Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

$$\{a, b, c\} \quad \text{head} \to a \to b \to c$$

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

$$\{a, b, c\} \quad \text{head} \to a \to b \to c \qquad \text{find\_set}(b):$$

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

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

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

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

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

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

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

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

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

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

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

$\{d, e\}$ $\quad \text{head} \to d \to e$

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

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

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

$$\{d, e\} \quad \text{head} \to d \to e$$

$$\text{union}(a, b)$$

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

$\{a, b, c\}$ $\quad \text{head} \to a \to b \to c$ $\qquad$ find_set(b): $O(1)$

make_set(v): $O(1)$

$\{d, e\}$ $\quad \text{head} \to d \to e$

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

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

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

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

$\{d, e\}$ $\quad \text{head} \to d \to e$

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

Cost of union:

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

$\{a, b, c\}$ $\text{head} \rightarrow a \rightarrow b \rightarrow c$ find_set(b): $O(1)$

make_set(v): $O(1)$

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

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

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

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

Using linked list:

$\{a, b, c\}$ head $\to a \to b \to c$

find_set(b): $O(1)$

make_set(v): $O(1)$

$\{d, e\}$ head $\to d \to e$

union(a, b) head $\to a \to b \to c \to d \to e$

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

$\xrightarrow{\text{union}}$

Running time of Kruskal's algorithm (I)

Depends on how we implement make_set, find_set, and union

head → d → e → a → b → c

Using linked list:

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

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

$\{d, e\}$ $\quad \text{head} \to d \to e$

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

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

$\xrightarrow{\text{union}}$

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

Running time of Kruskal's algorithm (II)

```
def KRUSKAL_MST(undirected G = (V, E), weights w = (w_e)_{e∈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);
```

Running time of Kruskal's algorithm (II)

```
def KRUSKAL_MST(undirected G = (V, E), weights w = (w_e)_{e∈E}):
    Set A := { };
    for v ∈ V:
        make_set(v) ;                                   // O(|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);
```

$|E| = O(|V|^2)$

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

worst case $O(|E| \cdot |V|)$

$O(|V|)$ worst-case

Running time of Kruskal's algorithm (II)

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

Running time of Kruskal's algorithm (II)

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

Worst-case cost for union:

Running time of Kruskal's algorithm (II)

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

  Set $A := \{\}$;

  for $v \in V$**:**

    make_set(v) ; // $O(|V|)$

  Sort E in increasing order of edge weights ; // $O(|E| \log |V|)$

  for $(u, v) \in E$**:**

    if $\text{find\_set}(u) \neq \text{find\_set}(v)$**:**

      $A := A \cup \{(u, v)\}$;

      $\text{union}(u, v)$;

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

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?

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.

Running time of Kruskal's algorithm (II)

1 **def** KRUSKAL_MST(*undirected* $G = (V, E)$*, weights* $w = (w_e)_{e \in E}$)**:**
2   Set $A := \{\}$;
3   **for** $v \in V$**:**
4   make_set(v) ;   // $O(|V|)$
5   Sort E in increasing order of edge weights ;   // $O(|E| \log |V|)$
6   **for** $(u, v) \in E$**:**
7   **if** $\mathrm{find\_set}(u) \neq \mathrm{find\_set}(v)$**:**
8             $A := A \cup \{(u, v)\}$;
9             $\mathrm{union}(u, v)$;

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

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:

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

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

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

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:

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:

$$\{a\} \quad a$$

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:

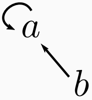
$$\{a\} \quad \circlearrowleft a \qquad \{a, b\}$$

Alternative data structure

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

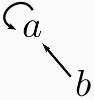
Directed tree disjoint set:

$\{a\}$ 

$\{a, b\}$ 

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

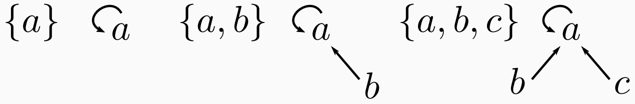
Directed tree disjoint set:

$\{a\}$  $\{a, b\}$  $\{a, b, c\}$

Alternative data structure

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

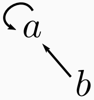
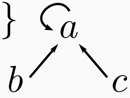
Directed tree disjoint set:



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:

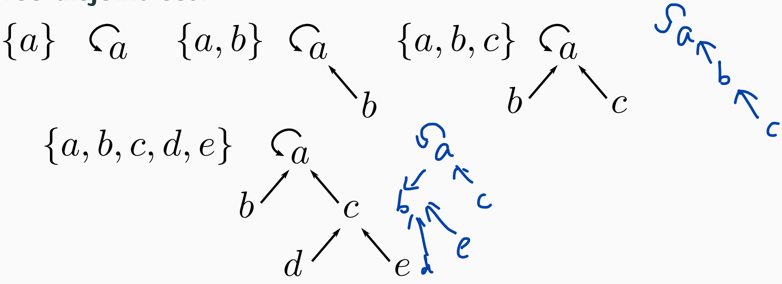
$\{a\}$  $\{a, b\}$  $\{a, b, c\}$ 

$\{a, b, c, d, e\}$

Alternative data structure

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

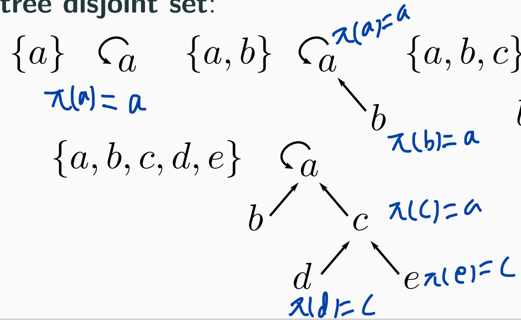
Directed tree disjoint set:



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

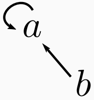
Alternative data structure

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

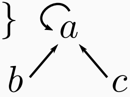
Directed tree disjoint set:

$\{a\}$

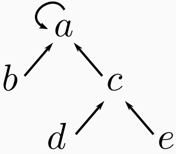
$\{a, b\}$



$\{a, b, c\}$



$\{a, b, c, d, e\}$



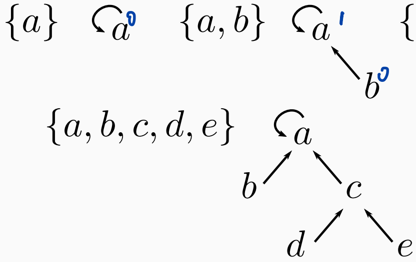
Definition

$\pi(x)$: parent of x

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

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. $\pi(x) = x$

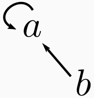
$\mathrm{rank}(x)$: number of the edges in the longest simple path from x to a leaf

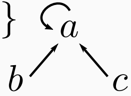
Alternative data structure

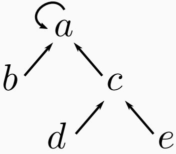
The linked-list implementation is good enough, but there exist better data structures to improve the worst-case cost for union

Directed tree disjoint set:

$\{a\}$ 

$\{a, b\}$ 

$\{a, b, c\}$ 

$\{a, b, c, d, e\}$ 

Definition

$\pi(x)$: parent of x

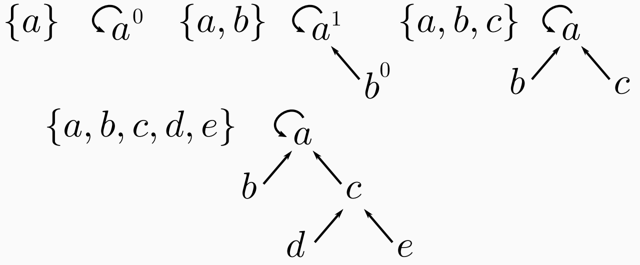
root node: x s.t. $\pi(x) = x$

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

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

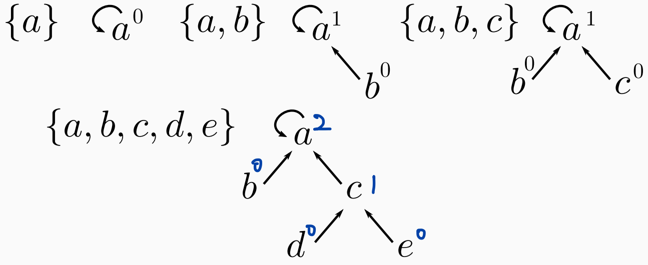
root node: x s.t. $\pi(x) = x$

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

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

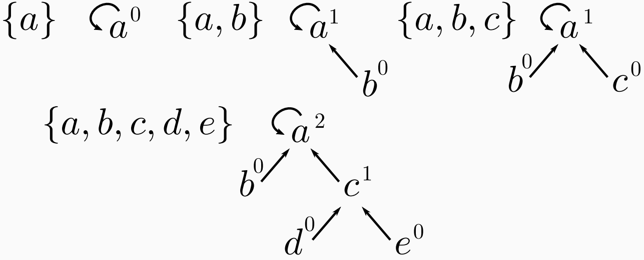
root node: x s.t. $\pi(x) = x$

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

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)

