Lecture 20

Recap part B

CS 161 Design and Analysis of Algorithms

Ioannis Panageas

Greedy method

The greedy method is a general algorithm design technique, in which given:

- configurations: different choices we need to make
- objective function: a score assigned to all configurations, which we want to either maximize or minimize

We should make choices greedily: We can find a globally-optimal solution by a series of local improvements from a starting configuration.

Example: Maxflow problem.

Configurations: All possible flow functions. Objective function: Maximize flow value.

Ford-Fulkerson makes choices greedily starting from flow $f = 0$.

Greedy does not always work

Problem 1: Given a value X and notes $\{1, 2, 5, 10, 20, 50, 100\}$, find the minimum number of notes to create value X. You can use each note as many times as you want.

Answer: Greedy approach works. Pick largest note that is at most X and subtract from X. Repeat until value becomes 0.
E.g., for X=1477, you need fourteen 100s, one 50, one 20, one 5 and one 2.

Problem 2: Given a value X and notes $\{1, 2, 7, 10\}$, find the minimum number of notes to create value X. You can use each note as many times as you want.

Answer: Greedy approach does not work as before.
E.g., for X=14, you need two 7s, but greedy will give one 10, two 2s.

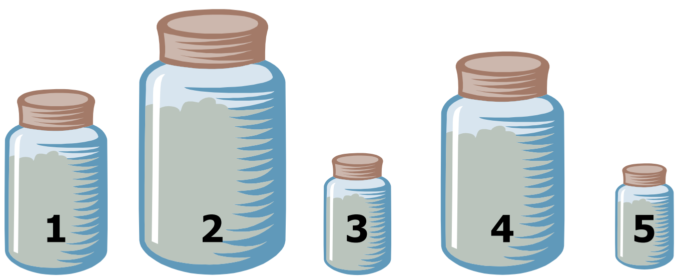
Greedy does not work always

Fractional Knapsack

Problem: A set of n items, with each item i having positive weight w_i and positive value v_i. You are asked to choose items with maximum total value so that the total weight is at most W. We are allowed to take fractional amounts (some percentage of each item).

Example:

Items:



Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Value:	\$12	\$32	\$40	\$30	\$50
Value: (\$ per ml)	\$3	\$4	\$20	\$5	\$50

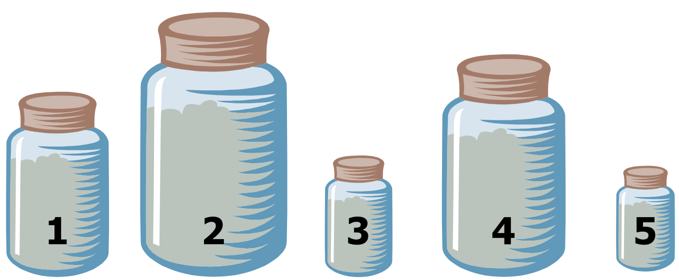
"knapsack" with 10ml

Fractional Knapsack

Problem: A set of n items, with each item i having positive weight w_i and positive value v_i. You are asked to choose items with **maximum total value** so that the **total weight** is **at most** W. We are allowed to take **fractional amounts** (some percentage of each item).

Example:

Items:



Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Value:	\$12	\$32	\$40	\$30	\$50
Value: (\$ per ml)	\$3	\$4	\$20	\$5	\$50

“knapsack” with 10ml

Solution:

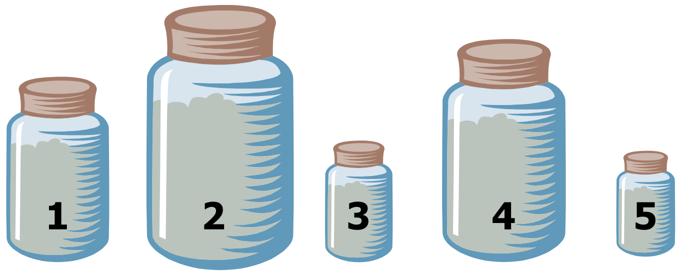
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

Total Value: \$124

Fractional Knapsack

Idea: Greedy approach. Keep taking item with highest value to weight ratio until knapsack is full or run out of items.

Items:



Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Value:	\$12	\$32	\$40	\$30	\$50
Value:	\$3	\$4	\$20	\$5	\$50

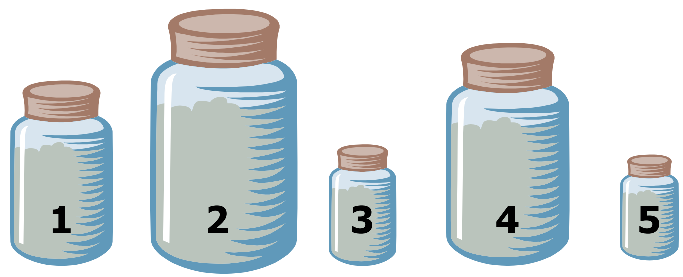
W = 10 ml

$value$ = \$0

Fractional Knapsack

Idea: Greedy approach. Keep taking item with highest value to weight ratio until knapsack is full or run out of items.

Items:



Weight:	4 ml	8 ml	2 ml	6 ml	**1 ml**
Value:	\$12	\$32	\$40	\$30	\$50
Value:	\$3	\$4	\$20	\$5	**\$50**

W = 10 ml

$value$ = \$0

Fractional Knapsack

Idea: Greedy approach. Keep taking item with highest value to weight ratio until knapsack is full or run out of items.

Items:

Weight:	4 ml	8 ml	2 ml	6 ml
Value:	\$12	\$32	\$40	\$30
Value:	\$3	\$4	\$20	\$5

$W = 9$ ml

$value = \$50$

Fractional Knapsack

Idea: Greedy approach. Keep taking item with highest value to weight ratio until knapsack is full or run out of items.

Items:

Weight:	4 ml	8 ml	**2 ml**	6 ml
Value:	\$12	\$32	\$40	\$30
Value:	\$3	\$4	**\$20**	\$5

W = 9 ml
$value$ = \$50

Fractional Knapsack

Idea: Greedy approach. Keep taking item with highest value to weight ratio until knapsack is full or run out of items.

Items:

Weight:	4 ml	8 ml	6 ml
Value:	$12	$32	$30
Value:	$3	$4	$5

$W = 7$ ml

$value = \$90$

Fractional Knapsack

Idea: Greedy approach. Keep taking item with highest value to weight ratio until knapsack is full or run out of items.

Items:

Weight:	4 ml	8 ml	**6 ml**
Value:	\$12	\$32	\$30
Value:	\$3	\$4	**\$5**

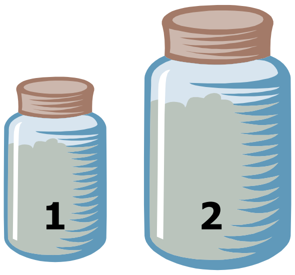
$W = 7$ ml

$value = \$90$

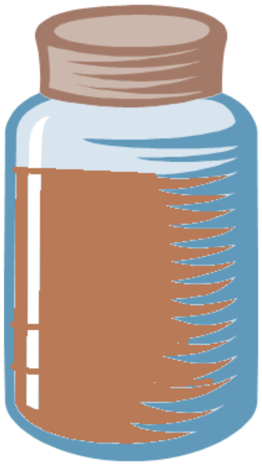
Fractional Knapsack

Idea: Greedy approach. Keep taking item with highest value to weight ratio until knapsack is full or run out of items.

Items:



Weight:	4 ml	8 ml
Value:	$12	$32
Value:	$3	$4

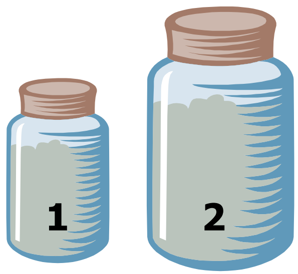


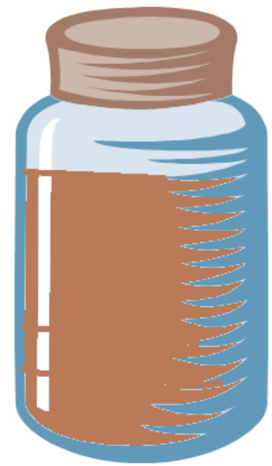
$W = 1$ ml

$value = \$120$

Fractional Knapsack

Idea: Greedy approach. Keep taking item with highest value to weight ratio until knapsack is full or run out of items.

Items:		
Weight:	4 ml	8 ml
Value:	\$12	\$32
Value:	\$3	**\$4**



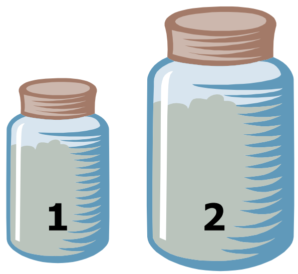
$W = 1$ ml

$value = \$120$

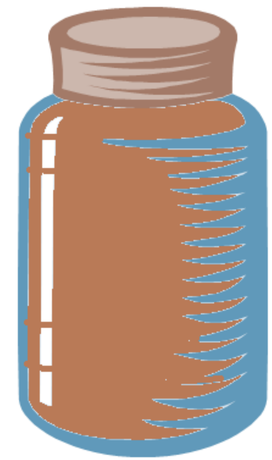
Fractional Knapsack

Idea: Greedy approach. Keep taking item with highest value to weight ratio until knapsack is full or run out of items.

Items:



	Item 1	Item 2
Weight:	4 ml	7 ml
Value:	\$12	\$32
Value:	\$3	**\$4**



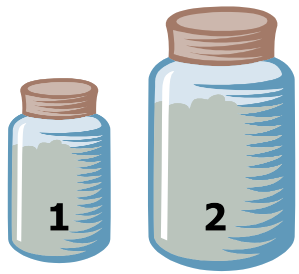
$W = 0$ ml

$value = \$124$

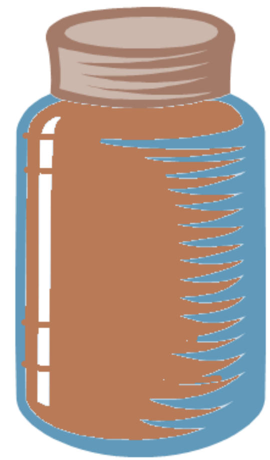
Running time: ?

Fractional Knapsack

Idea: Greedy approach. Keep taking item with highest value to weight ratio until knapsack is full or run out of items.

Items: 

Weight:	4 ml	7 ml
Value:	\$12	\$32
Value:	\$3	**\$4**



$W = 0$ ml

$value = \$124$

Running time: If we sort the items with respect to value to weight ratio then $\Theta(n \log n)$.

Fractional Knapsack

Pseudocode:

Items with $v[], w[]$, knapsack with W

For $i = 1$ to n **do**

$\mathrm{r}[i] \leftarrow \frac{v[i]}{w[i]}$

$w \leftarrow 0$

$val \leftarrow 0$

While $w < W$ **do**

Remove item i with highest $r[i]$

If $w + w_i \leq W$ **then**

$w \leftarrow w + w_i$

$val \leftarrow val + v[i]$

Else

$w \leftarrow W,\ val \leftarrow val + (W - w) \cdot r[i]$

return val

Compute the ratios

Initialization

While knapsack not full

If whole item fits

Scheduling jobs/tasks

Problem 1: Given: a set T of n tasks, each having a start time s_i and a finish time f_i (where $s_i < f_i$)
Goal: Perform all the tasks using a minimum number of machines.
A machine can serve one task at a given time.

Scheduling jobs/tasks

Problem 1: Given: a set T of n tasks, each having a start time s_i and a finish time f_i (where $s_i < f_i$)
Goal: Perform all the tasks using a minimum number of machines.
A machine can serve one task at a given time.

Idea: Sort tasks in increasing order of their **start** time. Assign first task to machine 1 and set $K = 1$.
When considering a new task, if all machines are busy, create a new machine, set $K = K + 1$ and assign the new task to the new machine otherwise assign the new task to an available machine.

Scheduling jobs/tasks

Problem 2: Given: a set T of n tasks, each having a start time s_i and a finish time f_i (where $s_i < f_i$)
Goal: Perform as many tasks as possible using one machine.
In other words, find the maximum number of non-overlapping intervals.

