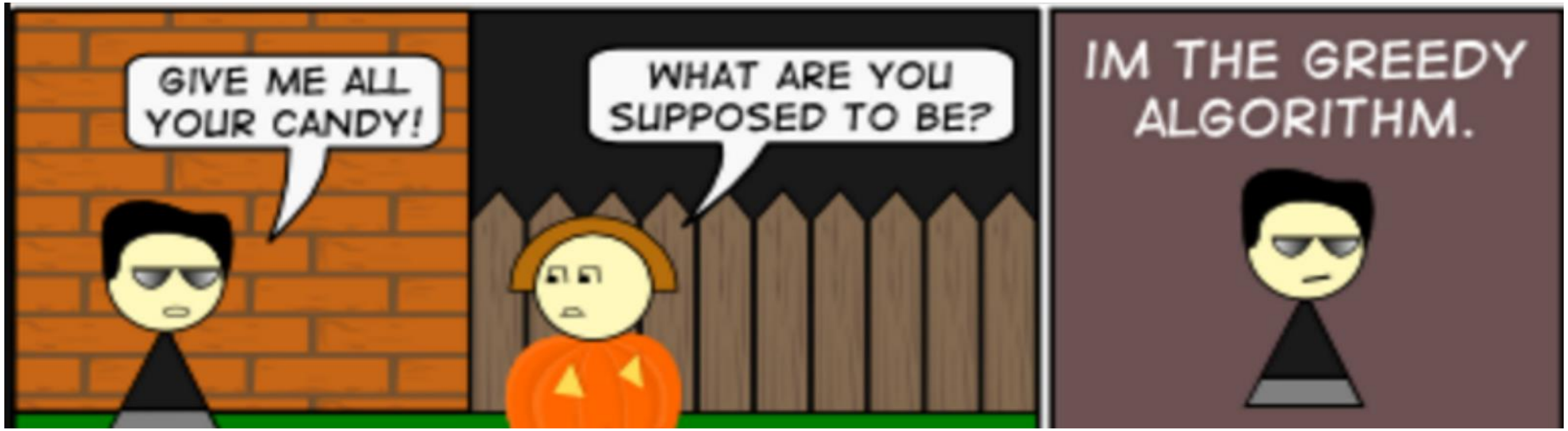
Algorithm Design : Greedy Algorithms

The seven deadly sins in Christianity (Also in other religions) :

Envy, gluttony, greed , lust, pride, sloth, wrath



Optimization problems

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A “greedy algorithm” sometimes works well for most optimization problems
- A greedy algorithm works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at *global* optimum

In order to get what you want,just grab what looks best!

Greedy Properties

1. "greedy-choice property" It says that a globally optimal solution can be arrived at by making a series of locally optimal choices.

2. "optimal substructure" A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the sub-problems.

→In order for greedy heuristic to solve the problem, the optimal solution to the big problem should contain optimal solutions to sub problems

The Structure of Greedy Algorithms

```
Algorithm Greedy (a , n)
//a[1:n]contains the n inputs.
   solution = 0    //initialize the solution.
   for i =1 to n do
        {
           x =SelectFrom(a)
           if  Feasible( solution, x) then
              solution = Union(solution , x)   //Accept partial sol.
        }
   return solution
```

The Structure of Greedy Algorithms

- *Select*() selects an input from a[] and removes it.

  the selected input value is assigned to x.
- *Feasible*() is a boolean-valued function that determines whether x can be included into the solution vector (no constraints are violated?).
- Union() combines x with the solution so far and updates the objective function.

Example : Coin Changing Problem

- Greedy Algorithm works by making the decision that seems most promising at any moment; it never reconsiders this decision, whatever situation may arise later.
- As an example consider the Change Making problem:

"What is the minimum number of coins that add up to a given amount of money ?“

Assume available coins are:

  - 100 cents, 25 cents, 10 cents ,5 cents,1 cent

- A Greedy Algorithm : Use fewest possible coins always.

→At each step, take the largest possible bill or coin that does not make the sum > required

Coin changing problem

//Goal: Make change for n units using the least possible number of coins.

MAKE-CHANGE (n)

```
C ← {100, 25, 10, 5, 1}     // constant.
S← {};                      // Initially empty set that will hold the solutions
Sum ← 0 sum of items in solution set
WHILE sum != n
   x = largest item in set C such that (sum + x ) ≤ n  //Greedy choice
   IF no such item THEN
       RETURN   "No Solution“     //Not feasible
   S ← S Union (value of x)        //Include x to solution set
   sum ← sum + x
RETURN S
```

How the Algorithm Works

- **Example :** Make a change for 2.89 (289 cents) here n = 2.89 and the solution contains 2 dollars, 3 quarters, 1 dime and 4 pennies.
- The algorithm is greedy because at every stage it chooses the largest coin without worrying about the consequences.
- Moreover, it never changes its choice : once a coin has been included in the solution set, it remains there.

Note:The greedy solution may not work for some currency systems.

Some Greedy Algorithms

- Dijkstra’s algorithm for finding single source shortest paths in a graph
- Kruskal’s algorithm for finding a minimum-cost spanning tree
- Prim’s algorithm for finding a minimum-cost spanning tree
- Huffman algorithm for finding minimal length codes for data compression
- Knapsack algorithm for filling optimally a space with different items

Single Source Weighted Shortest Paths

- The problem: Given a weighted graph $G=(V, E,W)$ for which there is no negative weight and one of the vertices is specifed to be the source vertex.
- Determine the cost of the shortest paths from the source to every other vertex in V.

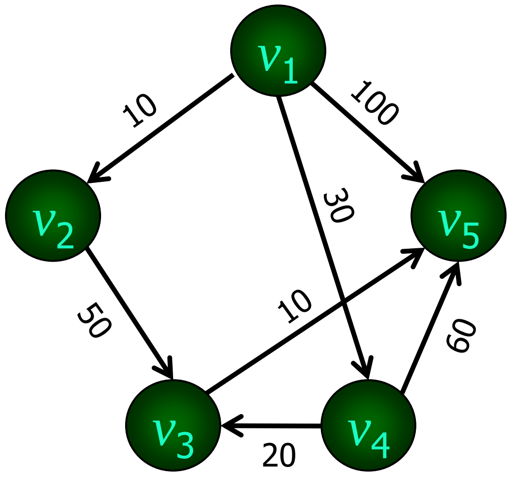
Dijkstra's Algorithm can be used to solve this problem.

(Developed by E.W.Dijkstra,1959)

An Application of Weighted Shortest Paths

The graph G represents an airline map.

- Vertices in G represent cities
- Each directed edge in G represents a route from one city to another city.
- Each weight represents the time required for flying from one city to another city.
- When we solve the single source shortest path problem, we will be able to determine the minimum flight time from a given city (v1) to every other city on the map.



Dijkstra's Algorithm

- Dijkstra's algorithm – A greedy solution to the single-source shortest paths problem in graph theory.
- Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

  Input: Weighted graph G={E,V,W} and source vertex $s \in V$.

  Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $s \in V$ to all other vertices

Dijkstra's Algorithm : Idea

- Problem:We have a weighted graph G= (V,E,W).

  Find shortest paths from a given node to all other nodes.
- Start with source vertex s and iteratively construct a tree rooted at s
- Each vertex keeps track of current cheapest path from s
- At each iteration, include the vertex whose cheapest path from s is the overall cheapest :

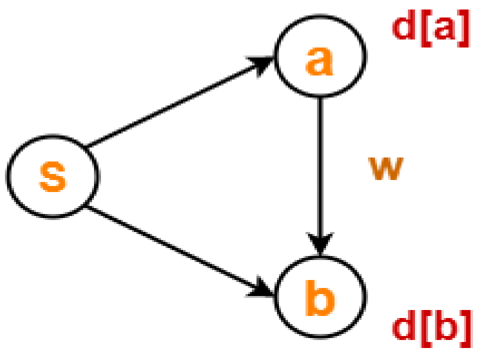
→ The choice is greedy !

Dijkstra's Algorithm - Pseudocode

```
//Weighted graph G and source vertex  s are available
Dijkstra(G,s)
dist[s] ← 0                              //distance to source vertex is zero
for  all v ∈ V–{s}
     do  dist[v] ←∞                      //set all other distances to infinity
S←∅                                     //S, visited vertices set. Initially empty
Q←V                                     //Q, the queue initially contains all vertices)
while Q ≠∅                               //while the queue is not empty
    do   u ← mindistance(Q, dist[v])//select the el. of Q with the min. dist. from s
    S←S∪{u}                              //add u to list of visited vertices
     for all v ∈ neighbors[u]
         do  if   dist[u] + w(u, v) < dist[v]        // if new shortest path found
                then      d[v] ←d[u] + w(u, v)   //Update shortest path
          // .................... (if desired, add code here to display the path
return dist                               //dist now includes shortest distances from s
end   //Dijkstra
```

How Distances are Updated ?

Let d[a] and d[b] denote the previous shortest paths for vertices a and b respectively from the source vertex S .
Can we find a shorter path to b if we go over a ?



Yes if $d[a] + w < d[b]$ then we update: $d[b] = d[a] + w$
We repeat this for all neighbors of a.

Example : Initialization

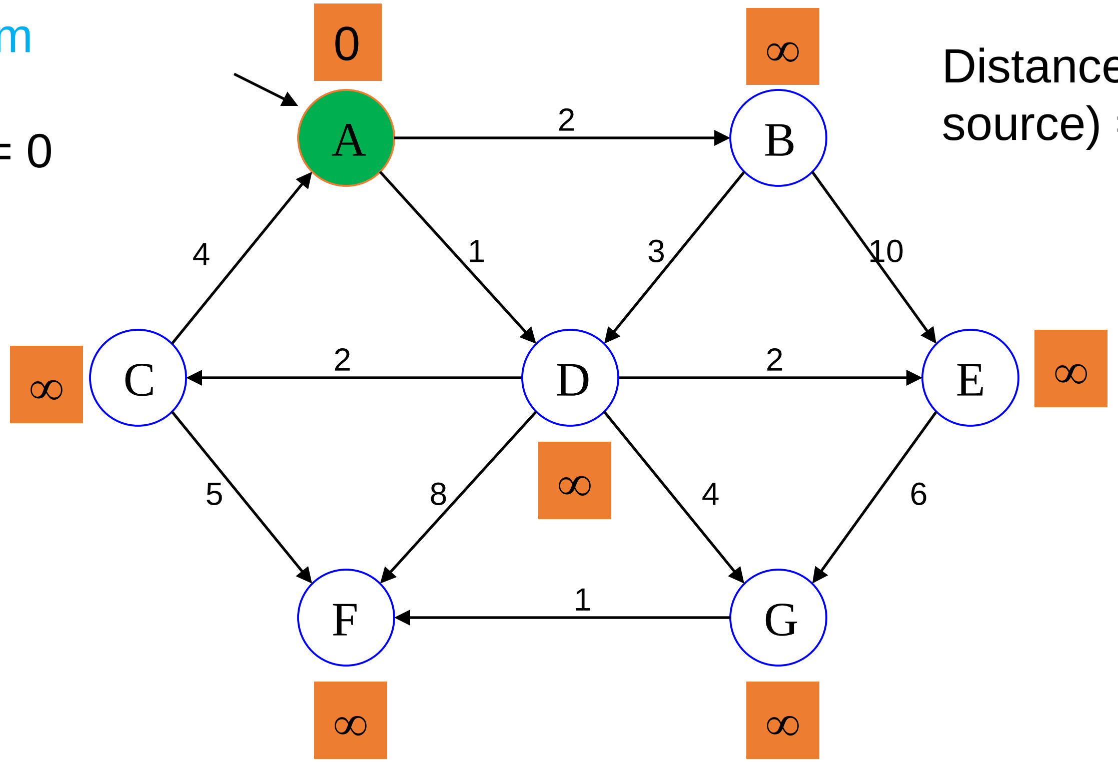
Initialize : Q ={A,B,C,D,E,F,G} , source A, S={A}, A is known(visited)

Distances of neighbors from source:

Distance(source) = 0

Distance(D) = 1

Distance(B) = 2

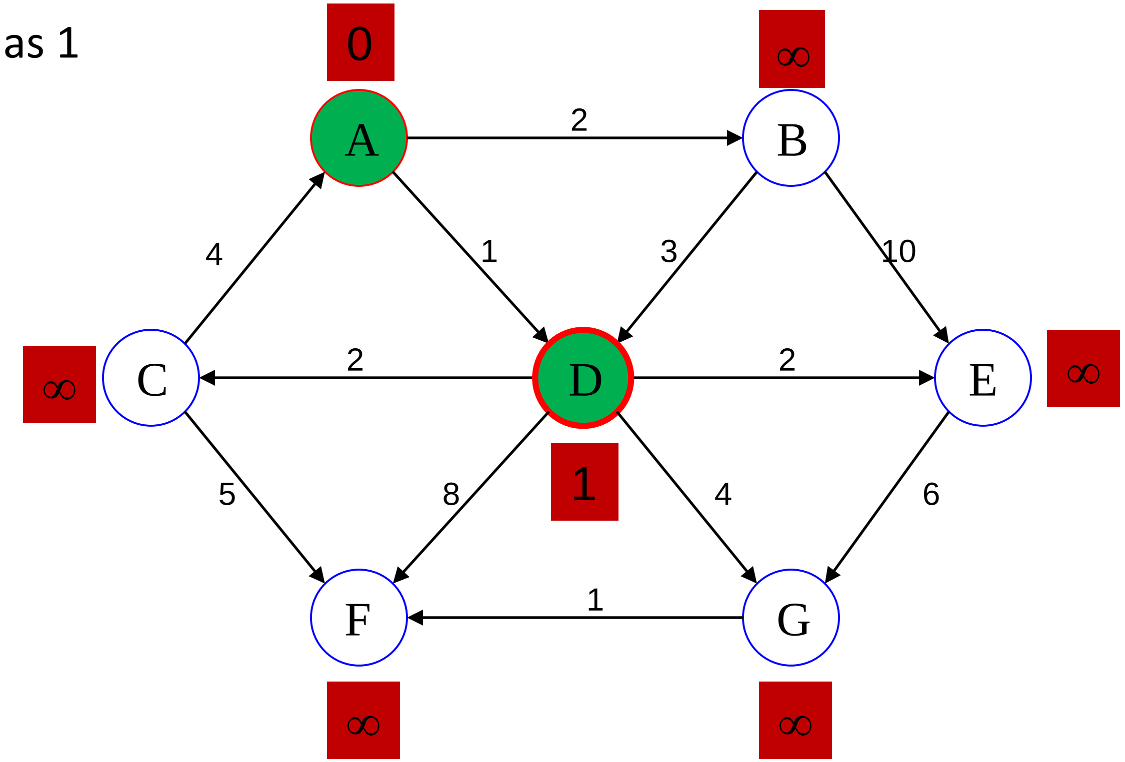


Distance (all vertices but source) = ∞

Pick vertex in List with minimum distance: D.   (u) in the algorithm)

Remove the Vertex With Minimum Distance

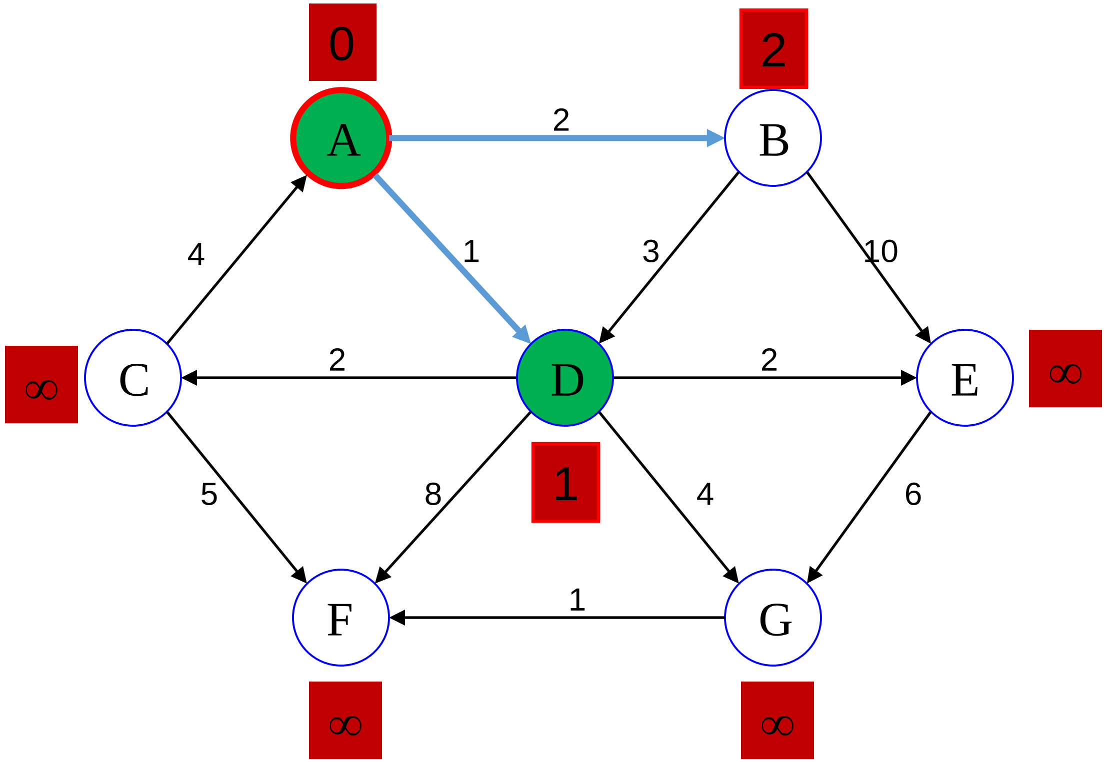
Update D's distance as 1



Mark D as **known. Add D: S= { A,D }**

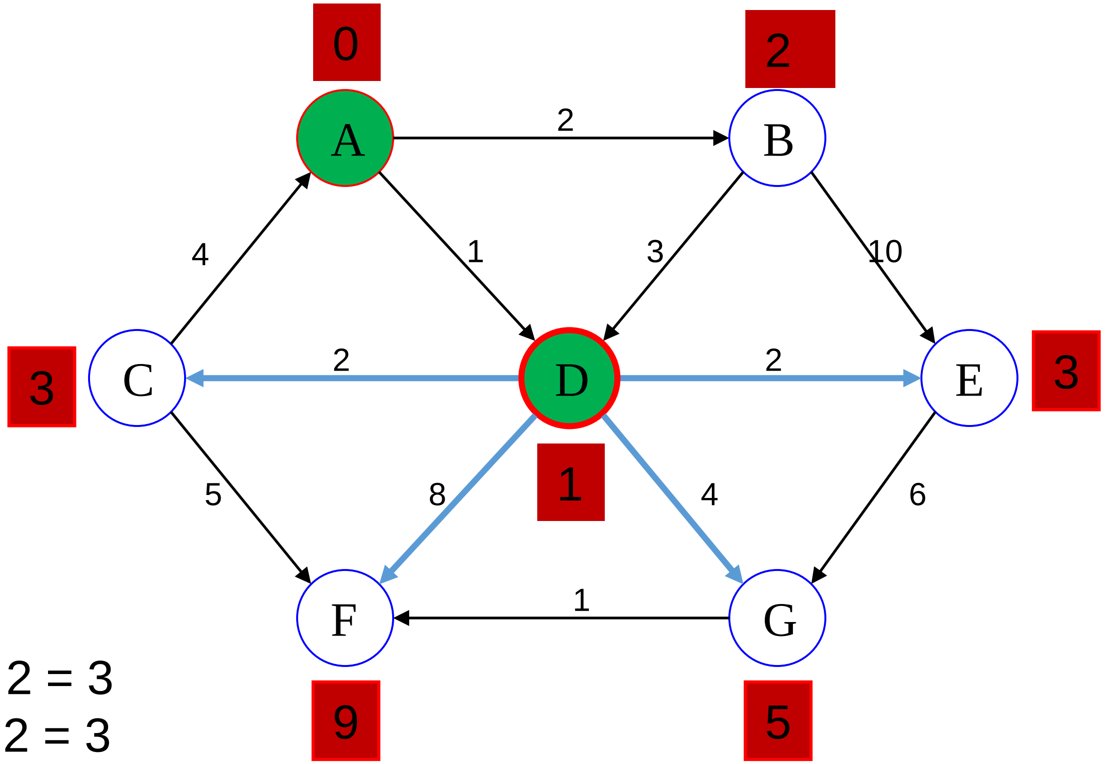
Red arrows indicate paths from destination to source.

Update Neighbor Distances of A and D: B,C,E,F,G



Update neighbor distances:

Find all distances from A to others (Check one edge paths and all paths over D)

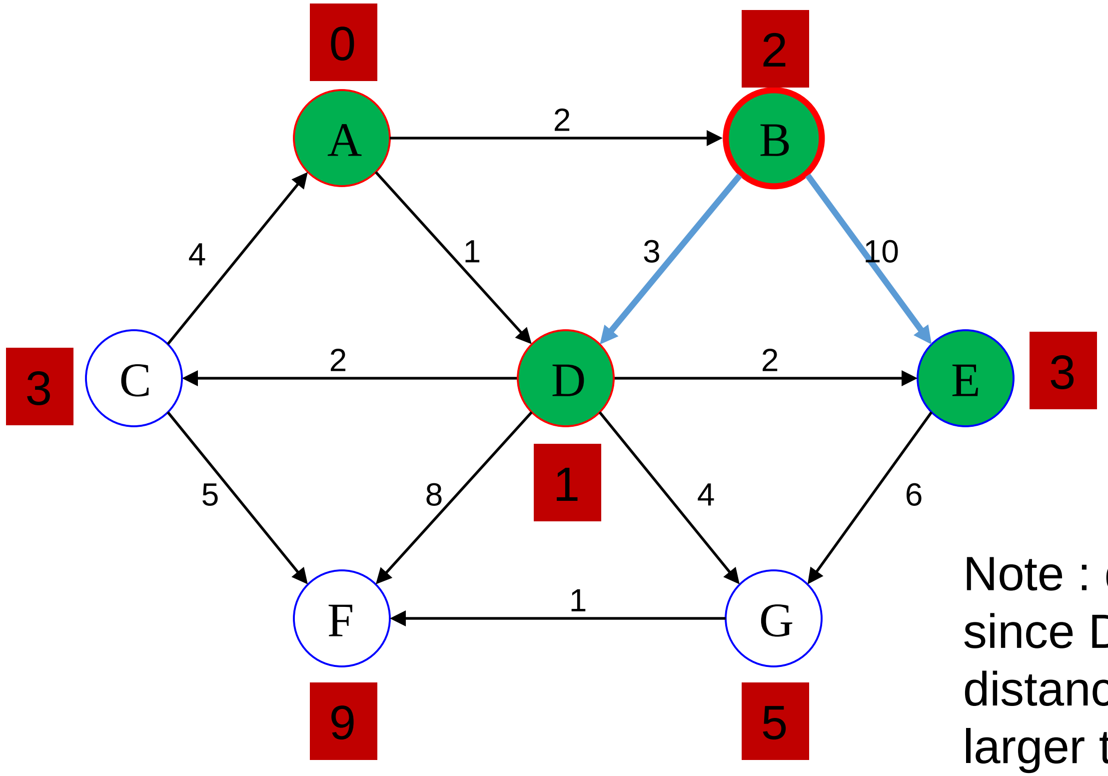


Update distances:

Distance(B) = 2
Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Update Neighbors