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

Operations of direct tree disjoint set (I)

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

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

$\quad \pi(v) := v;$

$\quad \mathrm{rank}(v) = 0;$

v 0

Operations of direct tree disjoint set (I)

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

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

  $\pi(v) := v;$

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

  Cost: $O(1)$

Operations of direct tree disjoint set (I)

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

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

  $\pi(v) := v;$

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

  Cost: $O(1)$

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

Operations of direct tree disjoint set (I)

- make_set(v)

  def make_set(v)**:**

  $\pi(v) := v$;

  rank(v) $= 0$;

  Cost: $O(1)$

- find_set(v)

  def find_set(v)**:**

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

  $v := \pi(v)$;

  return v;

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: $O(\text{depth of the node in the tree})$

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: $O(\text{depth of the node in the tree})$

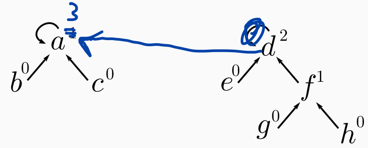
- what about union?

Operations of direct tree disjoint set (II)

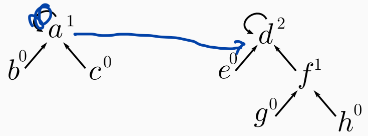
- union:

Operations of direct tree disjoint set (II)

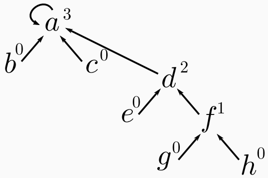
- union:



- union:

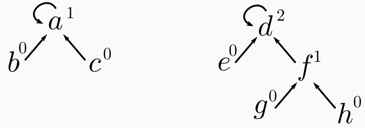


Option 1

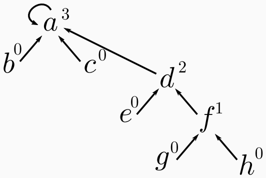


Operations of direct tree disjoint set (II)

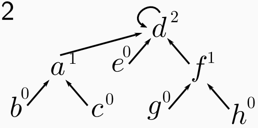
- union:



Option 1



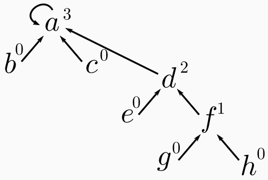
Option 2



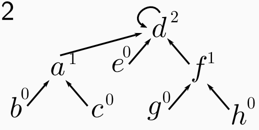
Operations of direct tree disjoint set (II)

- union:

Option 1



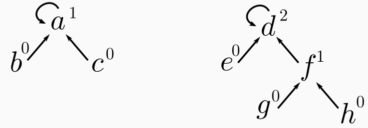
Option 2



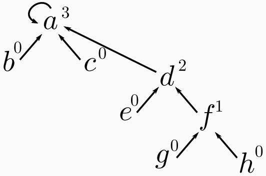
better!

Operations of direct tree disjoint set (II)

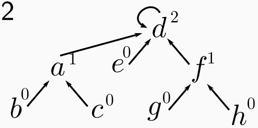
- union:



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)**:**

Operations of direct tree disjoint set (II)

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

    $r_x :=$ find_set(x), $r_y :=$ find_set(y);

Operations of direct tree disjoint set (II)

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

  $r_x := \text{find\_set}(x)$, $r_y := \text{find\_set}(y)$;

  if $\text{rank}(r_x) > \text{rank}(r_y)$**:**

Operations of direct tree disjoint set (II)

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

  $r_x := \mathrm{find\_set}(x)$, $r_y := \mathrm{find\_set}(y)$;

  if $\mathrm{rank}(r_x) > \mathrm{rank}(r_y)$**:**

    $\pi(r_y) := r_x$;

