Data Structure & Algorithms

Lec 16: Algorithms Design Techniques (Greedy Algorithms)

Fall 2017 - University of Windsor

Dr. Sherif Saad

Outlines

Introduction To Greedy Algorithms

Greedy Algorithms Design

Solving Problems with Greedy Algorithms

Greedy Algorithm: Definition

What is Greedy Search Algorithm?

A problem solving approach that solve the problem by making locally optimal choice at each stage with the hope of finding a global optimum.

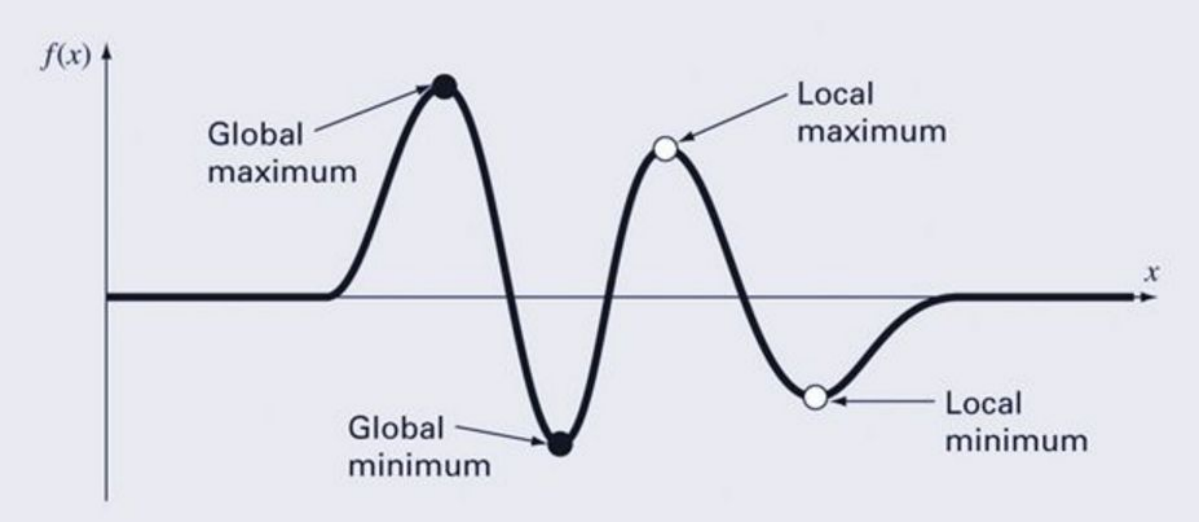
It is an optimistic approach for problem solving.

Making the best choice at each step using the available information.

Will never reconsider a decision once made.

Greedy Algorithm: Motivation

Ability to find good enough solution for complex combinatorial problems in reasonable time where traditional techniques usually fail.

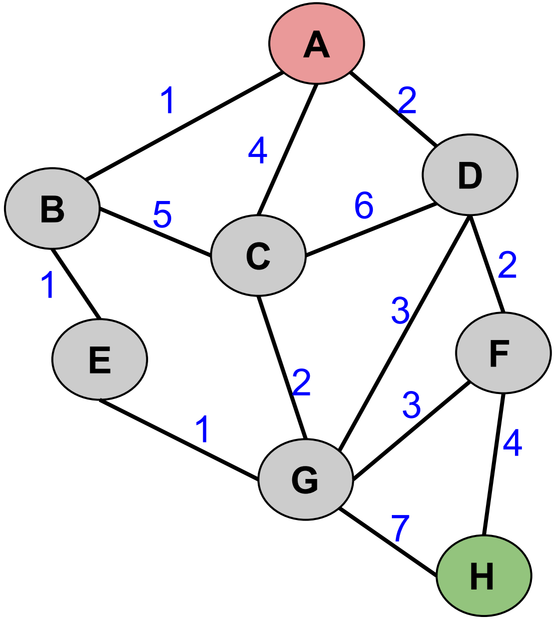


Greedy Search Algorithm: Example

Using greedy search find the path from node A to node H?

How is greedy search is different from Dijkstra?

What are the advantages of greedy search?



Greedy Algorithm: Main Idea

- I wish
  - I had some way to order my choices by value
  - I had some way of making the next choice.
  - greed works for my problem
- Advantages:
  - Greed is easy to implement
  - Less thinking and (cheaper), time complexity
- Disadvantages:
  - Design a correct greedy solution is hard
  - Verifying (proving) the correctness of greedy is hard
  - Doesn't work for all the problems

Greedy Algorithm: When??

Works very well for problems with the following properties:

Optimal Substructures: the problem can be divided into similar sub problems and the global solution can be reconstructed efficiently form optimal solutions of the sub-problmes.

Greedy Property Exist: at any point you are making a choice that seems to be the best at this point. You will never have to reconsider your earlier choices. A global optimal solution can be reached by choosing the optimal choice at each step.

Note: no guaranteed way to recognize problems that can be solved by a greedy algorithm

Greedy Algorithm: How??

Identify an optimal substructure or subproblem in the problem.

Design a selection criteria (what the solution will include e.g the largest sum, the shortest path, etc.)

Create a procedure (iterative or recursive) to go through all of the subproblems and build a solution

Greedy Algorithm: Why it Fails?

For some problem the idea of selecting the local optimal solutions does not guarantee finding the global optimum.

The greedy algorithm it is not aware of future choices it could make. It use only the current available information to find the local optimal, and it does not reconsider its decision.

How to Evaluate a Greedy Algorithm?

Correctness:

- Feasibility: you need to show that the algorithm produces a feasible solution, a solution to the problem that fulfills the constraints.
- Optimality: you need to show that the algorithm produces the optimal solution, (e.g., maximizes or minimizes). Mostly using mathematical induction.

To show that a greedy algorithm is not correct you only need one counter example.

Greedy Algorithm: Example

Things to do in Vancouver. You are visiting Vancouver for one day only here is the list of the places you can visit in Vancouver during this day and the time it will take you to visit each place.

You only have 8 hours and your objective is to visit as many places as you can

Things to do	Time = 8 hours
Stanley Park	2.5 hours
Capilano Suspension Bridge	1.5 hours
Vancouver Aquarium	3 hours
Science World	2 hours
Canada Place	1 hours
English Bay	½ hours
Grouse Mountain	3 hours

Greedy Algorithm: Vancouver Visit

Design a greedy algorithm to maximize the number of things you can do during your visit.

Greedy Solution Outlines:

- Sort the the things you can do in an ascending order based on the time.
- Pick an item from the sorted list and do it and mark it as done
- Update your available time based on the items you marked as done
- Repeat the above steps as long as the available time is bigger that the time of the item at the head of the list

Greedy Search Algorithm: Vancouver Visit

Is the proposed greedy solution complete?

Does it find the optimal solution?

What if we add pleasure value to each place, and we want to maximize the pleasure gain based on the available time

Things to do	Time = 8 hours	Order
Stanley Park	2.5 hours	5
Capilano Suspension Bridge	1.5 hours	3
Vancouver Aquarium	3 hours	6
Science World	2 hours	4
Canada Place	1 hours	2
English Bay	½ hours	1
Grouse Mountain	3 hours	7

Greedy Search Algorithm: Vancouver Visit

Use a greedy search algorithm to solve the vancouver visit considering a pleasure where you want to maximize the number of places you can visit and gain the maximum pleasure

Things to do	Time	Pleasure
Stanley Park	2.5	4
Capilano Suspension Bridge	1.5	1.5
Vancouver Aquarium	3	2
Science World	2	5
Canada Place	1	3
English Bay	0.5	1
Grouse Mountain	3	6

Greedy Search Algorithm: Vancouver Visit

Use the pleasure per unit of time as your objective function

Sort the places based on the objective measure in descending order

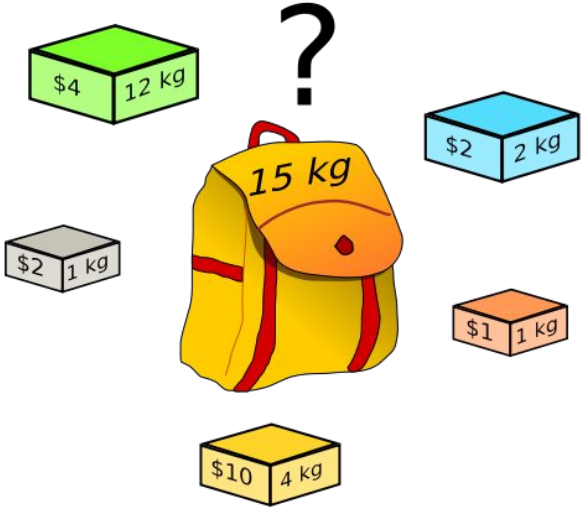
Then visit the places on the sorted list in order

Things to do	Time	Pleasure	Pleasure/time
Stanley Park	2.5	4	1.6
Capilano Bridge	1.5	1.5	1.0
Vancouver Aquarium	3	2	0.66
Science World	2	5	2.5
Canada Place	1	3	3
English Bay	0.5	1	2
Grouse Mountain	3	6	2

0-1 Knapsack Problem: Definition

Given weights and values of ***n*** items, put these items in a knapsack of capacity ***W*** to get the maximum total value in the knapsack.

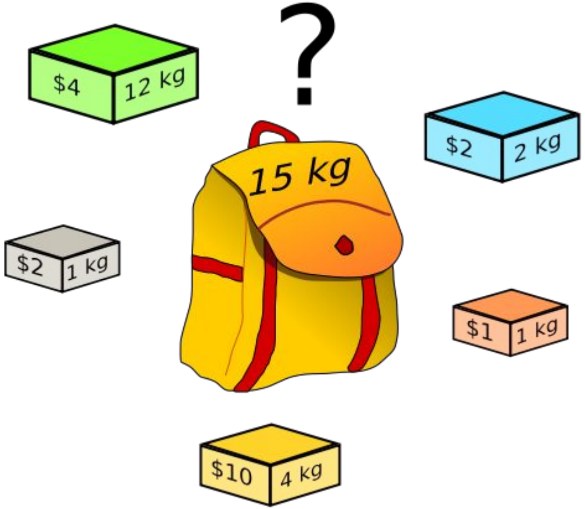
This an example of combinatorial optimization problem.



