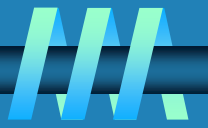
# Greedy Technique

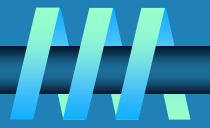


**Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:**

- ***feasible***
- ***locally optimal***
- ***irrevocable***

**For some problems, yields an optimal solution for every instance.**
**For most, does not but can be useful for fast approximations.**

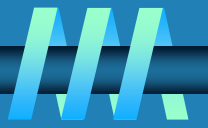
# Applications of the Greedy Strategy



- Optimal solutions:
  - change making for "normal" coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes

- Approximations:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

# Change-Making Problem



Given unlimited amounts of coins of denominations $d_1 > \ldots > d_m$, give change for amount $n$ with the least number of coins

Example: $d_1 = 25c$, $d_2 = 10c$, $d_3 = 5c$, $d_4 = 1c$ and $n = 48c$

Greedy solution:

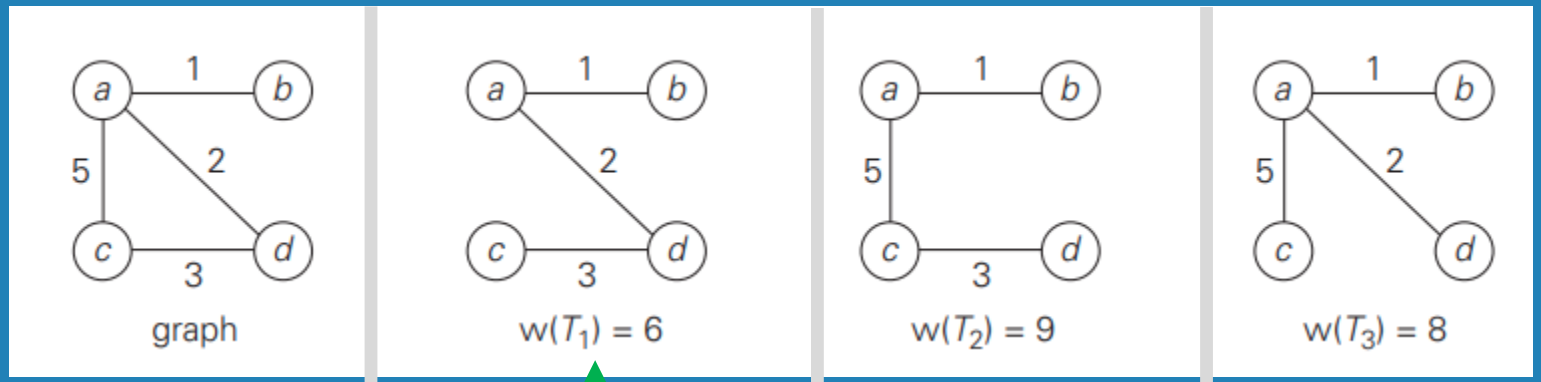
Greedy solution is

- optimal for any amount and "normal" set of denominations
- may not be optimal for arbitrary coin denominations

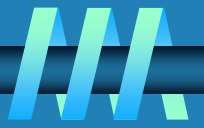
# Minimum Spanning Tree (MST)

- ***Spanning tree*** of a connected graph *G*: a connected acyclic subgraph of *G* that includes all of *G*'s vertices

- ***Minimum spanning tree*** of a weighted, connected graph *G*: a spanning tree of *G* of minimum total weight



# Prim's MST algorithm



- Start with tree $T_1$ consisting of one (any) vertex and "grow" tree one vertex at a time to produce MST through a series of expanding subtrees $T_1, T_2, \ldots, T_n$

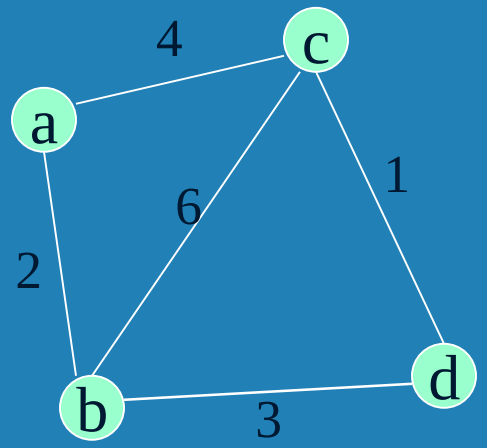
- On each iteration, construct $T_{i+1}$ from $T_i$ by adding vertex not in $T_i$ that is closest to those already in $T_i$ (this is a "greedy" step!)

- Stop when all vertices are included

# Example

**ALGORITHM** *Prim(G)*

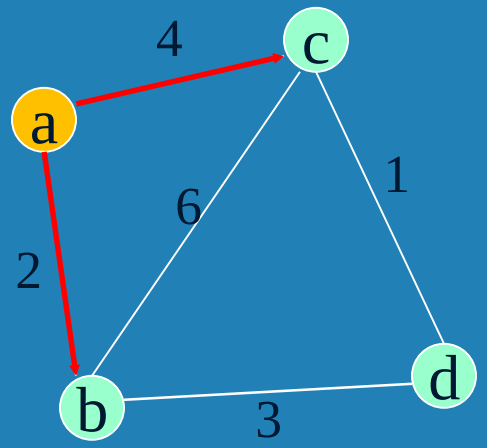
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
**for** $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
    such that $v$ is in $V_T$ and $u$ is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
**return** $E_T$



# Example

**ALGORITHM** *Prim(G)*

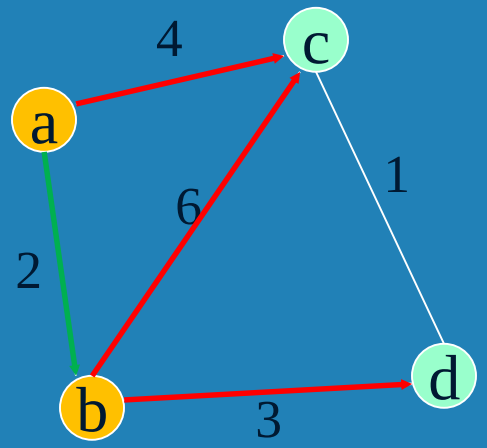
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
**for** $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
    such that $v$ is in $V_T$ and $u$ is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
**return** $E_T$



# Example

**ALGORITHM** *Prim(G)*

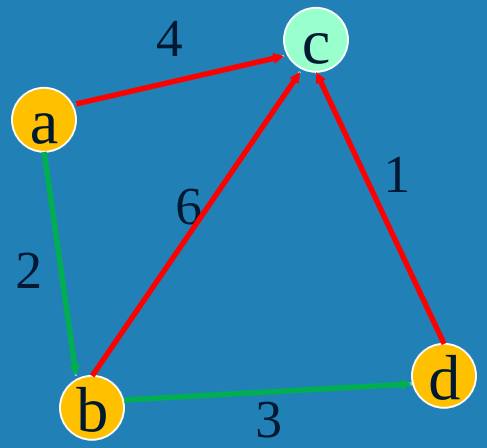
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
**for** $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
    such that $v$ is in $V_T$ and $u$ is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
**return** $E_T$



# Example

**ALGORITHM** *Prim(G)*

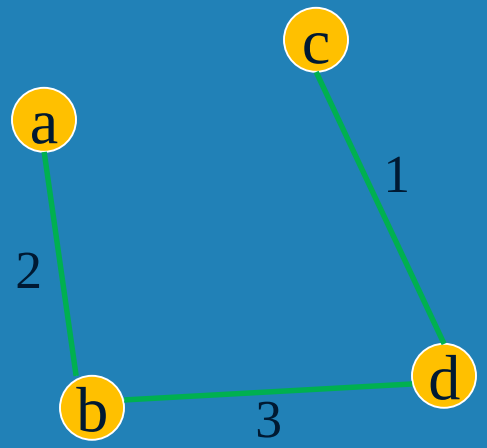
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
**for** $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
    such that $v$ is in $V_T$ and $u$ is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
**return** $E_T$