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:

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:

$\pi(r_x) := r_y$;

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):
```

Operations of direct tree disjoint set (II)

def $\text{union}(x, y)$**:**

$\quad r_x := \text{find\_set}(x)$, $r_y := \text{find\_set}(y)$;

$\quad$ **if** $\text{rank}(r_x) > \text{rank}(r_y)$**:**

$\quad\quad \pi(r_y) := r_x$;

$\quad$ **else:**

$\quad\quad \pi(r_x) := r_y$;

$\quad\quad$ **if** $\text{rank}(r_x) == \text{rank}(r_y)$**:**

$\quad\quad\quad \text{rank}(r_y) := \text{rank}(r_y) + 1$;

Operations of direct tree disjoint set (II)

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

  $r_x :=$ find_set(x), $r_y :=$ find_set(y);

  if $\text{rank}(r_x) > \text{rank}(r_y)$**:**

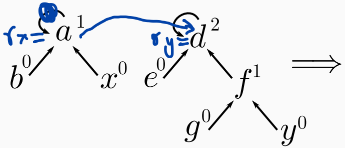
    $\pi(r_y) := r_x$;

  else:

    $\pi(r_x) := r_y$;

    if $\text{rank}(r_x) == \text{rank}(r_y)$**:**

      $\text{rank}(r_y) := \text{rank}(r_y) + 1$;



Operations of direct tree disjoint set (II)

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

  $r_x := \text{find\_set}(x)$, $r_y := \text{find\_set}(y)$;

  if $\text{rank}(r_x) > \text{rank}(r_y)$**:**

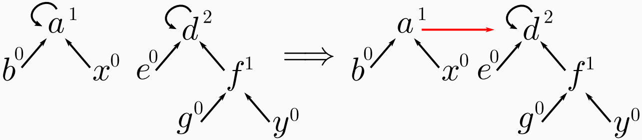
    $\pi(r_y) := r_x$;

  else:

    $\pi(r_x) := r_y$;

    if $\text{rank}(r_x) == \text{rank}(r_y)$**:**

      $\text{rank}(r_y) := \text{rank}(r_y) + 1$;



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)**:**

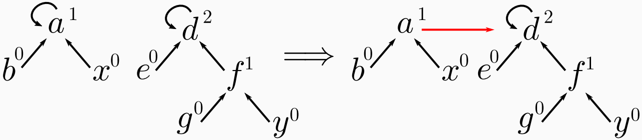
    $\pi(r_y) := r_x$;

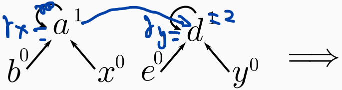
  else:

    $\pi(r_x) := r_y$;

    if rank(r_x) $==$ rank(r_y)**:**

      rank(r_y) $:=$ rank(r_y) $+ 1$;





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)**:**

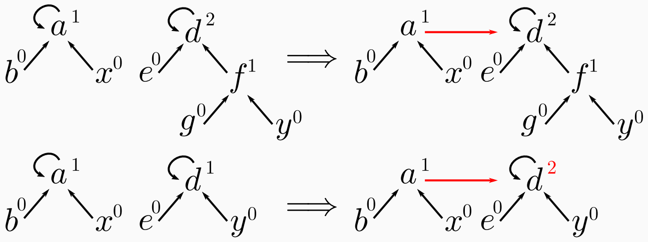
  $\pi(r_y) := r_x$;

else:

  $\pi(r_x) := r_y$;

  if rank(r_x) $==$ rank(r_y)**:**

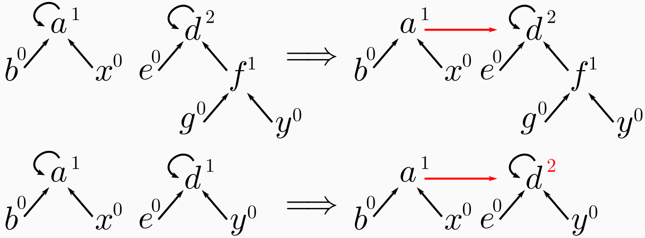
    rank(r_y) $:=$ rank(r_y) $+ 1$;



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

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

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 $2^0 = 1$.

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 $k - 1$.

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

log |V| |V|

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 $\geq 2^{k-1} + 2^{k-1} = 2^k$ $\square$

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

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 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 $\geq 2^{k-1} + 2^{k-1} = 2^k$ $\square$

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

- the cost of find_set: $O(\log |V|)$

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 $\geq 2^{k-1} + 2^{k-1} = 2^k$ $\square$

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

- the cost of find_set: $O(\log |V|)$
- the cost of union:

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 $\geq 2^{k-1} + 2^{k-1} = 2^k$ $\square$

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

- the cost of find_set: $O(\log |V|)$
- the cost of union: $O(\log |V|)$

Total running time of Kruskal using directed tree disjoint set

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

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

Total running time of Kruskal using directed tree disjoint set

1 **def** $\textsc{Kruskal\_MST}$(*undirected* $G = (V, E)$, *weights* $w = (w_e)_{e \in E}$):
2   Set $A := \{\}$;
3   **for** $v \in V$:
4         $\mathrm{make\_set}(\mathrm{v})$;   // $O(|V|)$
5   Sort E in increasing order of edge weights ;   // $O(|E| \log |V|)$
6   **for** $(u, v) \in E$:
7   **if** $\mathrm{find\_set}(u) \neq \mathrm{find\_set}(v)$:
8             $A := A \cup \{(u, v)\}$;
9             $\mathrm{union}(u, v)$;

Lines 6-9:

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 running time of Kruskal using directed tree disjoint set

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

2   Set $A := \{\}$;

3   **for** $v \in V$:

4   make_set(v) ; // $O(|V|)$

5   Sort E in increasing order of edge weights ; // $O(|E| \log |V|)$

6   **for** $(u, v) \in E$:

7   **if** $\text{find\_set}(u) \neq \text{find\_set}(v)$:

8             $A := A \cup \{(u, v)\}$;

9             $\text{union}(u, v)$;

Lines 6-9: $O(|E| \log |V|)$

Total cost:

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);
```