0-1 Knapsack Problem: Example

Assume we have the following set of items and a knapsack of capacity = 9. Find the items that maximize the profit with the capacity limit

Item	A	B	C	D	E	F	G
Profit	6	5	8	9	6	7	3
Weight	2	3	6	7	5	9	4

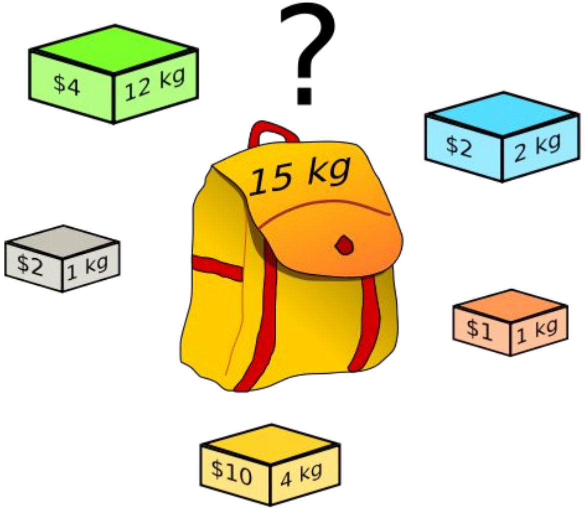


How can we solve this problem using greedy algorithm?

0-1 Knapsack Problem: Greedy Solution

Calculate the ratio of profit to weight

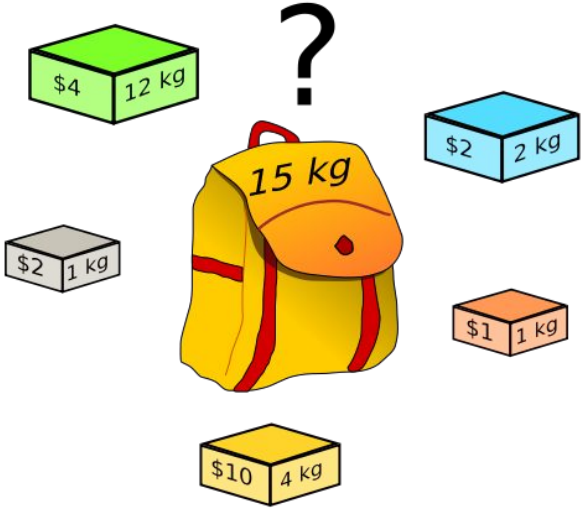
Item	A	B	C	D	E	F	G
Profit	6	5	8	9	6	7	3
Weight	2	3	6	7	5	9	4
P/W	3	1.67	1.33	1.29	1.2	0.78	0.75



The solution using greedy method is [A, B, G] , where the profit = 14 and the weight = 9 (Can you come up with a better solution?)

0-1 Knapsack Problem: Genetic Algorithm Solution

Item	A	B	C	D	E	F	G
Profit	6	5	8	9	6	7	3
Weight	2	3	6	7	5	9	4
P/W	3	1.67	1.33	1.29	1.2	0.78	0.75

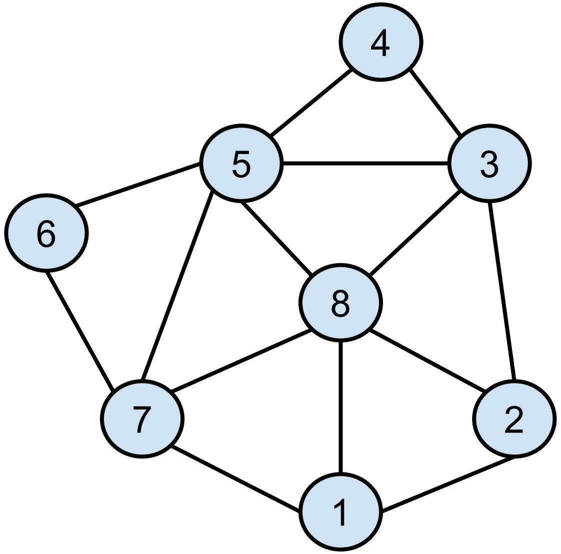


How big is the solution (search) space?

Minimum Spanning Tree

What is a spanning tree?

Given a graph G(V,E), the spanning tree is a tree in G that includes every vertex of G (span tree) and it is a subgraph of G (every edge in the tree belongs to G)



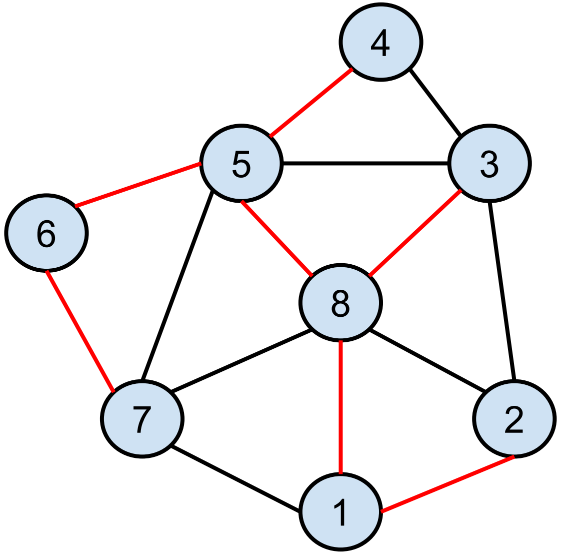
G = (V,E)

Minimum Spanning Tree

What is a spanning tree?

Given a graph G(V,E), the spanning tree is a tree in G that includes every vertex of G (span tree) and it is a subgraph of G (every edge in the tree belongs to G)

T = {(1, 8), (1, 2), (3, 8), (5, 8), (4, 5), (5, 6), (6, 7),}



G = (V,E)

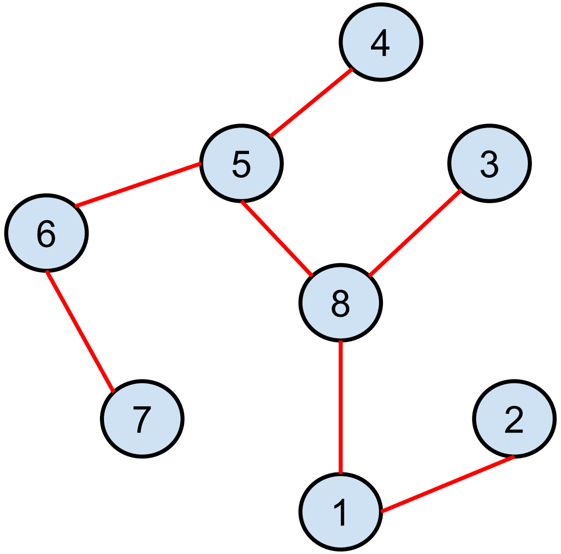
Minimum Spanning Tree

What is a spanning tree?

Given a graph G(V,E), the spanning tree is a tree in G that includes every vertex of G (span tree) and it is a subgraph of G (every edge in the tree belongs to G)

Every vertex in V(G) is in T, all the edges in E(T) are in E(G), T is connected and has no cycle. T is a spanning tree of G.

Write an algorithm to find all spanning trees in graph G



G = (V,E)

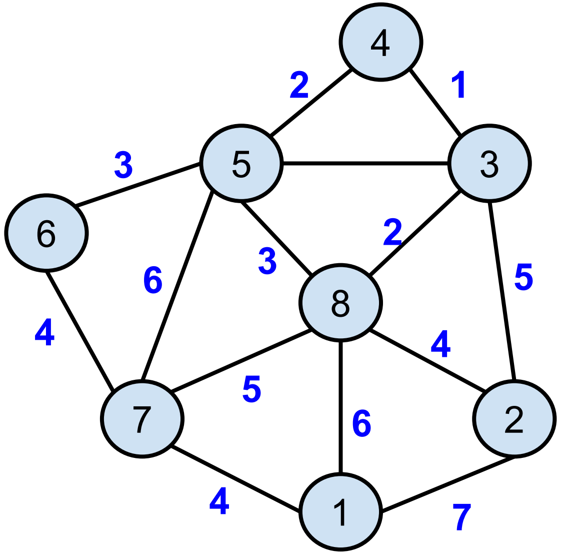
Minimum Spanning Tree

What is a minimum spanning tree?

For undirected weighted graph the cost of the spanning tree is the sum of the weights of all the edges in the tree.

A minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Is finding a minimum spanning tree is combinatorial optimization problem?



G = (V,E)

Minimum Spanning Tree

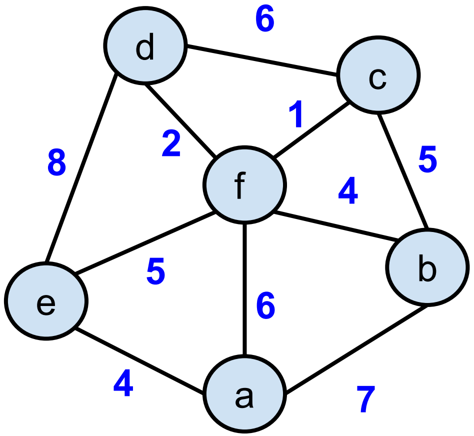
What are the applications of minimum spanning tree problem?

- Developing the best airline network of hubs and spokes.
- Determining the optimal way to deliver packages.
- Design utilities (e.g. electrical grid) networks.
- Cluster Analysis.
- Handwriting recognition.
- Image segmentation.

Minimum Spanning Tree: Greedy Algorithm

Kruskal's Algorithm

- Find the minimum spanning tree by adding edges one by one into a growing spanning tree.
- Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

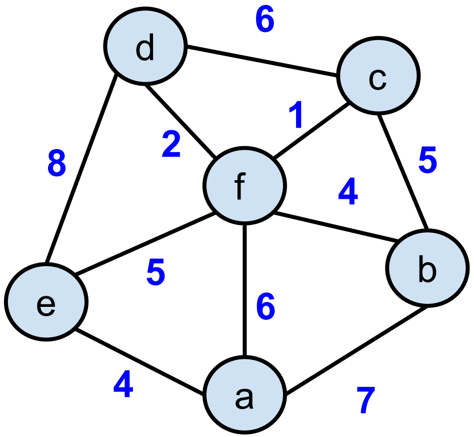


G = (V,E)

Minimum Spanning Tree: Kruskal’s Algorithm

How it works?