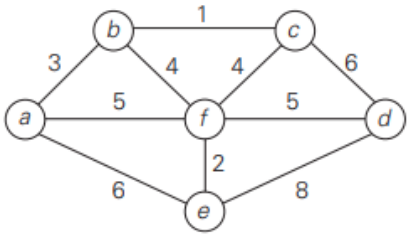
# Example

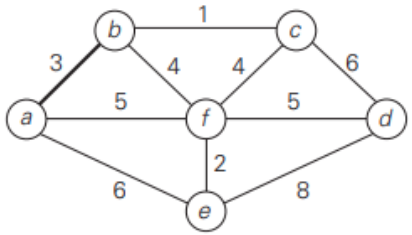
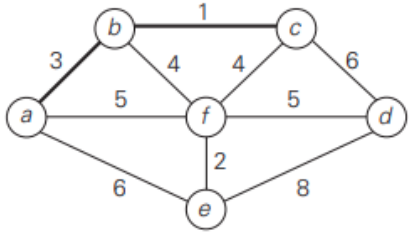
**ALGORITHM** *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
**for** $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
    such that $v$ is in $V_T$ and $u$ is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
**return** $E_T$



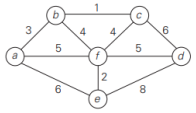
# Example

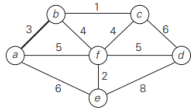
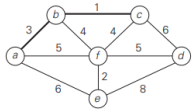



| Tree vertices | Remaining vertices | Illustration |
|---|---|---|
| a(−, −) | **b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5) |  |
| b(a, 3) | **c(b, 1)** d(−, ∞) e(a, 6) f(b, 4) |  |

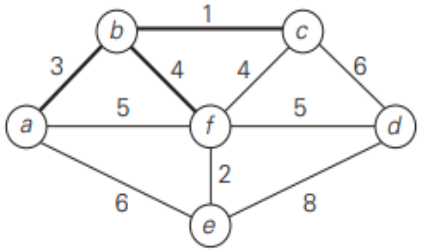
Application of Prim's algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.

# Example

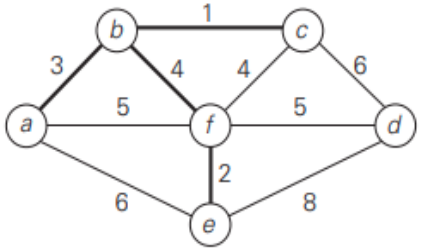



| Tree vertices | Remaining vertices | Illustration |
|---|---|---|
| a(−, −) | **b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5) |  |
| b(a, 3) | **c(b, 1)** d(−, ∞) e(a, 6) f(b, 4) |  |

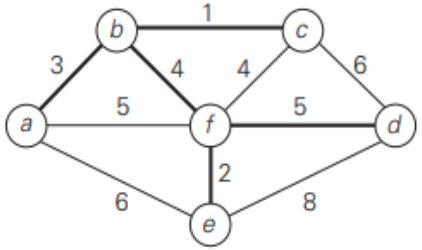
c(b, 1) d(c, 6) e(a, 6) **f(b, 4)**




f(b, 4) d(f, 5) **e(f, 2)**



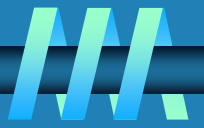

e(f, 2) **d(f, 5)**




d(f, 5)

Application of Prim's algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.

# Notes about Prim's algorithm

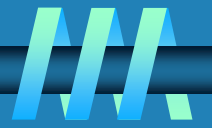


- Proof by induction that this construction actually yields MST
- Needs priority queue for locating closest fringe vertex
- Efficiency
  - $O(n^2)$ for weight matrix representation of graph and array implementation of priority queue
  - $O(m \log n)$ for adjacency list representation of graph with $n$ vertices and $m$ edges and min-heap implementation of priority queue

# Another greedy algorithm for MST: Kruskal's

- Sort the edges in nondecreasing order of lengths
- "Grow" tree one edge at a time to produce MST through a series of expanding forests $F_1, F_2, \ldots, F_{n-1}$
- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)

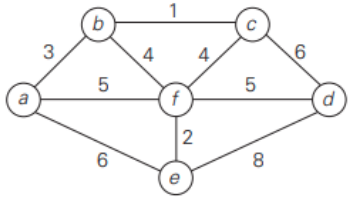
# Kruskal Algorithm

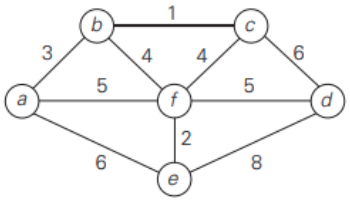
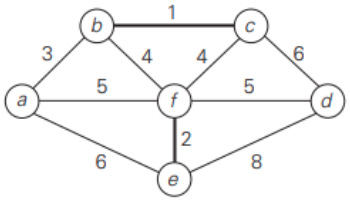
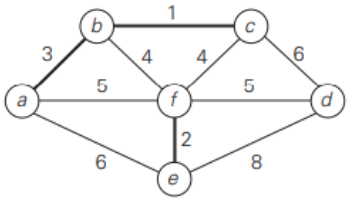


**ALGORITHM** *Kruskal*$(G)$

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
sort $E$ in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
**while** $ecounter < |V| - 1$ **do**
    $k \leftarrow k + 1$
    **if** $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
**return** $E_T$

# Example




| Tree edges | Sorted list of edges | Illustration |
|---|---|---|
| | **bc** ef ab bf cf af df ae cd de<br>1 2 3 4 4 5 5 6 6 8 |  |
| bc<br>1 | bc **ef** ab bf cf af df ae cd de<br>1 2 3 4 4 5 5 6 6 8 |  |
| ef<br>2 | bc ef **ab** bf cf af df ae cd de<br>1 2 3 4 4 5 5 6 6 8 |  |