Scheduling jobs/tasks

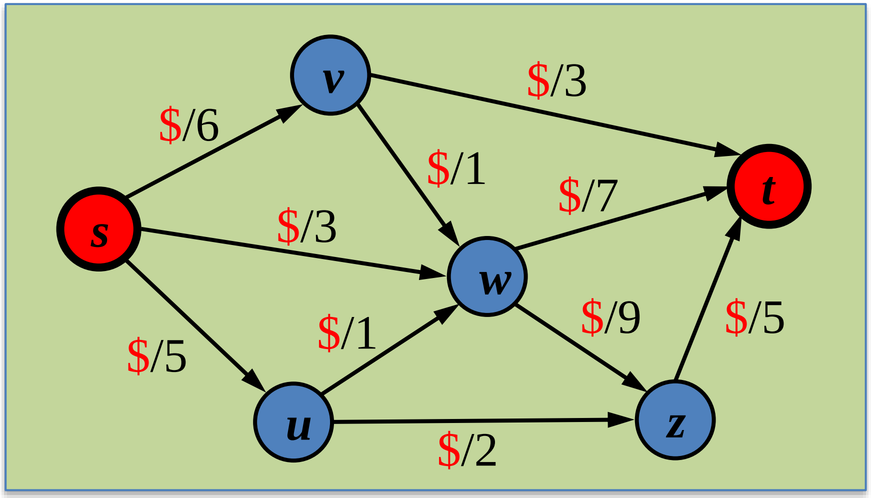
Problem 2: Given: a set T of n tasks, each having a start time s_i and a finish time f_i (where $s_i < f_i$)
Goal: Perform as many tasks as possible using one machine.
In other words, find the maximum number of non-overlapping intervals.

Idea: Sort tasks in increasing order of their **finish** time. Perform first task and remove all overlapping tasks with first task. Repeat the same process to the remaining tasks.

Maxflow Problem

Problem: Given a network G, a source $\boldsymbol{s}$ and a sink $\boldsymbol{t}$, and capacities on the edges, compute the maximum possible flow value $|f^*|$.

Example:



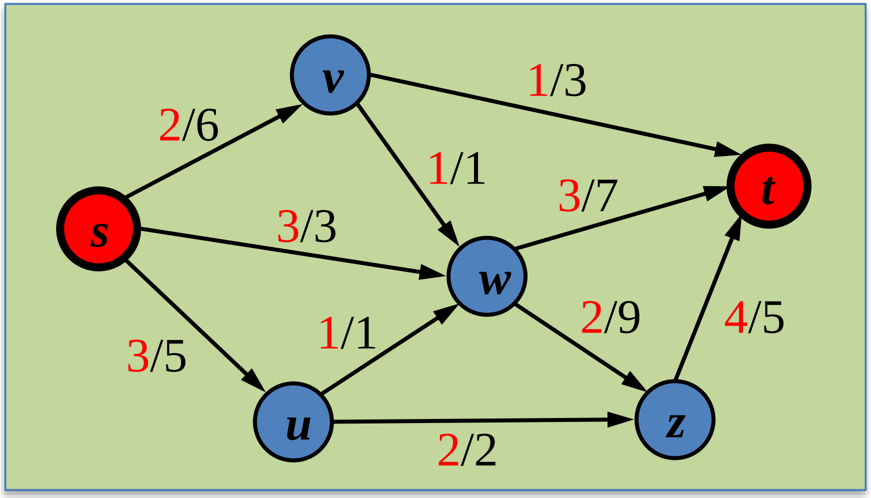
Find the \$ to get maxflow $|f^*|$

Augmenting paths

We are given a network G with edge capacities c and a flow f.
Let (u, v) be an edge from u to v.

Residual capacity from u to v is $\Delta_f(u, v) = c(u, v) - f(u, v)$.

Residual capacity from v to u is $\Delta_f(v, u) = f(u, v)$



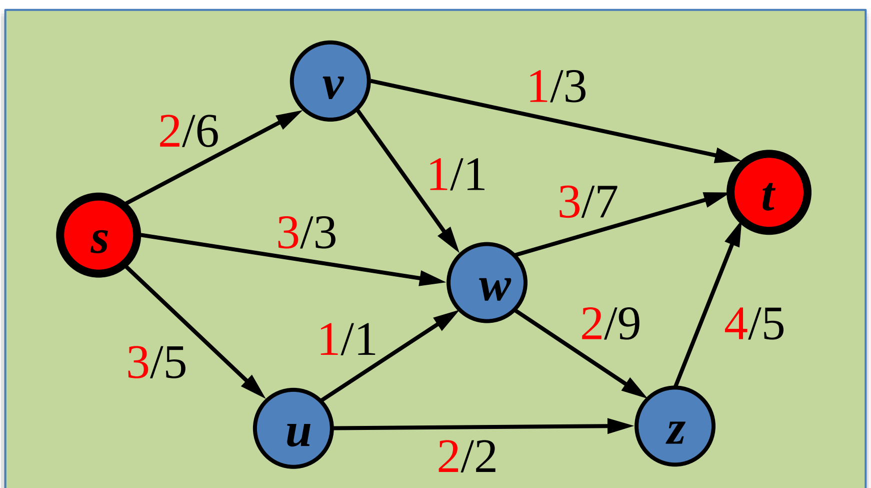
$$\Delta_f(s, v) = ?$$

Augmenting paths

We are given a network G with edge capacities c and a flow f.
Let (u, v) be an edge from u to v.

Residual capacity from u to v is $\Delta_f(u,v) = c(u,v) - f(u,v)$.

Residual capacity from v to u is $\Delta_f(v,u) = f(u,v)$



$$\Delta_f(s,v) = 4$$

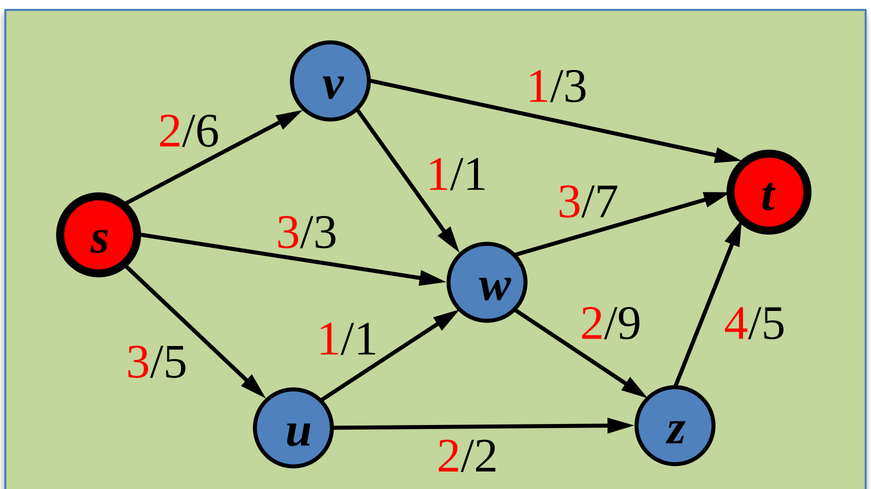
$$\Delta_f(v,w) = ?$$

Augmenting paths

We are given a network G with edge capacities c and a flow f.
Let (u, v) be an edge from u to v.

Residual capacity from u to v is $\Delta_f(u, v) = c(u, v) - f(u, v)$.

Residual capacity from v to u is $\Delta_f(v, u) = f(u, v)$



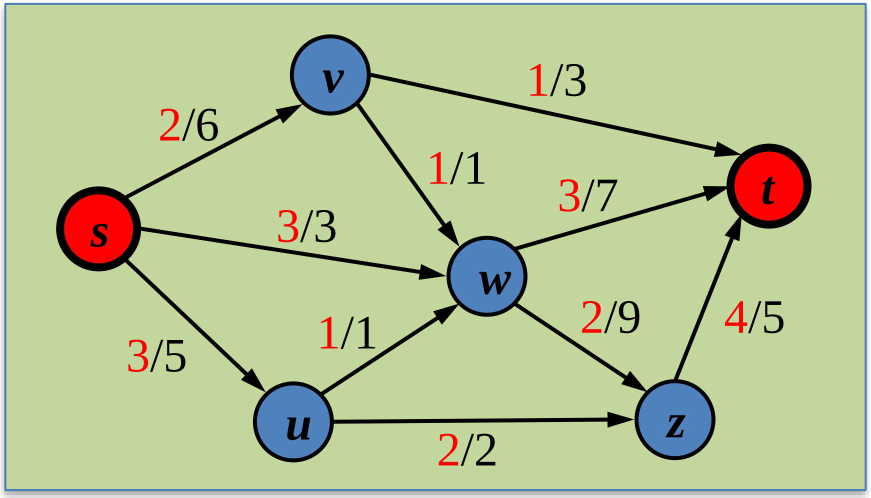
$$\Delta_f(s, v) = 4$$
$$\Delta_f(v, w) = 0$$
$$\Delta_f(w, u) = ?$$

Augmenting paths

We are given a network G with edge capacities c and a flow f.
Let (u, v) be an edge from u to v.

Residual capacity from u to v is $\Delta_f(u,v) = c(u,v) - f(u,v)$.

Residual capacity from v to u is $\Delta_f(v,u) = f(u,v)$



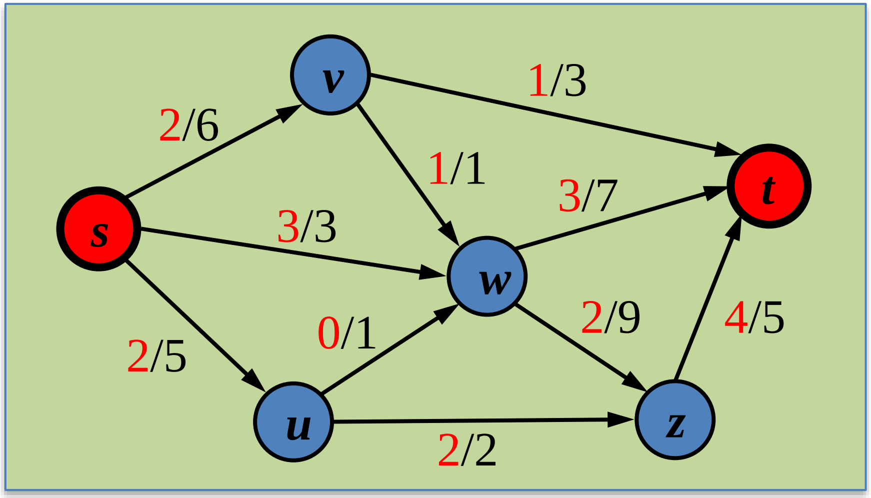
$$\Delta_f(s,v) = 4$$
$$\Delta_f(v,w) = 0$$
$$\Delta_f(w,u) = 1$$

Augmenting paths

We are given a network G with edge capacities c and a flow f.
Let (u, v) be an edge from u to v.

Residual capacity from u to v is $\Delta_f(u, v) = c(u, v) - f(u, v)$.

Residual capacity from v to u is $\Delta_f(v, u) = f(u, v)$



Augmenting path: Path from $\boldsymbol{s}$ to $\boldsymbol{t}$ with positive residual capacities.

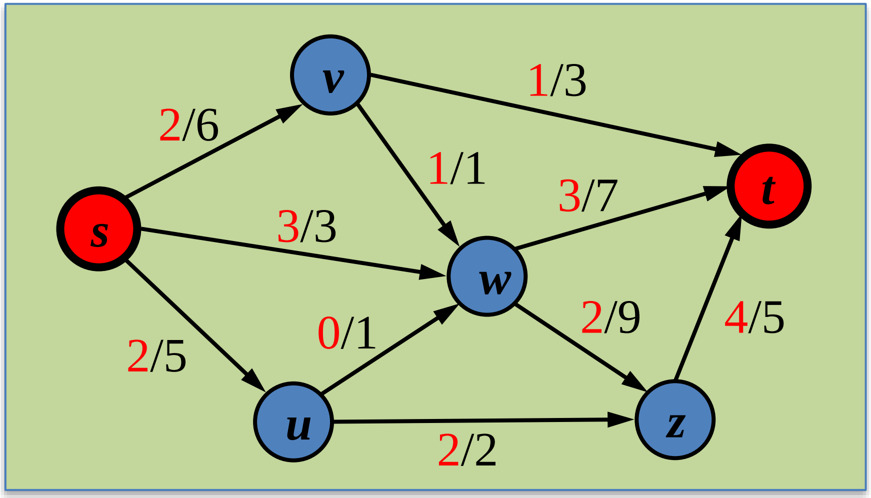
$s \to v \to t$ augmenting path
$s \to u \to w \to v \to t$ augmenting path

Augmenting paths

We are given a network G with edge capacities c and a flow f.
Let (u, v) be an edge from u to v.