1. Make disjoint sets of size |V|, each vertex is a disjoint set
2. Sort the graph edges with respect to their weights is ascending order
3. Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
4. Only add edges which does not form a cycle (only add edges that do not belong to the same disjoint set)
5. Apply Union operation on the vertex disjoint set



G = (V,E)

Minimum Spanning Tree: Kruskal’s Algorithm

(a, b)	7
(a, e)	4
(a, f)	6
(b, f)	4
(b, c)	5
(e, f)	5
(c, d)	6
(c, f)	1
(d, e)	8
(d, f)	2

Disjoint Sets

S_1 = {a}

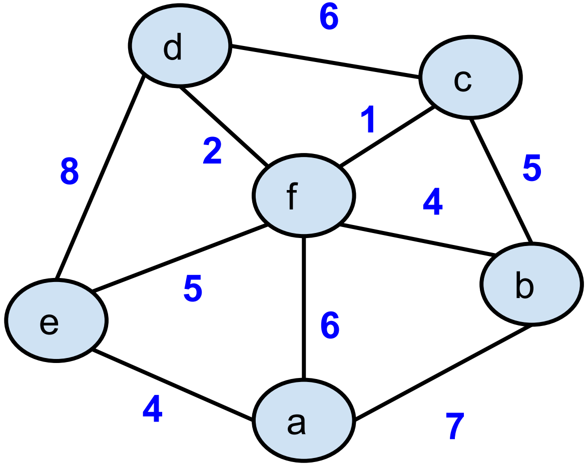
S_2 = {b}

S_3 = {c}

S_4 = {d}

S_5 = {e}

S_6 = {f}



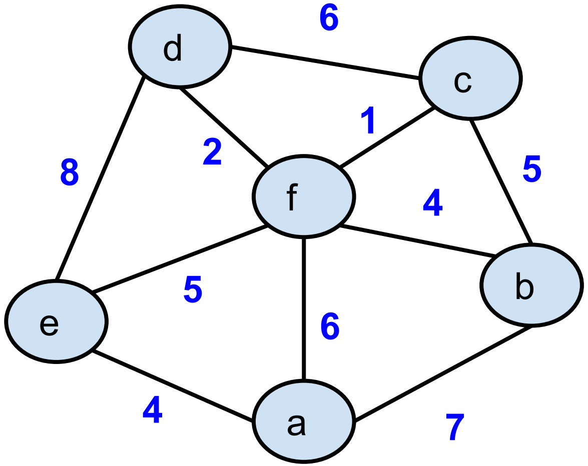
G = (V,E)

Minimum Spanning Tree: Kruskal’s Algorithm

(a, b)	7
(a, e)	4
(a, f)	6
(b, f)	4
(b, c)	5
(e, f)	5
(c, d)	6
(c, f)	1
(d, e)	8
(d, f)	2

Sort by weight

(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8



G = (V,E)

Minimum Spanning Tree: Kruskal’s Algorithm

(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8

Disjoint Sets

S_1 = {a}

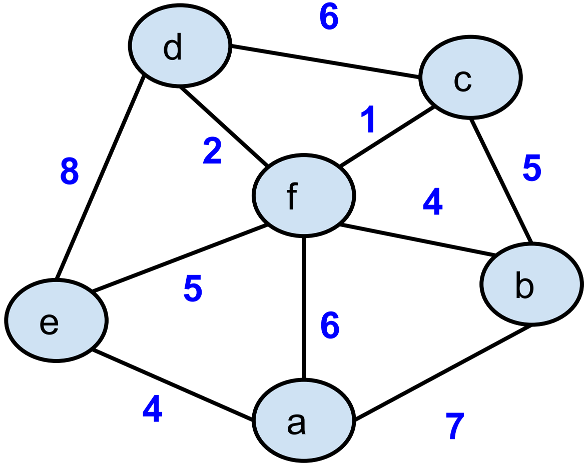
S_2 = {b}

S_3 = {c}

S_4 = {d}

S_5 = {e}

S_6 = {f}



G = (V,E)

Minimum Spanning Tree: Kruskal’s Algorithm

(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8

Disjoint Sets

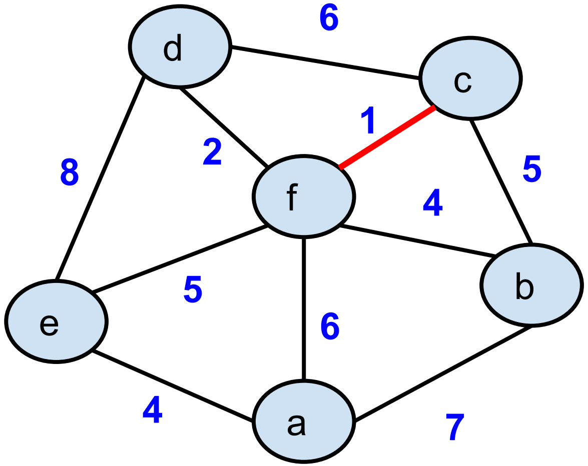
S_1 = {a}

S_2 = {b}

S_3 = {c, f}

S_4 = {d}

S_5 = {e}



G = (V,E)

Minimum Spanning Tree: Kruskal’s Algorithm

(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8

Disjoint Sets

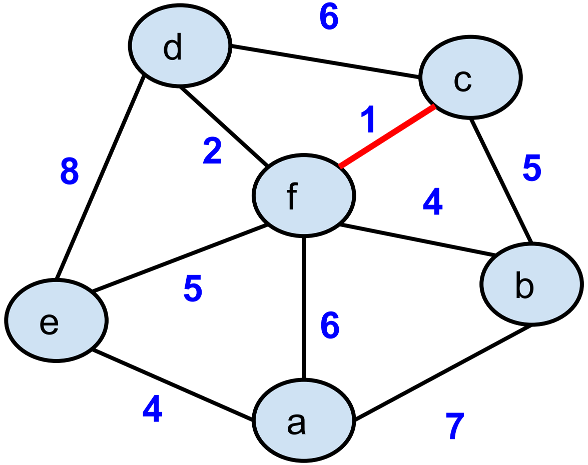
S_1 = {a}

S_2 = {b}

S_3 = {c, f}

S_4 = {d}

S_5 = {e}



G = (V,E)

Minimum Spanning Tree: Kruskal's Algorithm

(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8

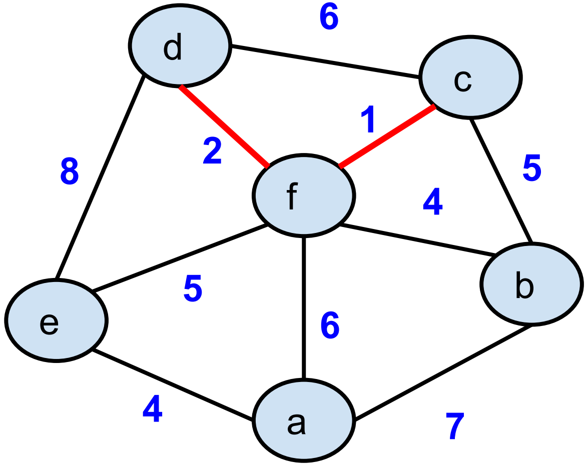
Disjoint Sets

S_1 = {a}

S_2 = {b}

S_3 = {c, f, d}

S_5 = {e}



G = (V,E)

Minimum Spanning Tree: Kruskal’s Algorithm

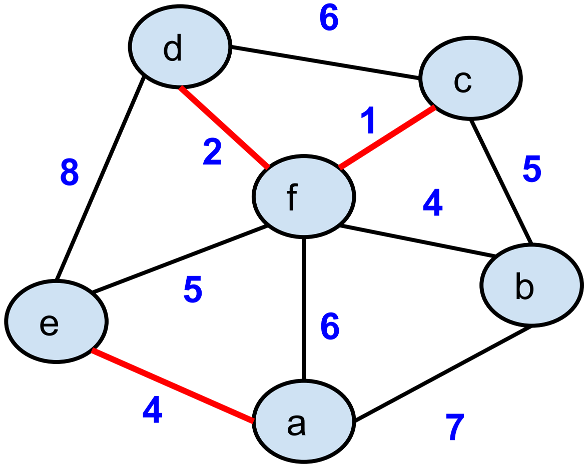
(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8

Disjoint Sets

S_1 = {a, e}

S_2 = {b}

S_3 = {c, f, d}



G = (V,E)

Minimum Spanning Tree: Kruskal’s Algorithm

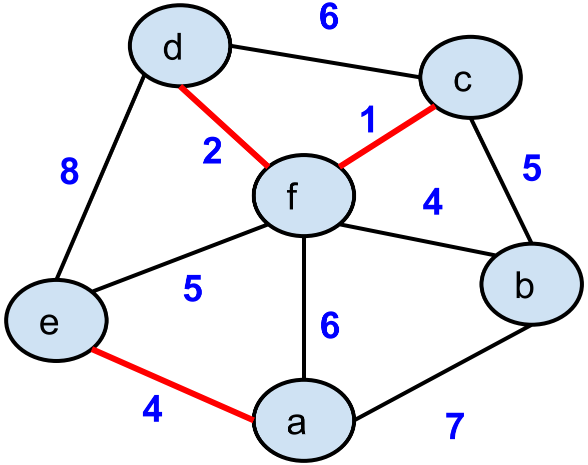
(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8

Disjoint Sets

S_1 = {a, e}

S_2 = {b}

S_3 = {c, f, d}



G = (V,E)

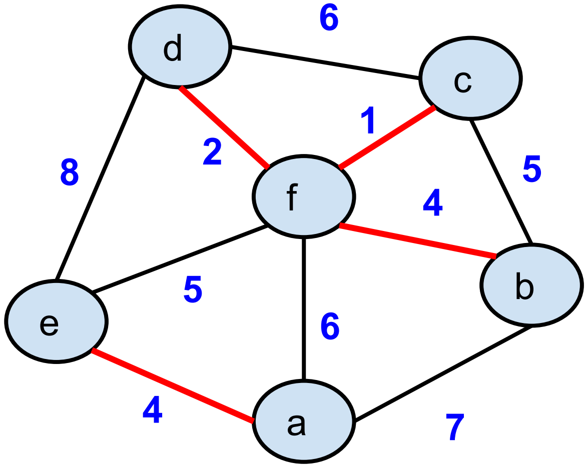
Minimum Spanning Tree: Kruskal's Algorithm

(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8

Disjoint Sets

$S_1 = \{a, e\}$

$S_3 = \{c, f, d, b\}$



G = (V,E)

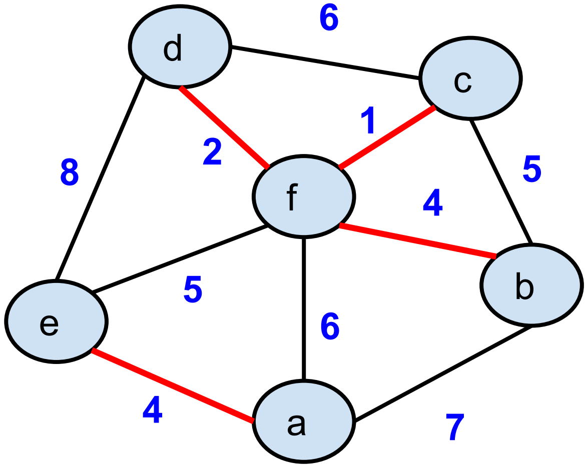
Minimum Spanning Tree: Kruskal’s Algorithm