Pick vertex in the list with minimum distance: (B) and update its neighbors



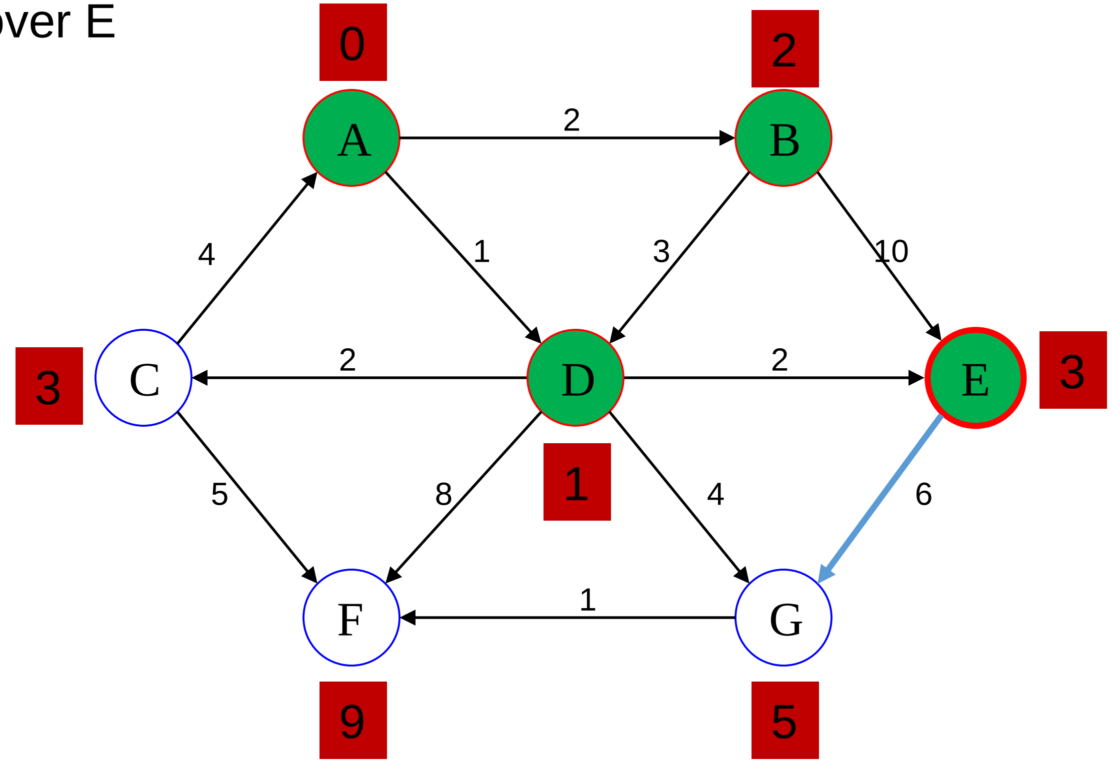
Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Mark B as known, S= {A,B,D}
Mark E as known , S={A,B,D,E}

Update Neighbors

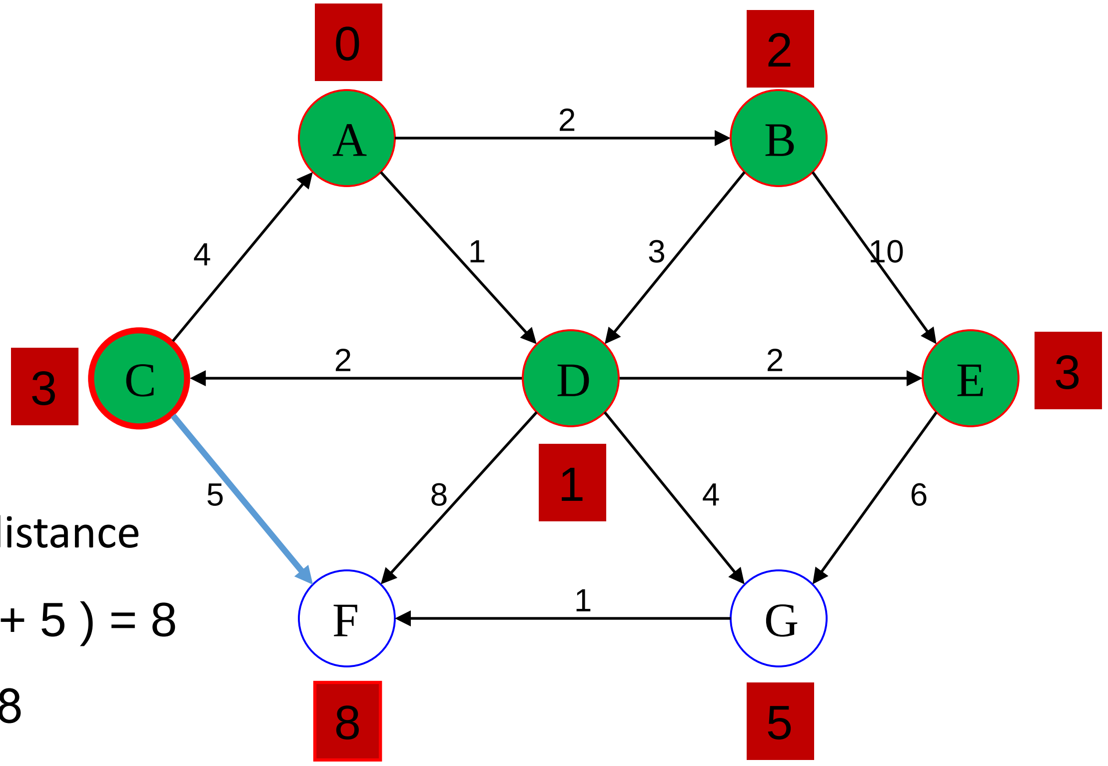
Pick List vertex over E with minimum distance: Update neighbors

No updating possible over E



Update Neighbors

Pick next list vertex with minimum distance: C. Update neighbors



Previous distance

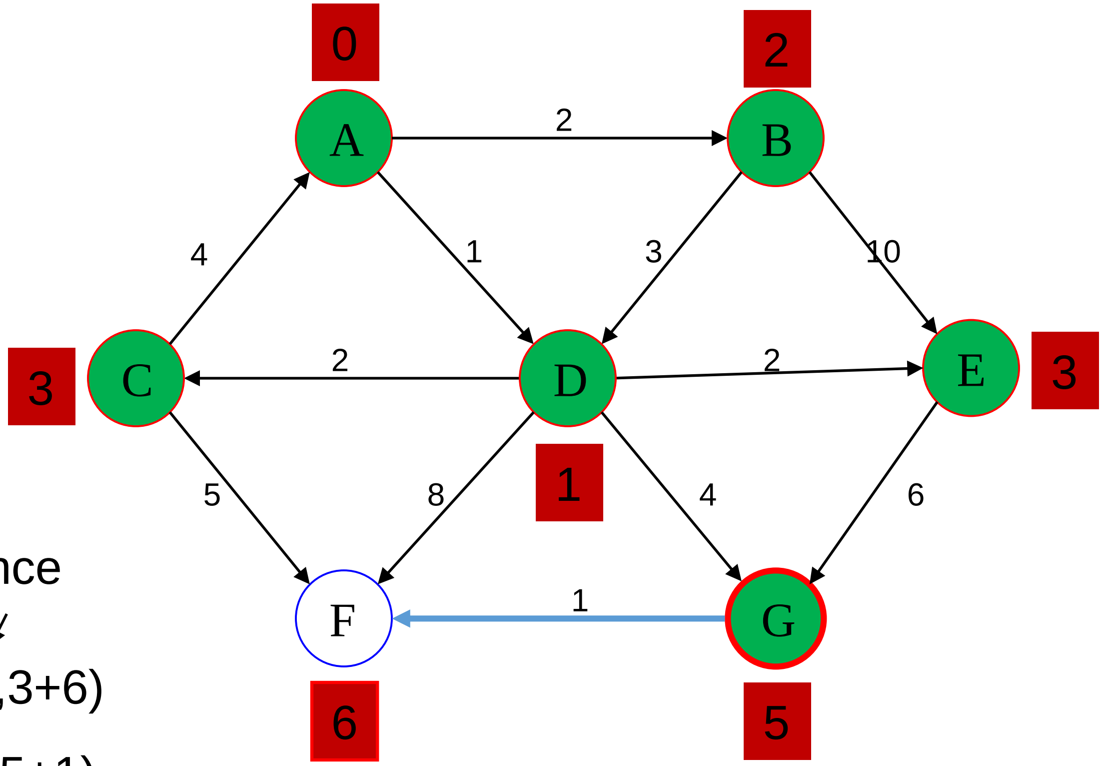
Distance(F) = min(9 , 3 + 5) = 8

Update F's distance as 8

Mark C as known. S={ A,B,C,D,E }

Update Neighbors

Pick vertex List with minimum distance : G. Update neighbors



Previous distance

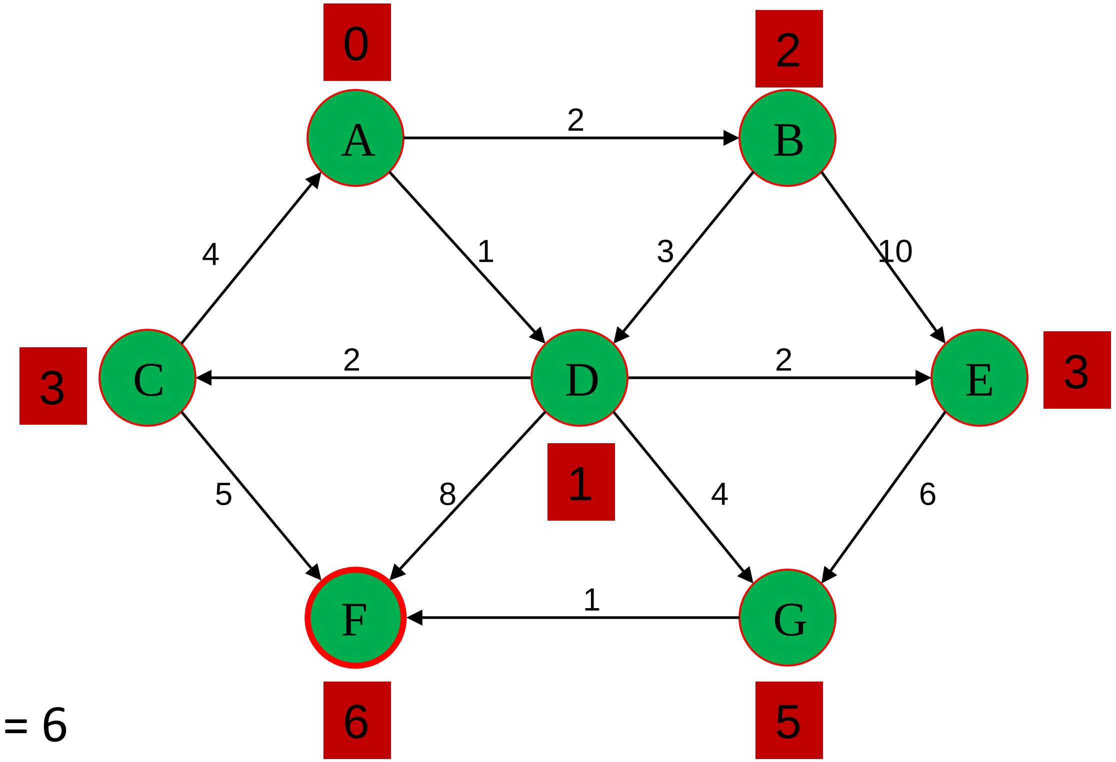
Distance(G) = min(5,3+6)

Distance(F)= min(8,5+1)

Mark G as known :S= {A,B,C,D,E,G}

Example Finished

Pick vertex not in S with lowest cost: F .Update neighbors. Last step.



Distance F = 6

Mark F as known. S= { A,B,C,D,E,F,G }

Minimal distances from A to all others have been determined(Numbers in squares)

Red arrows indicate paths from destination to source.

Correctness of Dijkstra’s Algorithm

- Djkistra is a greedy algorithm
  - makes choices that currently seem the best
  - However, locally optimal does not always mean globally optimal
- The algorithm is correct because it maintains following two properties:
  - for every known vertex , recorded distance is shortest distance from source vertex to that vertex
  - for every unknown vertex *v*, its recorded distance is shortest path distance to *v* from source vertex, as it considers only currently known shortest distance vertices so far and *v* .

Time Complexity of Djkstra

Assume a simple list implementation of graph G.