Residual capacity from u to v is $\Delta_f(u, v) = c(u, v) - f(u, v)$.

Residual capacity from v to u is $\Delta_f(v, u) = f(u, v)$



Augmenting path: Path from ***s*** to ***t*** with positive residual capacities.

$s \to v \to t$ augmenting path
$s \to u \to w \to v \to t$ augmenting path

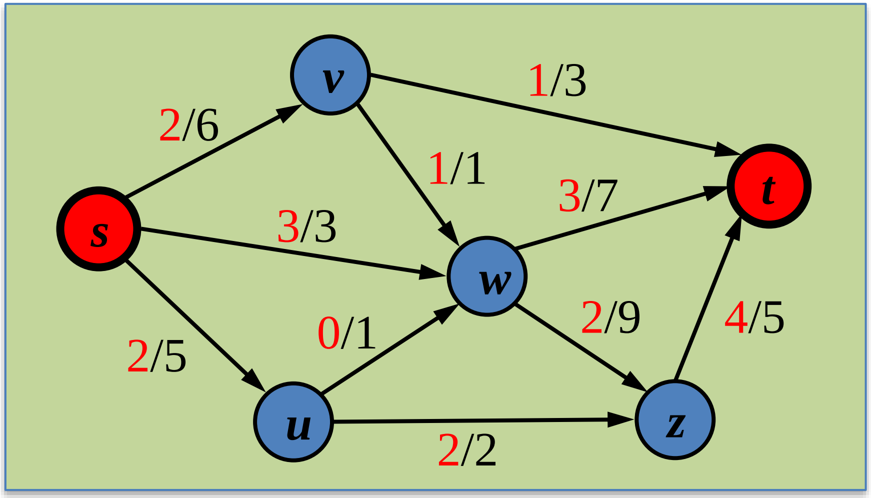
$s \to u \to z \to t$ is **not**

Augmenting paths

We are given a network G with edge capacities c and a flow f.
Let (u, v) be an edge from u to v.

Residual capacity from u to v is $\Delta_f(u, v) = c(u, v) - f(u, v)$.

Residual capacity from v to u is $\Delta_f(v, u) = f(u, v)$



$s \to v \to t$: 2 units of flow can be pushed (min over residual capacities).

$s \to u \to w \to v \to t$: 1 unit of flow can be pushed

$s \to u \to z \to t$: No flow can be pushed

The Ford-Fulkerson Algorithm

Main idea: Repeatedly search for an augmenting path $\boldsymbol{\pi}$:

- If there is an augmenting path, augment flow with $\Delta_f(\pi)$ (minimum residual capacity among the edges of π) along the edges of π.

The Ford-Fulkerson Algorithm

Main idea: Repeatedly search for an augmenting path $\boldsymbol{\pi}$:

- If there is an augmenting path, augment flow with $\Delta_f(\pi)$ (minimum residual capacity among the edges of π) along the edges of π.
- If there is no augmenting path, terminate.

Remark: You can use DFS (or BFS) to search for an augmenting path.

Running time: ?

The Ford-Fulkerson Algorithm

Main idea: Repeatedly search for an augmenting path $\boldsymbol{\pi}$:

- If there is an augmenting path, augment flow with $\Delta_f(\pi)$ (minimum residual capacity among the edges of π) along the edges of π.
- If there is no augmenting path, terminate.

Remark: You can use DFS (or BFS) to search for an augmenting path.

Running time:

Time to search for an augmenting path $\times$ number of updates.

The Ford-Fulkerson Algorithm

Main idea: Repeatedly search for an augmenting path $\boldsymbol{\pi}$:

- If there is an augmenting path, augment flow with $\Delta_f(\pi)$ (minimum residual capacity among the edges of π) along the edges of π.
- If there is no augmenting path, terminate.

Remark: You can use DFS (or BFS) to search for an augmenting path.

Running time:

Time to search for an augmenting path × number of updates.

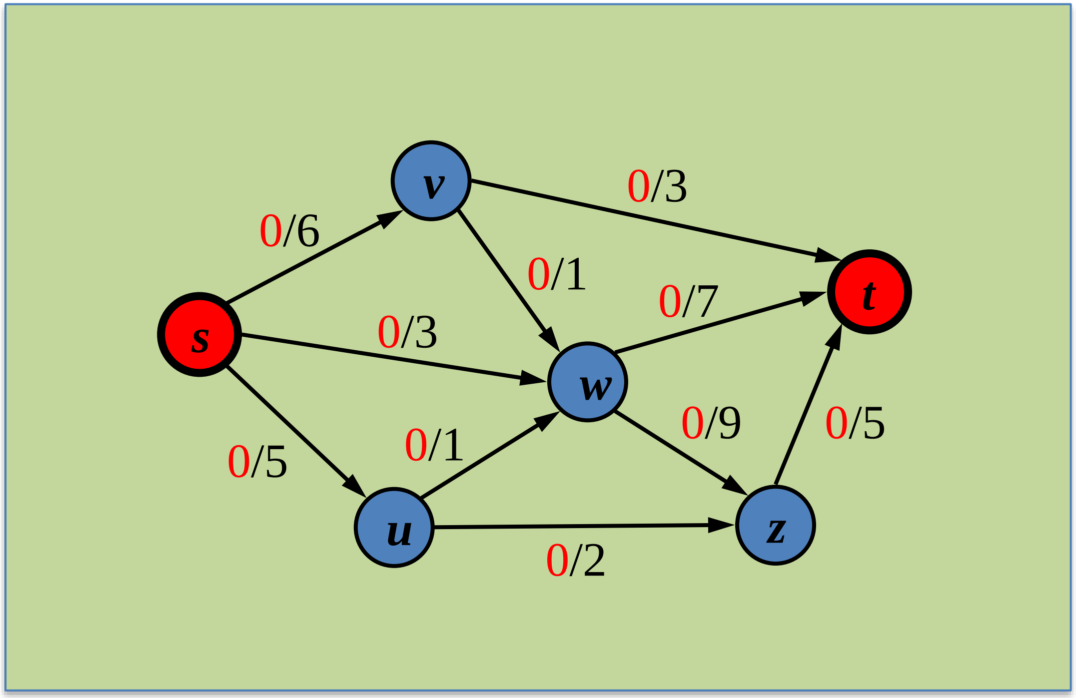
$$\Theta(|V| + |E|) \cdot |f^*|$$

Running time of DFS or BFS

Updates increase flow by 1 unit only

The Ford-Fulkerson Algorithm

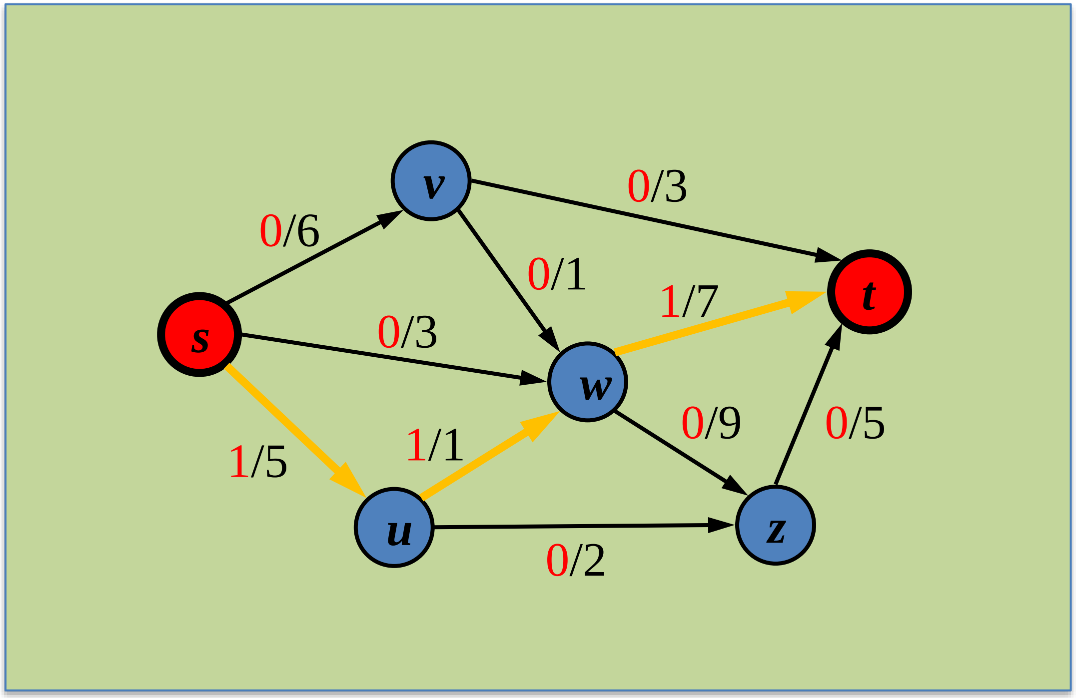
Example:



Total Flow $|f| = 0$

The Ford-Fulkerson Algorithm

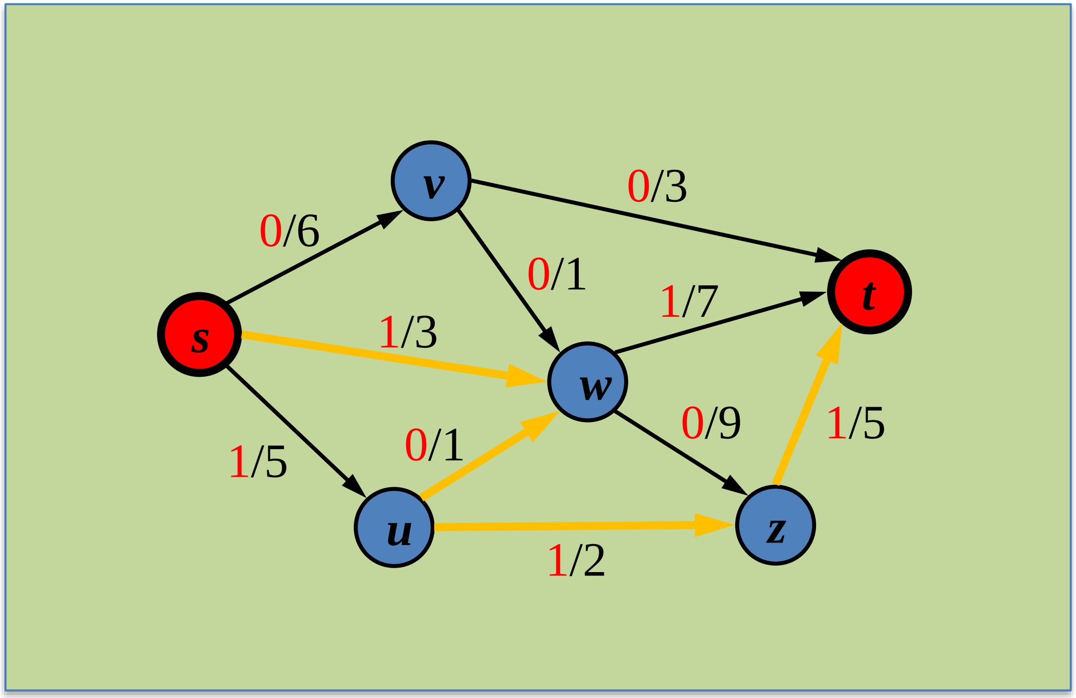
Example:



Total Flow $|f| = 1$

The Ford-Fulkerson Algorithm

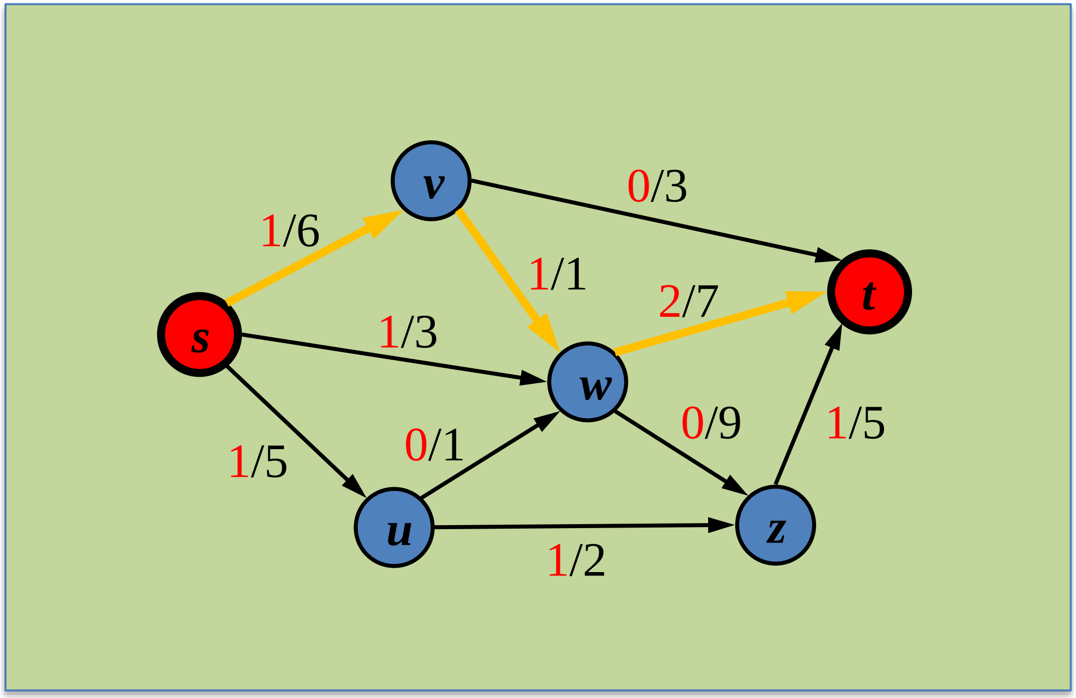
Example:



Total Flow $|f| = 2$

The Ford-Fulkerson Algorithm

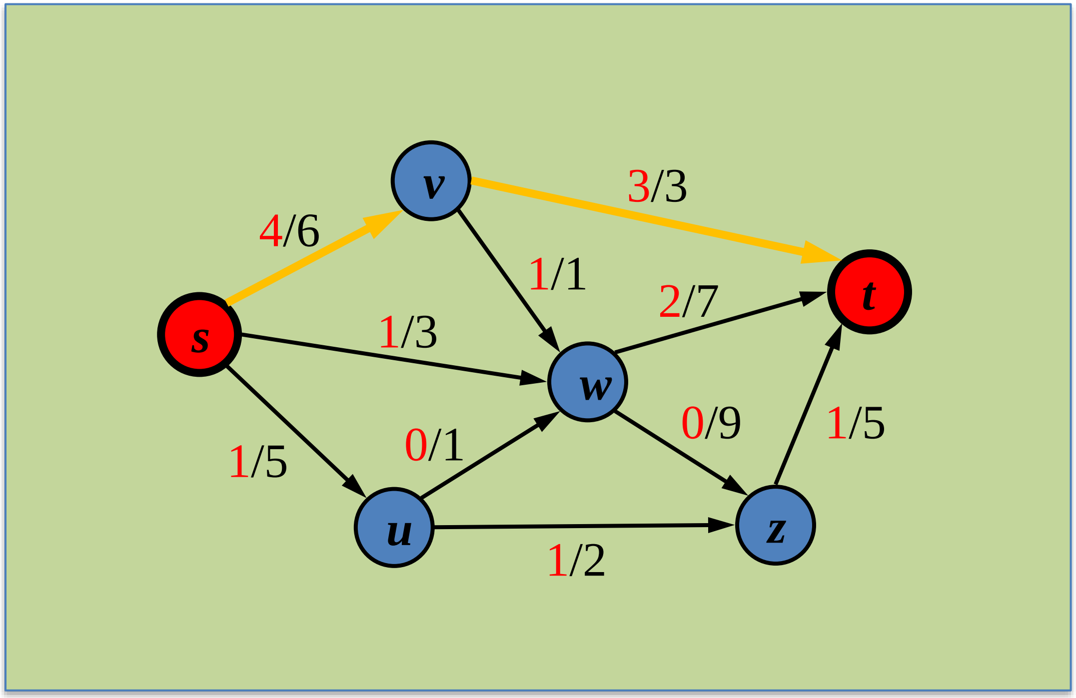
Example:



Total Flow $|f| = 3$

The Ford-Fulkerson Algorithm

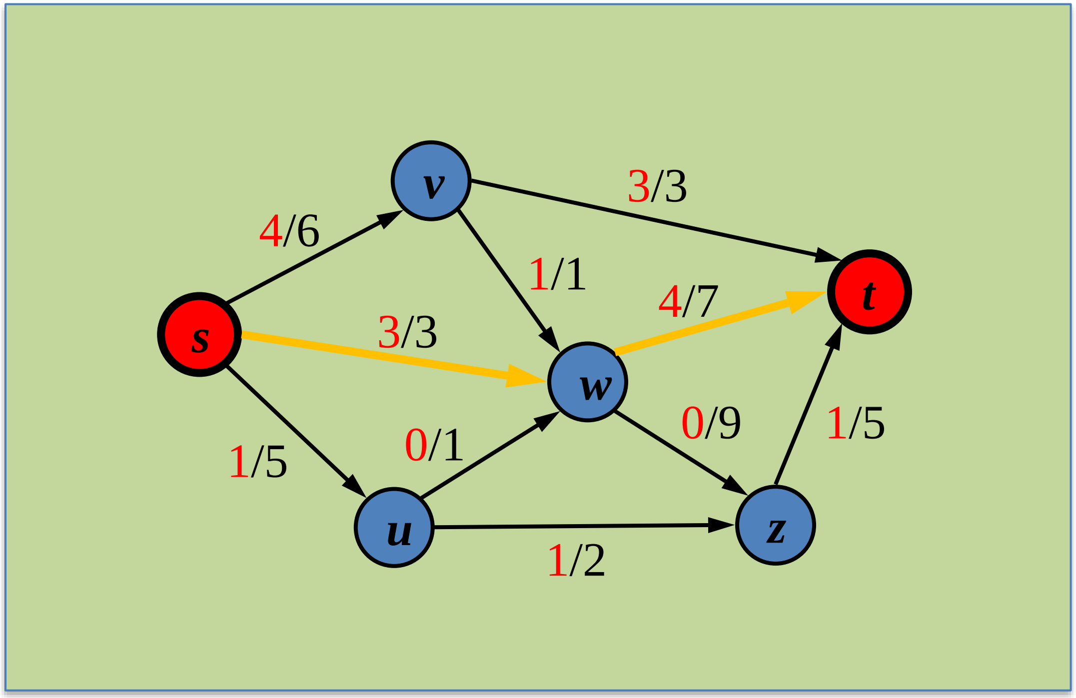
Example:



$\text{Total Flow } |f| = 6$

The Ford-Fulkerson Algorithm

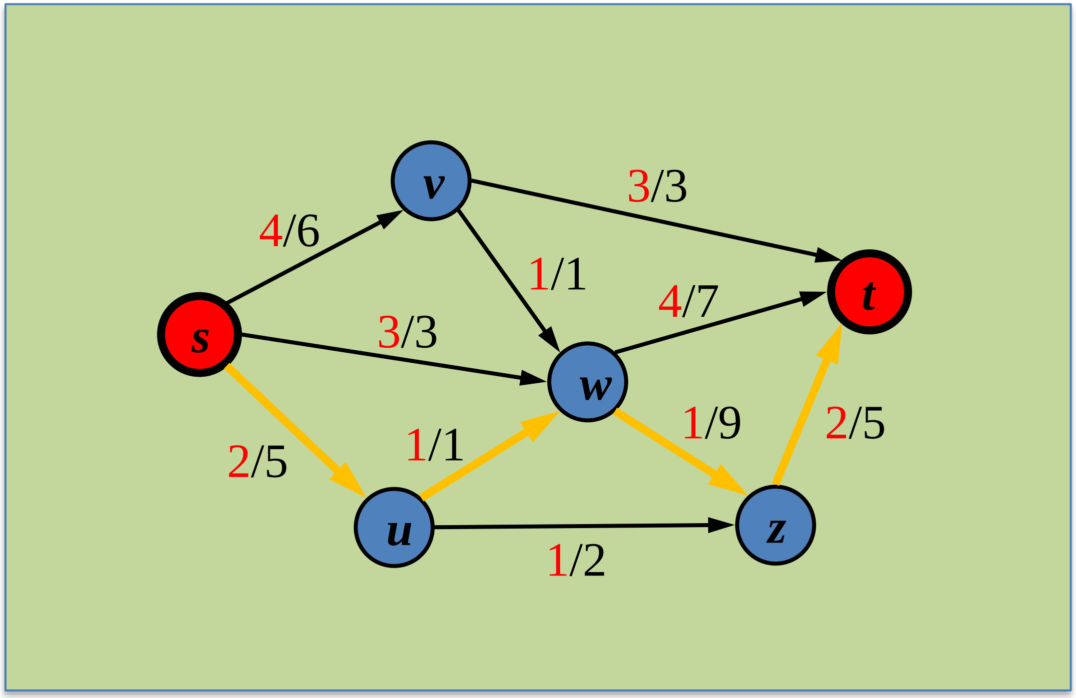
Example:



$$\text{Total Flow } |f| = 8$$

The Ford-Fulkerson Algorithm

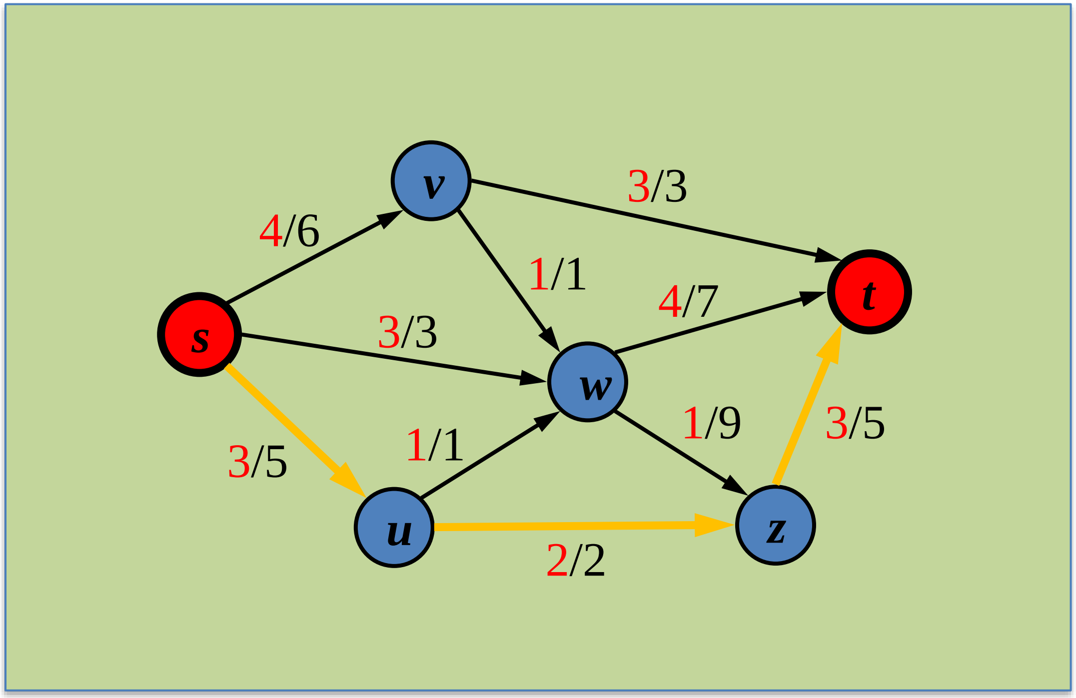
Example:



Total Flow $|f| = 9$

The Ford-Fulkerson Algorithm

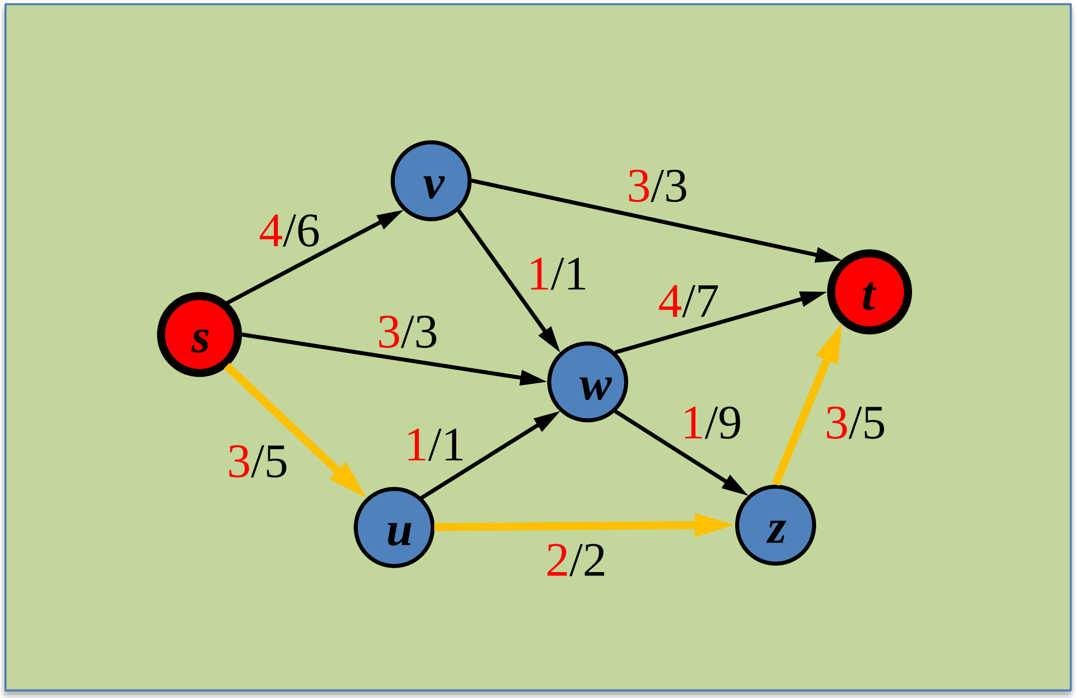
Example:



$$\text{Total Flow } |f| = 10$$

The Ford-Fulkerson Algorithm

Example:



$$\text{Total Flow } |f| = 10$$

No more augmenting paths!

The Ford-Fulkerson Algorithm

Pseudocode:

```
Algorithm MaxFlowFordFulkerson
   Input: Flow network (G, c, s, t)
   Output: A maximum flow f
  for each edge e
       f(e) ← 0
  stop ← false
  repeat
       traverse G starting at s to find an augmenting path for f
       if an augmenting path π exists then
            // Compute the residual capacity Δ_f(π) of π
            Δ ← +∞
            for each edge e ∈ π do
                 if Δ_f(e) < Δ then
                      Δ ← Δ_f(e)
            for each edge e ∈ π do  // push Δ = Δ_f(π) units along π
                 if e is a forward edge then
                      f(e) ← f(e) + Δ
                 else
                      f(e) ← f(e) − Δ        // e is a backward edge
       else
            stop ← true        // f is a maximum flow
  until stop
```

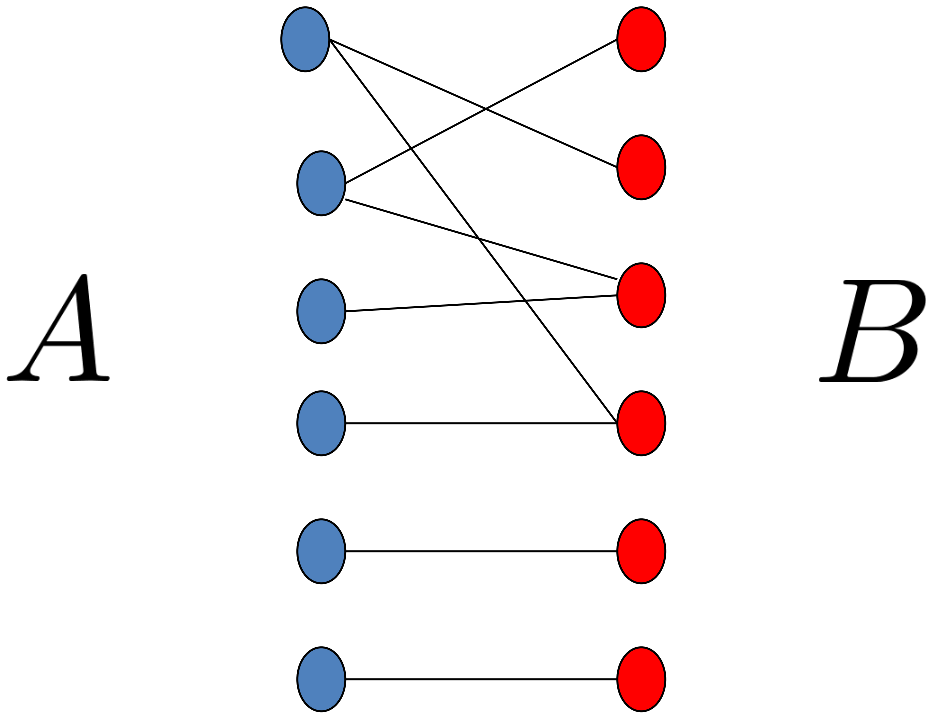
Initialization $f = 0$

Δ: min residual capacity on aug. path

Update flow on aug. path

No more aug. paths

Application: Maximum Matching

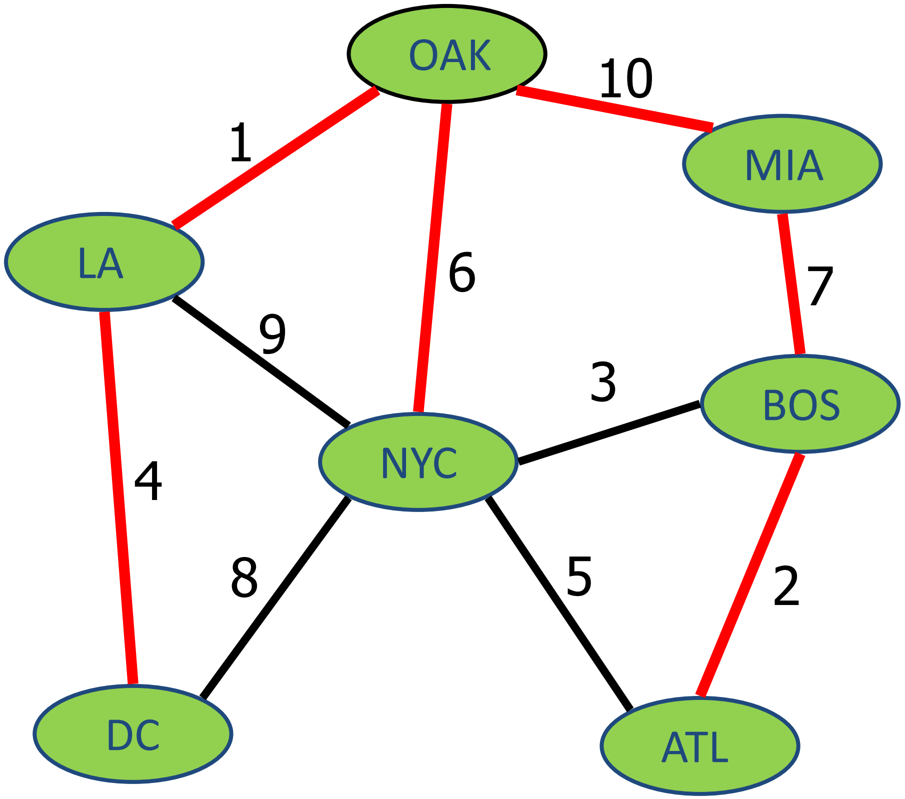


Definition: Given a **bipartite** graph, a **matching** is just a collection of edges that do not share a vertex.

Spanning Tree

Definition: We are given an undirected, weighted graph G. A spanning tree of G is a connected acyclic (tree) subgraph of G that includes all the vertices of G (spanning).

Example:



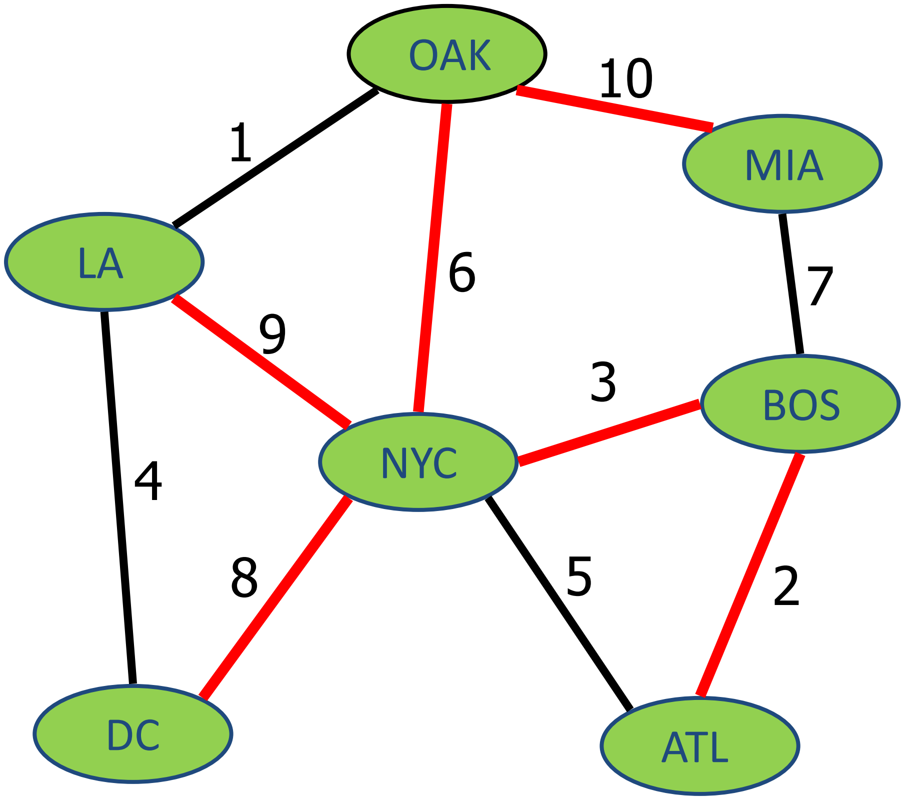
Total cost

$4+1+10+6+7+2 = 30$

Spanning Tree

Definition: We are given an undirected, weighted graph G. A spanning tree of G is a connected acyclic (tree) subgraph of G that includes all the vertices of G (spanning).