(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8

Disjoint Sets

S_1 = {a, e}

S_3 = {c, f, d, b}



G = (V,E)

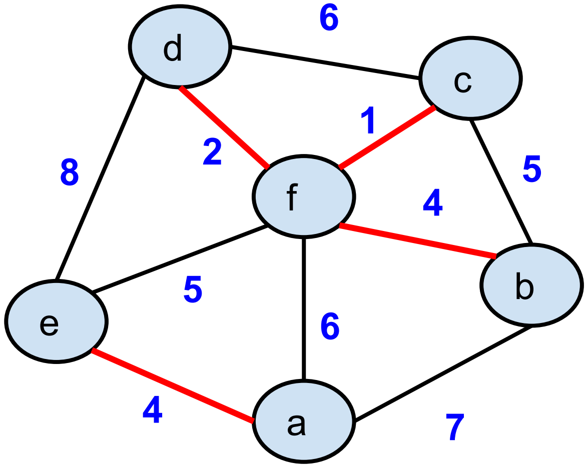
Minimum Spanning Tree: Kruskal's Algorithm

(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8

Disjoint Sets

S_1 = {a, e}

S_3 = {c, f, d, b}



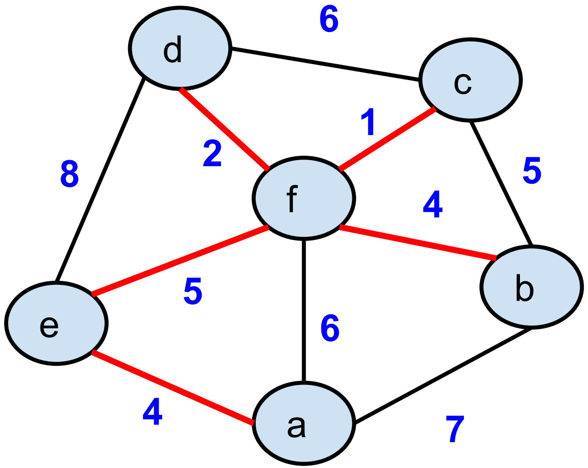
G = (V,E)

Minimum Spanning Tree: Kruskal’s Algorithm

(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8

Disjoint Sets

S_3 = {a, e, c, f, d, b}



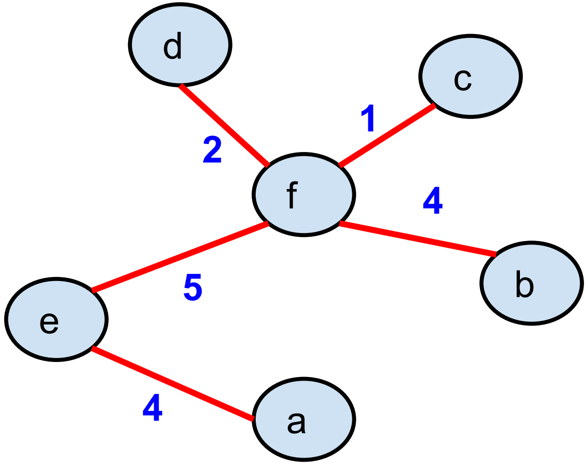
G = (V,E)

Minimum Spanning Tree: Kruskal's Algorithm

(c, f)	1
(d, f)	2
(a, e)	4
(b, f)	4
(b, c)	5
(e, f)	5
(a, f)	6
(c, d)	6
(a, b)	7
(d, e)	8

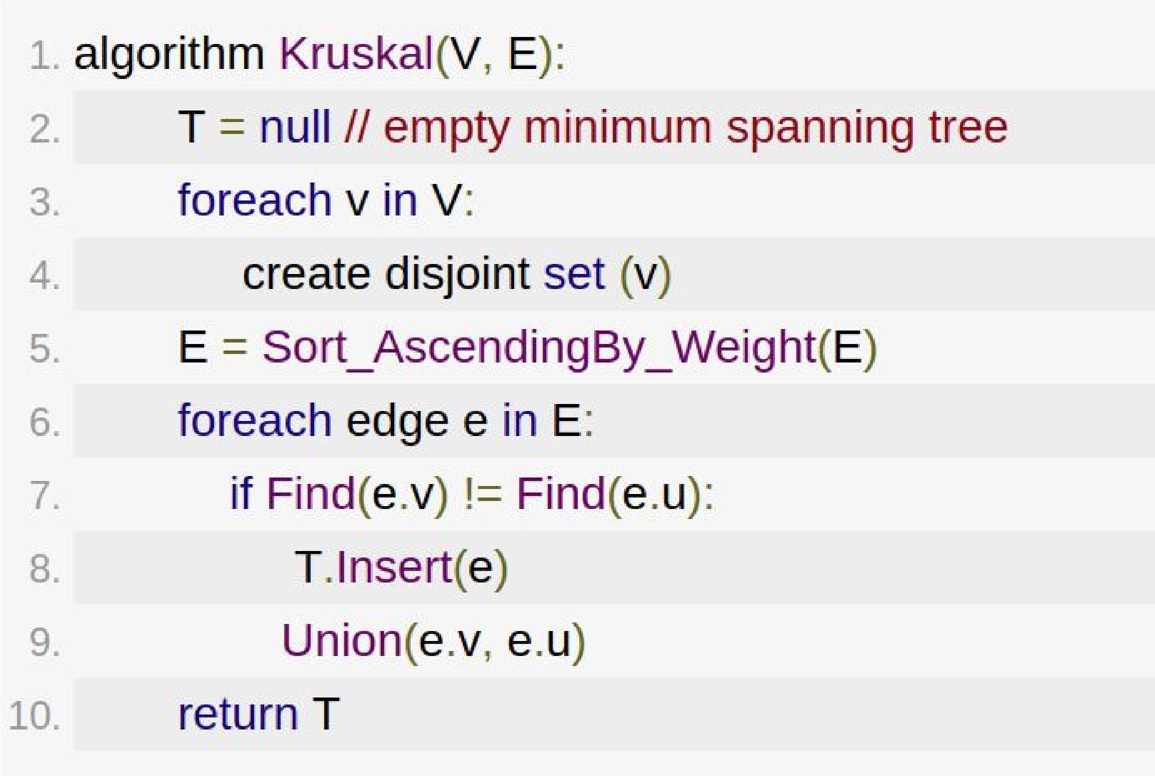
Disjoint Sets

S_3 = {a, e, c, f, d, b}



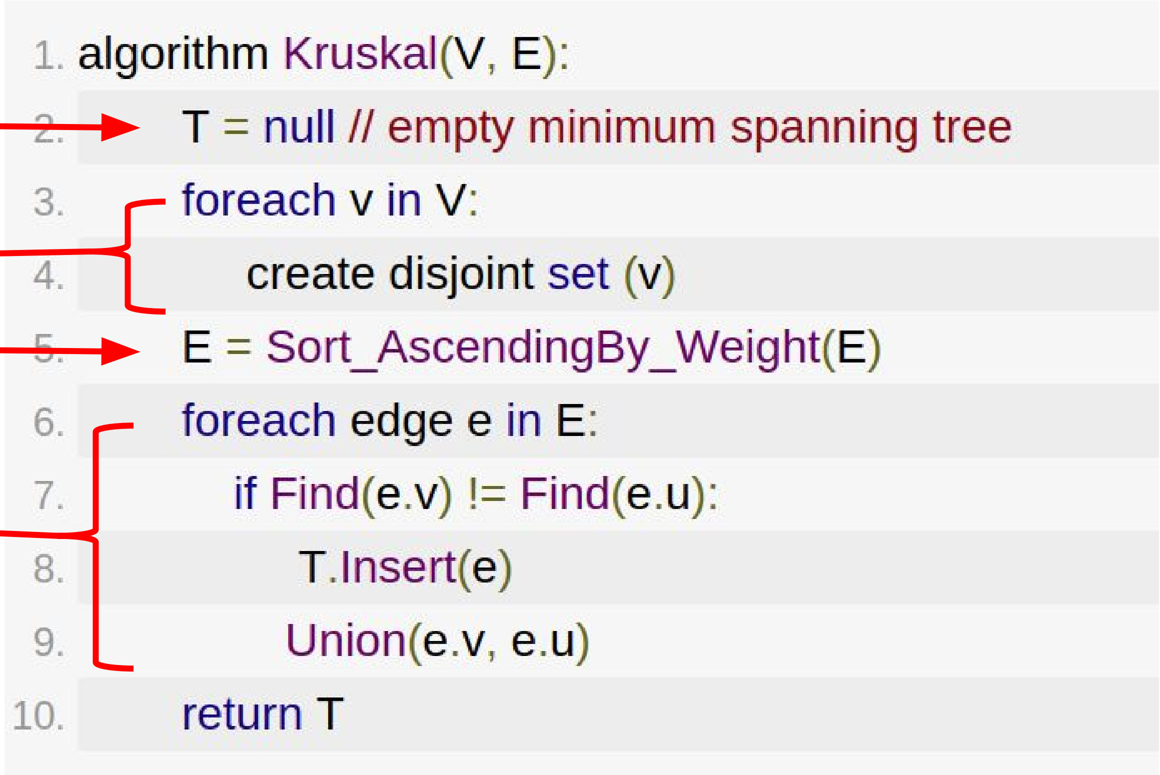
T: *minimum spanning tree of G*

Minimum Spanning Tree: Kruskal’s Algorithm



```
algorithm Kruskal(V, E):
    T = null // empty minimum spanning tree
    foreach v in V:
        create disjoint set (v)
    E = Sort_AscendingBy_Weight(E)
    foreach edge e in E:
        if Find(e.v) != Find(e.u):
            T.Insert(e)
            Union(e.v, e.u)
    return T
```

Minimum Spanning Tree: Kruskal's Algorithm



```
algorithm Kruskal(V, E):
    T = null // empty minimum spanning tree
    foreach v in V:
        create disjoint set (v)
    E = Sort_AscendingBy_Weight(E)
    foreach edge e in E:
        if Find(e.v) != Find(e.u):
            T.Insert(e)
            Union(e.v, e.u)
    return T
```

O(1)

O(V)