- We have $|V|$ vertices and $|E|$ edges. Costs in the algorithm:
  - Initialization: $O(|V|)$
  - While loop : $O(|V|)$
    - Nested for loop : Find and remove min distance vertices $O(|V|)$

    This must be repeated for every vertex : $O(|V|*|V|)$
    - Potentially $|E|$ updates,one for each edge , each update cost: $O(1)$

    $O(|E|)$

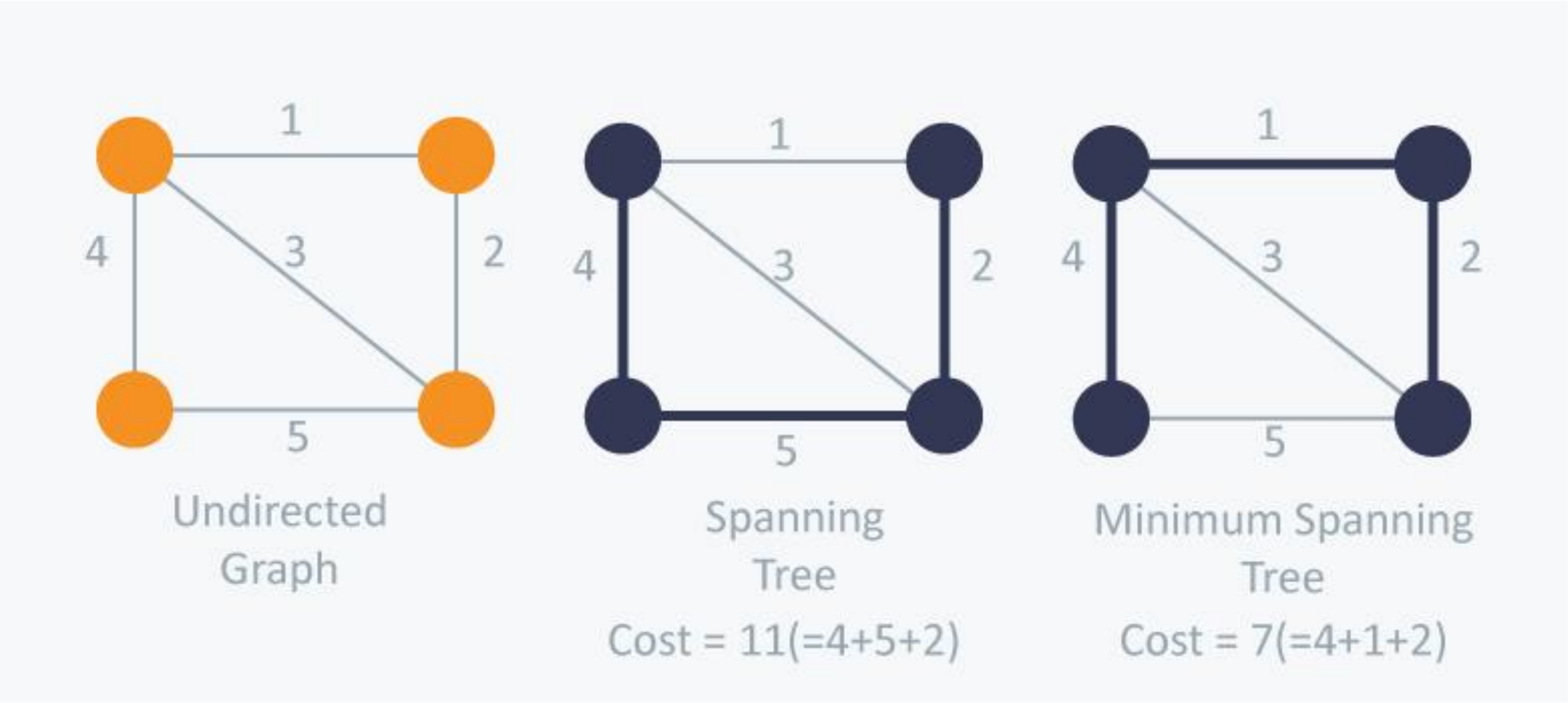
Total time $T(n)=O(|V|^2 + |E|) = O(|V|^2)$

Minimum Spanning Trees

Tree→ a connected, directed acyclic graph

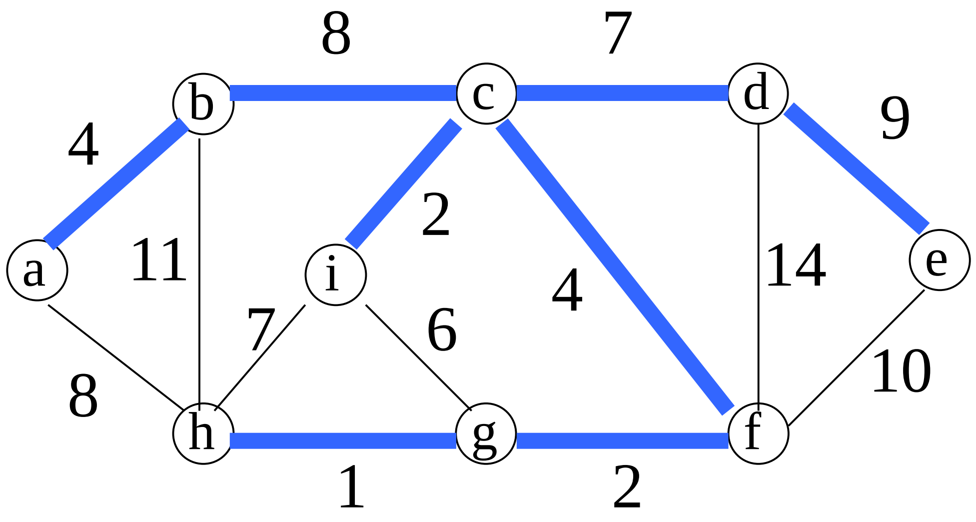
- Spanning tree: A subgraph of a graph, which meets the following constraints :
  - connected
  - acyclic
  - connects every vertex
- Minimum spanning tree(MST): A spanning tree with weight less than or equal to any other spanning tree for the given graph
- Two well-known algorithms for finding MST are Kruskal and Prim algorithms.

Minimum Spanning Tree : Example-1



Minimum Spanning Tree: Example-2

A connected graph and its MST:



Notice that the tree is not unique: replacing (b,c) with (a,h) yields another spanning tree with the same minimum weight.

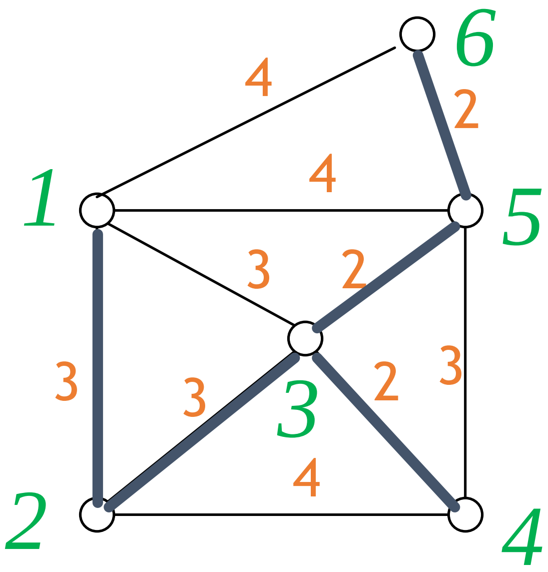
How to find? Given a connected weighted undirected graph, find subset of edges that spans all the nodes , creates no cycle, and minimizes the sum of weights.

Finding Minimum Spanning Tree: Greedy Solution

- Find a least-cost subset of the edges of a graph that connects all the nodes:
  - Start by picking any node and adding it to the tree
    - Repeatedly: Pick any *least-cost* edge from a node in the tree to a node not in the tree, and add the edge and new node to the tree.
    - Cycle is not permitted at any stage.
  - Stop when all nodes have been added to the tree.
  - This is the Kruskal algorithm.

Finding Minimum spanning tree: Greedy Solution

Example: Start from node 1



- Minimum spanning tree: 1-2-3-4-5-6

  The cost : 3+3+2+2+2=12
- Some other edge with a higher cost cannot be included in the spanning tree.

Generic MST Algorithm

Input: weighted undirected graph

G = (V,E,w)

T = ø (Initially empty ,it will include all the edges in the end)

while T is not yet a spanning tree of G

find an edge e in E such that T U {e} is a

subgraph of some MST of G

add e to T // Transfer one edge from E to T

return T //as MST of G

Kruskal’s MST Algorithm

//G = (V, E,w)

Algorithm Kruskal (G)

T ← ϕ   //The edges in MST , initially empty

Sort the m edges in G in increasing weight order

While | T |< |V| - 1 and E ≠ ϕ do

  Choose an edge (v, x) from E of lowest cost

  Delete (v, x) from E

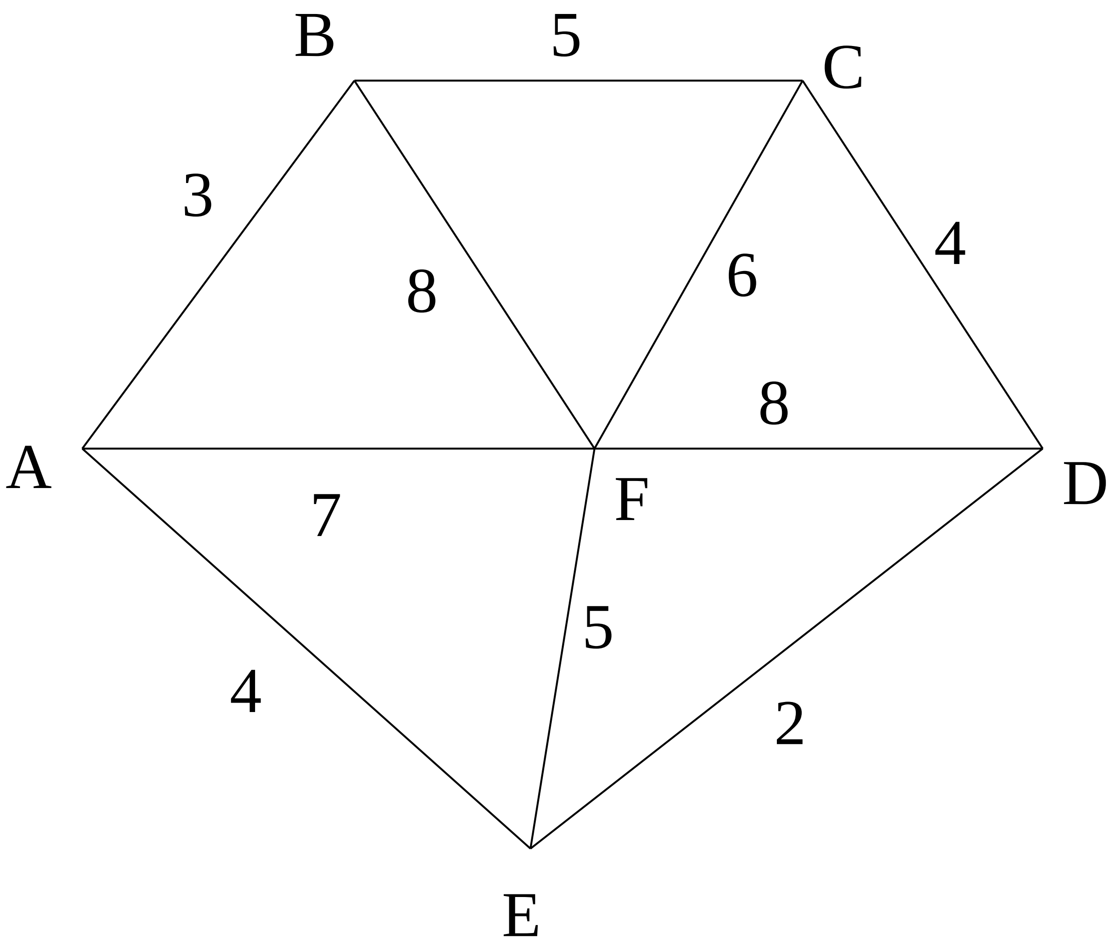
  If (v, x) does not create a cycle in T