	linked list	directed Tree Disjoint set
make_set	$O(1)$	$O(1)$
find_set	$O(1)$	$O(\log\|V\|)$
union	$O(\|V\|)$	$O(\log\|V\|)$

Lines 6-9: $O(|E| \log |V|)$

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

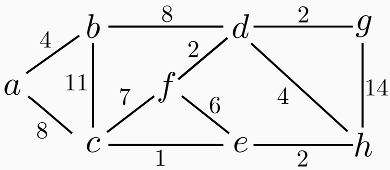
Prim's algorithm

Intuition: iteratively grows the tree

Prim's algorithm

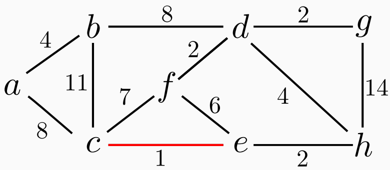
Intuition: iteratively grows the tree

Kruskal



Intuition: iteratively grows the tree

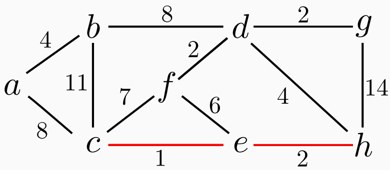
Kruskal



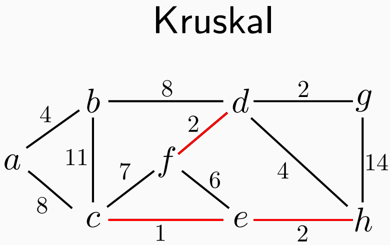
Prim's algorithm

Intuition: iteratively grows the tree

Kruskal



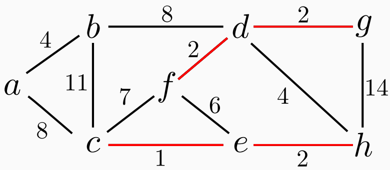
Intuition: iteratively grows the tree



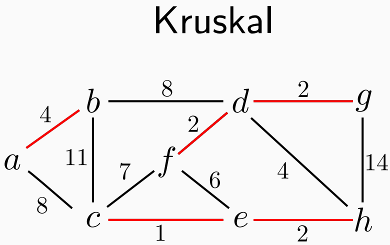
Prim's algorithm

Intuition: iterately grows the tree

Kruskal



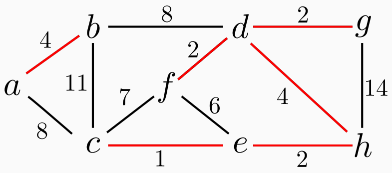
Intuition: iteratively grows the tree



Prim's algorithm

Intuition: iteratively grows the tree

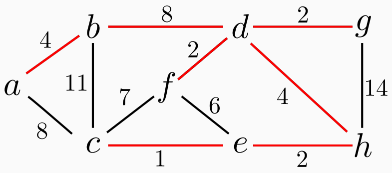
Kruskal



Prim's algorithm

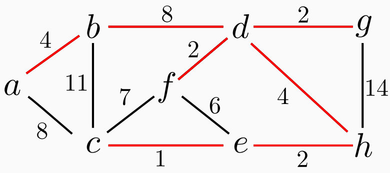
Intuition: iteratively grows the tree

Kruskal



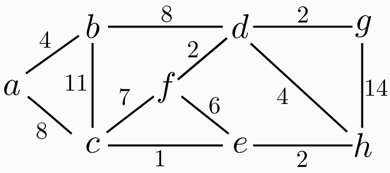
Intuition: iteratively grows the tree

Kruskal



Prim

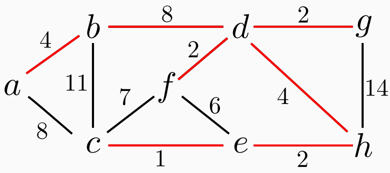
starting with c



Prim's algorithm

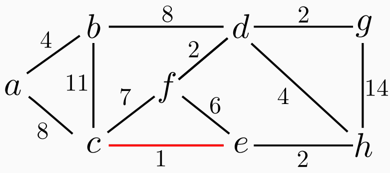
Intuition: iteratively grows the tree

Kruskal



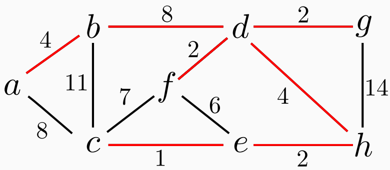
Prim

starting with c



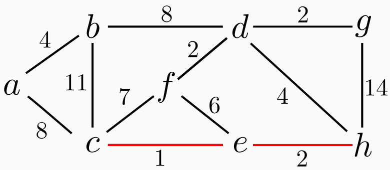
Intuition: iteratively grows the tree

Kruskal



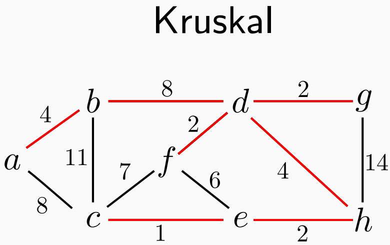
Prim

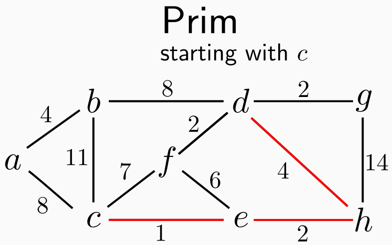
starting with c



Prim's algorithm