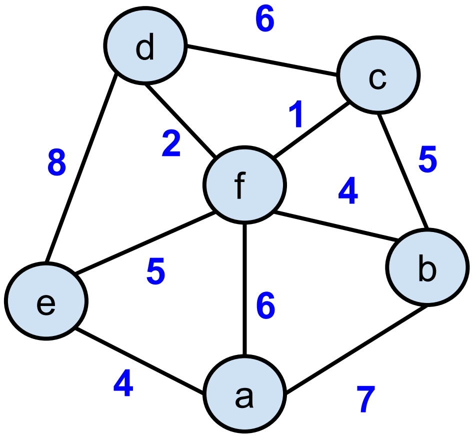
O(E log E)

O(V log V)

Minimum Spanning Tree: Prim’s Algorithm

Prim's Algorithm

- Find the minimum spanning tree by examine vertices one by one and add edges into a growing spanning tree.
- Follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

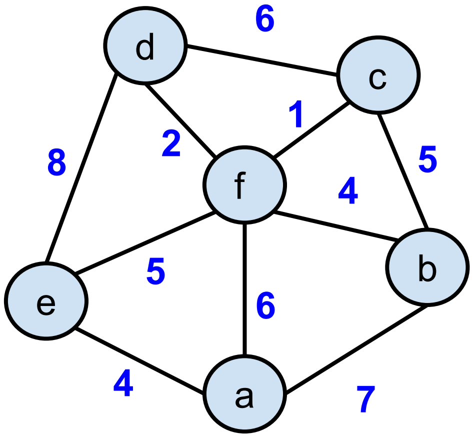


G = (V,E)

Minimum Spanning Tree: Prim's Algorithm

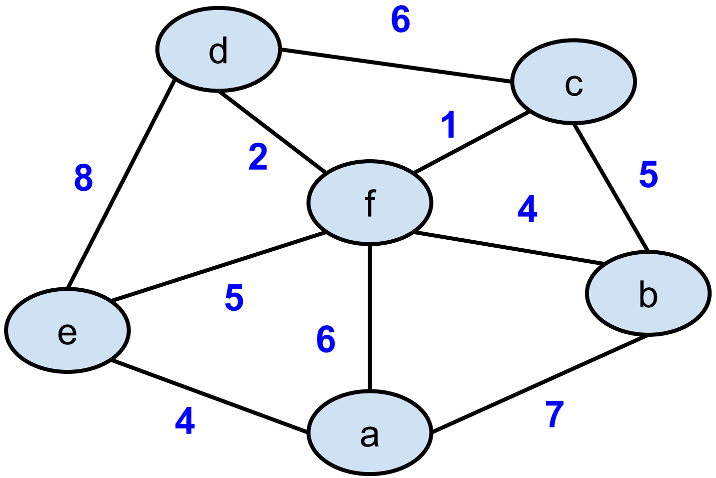
How it works?

1. Randomly select any vertex as the starting point
2. Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
3. Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree.
4. Mark the nodes which have been already selected and insert only those nodes that are not marked.

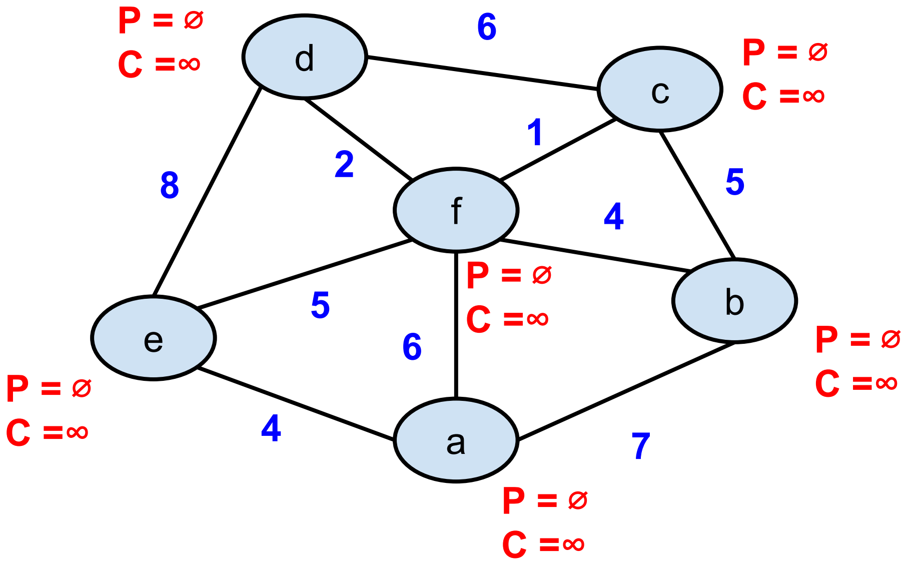


G = (V,E)

Minimum Spanning Tree: Prim’s Algorithm



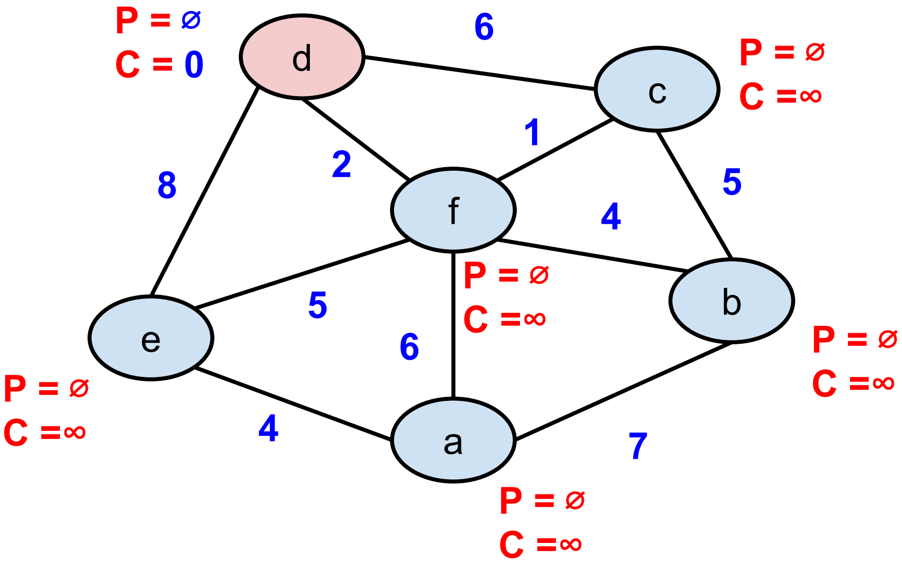
Minimum Spanning Tree: Prim's Algorithm



T = {}

Q = {a, b, c, d, e, f}

Minimum Spanning Tree: Prim's Algorithm

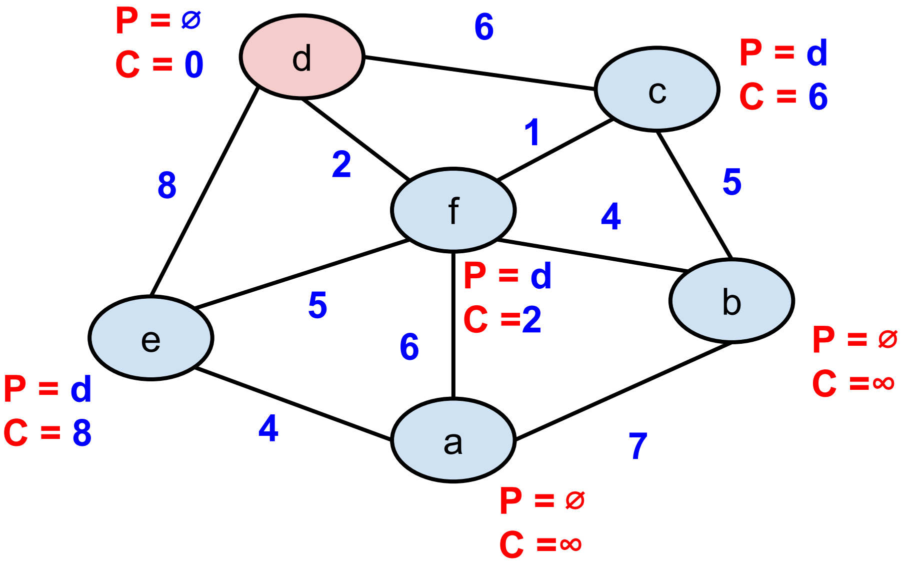


T = {}

Q = {a, b, c, e, f}

For each vertex in Q that is adjacent to the current selected vertex update the cost and the parent. ***As long as*** the new cost less that the current cost of that vertex

Minimum Spanning Tree: Prim's Algorithm

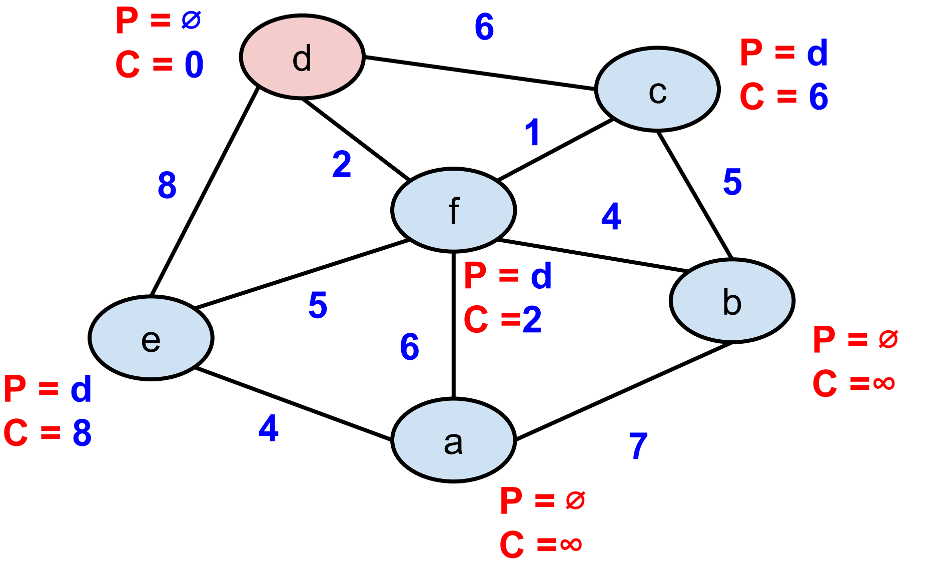


For each vertex in Q that is adjacent to the current selected vertex update the cost and the parent. ***As long as*** the new cost less that the current cost of that vertex