    then add (v, x) to T

    else discard (v, x) //Unfeasible

Return T

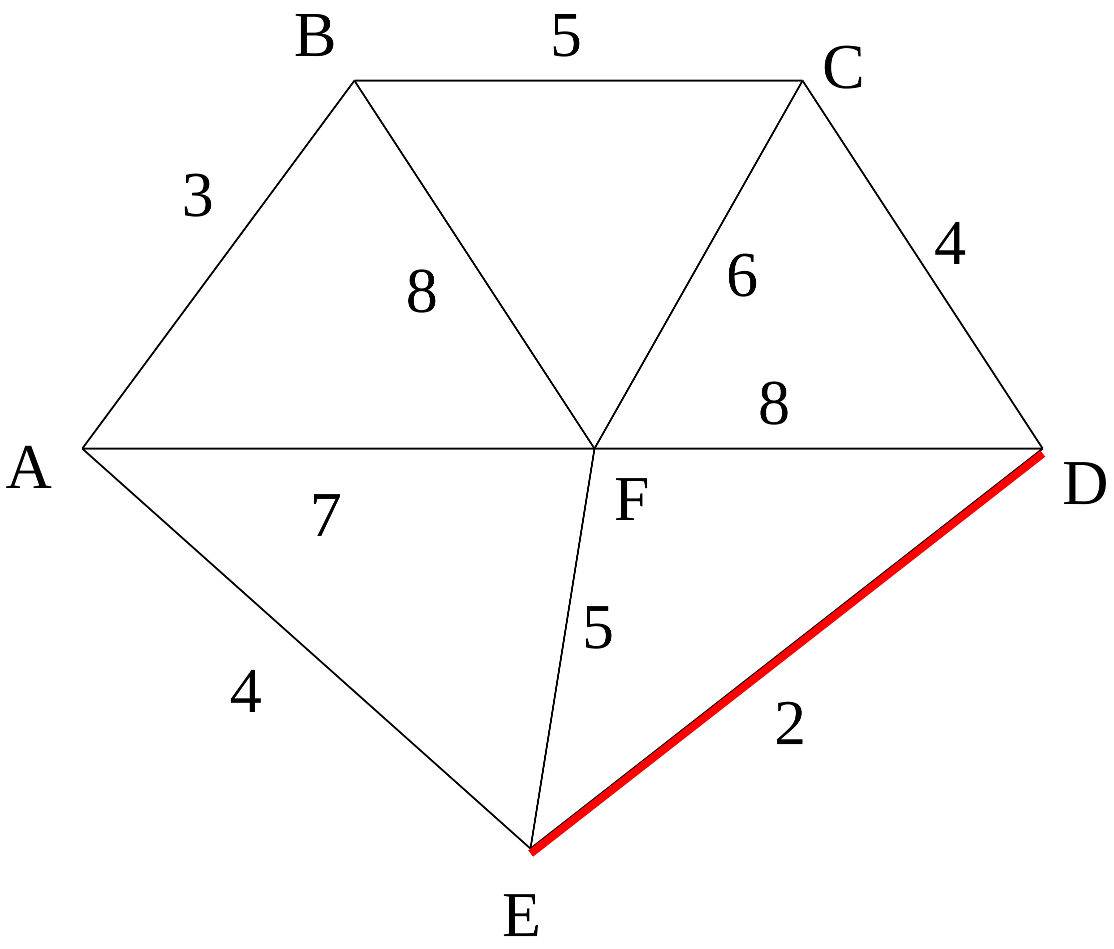
Trace of Kruskal’s Algorithm



List the edges in order of size(Sort):

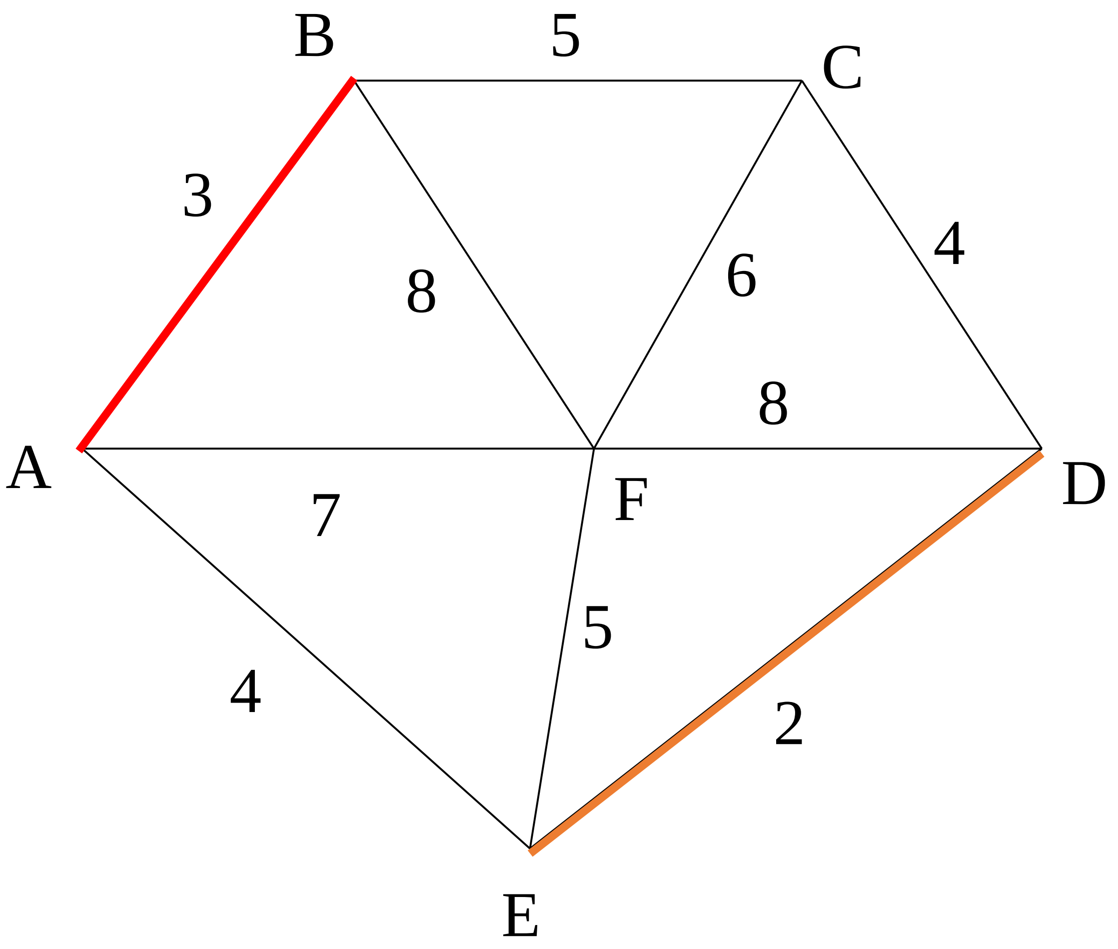
ED 2
AB 3
AE 4
CD 4
BC 5
EF 5
CF 6
AF 7
BF 8
CF 8

Kruskal’s Algorithm



Select the shortest edge in the network :ED
No cycle,add to T:
ED 2

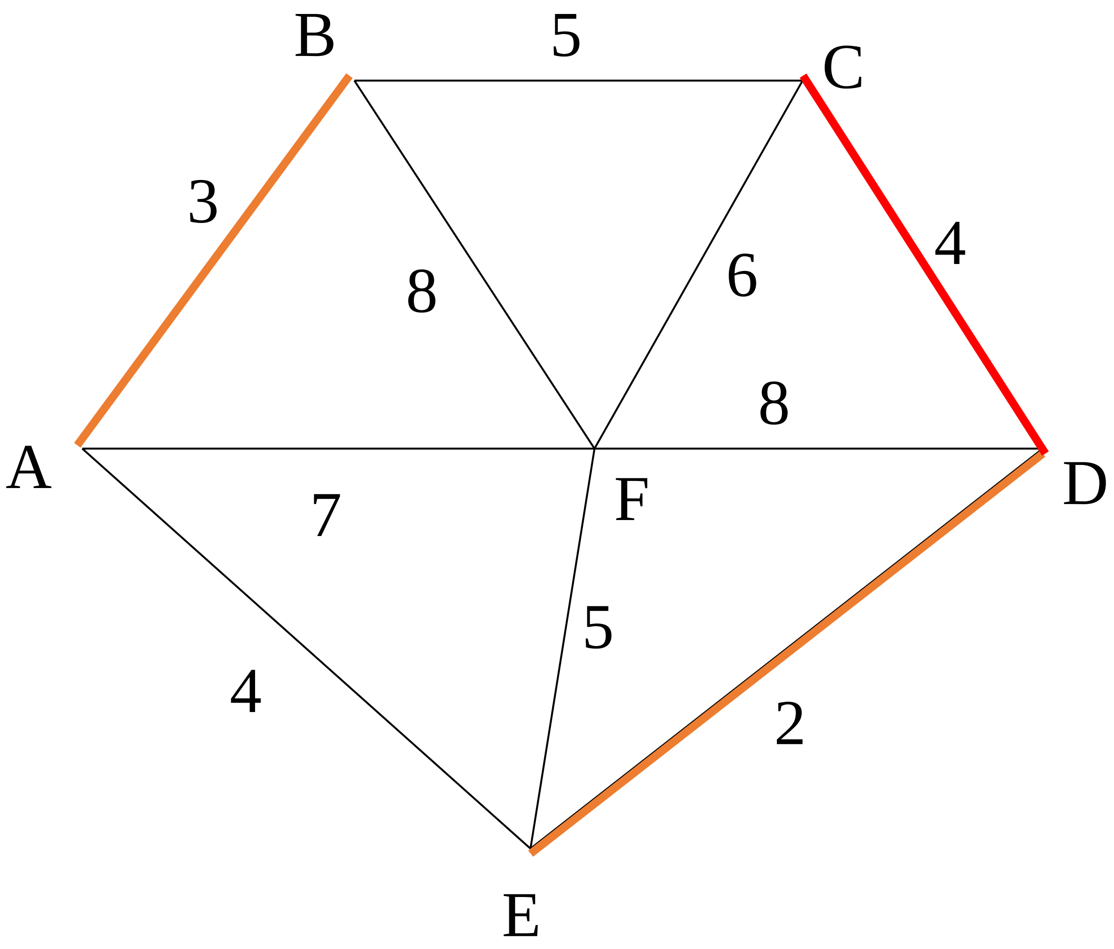
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle: AB