Intuition: iteratively grows the tree

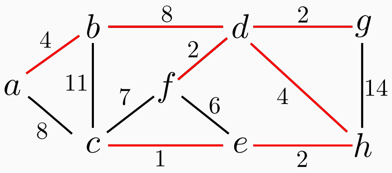




Prim's algorithm

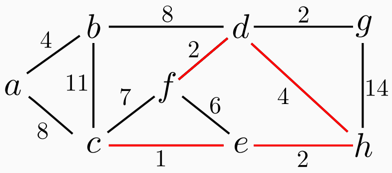
Intuition: iteratively grows the tree

Kruskal



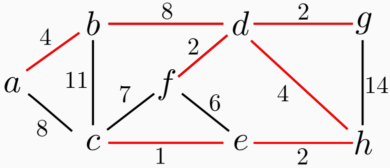
Prim

starting with c



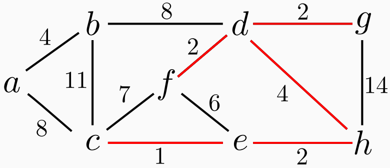
Intuition: iteratively grows the tree

Kruskal



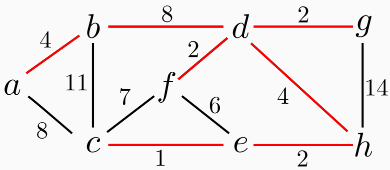
Prim

starting with c

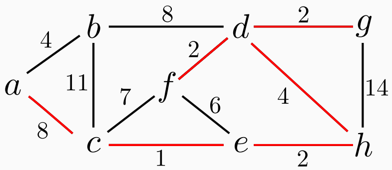


Intuition: iteratively grows the tree

Kruskal

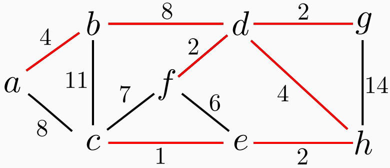


Prim
starting with c

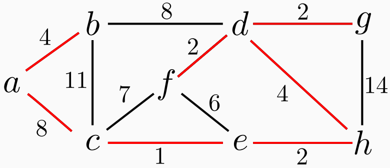


Intuition: iteratively grows the tree

Kruskal



Prim
starting with c



Prim's algorithm: pseudocode

Let S be the set included in the tree so far

Prim's algorithm: pseudocode

Let S be the set included in the tree so far

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

Prim's algorithm: pseudocode

Let S be the set included in the tree so far

$\mathrm{cost}(v) := \min_{e=(u,v) \text{ s.t. } u \in S} w_e$ and $\mathrm{prev}(\cdot)$ is used to keep track of the tree

Prim's algorithm: pseudocode

Let S be the set included in the tree so far

$\mathrm{cost}(v) := \min_{e=(u,v) \text{ s.t. } u \in S} w_e$ and $\mathrm{prev}(\cdot)$ is used to keep track of the tree

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

Prim's algorithm: pseudocode

Let S be the set included in the tree so far

$\mathrm{cost}(v) := \min_{e=(u,v) \text{ s.t. } u \in S} w_e$ and $\mathrm{prev}(\cdot)$ 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;
```

Prim's algorithm: pseudocode

Let S be the set included in the tree so far