T = {}

Q = {a, b, c, e, f}

Minimum Spanning Tree: Prim’s Algorithm

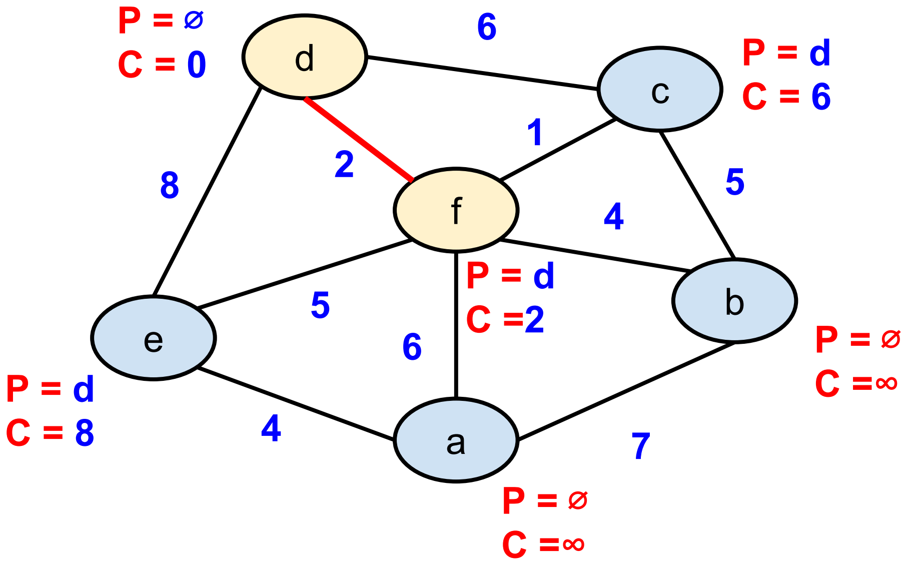


***Select** the vertex with the minimum cost in Q (vertex not in MST) if the parent of this vertex not !=* ∅ add the vertex and its parent to T

T = {}

Q = {a, b, c, e, f}

Minimum Spanning Tree: Prim's Algorithm

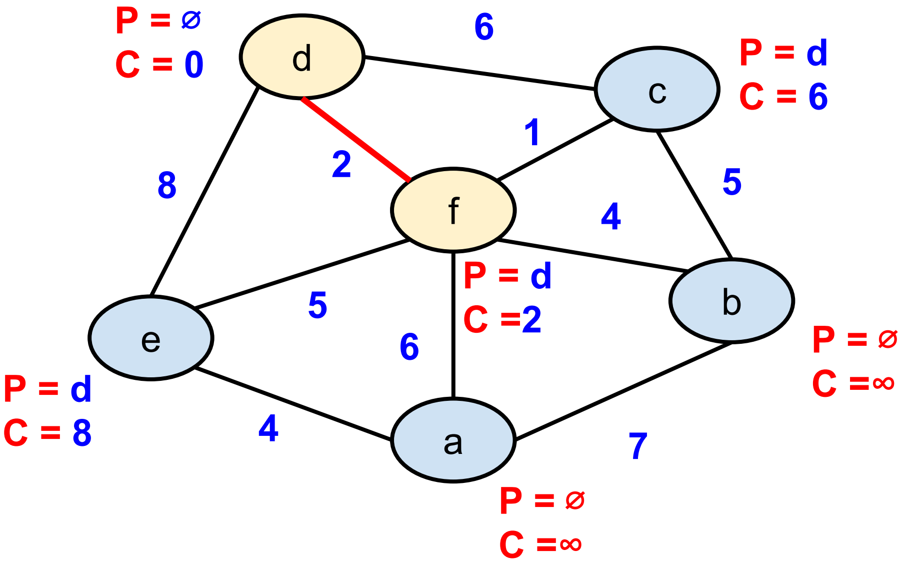


Select the vertex with the minimum cost in **Q** (vertex not in MST) if the **parent** of this vertex not != ∅, **remove** it from **Q**. add the vertex and its parent to T (MST)

T = { (d, f) }

Q = {a, b, c, e, f}

Minimum Spanning Tree: Prim's Algorithm



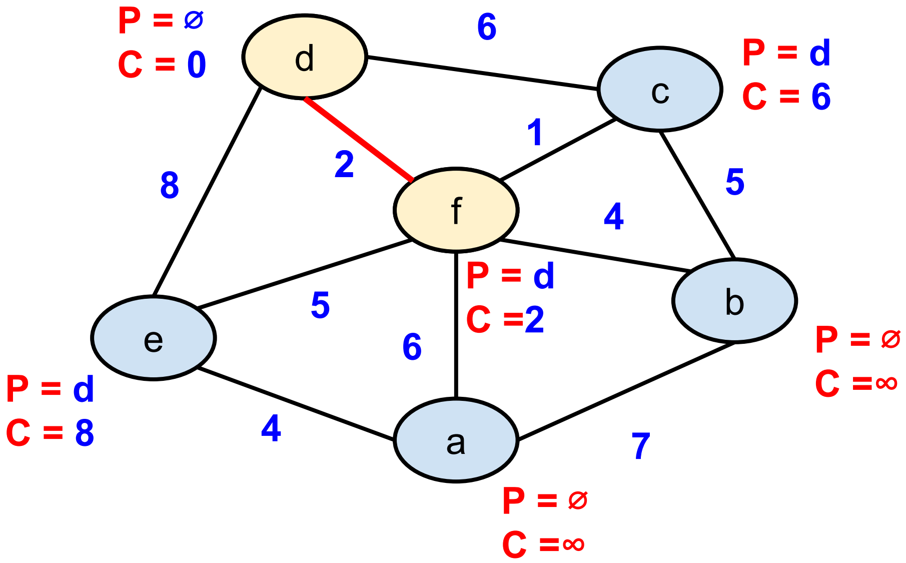
T = { (d, f) }

Q = {a, b, c, e, f}

Select the vertex with the minimum cost in **Q** (vertex not in MST) if the **parent** of this vertex not != ∅, **remove** it from **Q**. add the vertex and its parent to T (MST).

This vertex with minimum cost become the current selected vertex

Minimum Spanning Tree: Prim's Algorithm

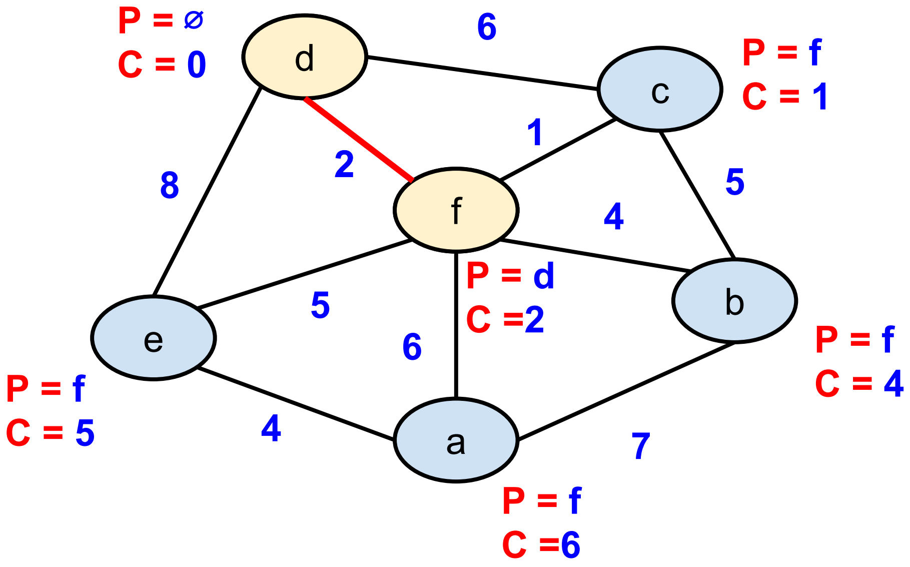


For each vertex in Q that is adjacent to the current selected vertex update the cost and the parent. ***As long as*** the new cost less that the current cost of that vertex

T = { (d, f) }

Q = {a, b, c, e}

Minimum Spanning Tree: Prim’s Algorithm

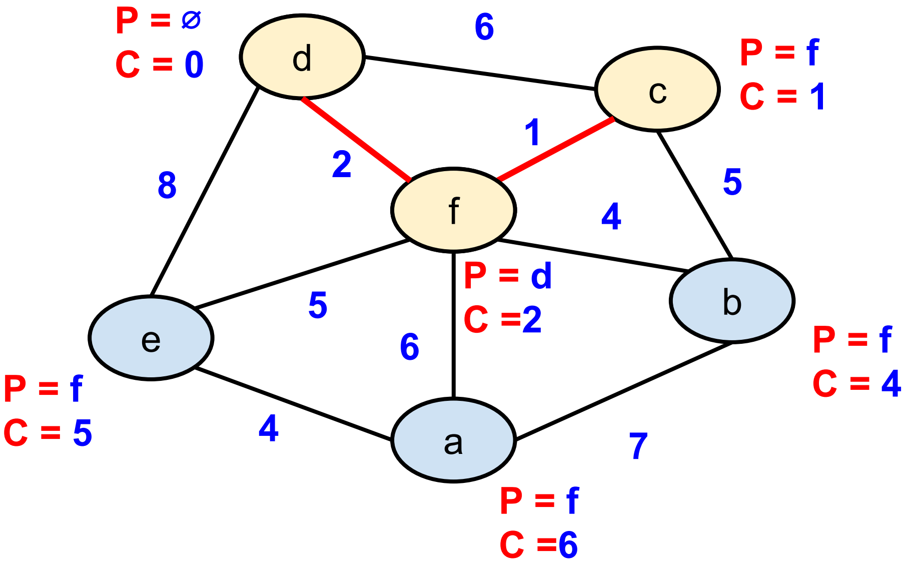


For each vertex in Q that is adjacent to the current selected vertex update the cost and the parent. ***As long as*** the new cost less that the current cost of that vertex

T = { (d, f) }

Q = {a, b, c, e}

Minimum Spanning Tree: Prim’s Algorithm



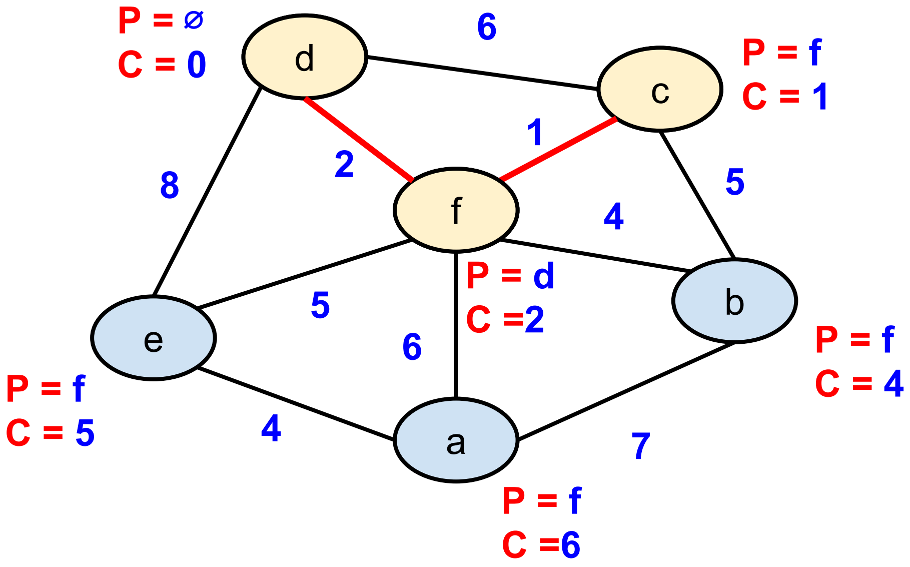
T = { (d, f), (f, c), }

Q = {a, b, e}

Select the vertex with the minimum cost in **Q** (vertex not in MST) if the **parent** of this vertex not != ∅, **remove** it from **Q**. add the vertex and its parent to T (MST).