ED 2
AB 3

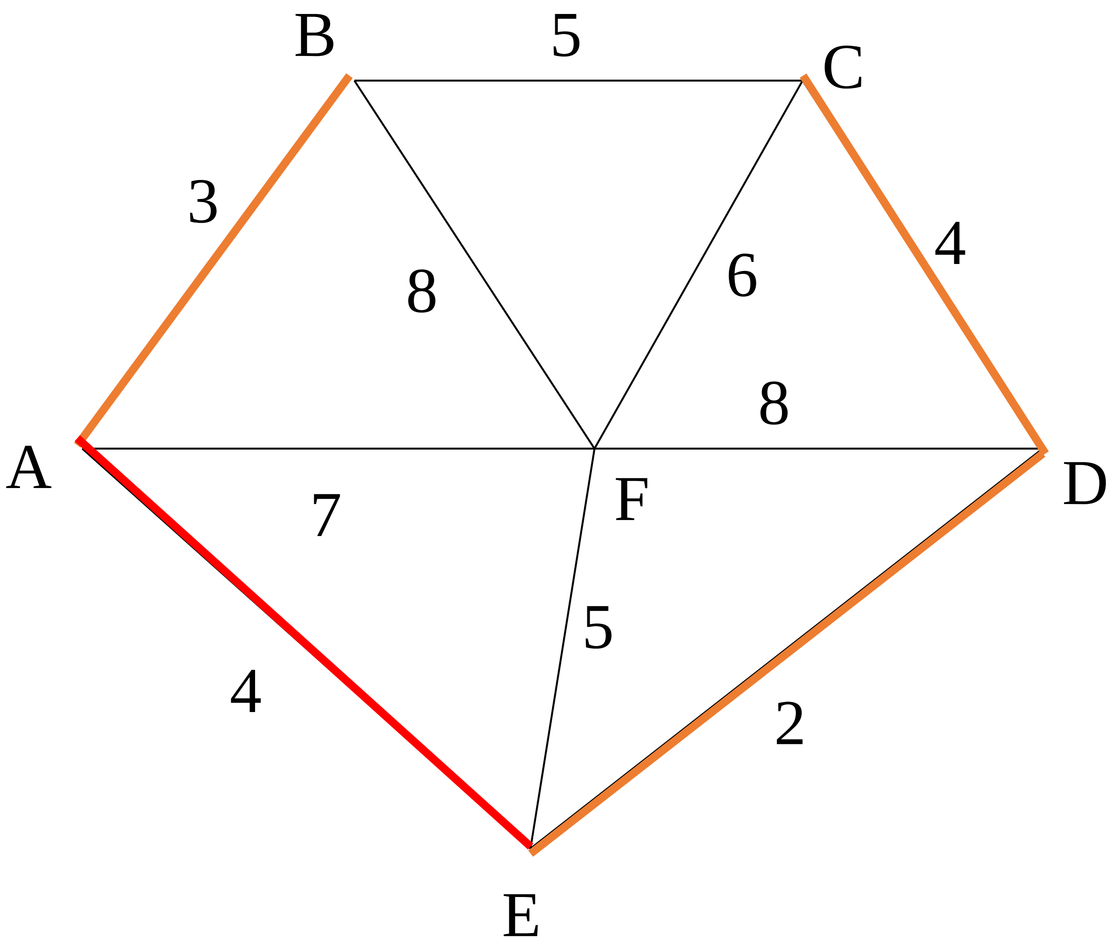
Kruskal’s Algorithm



Select the next shortest edge which does not create a cycle :CD,AE

ED 2
AB 3
CD 4 (or AE 4)

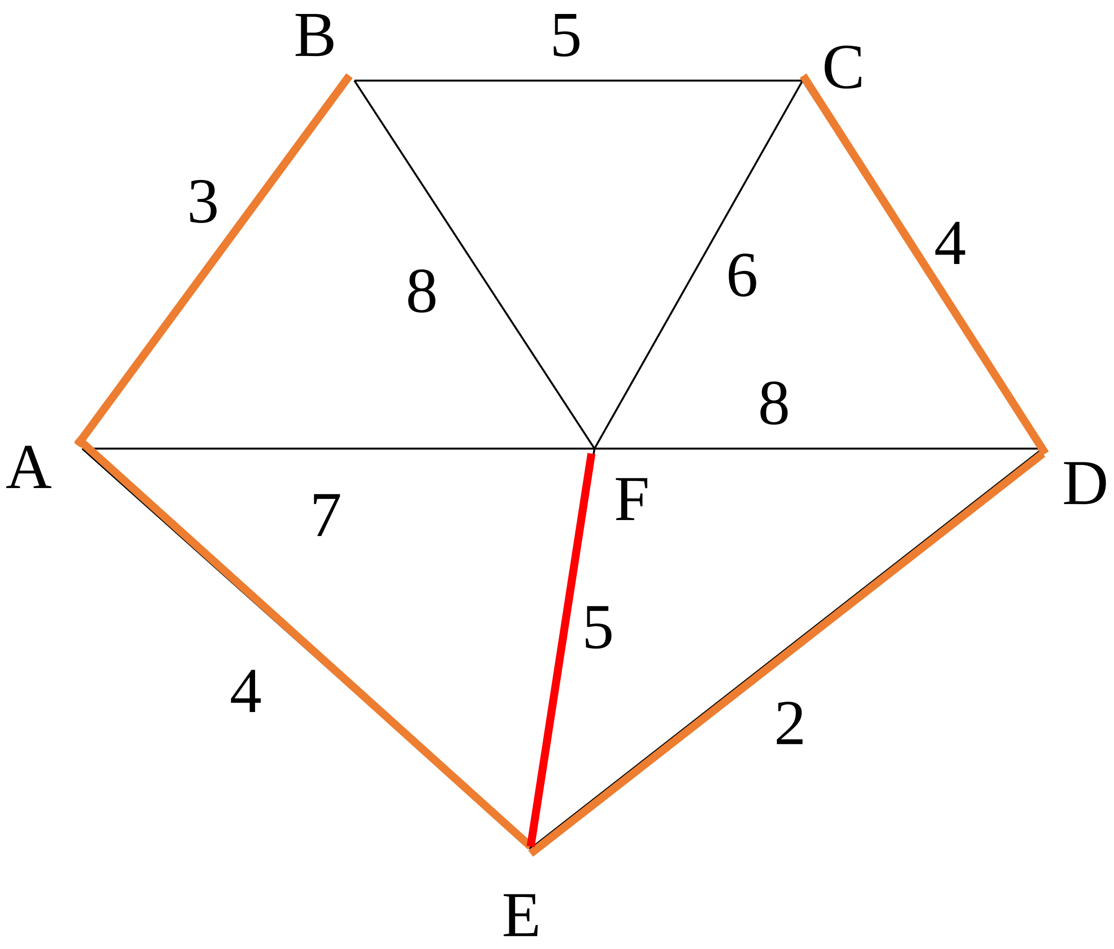
Kruskal’s Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4
AE 4

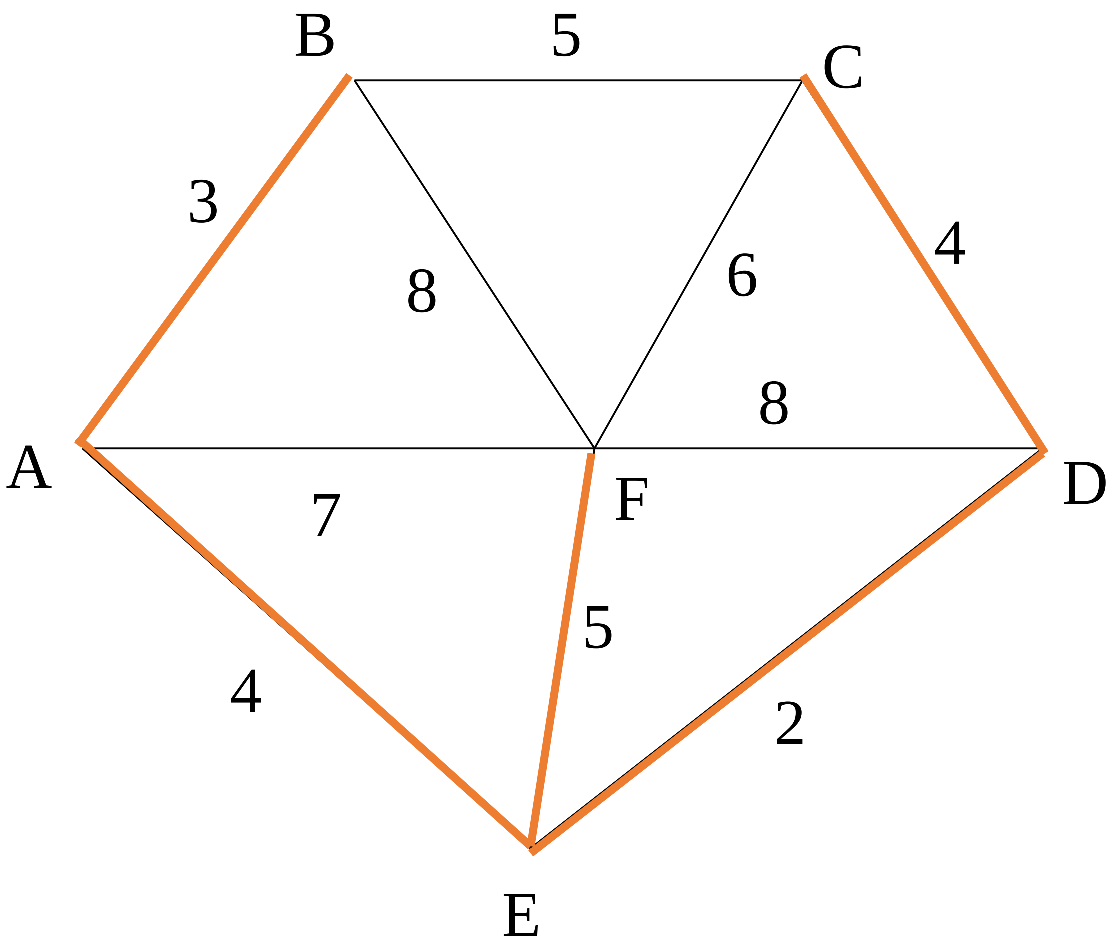
Kruskal’s Algorithm



Select the next shortest edge which does not create a cycle BC,EF

ED 2
AB 3
CD 4
AE 4
BC 5 – forms a cycle
EF 5

Kruskal's Algorithm



All vertices have been connected.

The solution is

ED 2
AB 3
CD 4
AE 4
EF 5

Total weight of MST = 18

Why is Kruskal's Algorithm Greedy?

- Algorithm manages a set of edges such that
  - these edges are a subset of some MST
- At each iteration:
  - choose an edge so that the MST-subset property remains true
  - Sub problem has to do the same with the remaining edges
- Always try to add cheapest available edge that will not violate the tree property

  →locally optimal choice

Time Complexity of Kruskal

- Sorting of edges takes $O(|E| \text{Log } |E|)$ time.
- After sorting, we iterate through all edges.
- The find and union operations can take at most $O(\text{Log}|V|)$ time for each tree edge [This result is based on a priority queue implementation which will be considered later]

→So overall complexity is $O(|E|\text{Log } |E| + |E|\text{Log } |V|)$ time.

- However, if we assume $|E| \sim |V|$, we can write

$$\log(|E|) <= \log(|V|^2)$$

$$O(\log(|E|) = O(2 \log(|V|)) = O(\log(|V|))$$

so $O(\text{Log}|V|)$ and $O(\text{Log}|E|)$ are the same .

- Therefore, overall time complexity is

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

Another Greedy MST Algorithm: Prim

- Kruskal's algorithm maintains a forest that grows until it forms a spanning tree
- Alternative idea is keep just one tree and grow it until it spans all the nodes:

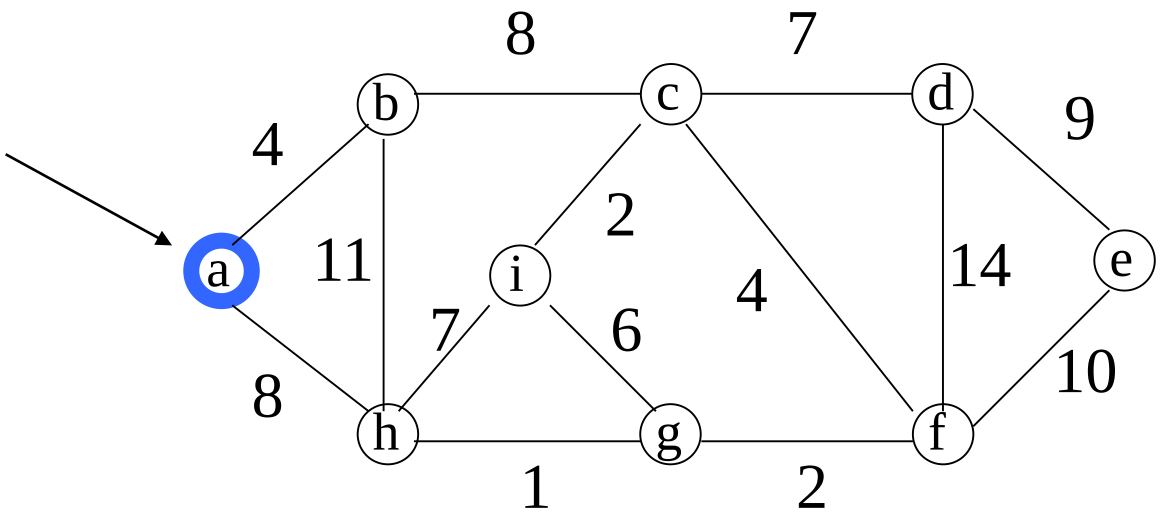
  Prim's algorithm
- At each iteration, choose the minimum weight outgoing edge : → Greedy!
- Problem: given a connected, undirected, weighted graph, find a *spanning tree* using edges that minimize the total weight.