# Example

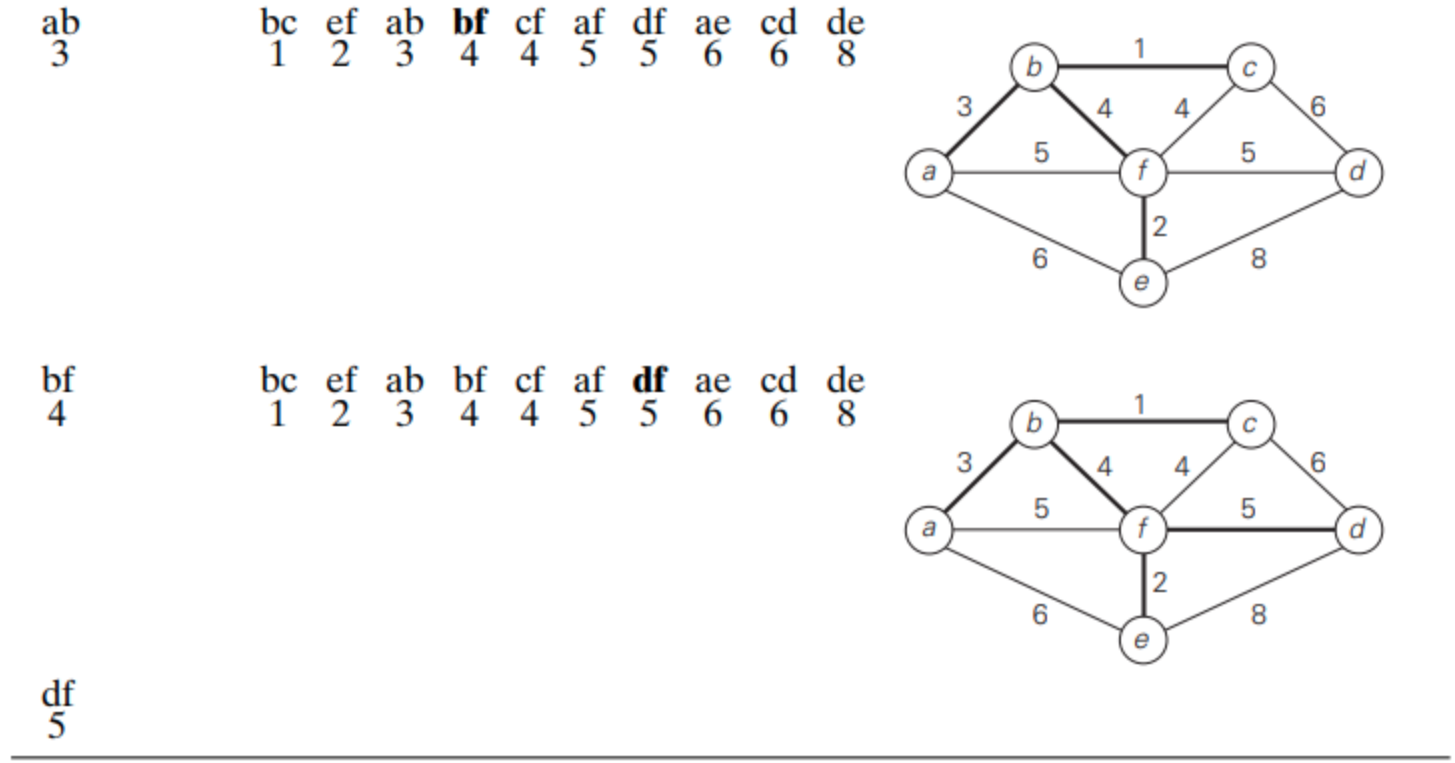



**FIGURE 9.5** Application of Kruskal's algorithm. Selected edges are shown in bold.

# Acyclic Subgraph Detection

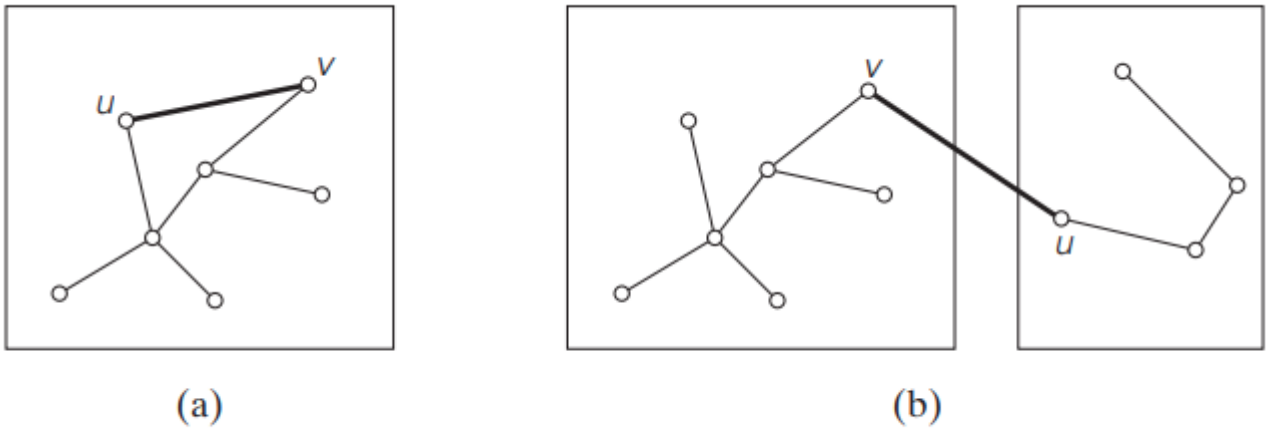
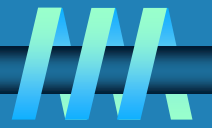



**FIGURE 9.6** New edge connecting two vertices may (a) or may not (b) create a cycle.

- **If the edge's vertices ( u and v) are in the same set, it causes a cycle**

# Implementation (union-find)



*makeset*($x$) creates a one-element set $\{x\}$. It is assumed that this operation can be applied to each of the elements of set $S$ only once.

*find*($x$) returns a subset containing $x$.

*union*($x$, $y$) constructs the union of the disjoint subsets $S_x$ and $S_y$ containing $x$ and $y$, respectively, and adds it to the collection to replace $S_x$ and $S_y$, which are deleted from it.

For example, let $S = \{1, 2, 3, 4, 5, 6\}$. Then *makeset*($i$) creates the set $\{i\}$ and applying this operation six times initializes the structure to the collection of six singleton sets:

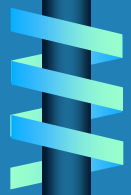
$$\{1\}, \quad \{2\}, \quad \{3\}, \quad \{4\}, \quad \{5\}, \quad \{6\}.$$

Performing *union*(1, 4) and *union*(5, 2) yields

$$\{1, 4\}, \quad \{5, 2\}, \quad \{3\}, \quad \{6\},$$

and, if followed by *union*(4, 5) and then by *union*(3, 6), we end up with the disjoint subsets

$$\{1, 4, 5, 2\}, \quad \{3, 6\}.$$



# Implementation - Datastructure

| subset representatives | |
|---|---|
| element index | representative |
| 1 | 1 |
| 2 | 1 |
| 3 | 3 |
| 4 | 1 |
| 5 | 1 |
| 6 | 3 |

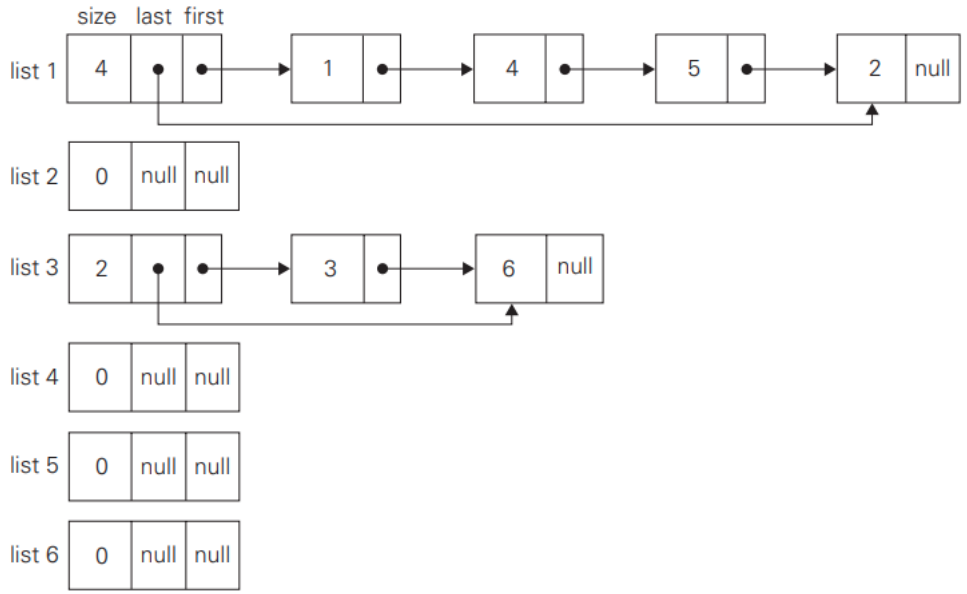
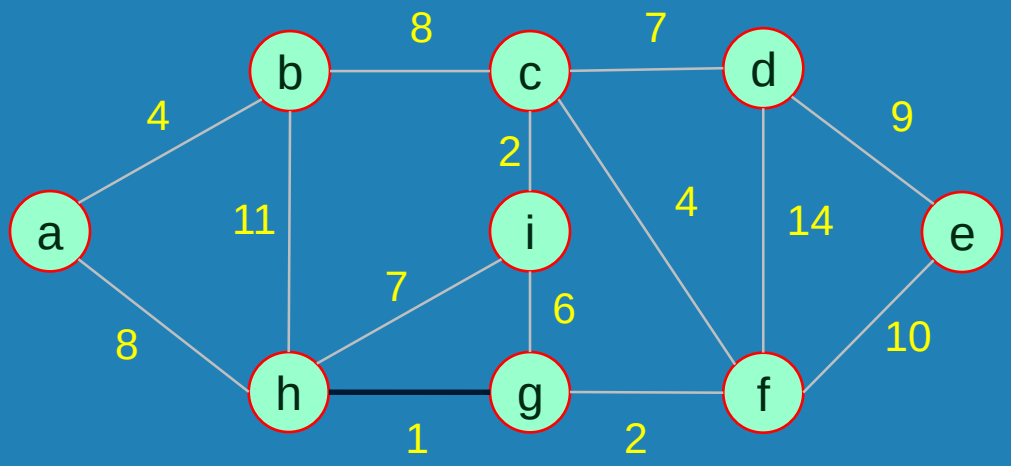