$\mathrm{cost}(v) := \min_{e=(u,v) \text{ s.t. } u \in S} w_e$ and $\mathrm{prev}(\cdot)$ is used to keep track of the tree

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

    for $v \in V$:

        $\mathrm{cost}(v) := \infty$;

        $\mathrm{prev} := \mathrm{nil}$;

    Pick any initial vertex u_0;

Prim's algorithm: pseudocode

Let S be the set included in the tree so far

$\mathrm{cost}(v) := \min_{e=(u,v) \text{ s.t. } u \in S} w_e$ and $\mathrm{prev}(\cdot)$ is used to keep track of the tree

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

  for $v \in V$**:**

    $\mathrm{cost}(v) := \infty$;

    $\mathrm{prev} := \mathrm{nil}$;

  Pick any initial vertex u_0;

  $\mathrm{cost}(u_0) := 0$;

Prim's algorithm: pseudocode

Let S be the set included in the tree so far

$\mathrm{cost}(v) := \min_{e=(u,v) \text{ s.t. } u \in S} w_e$ and $\mathrm{prev}(\cdot)$ is used to keep track of the tree

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

- **for** $v \in V$:
  - $\mathrm{cost}(v) := \infty$;
  - $\mathrm{prev} := \mathrm{nil}$;
- Pick any initial vertex u_0;
- $\mathrm{cost}(u_0) := 0$;
- $H := \mathrm{make\_queue}(V)$; // keys are $\mathrm{cost}(v)$

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

    for $v \in V$**:**

        $\text{cost}(v) := \infty$;

        $\text{prev} := \text{nil}$;

    Pick any initial vertex u_0;

    $\text{cost}(u_0) := 0$;

    $H := \text{make\_queue}(V)$; // keys are $\text{cost}(v)$

    while *H is not empty***:**

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 := 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);
```