Prim's Algorithm

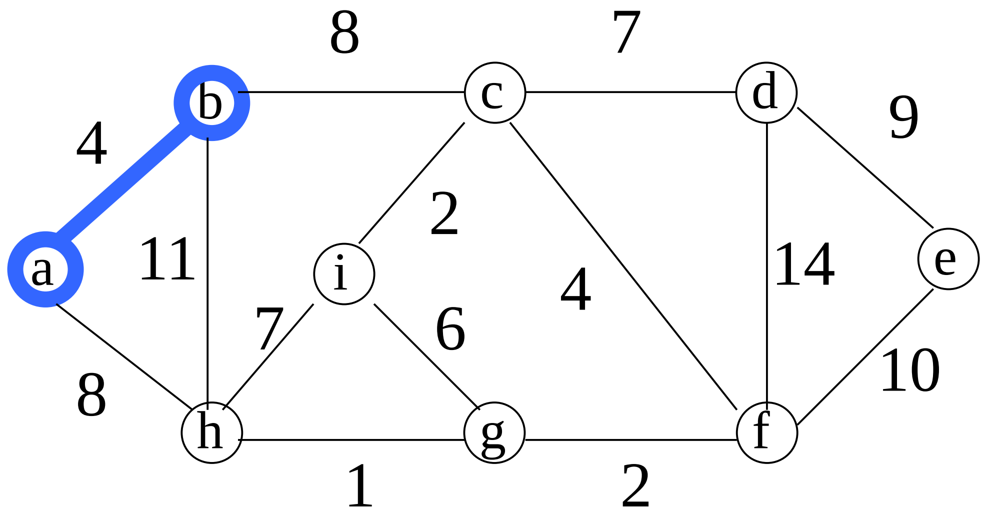
```
// input:  weighted undirected graph G = (V,E,w)
// r : Start node
MST_PRIM(G,r)
  T ← ϕ            //Edges in MST
  S←{r}          //The nodes in current tree
  Q←V-{r}    // Q: Remaning(outside) nodes
  while |T| < |V| - 1 do
     If  (u,v) is a min wt. outgoing edge  such that
          (u in S  and  v not in S)
     add (u,v) to T
     add v to S
     delete v from Q
return T
```

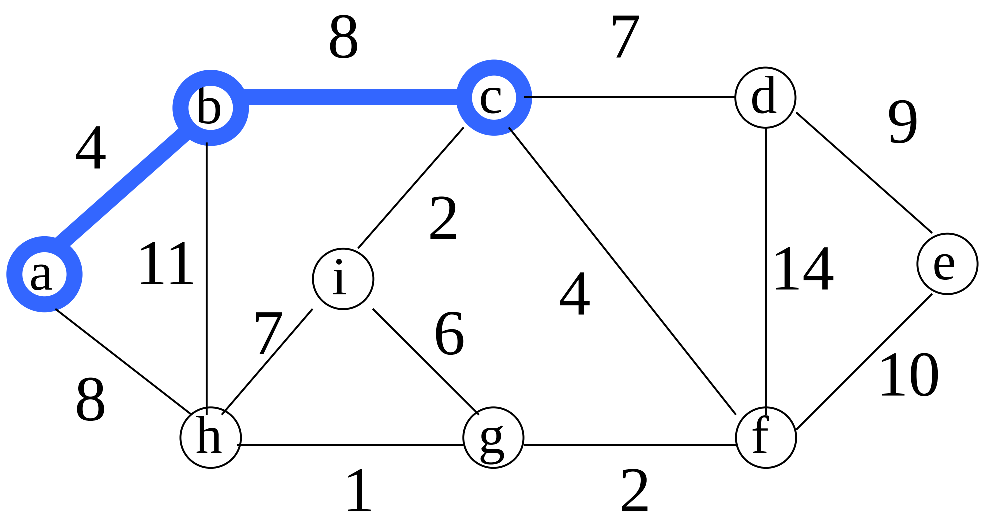
The execution of Prim's algorithm : Example

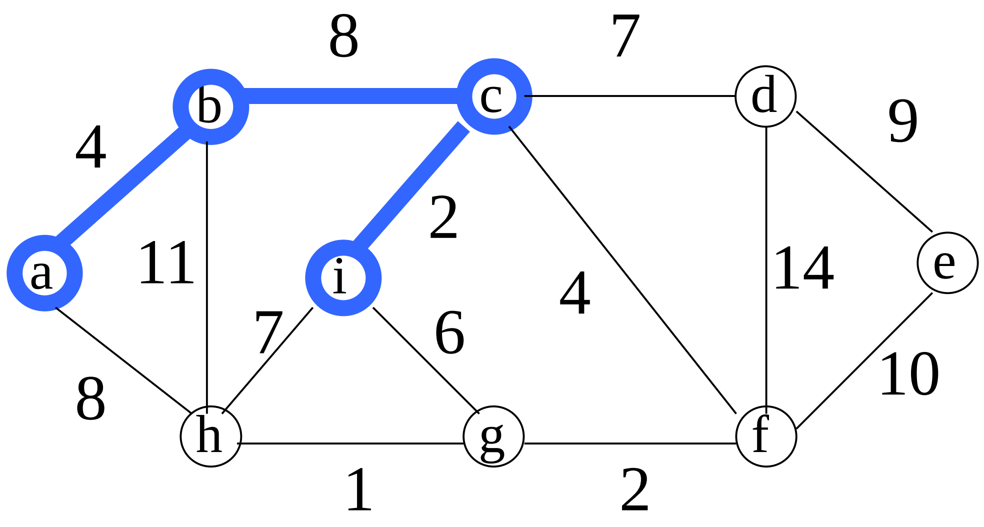
the root
vertex: a

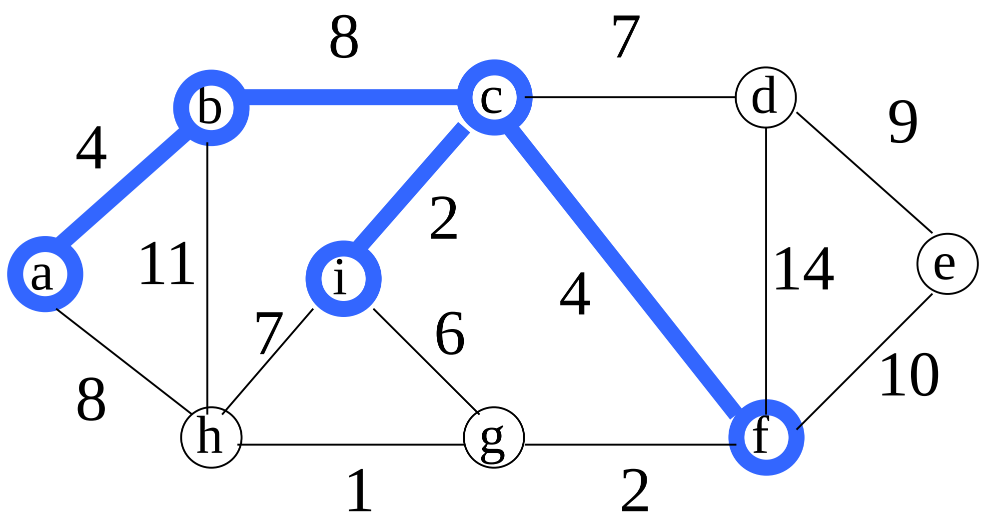


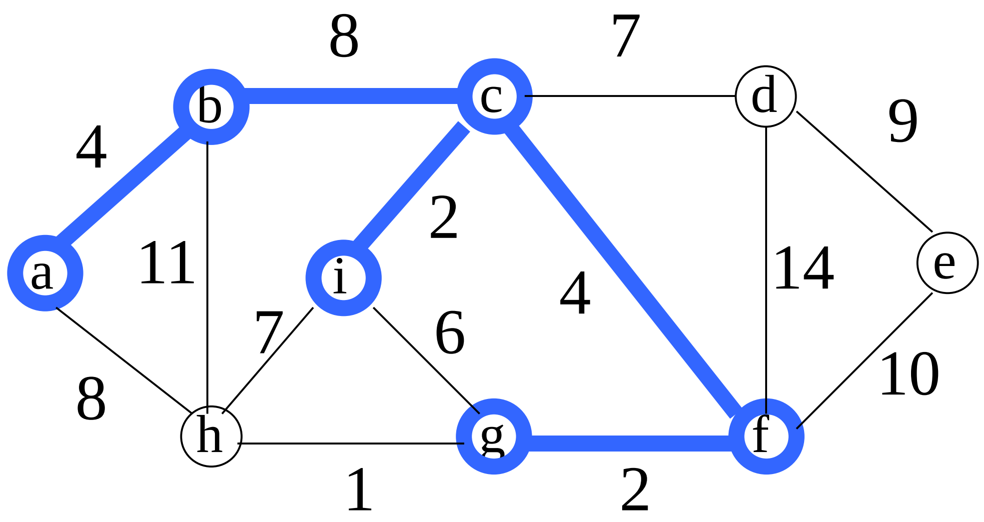
Min edge: (a,b)