**FIGURE 9.7** Linked-list representation of subsets {1, 4, 5, 2} and {3, 6} obtained by quick find after performing *union*(1, 4), *union*(5, 2), *union*(4, 5), and *union*(3, 6). The lists of size 0 are considered deleted from the collection.

# Example: Kruskal's MST

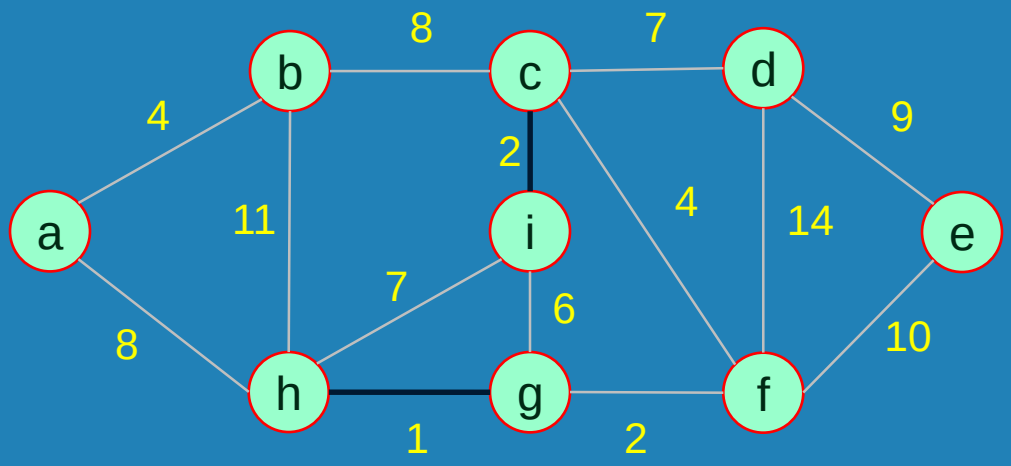



Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg} }

# Example: Kruskal's MST

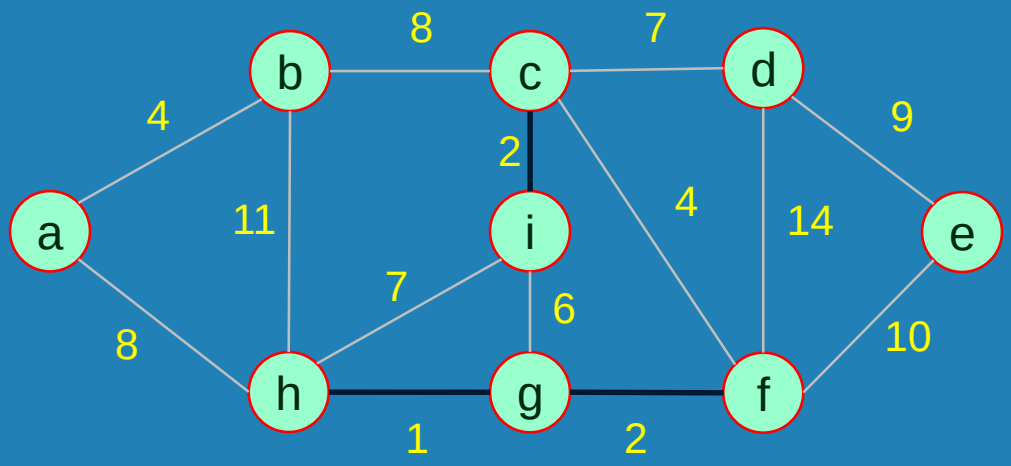



Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg}, {ic} }

# Example: Kruskal's MST

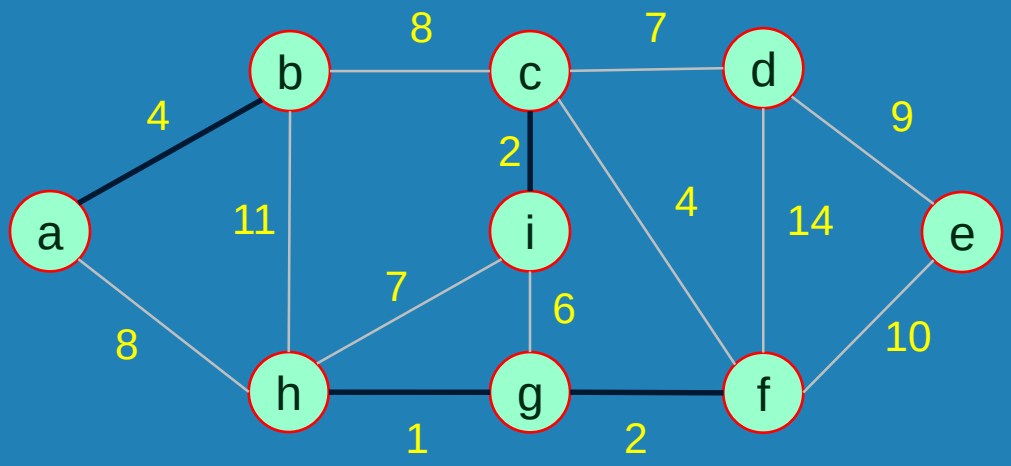



Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg, gf}, {ic} }

# Example: Kruskal's MST




Sorted edge list:

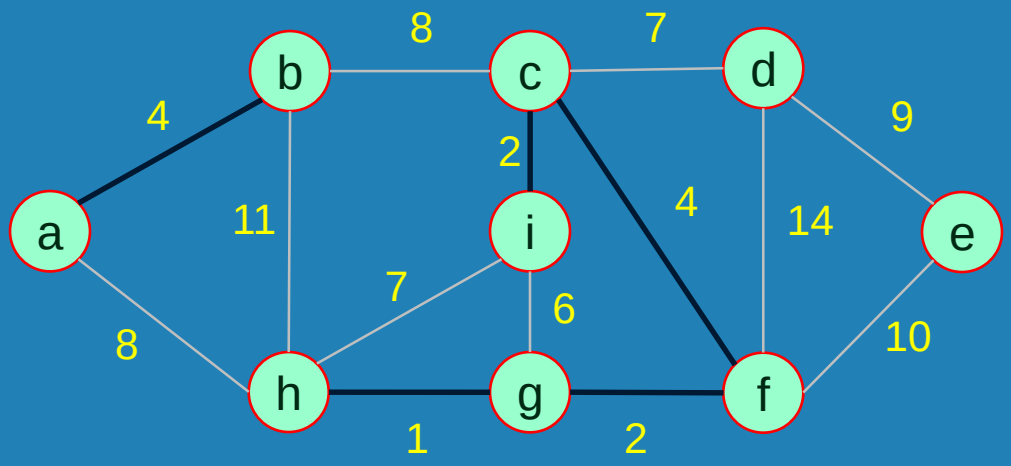
| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg, gf}, {ic}, {ab} }