Example:



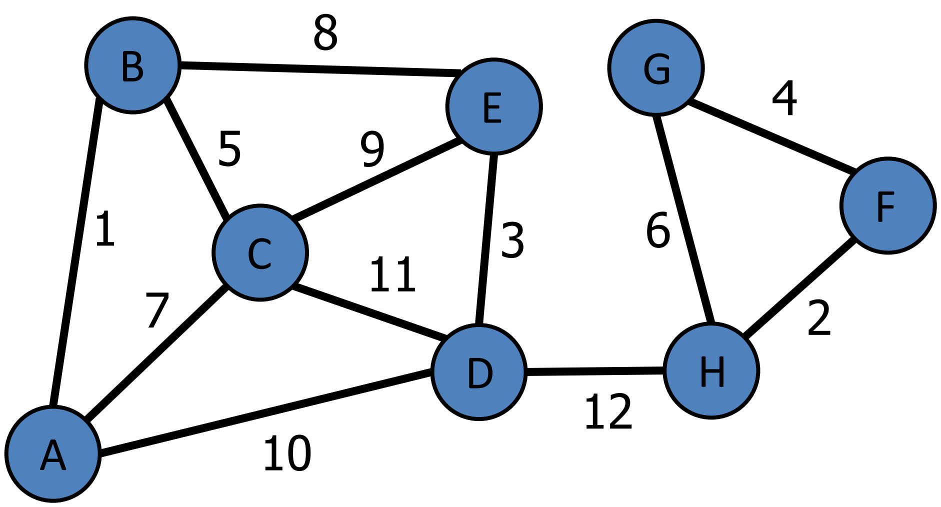
Total cost

$8+9+6+10+3+2 = 38$

Kruskal's Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

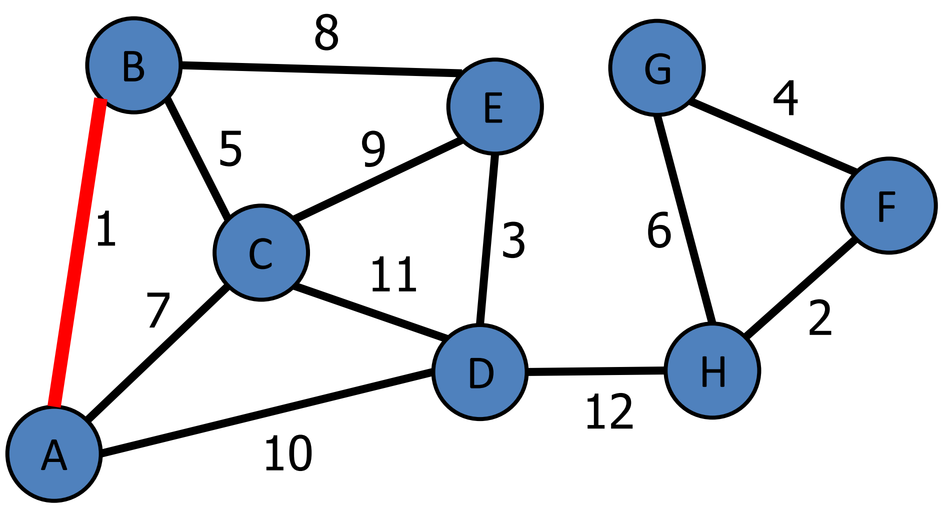
Example:



Kruskal's Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

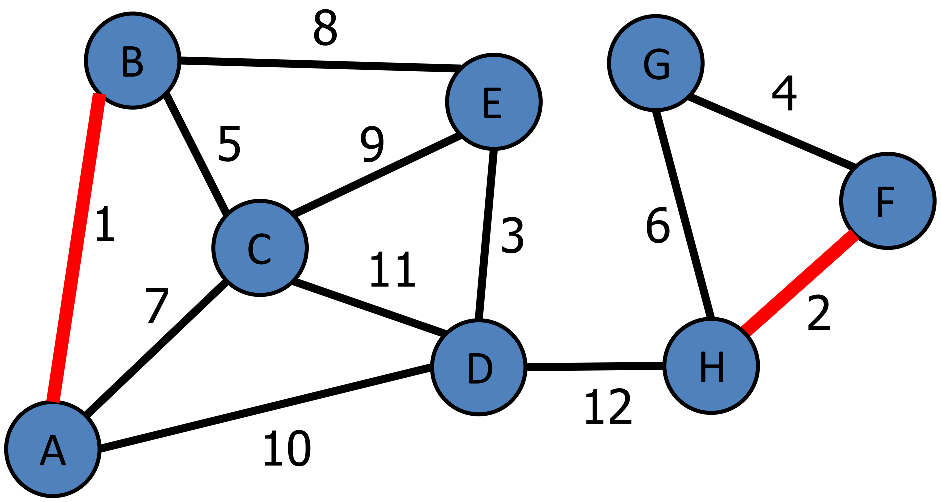
Example:



Kruskal’s Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

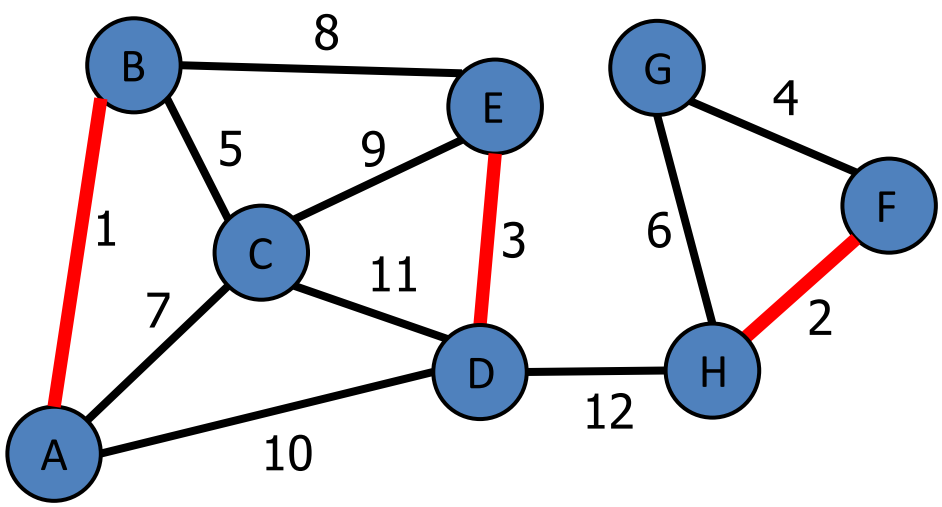
Example:



Kruskal’s Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

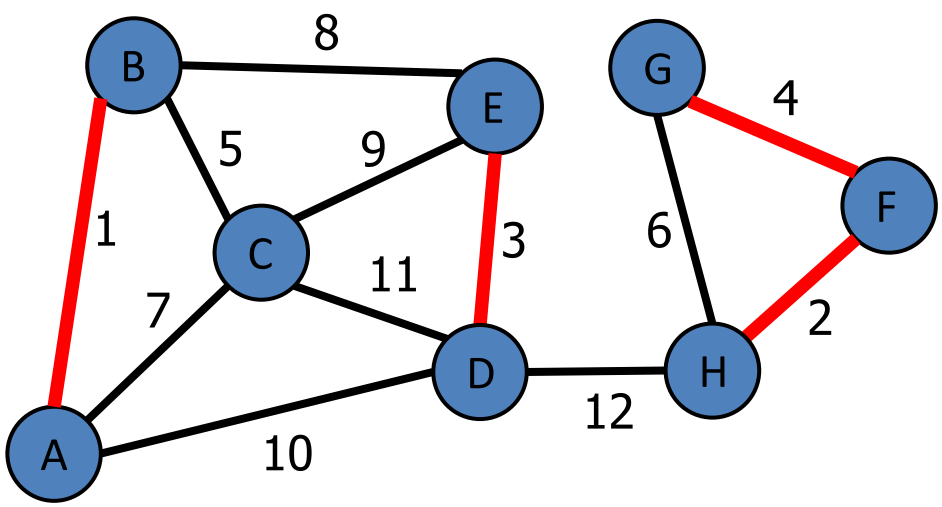
Example:



Kruskal's Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

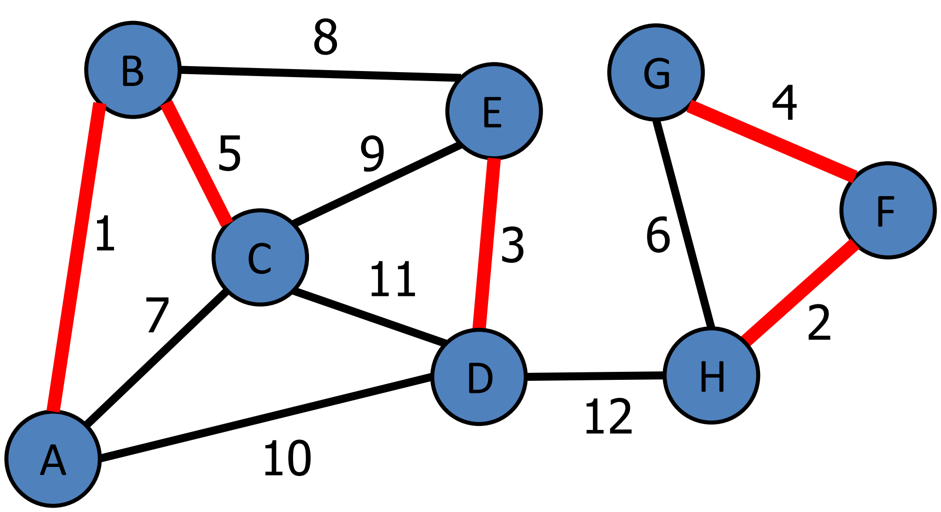
Example:



Kruskal's Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

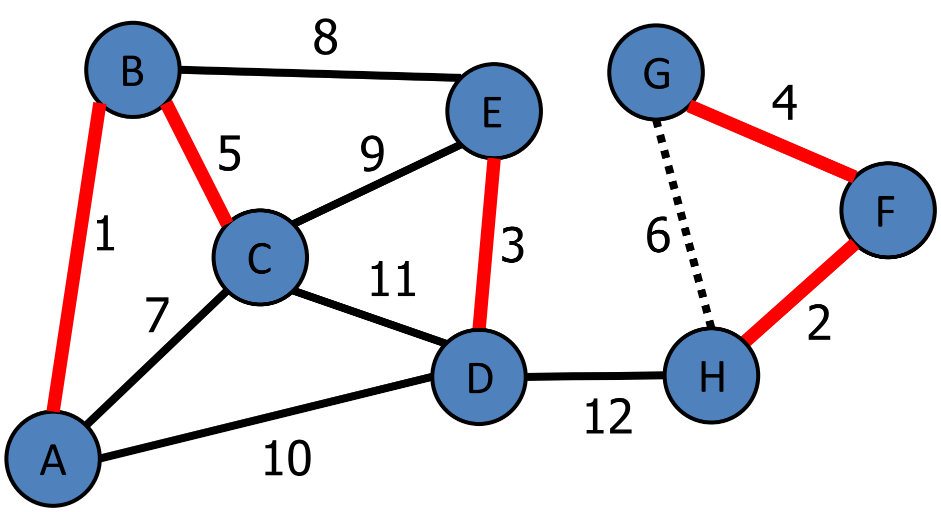
Example:



Kruskal's Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

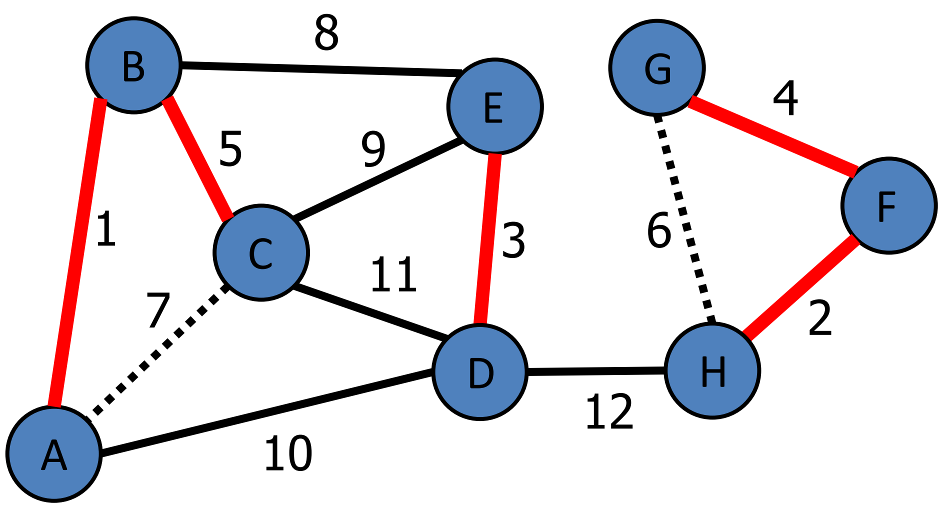
Example:



Kruskal’s Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

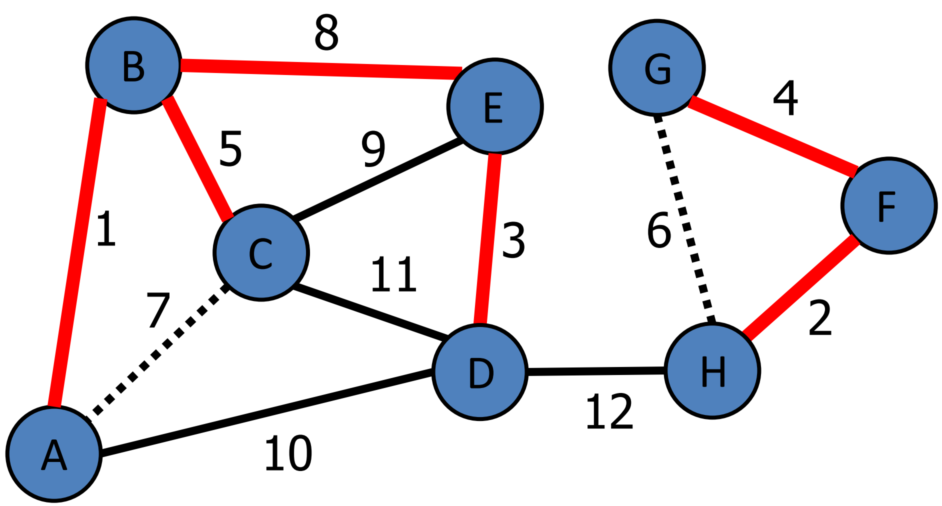
Example:



Kruskal's Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

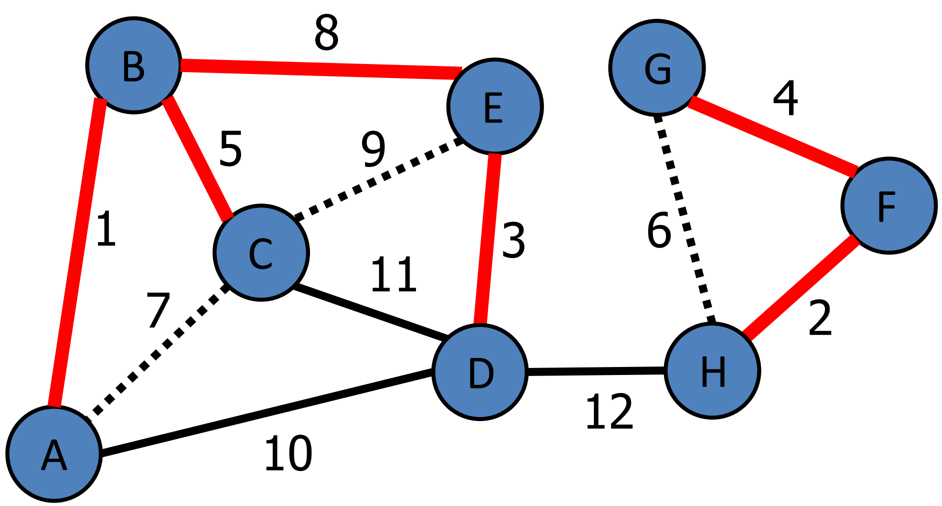
Example:



Kruskal's Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

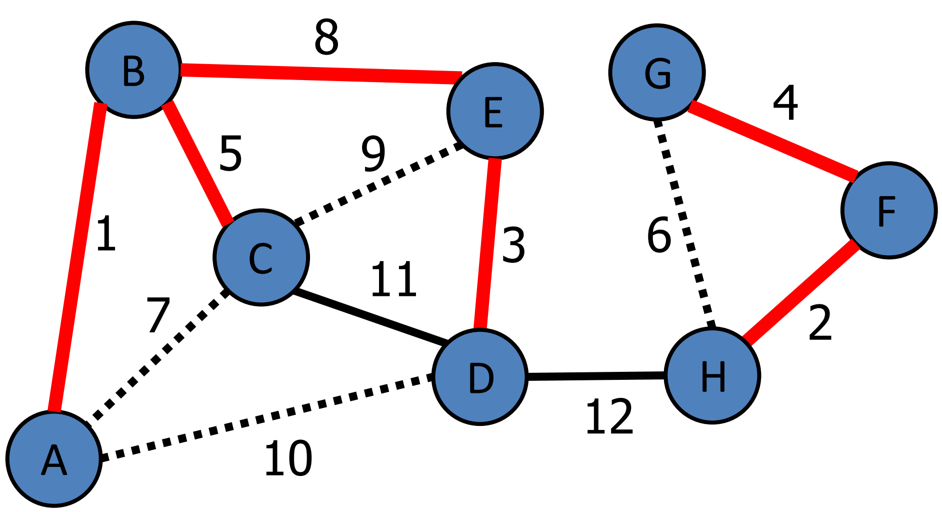
Example:



Kruskal's Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

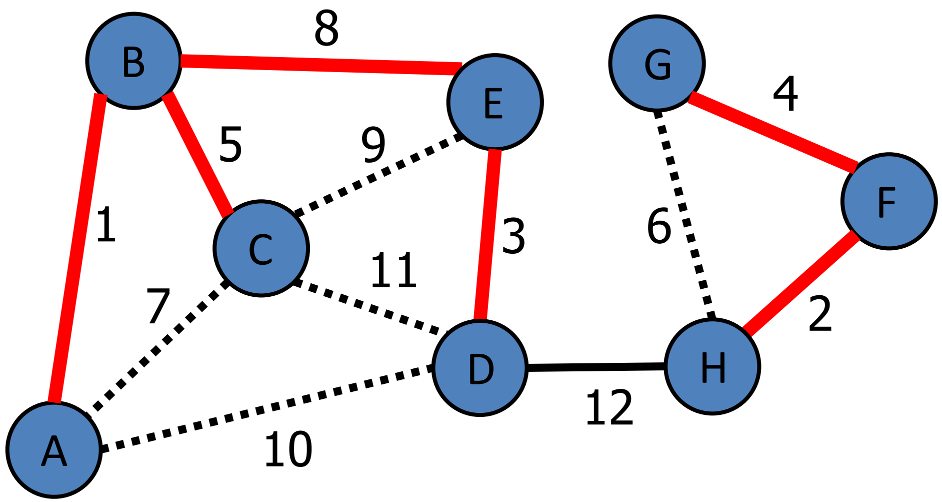
Example:



Kruskal's Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

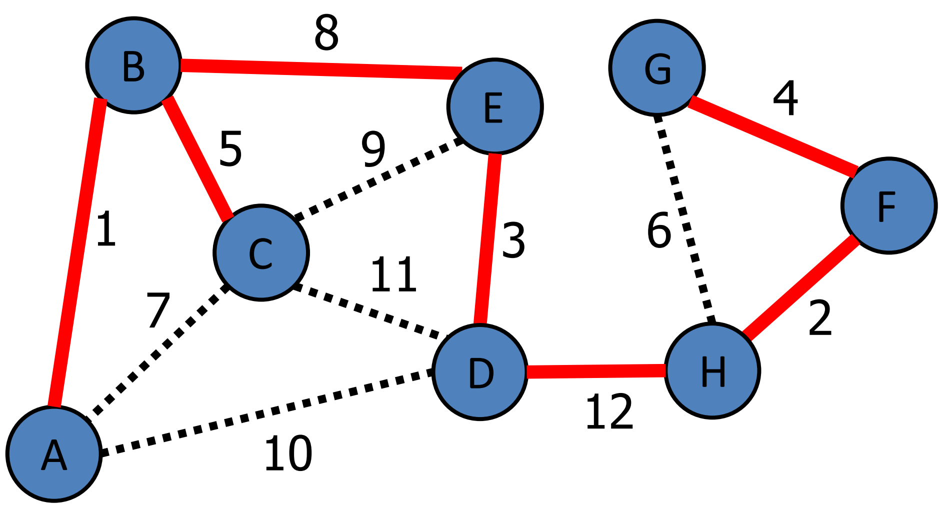
Example:



Kruskal's Algorithm for MSTs

Idea 1: Greedy approach. Consider the edges from smaller weight to larger. Include each edge in the current solution as long as it does not create a cycle, otherwise discard it.

Example:



Total cost

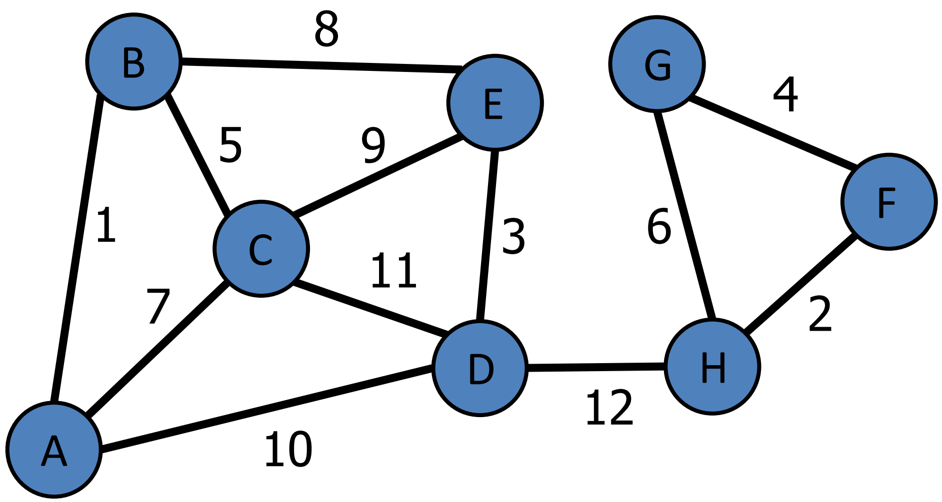
$1+2+3+4+5+8+12 = 35$

Prim's Algorithm for MSTs

Idea 2: Similar to Dijkstra's algorithm. We pick an arbitrary vertex s. At each step:

- We add to the current tree the vertex u with the smallest $d[u]$ and the corresponding incident to u edge.
- We update the labels of the vertices adjacent to u.

$d[A] = 0$
$d[B] = \infty$
$d[C] = \infty$
$d[D] = \infty$
$d[E] = \infty$
$d[F] = \infty$
$d[G] = \infty$
$d[H] = \infty$

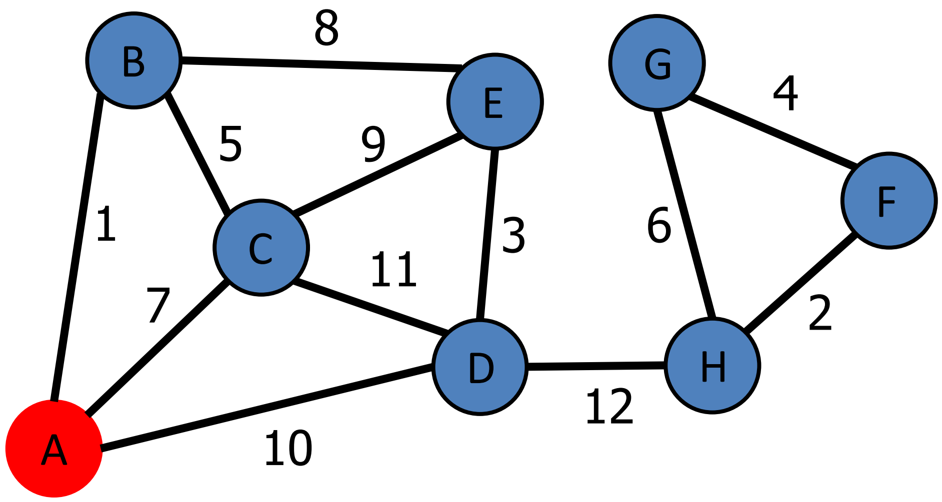


Prim's Algorithm for MSTs

Idea 2: Similar to Dijkstra's algorithm. We pick an arbitrary vertex s. At each step:

- We add to the current tree the vertex u with the smallest $d[u]$ and the corresponding incident to u edge.
- We update the labels of the vertices adjacent to u.