This vertex with minimum cost become the current selected vertex

Minimum Spanning Tree: Prim's Algorithm

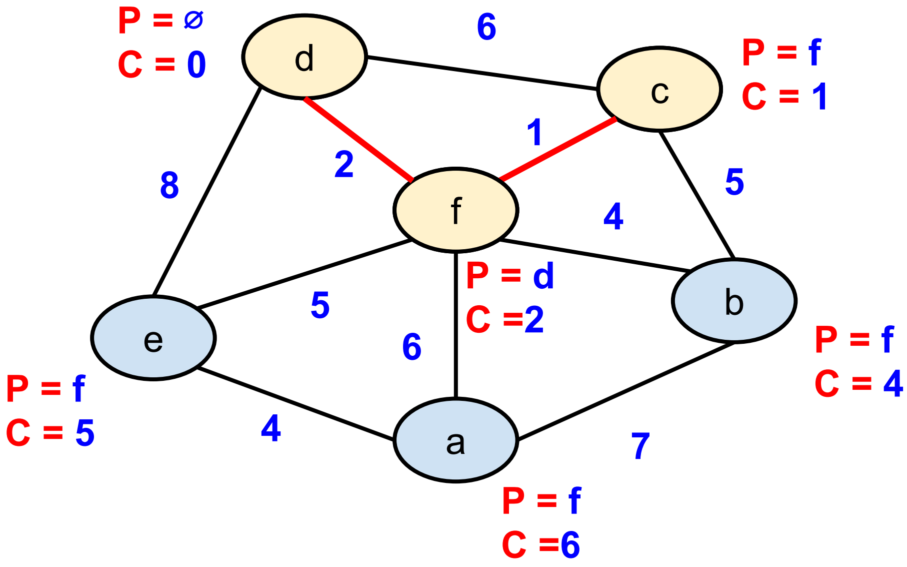


For each vertex in Q that is adjacent to the current selected vertex update the cost and the parent. ***As long as*** the new cost less that the current cost of that vertex

T = { (d, f), (f, c), }

Q = {a, b, e}

Minimum Spanning Tree: Prim's Algorithm



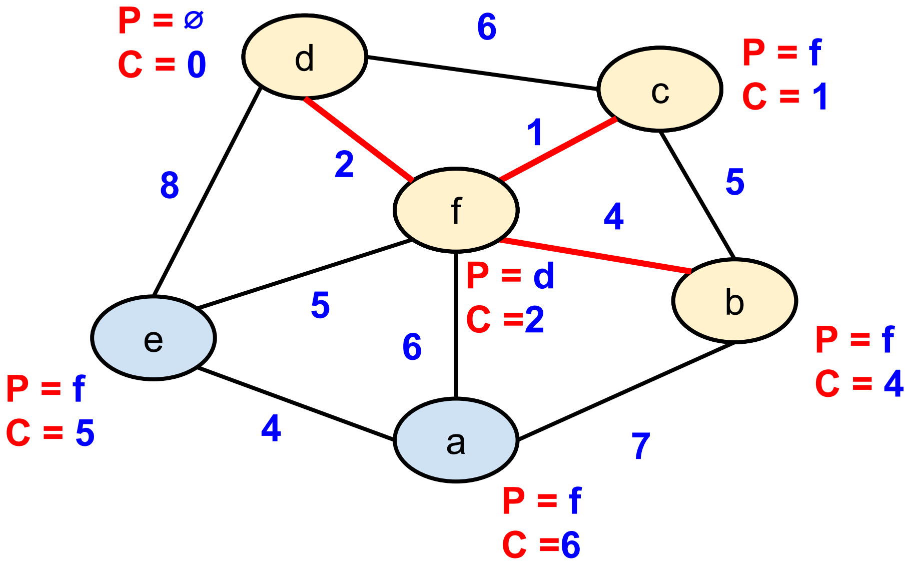
T = { (d, f), (f, c), }

Q = {a, b, e}

Select the vertex with the minimum cost in **Q** (vertex not in MST) if the **parent** of this vertex not != ∅, **remove** it from **Q**. add the vertex and its parent to T (MST).

This vertex with minimum cost become the current selected vertex

Minimum Spanning Tree: Prim’s Algorithm



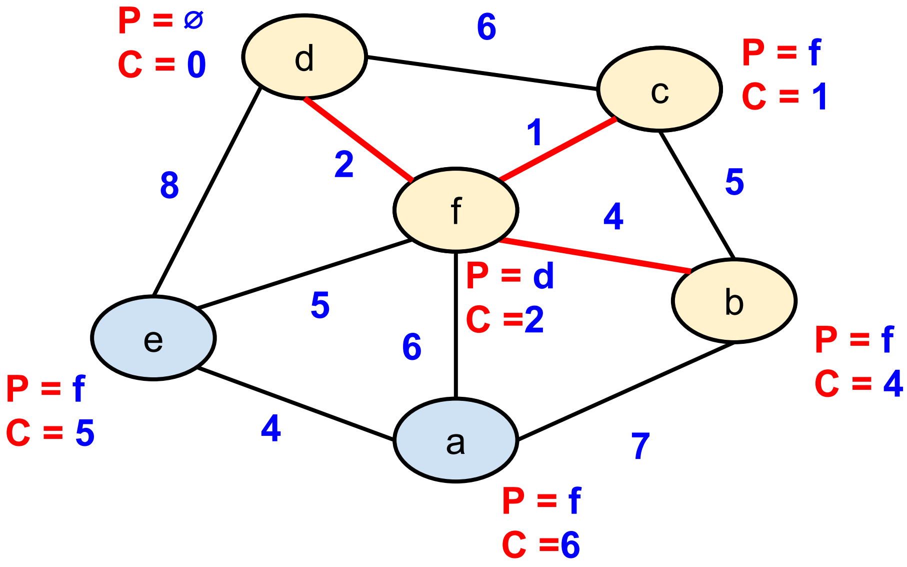
T = { (d, f), (f, c), (f, b) }

Q = {a, e}

Select the vertex with the minimum cost in **Q** (vertex not in MST) if the **parent** of this vertex not != ∅, **remove** it from **Q**. add the vertex and its parent to T (MST).

This vertex with minimum cost become the current selected vertex

Minimum Spanning Tree: Prim's Algorithm

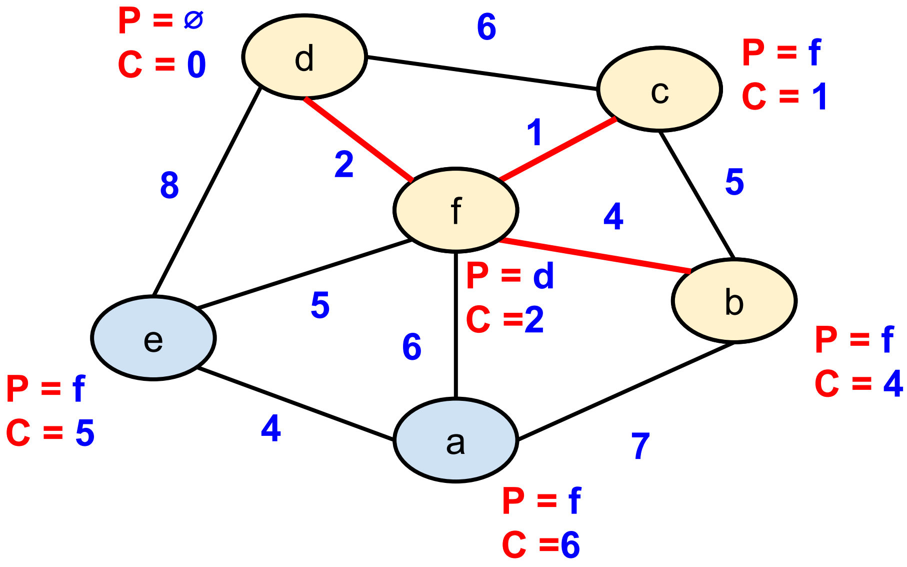


T = { (d, f), (f, c), (f, b) }

Q = {a, e}

For each vertex in Q that is adjacent to the current selected vertex update the cost and the parent. ***As long as*** the new cost less that the current cost of that vertex

Minimum Spanning Tree: Prim’s Algorithm



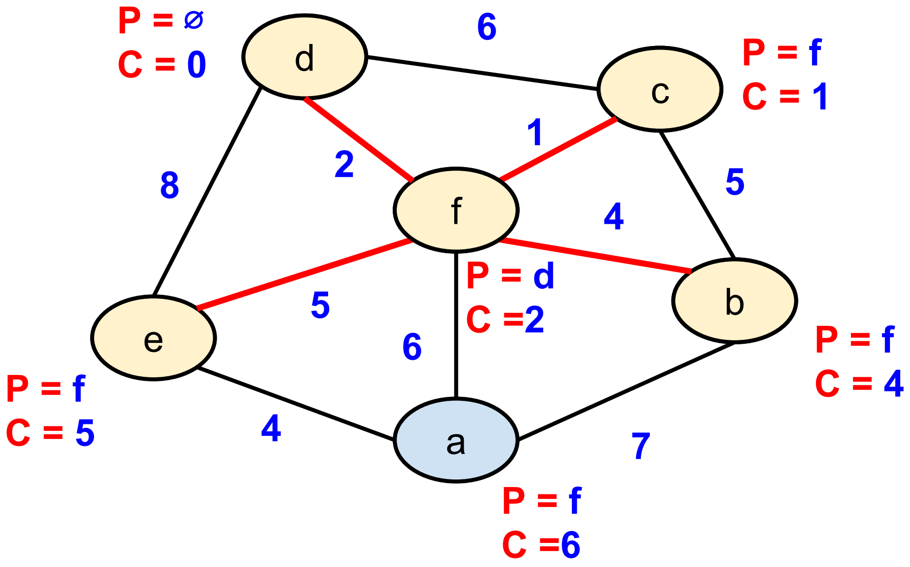
T = { (d, f), (f, c), (f, b) }

Q = {a, e}

Select the vertex with the minimum cost in **Q** (vertex not in MST) if the **parent** of this vertex not != ∅, **remove** it from **Q**. add the vertex and its parent to T (MST).

