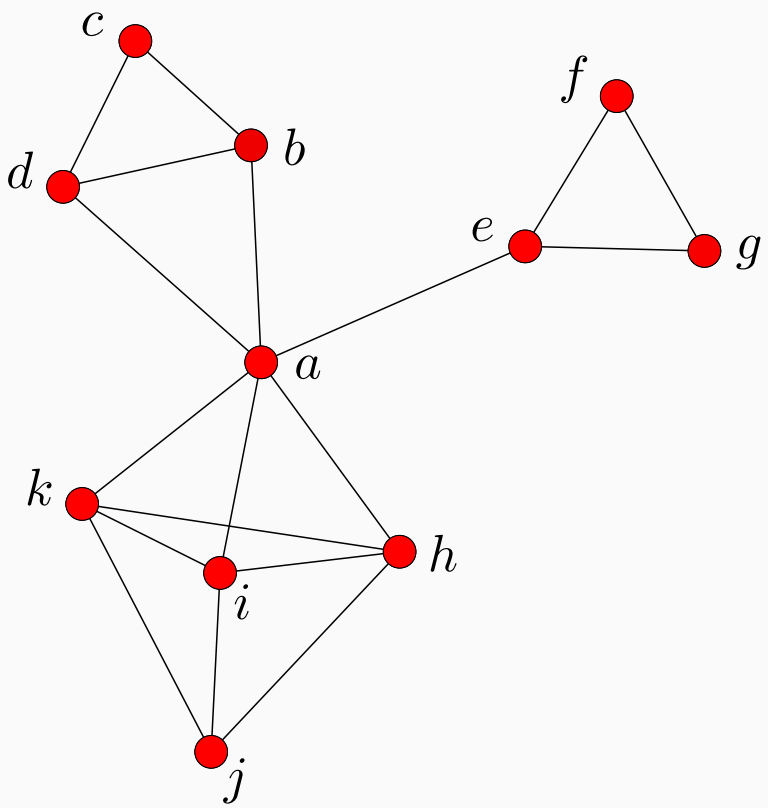
$$S_f = \{f, g, e\}$$

$$S_c = \{c, b, d\}$$

$$S_j = \{i, k, j, h\}$$

Set cover: greedy heuristic

Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



$$S_a = \{a, b, d, e, h, i, k\}$$

$$S_f = \{f, g, e\}$$

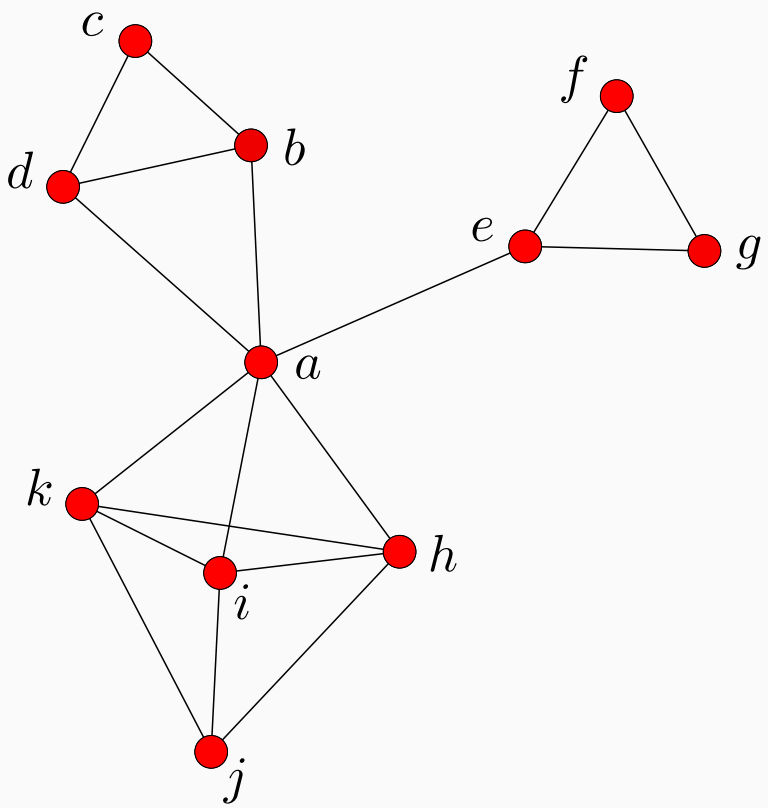
$$S_c = \{c, b, d\}$$

$$S_j = \{i, k, j, h\}$$

Is this optimal?

Set cover: greedy heuristic

Greedy heuristic: choose the next subset with the most number of uncovered items, until B gets covered



$$S_a = \{a, b, d, e, h, i, k\}$$

$$S_f = \{f, g, e\}$$

$$S_c = \{c, b, d\}$$

$$S_j = \{i, k, j, h\}$$

Is this optimal?

Optimal solution: S_b, S_e, S_i

Greedy solution is not too bad

Although the greedy solution is not optimal, but it's not off by much

Theorem

Assume $|B| = n$ and the optimal solution uses k subsets.

Then the greedy algorithm uses at most $k \ln(n)$ subsets

$\ln(n)$: *approximation ratio*

More about **approximation algorithms**: CSE 565

Proof: Let n_t be the number of elements not covered by the greedy algorithm after t iterations.

These remaining n_t elements are covered by the optimal k subsets.

So some subsets has $\geq \frac{n_t}{k}$ of these uncovered elements,

and the greedy algorithm will pick a set of size at least $\frac{n_t}{k}$.

So, $n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right)$

Repeatedly applying this:

$$n_t \leq n_{t-1}\left(1-\frac{1}{k}\right) \leq n_{t-2}\left(1-\frac{1}{k}\right)^2 \leq \cdots \leq n_0\left(1-\frac{1}{k}\right)^t = n\left(1-\frac{1}{k}\right)^t$$

Using the fact: $1 - x \leq e^{-x}$ (equality when $x = 0$)

$$n_t \leq n\left(1-\frac{1}{k}\right)^t \leq ne^{-t/k}$$

Greedy algorithm terminates when $n_t < 1$. Let's find out what t makes $n_t < 1$

Since $n_t < ne^{-t/k}$, it suffices to make $ne^{-t/k} \leq 1$

Solving $ne^{-t/k} \leq 1$

$$\iff e^{-t/k} \leq \frac{1}{n} \iff -\frac{t}{k} \leq \ln\left(\frac{1}{n}\right) \iff t \geq -k\ln\left(\frac{1}{n}\right) = k\ln(n)$$

At $t = k\ln(n)$, $n_t < 1$. Everything is covered $\square$

Proof of the fact $1 - x \leq e^{-x}$ (equality when $x = 0$):

Consider $f(x) = e^{-x} - (1 - x) \geq 0$

$f'(x) = -e^{-x} + 1$. Critical point at $x = 0$, achieving minimum $\square$