$d[A] = 0$
$\mathbf{d[B]=1}$
$\mathbf{d[C]=7}$
$\mathbf{d[D]=10}$
$d[E] = \infty$
$d[F] = \infty$
$d[G] = \infty$
$d[H] = \infty$

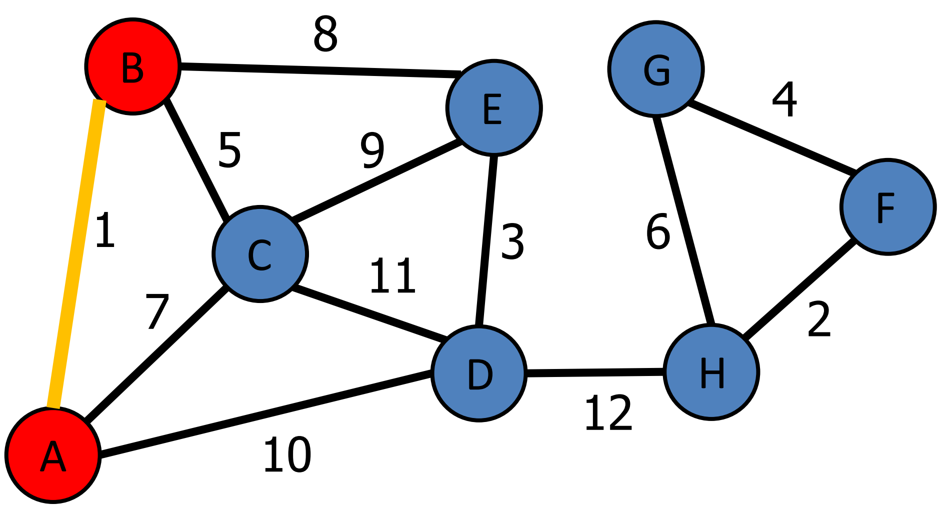


Prim's Algorithm for MSTs

Idea 2: Similar to Dijkstra's algorithm. We pick an arbitrary vertex s. At each step:

- We add to the current tree the vertex u with the smallest $d[u]$ and the corresponding incident to u edge.
- We update the labels of the vertices adjacent to u.

$d[A] = 0$
$d[B] = 1$
$\mathbf{d[C]=5}$
$d[D] = 10$
$\mathbf{d[E]=8}$
$d[F] = \infty$
$d[G] = \infty$
$d[H] = \infty$

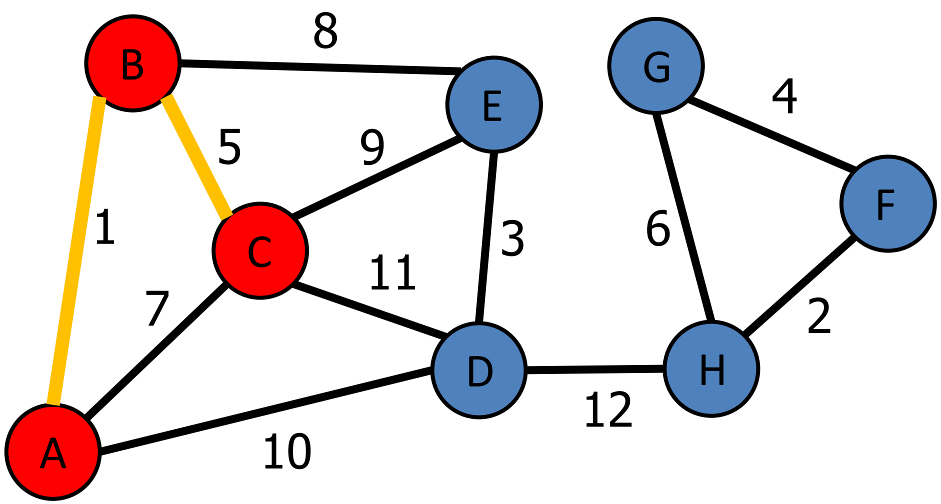


Prim's Algorithm for MSTs

Idea 2: Similar to Dijkstra's algorithm. We pick an arbitrary vertex s. At each step:

- We add to the current tree the vertex u with the smallest $d[u]$ and the corresponding incident to u edge.
- We update the labels of the vertices adjacent to u.

$d[A] = 0$
$d[B] = 1$
$d[C] = 5$
$d[D] = 10$
$d[E] = 8$
$d[F] = \infty$
$d[G] = \infty$
$d[H] = \infty$

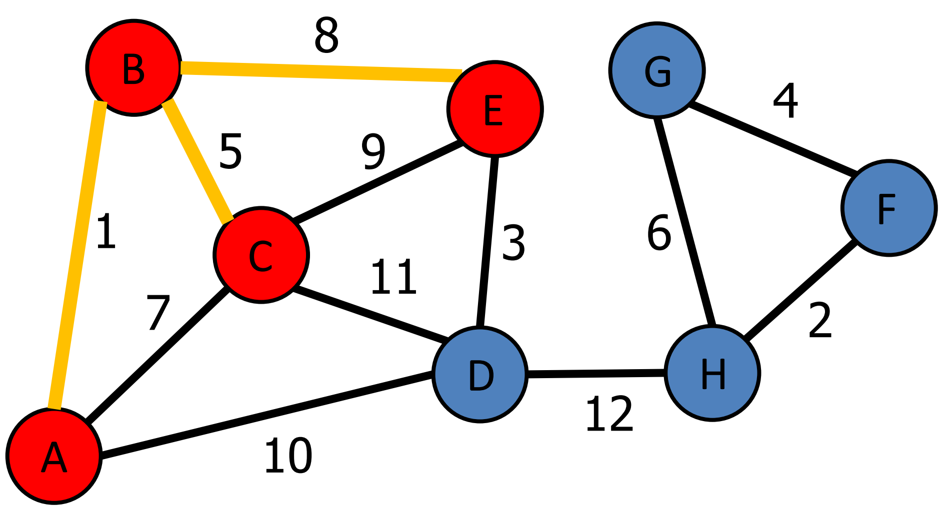


Prim's Algorithm for MSTs

Idea 2: Similar to Dijkstra's algorithm. We pick an arbitrary vertex s. At each step:

- We add to the current tree the vertex u with the smallest $d[u]$ and the corresponding incident to u edge.
- We update the labels of the vertices adjacent to u.

$d[A] = 0$
$d[B] = 1$
$d[C] = 5$
$\mathbf{d[D]=3}$
$d[E] = 8$
$d[F] = \infty$
$d[G] = \infty$
$d[H] = \infty$

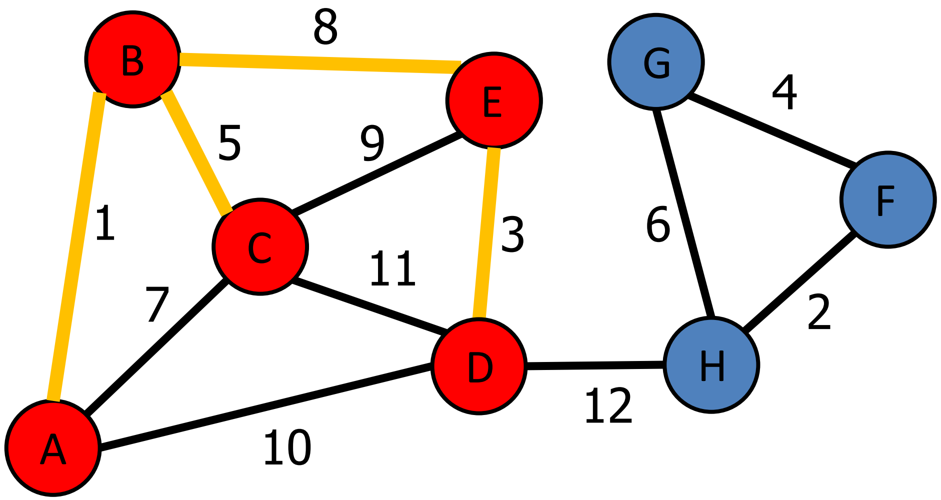


Prim's Algorithm for MSTs

Idea 2: Similar to Dijkstra's algorithm. We pick an arbitrary vertex s. At each step:

- We add to the current tree the vertex u with the smallest $d[u]$ and the corresponding incident to u edge.
- We update the labels of the vertices adjacent to u.

$d[A] = 0$
$d[B] = 1$
$d[C] = 5$
$d[D] = 3$
$d[E] = 8$
$d[F] = \infty$
$d[G] = \infty$
$\mathbf{d[H]=12}$

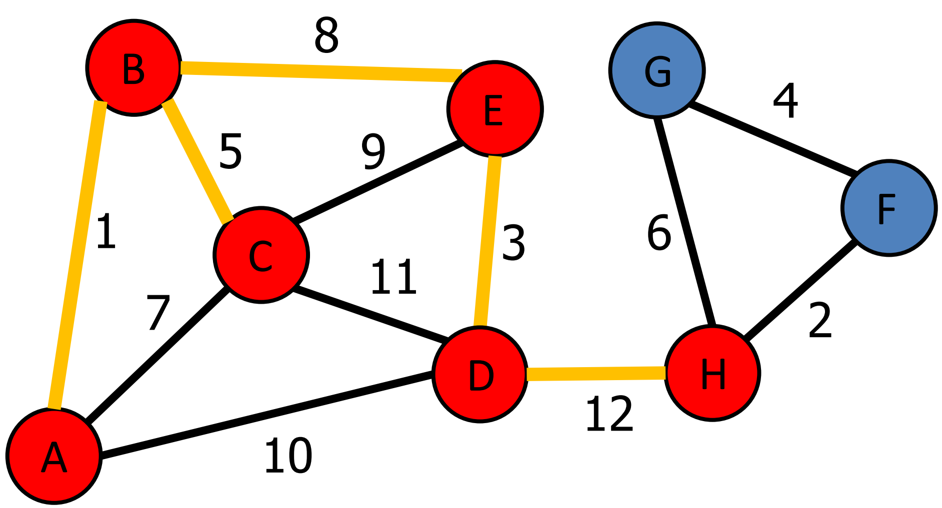


Prim's Algorithm for MSTs

Idea 2: Similar to Dijkstra's algorithm. We pick an arbitrary vertex s. At each step:

- We add to the current tree the vertex u with the smallest $d[u]$ and the corresponding incident to u edge.
- We update the labels of the vertices adjacent to u.

$d[A] = 0$
$d[B] = 1$
$d[C] = 5$
$d[D] = 3$
$d[E] = 8$
$\mathbf{d[F]=2}$
$\mathbf{d[G]=6}$
$d[H] = 12$

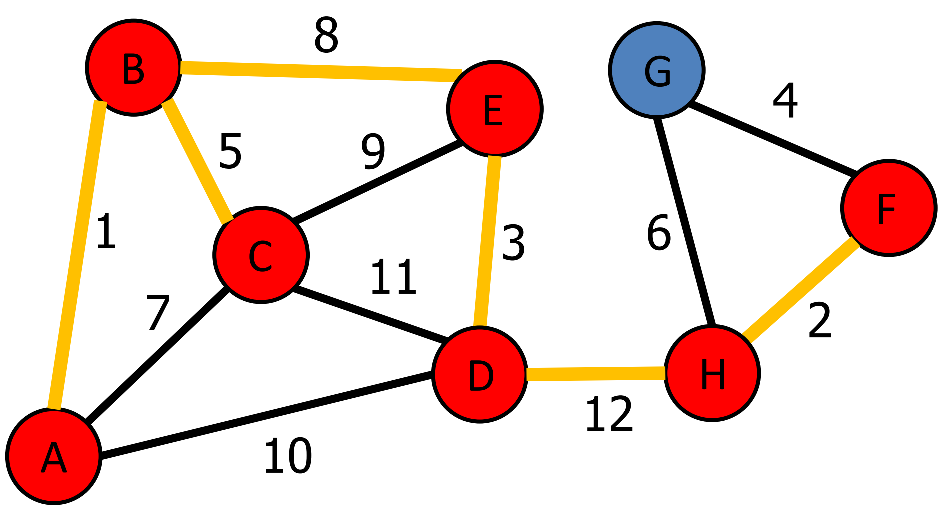


Prim's Algorithm for MSTs

Idea 2: Similar to Dijkstra's algorithm. We pick an arbitrary vertex s. At each step:

- We add to the current tree the vertex u with the smallest $d[u]$ and the corresponding incident to u edge.
- We update the labels of the vertices adjacent to u.

$d[A] = 0$
$d[B] = 1$
$d[C] = 5$
$d[D] = 3$
$d[E] = 8$
$d[F] = 2$
$\mathbf{d[G]=4}$
$d[H] = 12$



Prim's Algorithm for MSTs

Idea 2: Similar to Dijkstra's algorithm. We pick an arbitrary vertex s. At each step:

- We add to the current tree the vertex u with the smallest $d[u]$ and the corresponding incident to u edge.
- We update the labels of the vertices adjacent to u.

$d[A] = 0$
$d[B] = 1$
$d[C] = 5$
$d[D] = 3$
$d[E] = 8$
$d[F] = 2$
$d[G] = 4$
$d[H] = 12$