This vertex with minimum cost become the current selected vertex

Minimum Spanning Tree: Prim's Algorithm

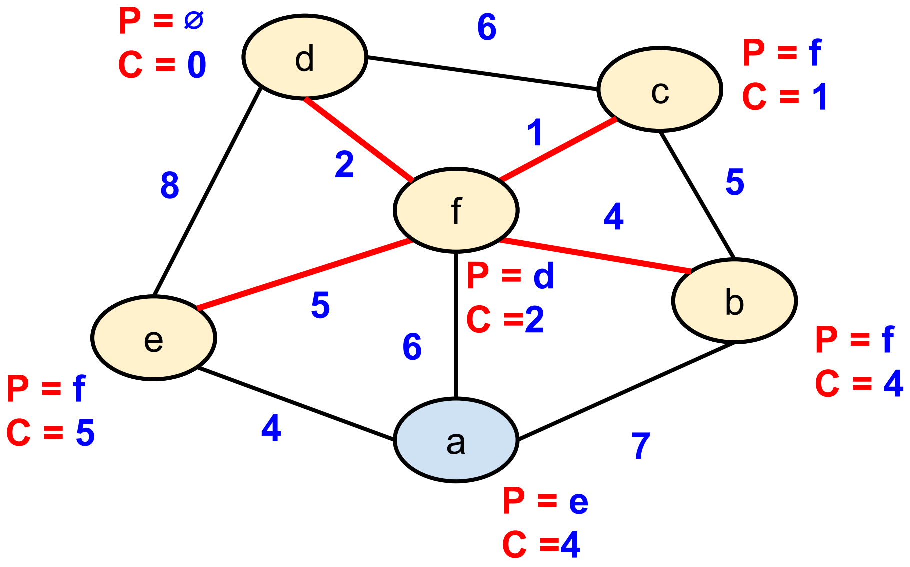


T = { (d, f), (f, c), (f, b), (f,e) }

Q = {a}

For each vertex in Q that is adjacent to the current selected vertex update the cost and the parent. ***As long as*** the new cost less that the current cost of that vertex

Minimum Spanning Tree: Prim's Algorithm

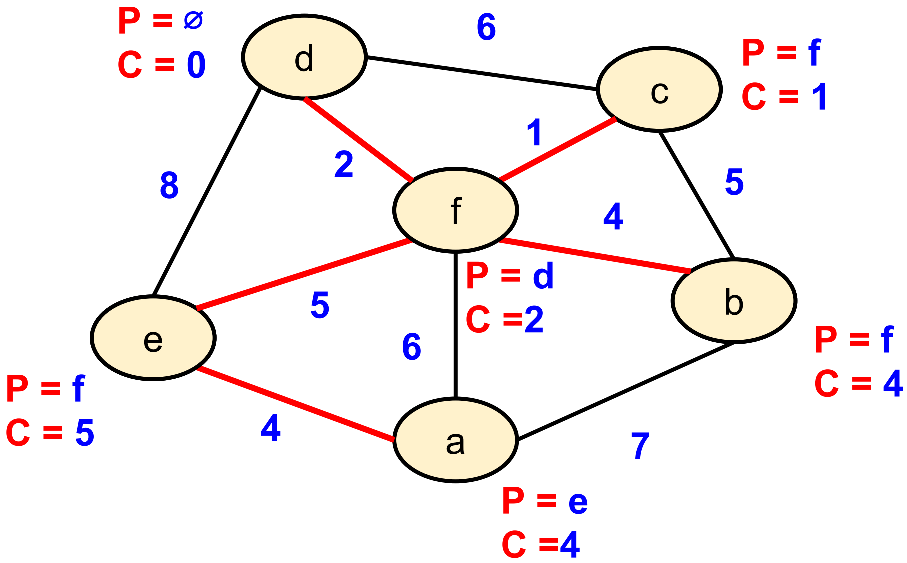


For each vertex in Q that is adjacent to the current selected vertex update the cost and the parent. ***As long as*** the new cost less that the current cost of that vertex

T = { (d, f), (f, c), (f, b), (f,e) }

Q = {a}

Minimum Spanning Tree: Prim's Algorithm

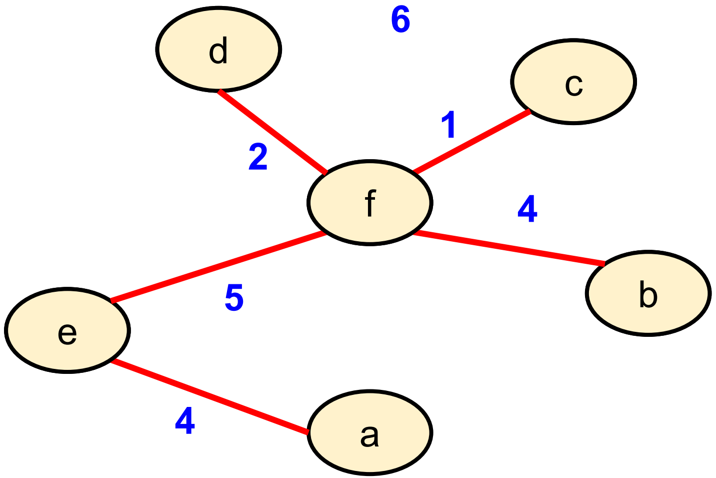


When Q is empty the algorithm will terminate

T = { (d, f), (f, c), (f, b), (f,e). (e, a) }

Q = {}

Minimum Spanning Tree: Prim's Algorithm

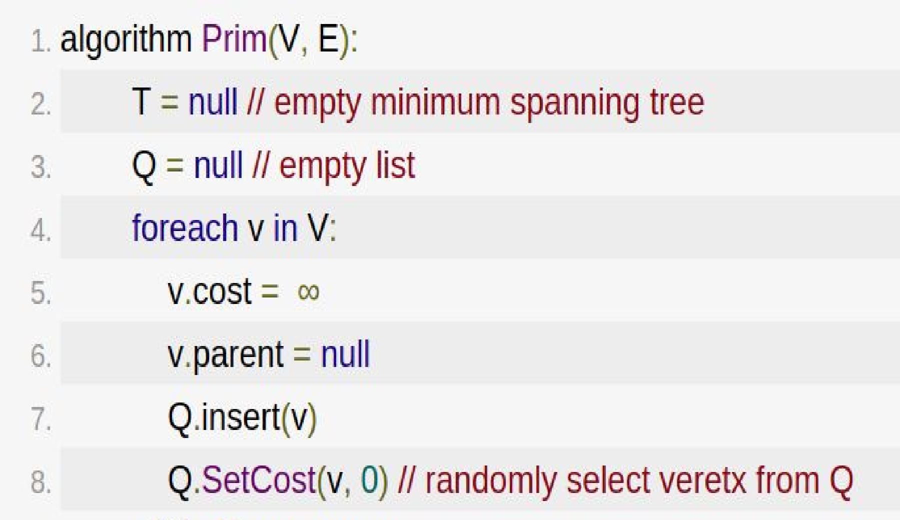


When Q is empty the algorithm will terminate

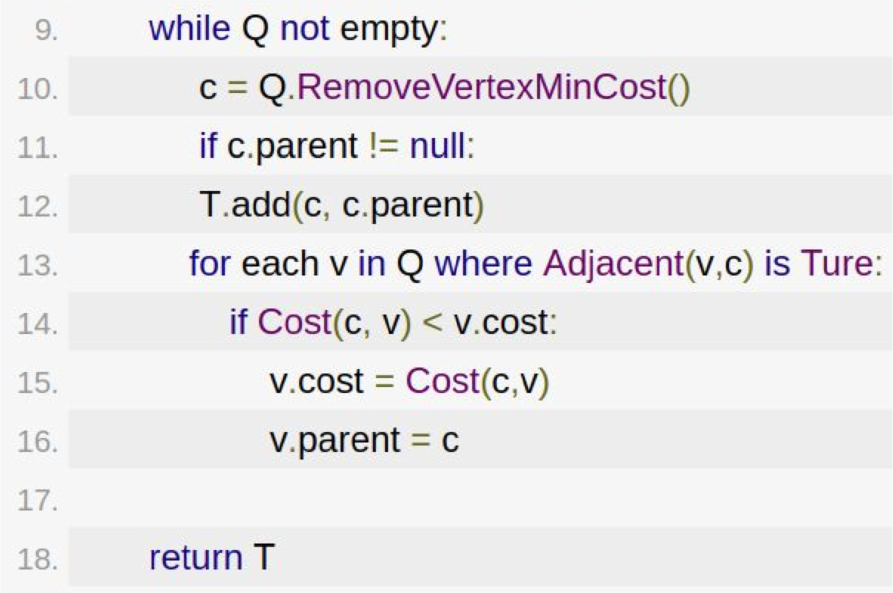
T = { (d, f), (f, c), (f, b), (f,e). (e, a) }

Q = {}

Minimum Spanning Tree



```
algorithm Prim(V, E):
    T = null // empty minimum spanning tree
    Q = null // empty list
    foreach v in V:
        v.cost = ∞
        v.parent = null
        Q.insert(v)
        Q.SetCost(v, 0) // randomly select veretx from Q
```



```
    while Q not empty:
        c = Q.RemoveVertexMinCost()
        if c.parent != null:
        T.add(c, c.parent)
        for each v in Q where Adjacent(v,c) is Ture:
            if Cost(c, v) < v.cost:
                v.cost = Cost(c,v)
                v.parent = c

    return T
```

Self-Assessment

Given a graph G with more than one minimum spanning tree. Explain why Kruskal's algorithm will always generate the same MST, while Prim's algorithm may generate different MST.