***union(c,f)*** takes the union of two disjoint sets {hg, gf} and {ic}

{hg, gf, ic, cf}

# Example: Kruskal's MST



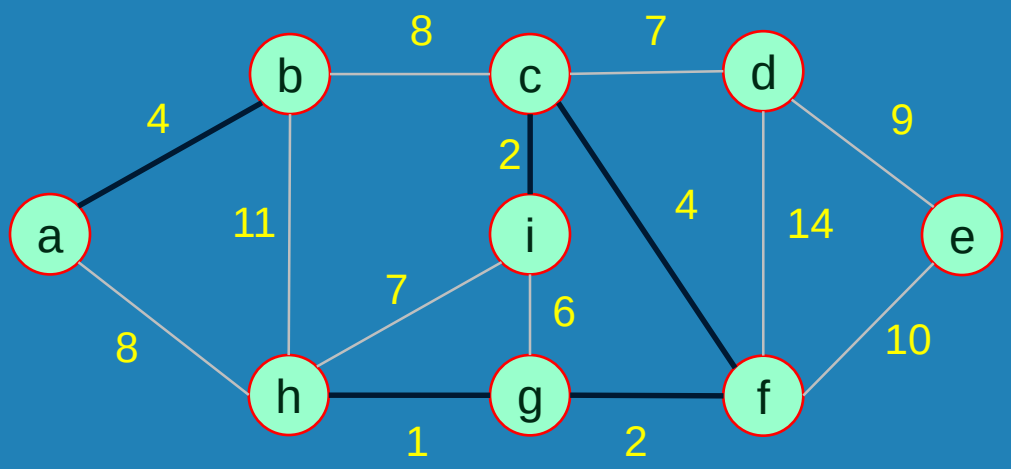

Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg, gf, ic, cf}, {ab} }

***find(i)*** *and* ***find(g)*** *return the same disjoint set, therefore, it forms a cycle. Therefore, "ig" cannot be part of MST.*