Prim's algorithm: pseudocode

Let S be the set included in the tree so far

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

Prim's algorithm: pseudocode

Let S be the set included in the tree so far

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

Prim's algorithm: pseudocode

Let S be the set included in the tree so far

$\mathrm{cost}(v) := \min\limits_{e=(u,v) \text{ s.t. } u \in S} w_e$ and $\mathrm{prev}(\cdot)$ 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 := 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;
```

Prim's algorithm: pseudocode

Let S be the set included in the tree so far

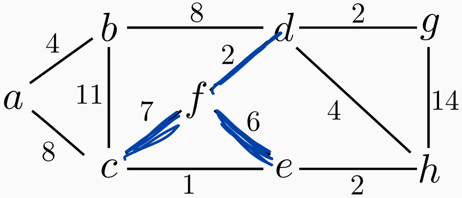
$\mathrm{cost}(v) := \min_{e=(u,v) \text{ s.t. } u \in S} w_e$ and $\mathrm{prev}(\cdot)$ is used to keep track of the tree

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

- **for** $v \in V$:
  - $\mathrm{cost}(v) := \infty$;
  - $\mathrm{prev} := \mathrm{nil}$;
- Pick any initial vertex u_0;
- $\mathrm{cost}(u_0) := 0$;
- $H := \mathrm{make\_queue}(V)$; // keys are $\mathrm{cost}(v)$
- **while** *H is not empty*:
  - $v = \mathrm{delete\_min}(H)$;
  - **for** $e := (v, z) \in E$:
    - **if** $\mathrm{cost}(z) > w_e$:
      - $\mathrm{cost}(z) := w_e$;
      - $\mathrm{prev}(z) := v$;

Prim's algorithm: a running example

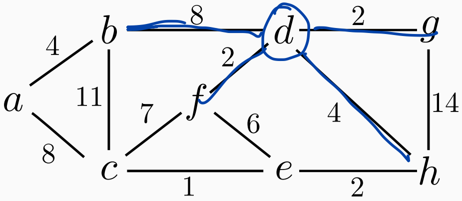
Starting with f



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

Prim's algorithm: a running example

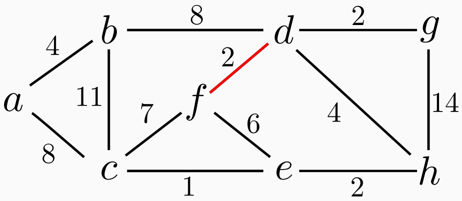
Starting with f



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

Prim's algorithm: a running example

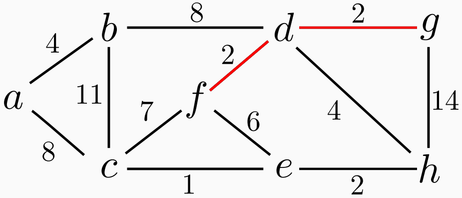
Starting with f



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

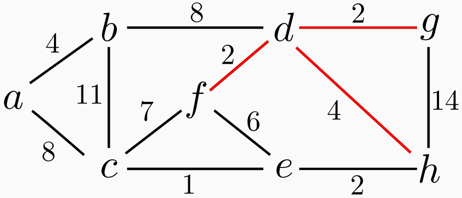
Starting with f



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

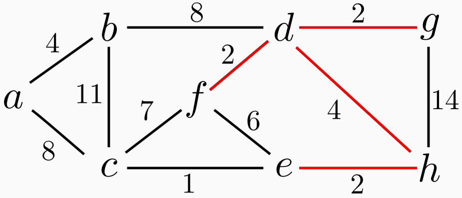
Starting with f



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

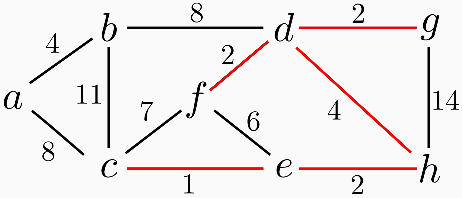
Starting with f



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

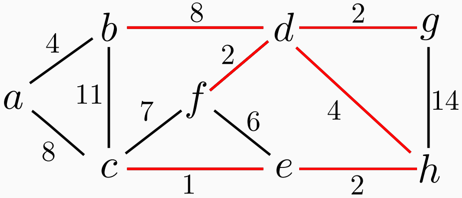
Starting with f



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

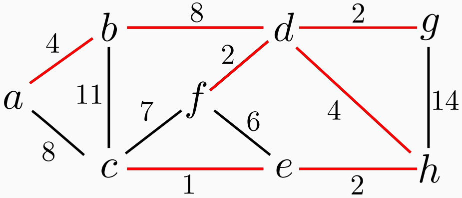
Starting with f



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								

Greedy algorithms

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

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 $e_2 :$ $\begin{aligned} a &\to 0 \\ b &\to 1 \\ c &\to 01 \end{aligned}$ 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