# Example: Kruskal's MST



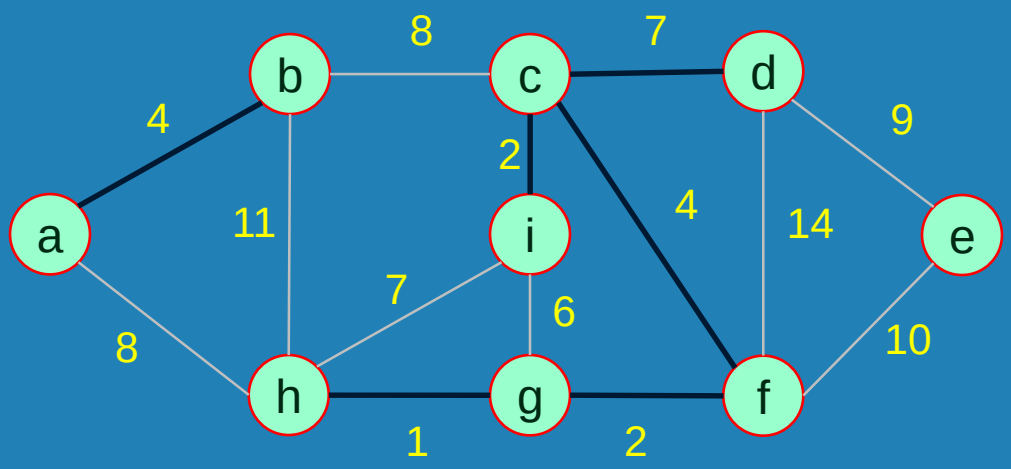

Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg, gf, ic, cf}, {ab} }

***c** and **d** are not in the same set, so add "cd" into the set.*

# Example: Kruskal's MST



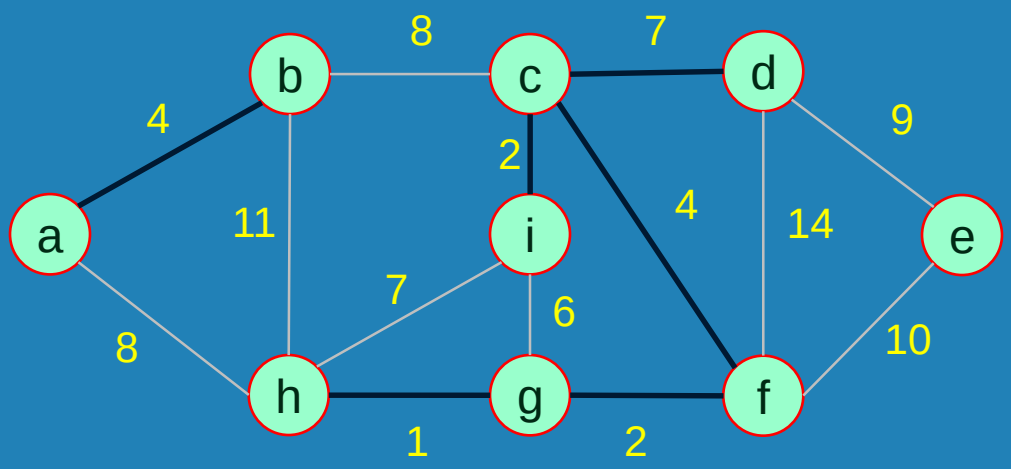

Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg, gf, ic, cf, cd}, {ab} }

***h** and **i** are in the same set, so it causes a cycle, skip "hi"*

# Example: Kruskal's MST



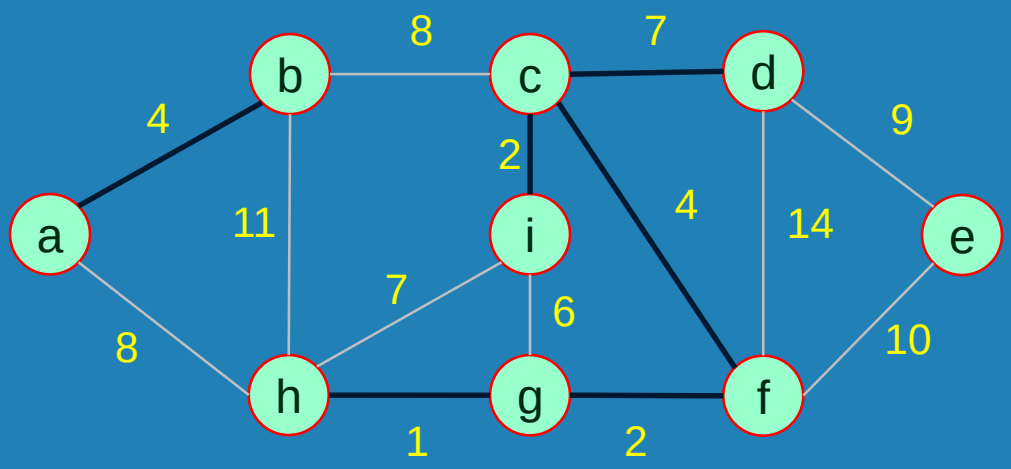

Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg, gf, ic, cf, cd}, {ab} }

***b** and **c** are in the different sets, so take union(b,c), and add "bc"*

# Example: Kruskal's MST



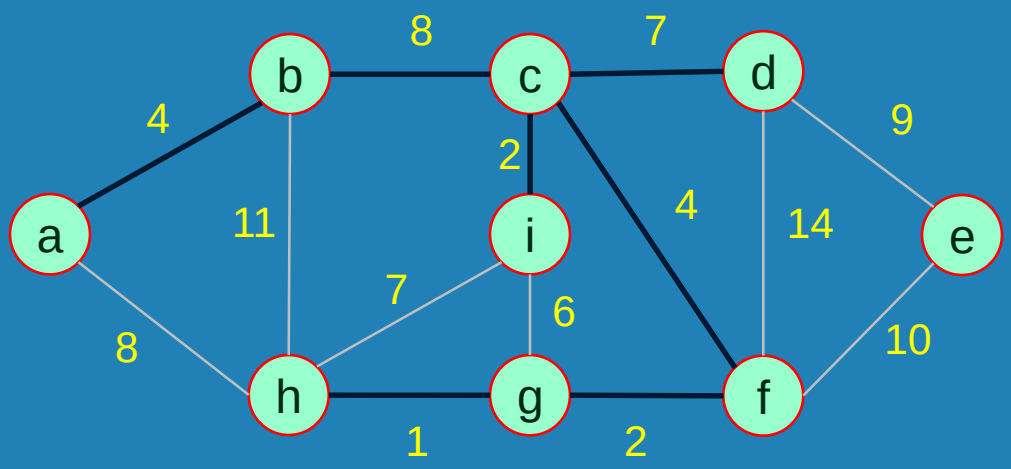

Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = {{hg, gf, ic, cf, cd, ab, bc} }

***b** and **c** are in the different sets, so take union(b,c), and add "bc"*

# Example: Kruskal's MST



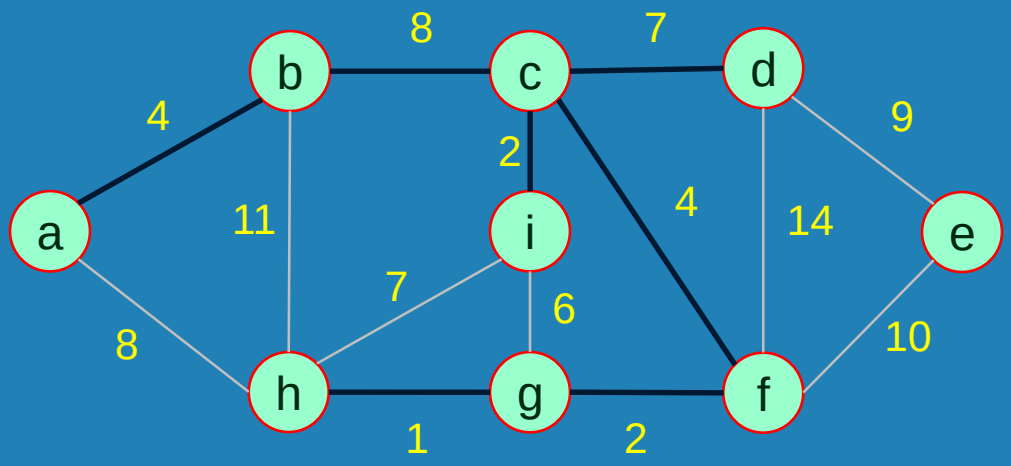

Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg, gf, ic, cf, cd, ab, bc} }

*skip the edge "ah", since "a" and "h" are in the same set.*

# Example: Kruskal's MST

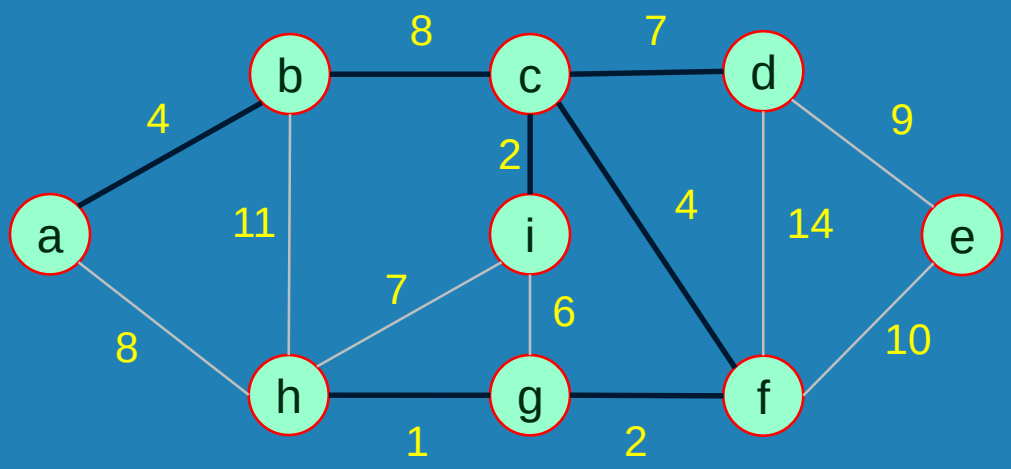



Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg, gf, ic, cf, cd, ab, bc} }

# Example: Kruskal's MST



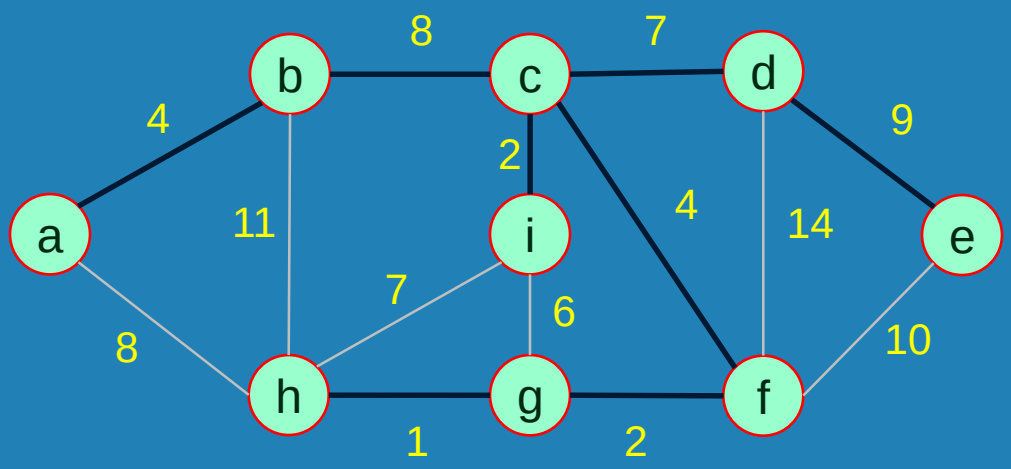

Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg, gf, ic, cf, cd, ab, bc} }

*"d" and "e" are not in the same set. Therefore, add "de" into the list*

# Example: Kruskal's MST



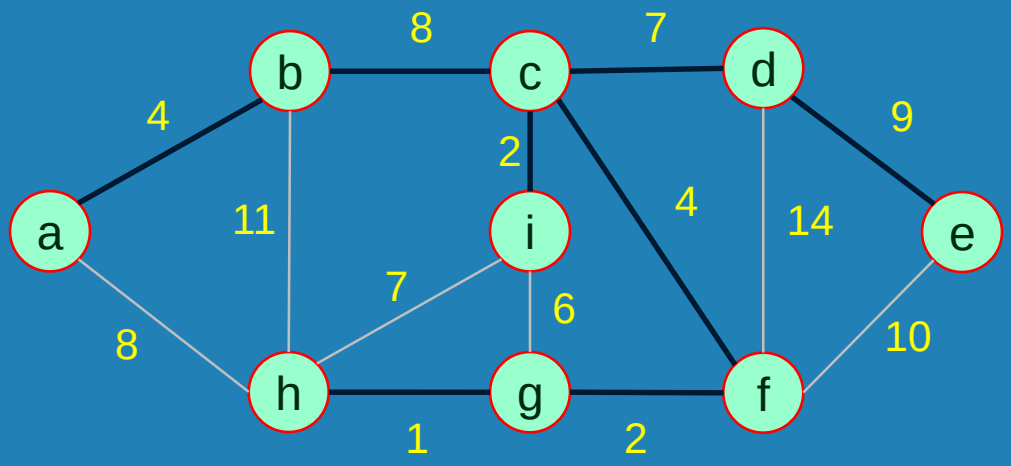

Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

Edge List = { {hg, gf, ic, cf, cd, ab, bc, de} }

*add "de", since "d" and "e" are not in the same set.*

# Example: Kruskal's MST



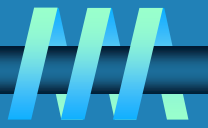

Sorted edge list:

| hg | ic | gf | ab | cf | ig | cd | hi | bc | ah | de | fe | bh | df |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 4 | 4 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 11 | 14 |

MST = { {hg, gf, ic, cf, cd, ab, bc, de} }

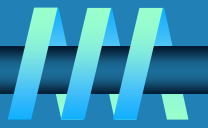
*Since there are 8 edges ( v – 1 = 9 – 1 = 8). MST is built already. No need to check remaining edges, "fe, bh, df"*

# Notes about Kruskal's algorithm



- Algorithm looks easier than Prim's but is harder to implement (checking for cycles!)
- Cycle checking: a cycle is created iff added edge connects vertices in the same connected component
- *Union-find* algorithms – see section 9.2

# Shortest paths – Dijkstra's algorithm



***Single Source Shortest Paths Problem***: Given a weighted connected graph G, find shortest paths from source vertex *s* to each of the other vertices

***Dijkstra's algorithm***: Similar to Prim's MST algorithm, with a different way of computing numerical labels: Among vertices not already in the tree, it finds vertex *u* with the smallest sum

$$d_v + w(v,u)$$

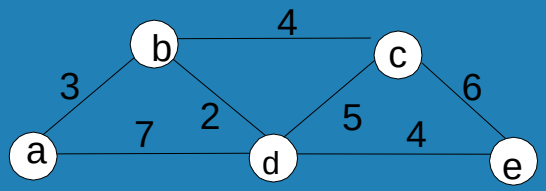
where

*v* is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree)

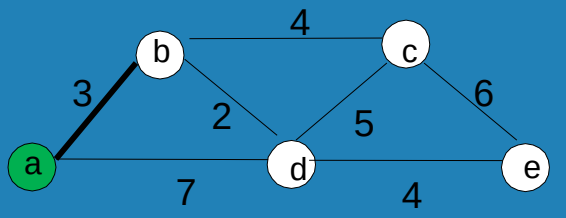
$d_v$ is the length of the shortest path form source to *v*

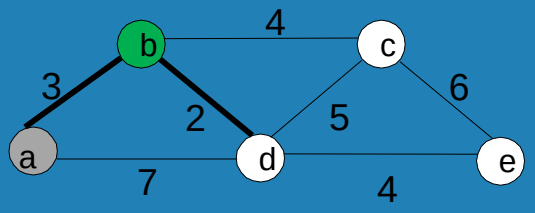
$w(v,u)$ is the length (weight) of edge from *v* to *u*

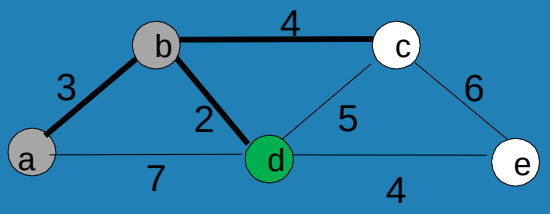
# Example

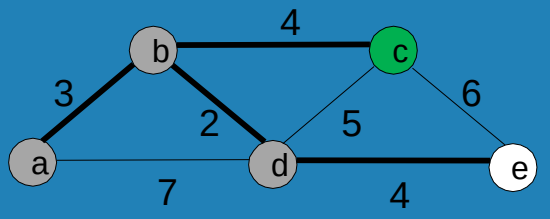



| Tree vertices | Remaining vertices | |
|---|---|---|
| a(-,0) | <u>b(a,3)</u> | c(-,∞) d(a,7) e(-,∞) |
| b(a,3) | c(b,3+4) | <u>d(b,3+2)</u> e(-,∞) |
| d(b,5) | <u>c(b,7)</u> | e(d,5+4) |
| c(b,7) | <u>e(d,9)</u> | |
| e(d,9) | | |



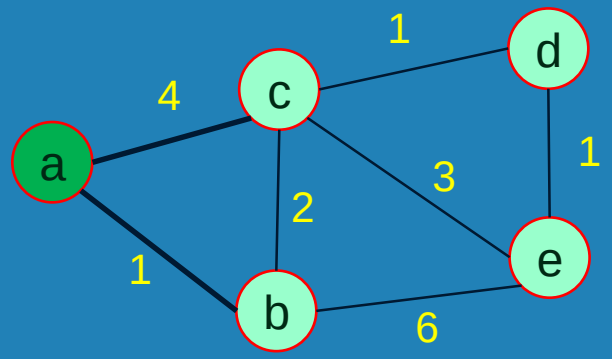



# Example (array implementation)

| | a | b | c | d | e |
|---|---|---|---|---|---|
| Distance | **0** | 1 | 4 | ∞ | ∞ |
| Visited | T | F | F | F | F |
| Previous | Null | a | a | Null | Null |

Source : a

Target : any node

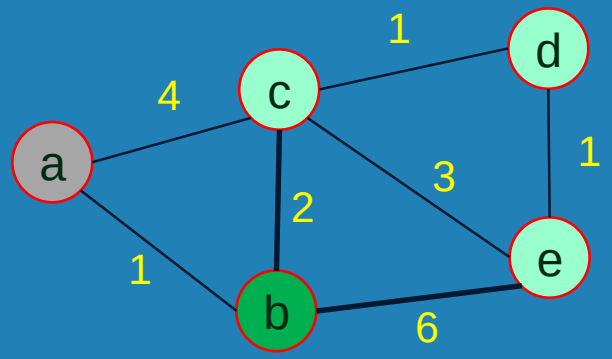


# Example (array implementation)

| | a | b | c | d | e |
|---|---|---|---|---|---|
| Distance | **0** | **1** | 4 or 1+2 | ∞ | ∞ or 1+6 |
| Visited | T | T | F | F | F |
| Previous | Null | a | a or b | Null | b |

Source : a

Target : any node

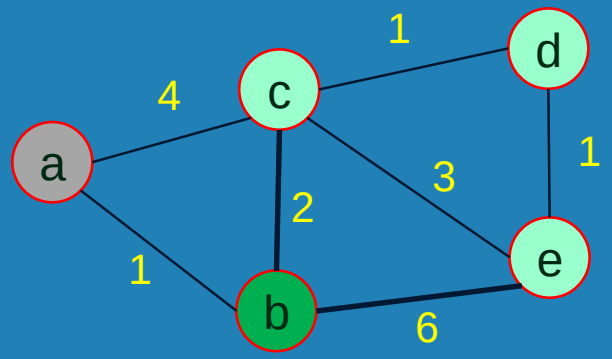


# Example (array implementation)

| | a | b | c | d | e |
|---|---|---|---|---|---|
| Distance | **0** | **1** | **3** | ∞ | 7 |
| Visited | T | T | F | F | F |
| Previous | Null | a | b | Null | b |

Source : a

Target : any node

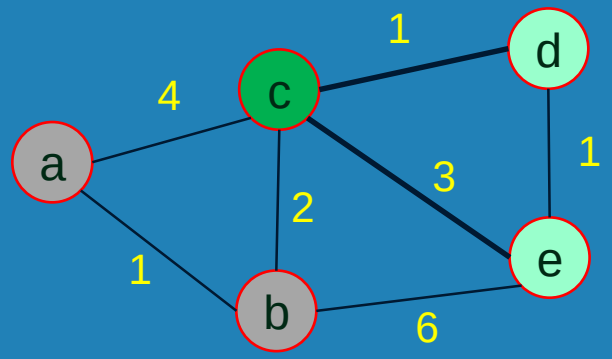


# Example (array implementation)

| | a | b | c | d | e |
|---|---|---|---|---|---|
| Distance | **0** | **1** | **3** | ∞ or 3+1 | 7 or 3+3 |
| Visited | T | T | T | F | F |
| Previous | Null | a | b | c | b or c |

Source : a

Target : any node

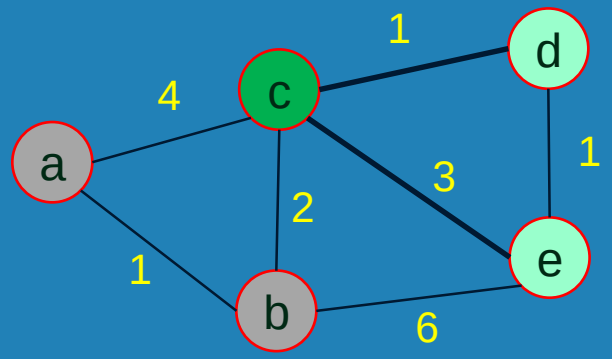


# Example (array implementation)

| | a | b | c | d | e |
|---|---|---|---|---|---|
| Distance | **0** | **1** | **3** | **4** | 6 |
| Visited | T | T | T | F | F |
| Previous | Null | a | b | c | c |

Source : a

Target : any node

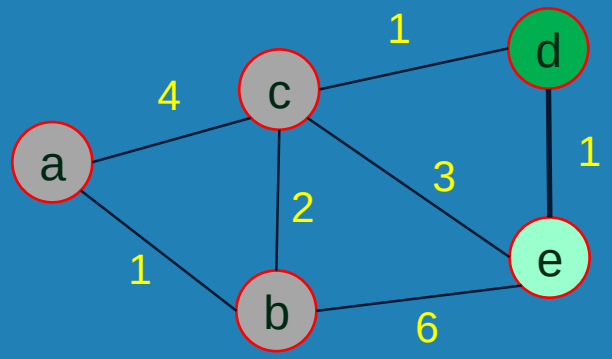


# Example (array implementation)

| | a | b | c | d | e |
|---|---|---|---|---|---|
| Distance | **0** | **1** | **3** | **4** | 6 |
| Visited | T | T | T | T | F |
| Previous | Null | a | b | c | c |

Source : a

Target : any node

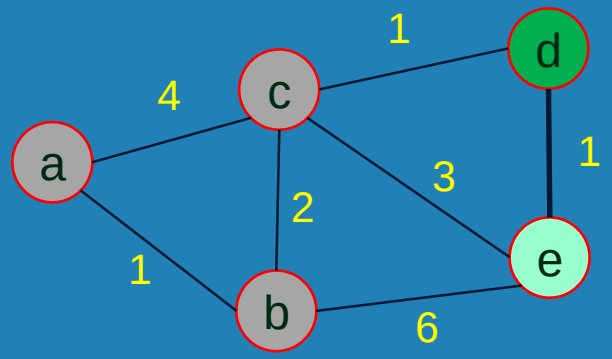


# Example (array implementation)

| | a | b | c | d | e |
|---|---|---|---|---|---|
| Distance | **0** | **1** | **3** | **4** | 6 or 4+1 |
| Visited | T | T | T | T | F |
| Previous | Null | a | b | c | c or d |

Source : a

Target : any node

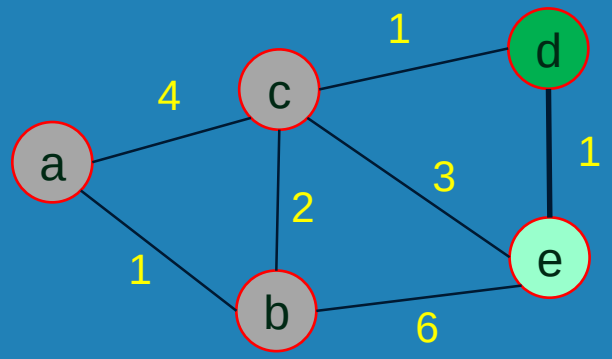


# Example (array implementation)

| | a | b | c | d | e |
|---|---|---|---|---|---|
| Distance | **0** | **1** | **3** | **4** | 5 |
| Visited | T | T | T | T | F |
| Previous | Null | a | b | c | d |

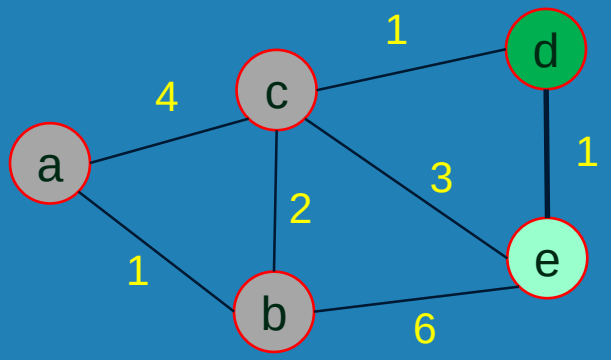
Source : a

Target : any node



# Example (array implementation)

| | a | b | c | d | e |
|---|---|---|---|---|---|
| Distance | 0 | 1 | 3 | 4 | 5 |
| Previous | Null | a | b | c | d |




Source : a Target : e

Backtracking from e

Backtrack : e d c b a

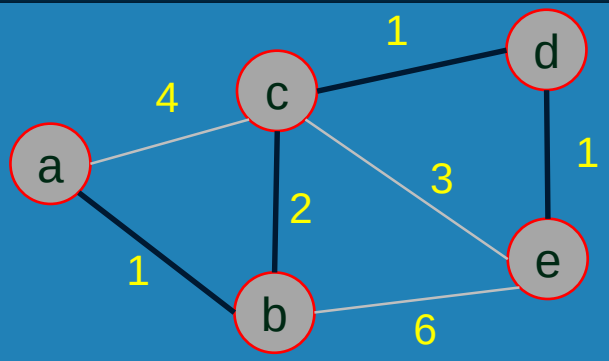
Path : a b c d e

We can find all shortest paths from node "a" to any node in the weighted graph

# Tracing Dijsktra's Algorithm

Source : a

Target : e

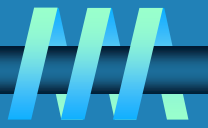



| Tree vertices | Remaining vertices |
|---|---|
| a(-,0) | b(a,1) , c(a,4), d(-, ∞), e(-, ∞) |
| b(a,1) | c(b,1+2), d(-, ∞), e(b, 1+6) |
| c(b,3) | d(c, 3+1), e(c, 3+3) |
| d(c,4) | e(d, 4+1) |
| e(d,5) | |

Backtracking from e to a : e, d, c, b, a

Path a to e : a, b, c, d, e

# Notes on Dijkstra's algorithm



- Doesn't work for graphs with negative weights
- Applicable to both undirected and directed graphs
- Efficiency
  - $O(|V|^2)$ for graphs represented by weight matrix and array implementation of priority queue
  - $O(|E|\log|V|)$ for graphs represented by adj. lists and min-heap implementation of priority queue
- Don't mix up Dijkstra's algorithm with Prim's algorithm!