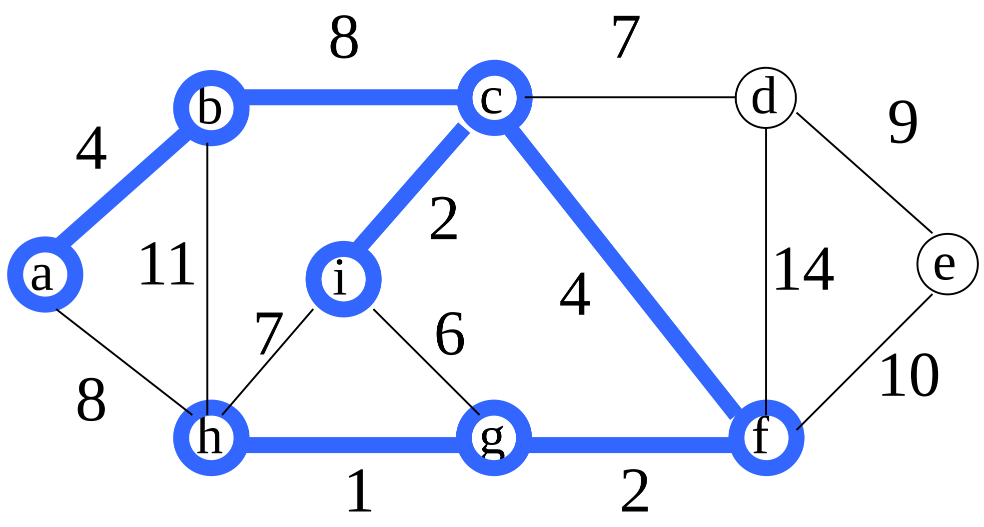



8
7
b
c
d
4
9
2
a
11
i
14
e
4
7
6
10
8
h
g
f
1
2

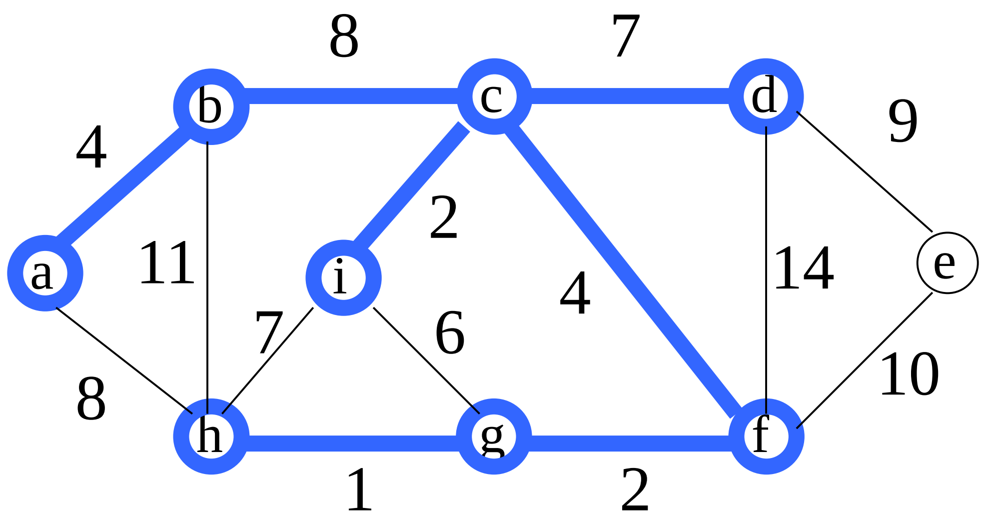

8
7
b
c
d
4
9
2
a
11
i
14
e
4
7
6
10
8
h
g
f
1
2


8
7
b
c
d
4
9
2
a
11
i
e
14
4
7
6
10
8
h
g
f
1
2

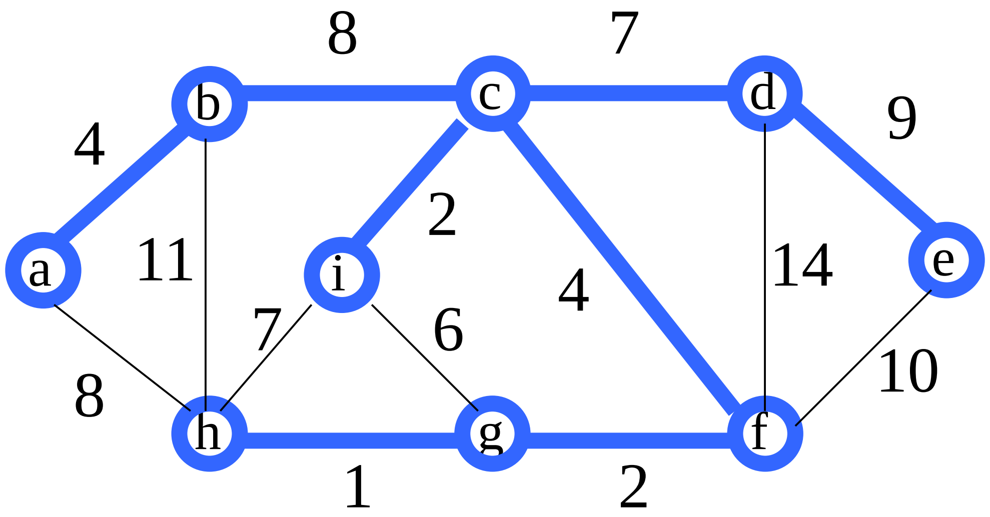

8
7
b
c
d
4
9
2
a
11
i
e
14
4
7
6
10
8
h
g
f
1
2



8
7
b
c
d
4
9
2
a
11
i
4
14
e
7
6
10
8
h
g
f
1
2



8
7
b
c
d
4
9
2
a
11
i
4
14
e
7
6
10
8
h
g
f
1
2



Total MST length = 37

Complexity of Prim's Algorithm

- Complexity depends on implementation. In our case:
- Initializations : O(|V|)
- While loop iteration: |V| times
- Each time we find a vertex, we must check all of its neighbors : O(|V|)
- Edge operations will be performed at most |E| times

With an adjacency list, complexity is:

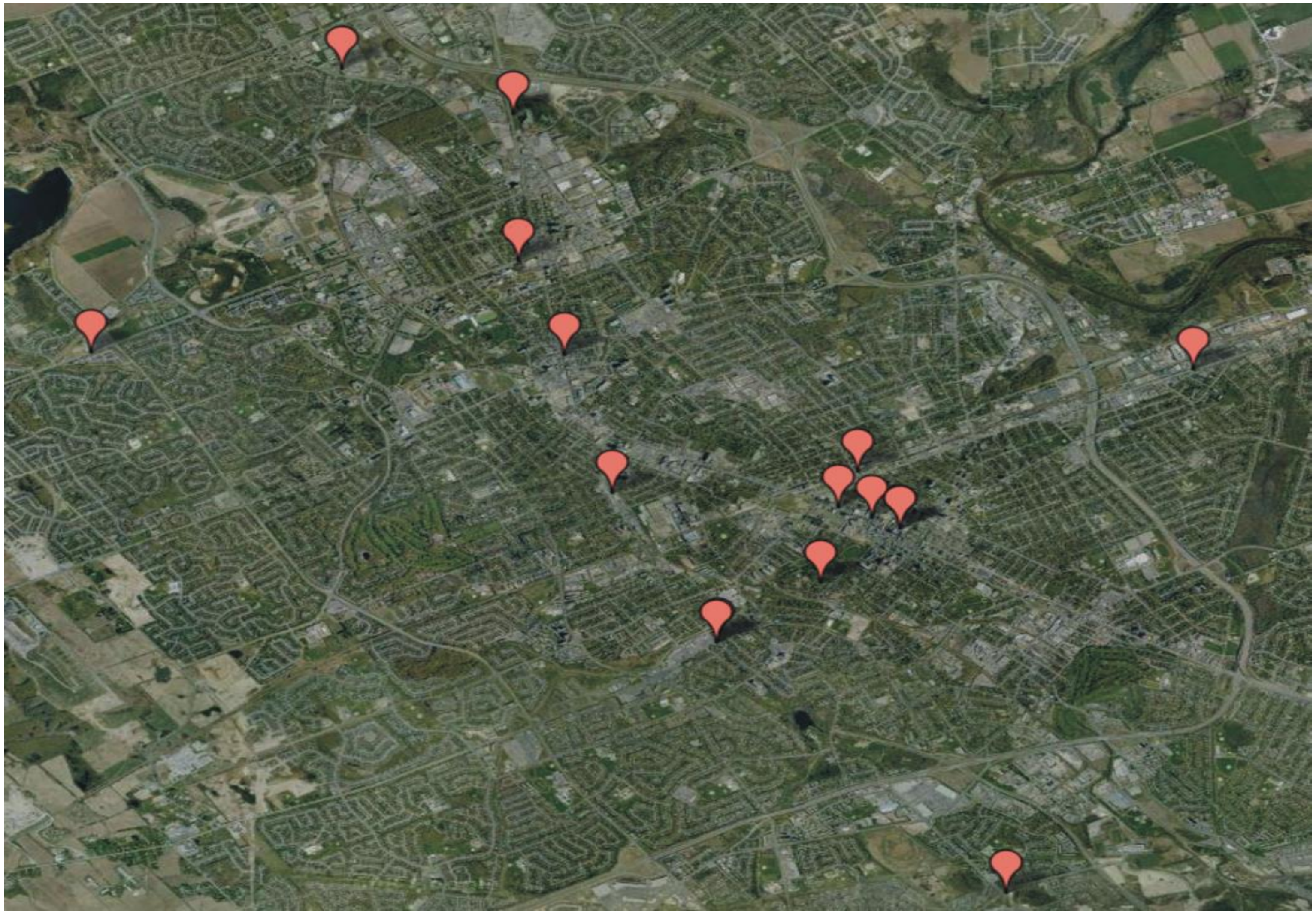
$$O(|V|^2 + |E|) = O(2|V|^2) \quad (\text{ Using } |E| = O(|V|^2))$$
$$= O(|V|^2)$$

This complexity can be reduced using more efficient implementations involving priority queues.

Applications of MST-1

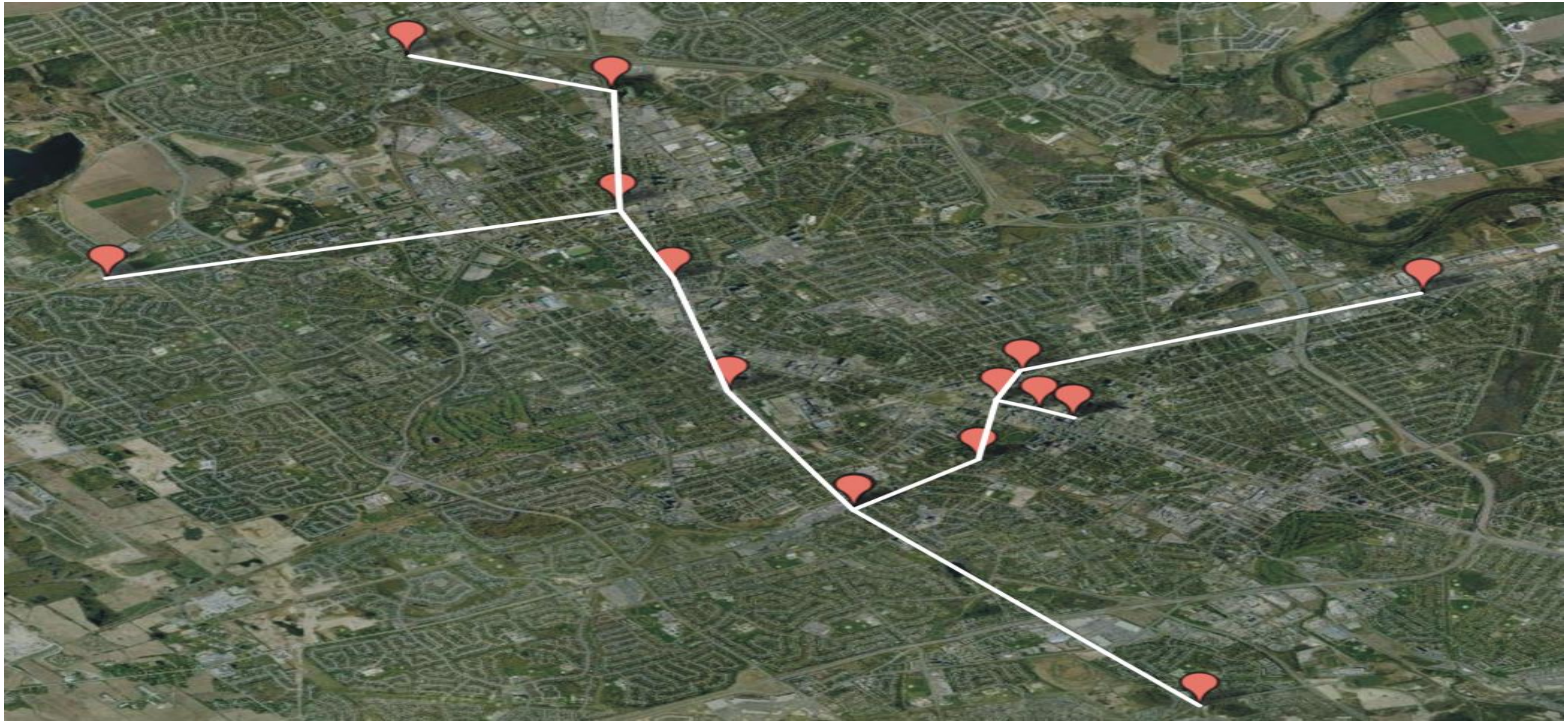
Consider attempting to find the best means of connecting a number of LANs

- Minimize the number of bridges
- Costs not strictly dependant on distances



Applications of MST-1

A minimum spanning tree will provide the optimal solution:



Applications of MST-2

- In the design of electronic circuitry, it is often necessary to make a set of pins electrically equivalent by wiring them together.
- To interconnect n pins, we can use n-1 wires, each connecting two pins.
- We want to minimize the total length of the wires.
- Minimum Spanning Trees can be used to model this problem.

Applications of MST-3

- Consider a cable TV company laying cable to a new neighborhood...
  - If it is constrained to bury the cable only along certain paths, then there would be a graph representing which points are connected by those paths.
  - Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper.
    - These paths would be represented by edges with larger weights.
  - A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house.
    - There might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost.

Prim's vs. Dijkstra's

- Both Prim's and Kruskal's Algorithms work with undirected graphs
- Both work with weighted and unweighted graphs but are more interesting when edges are weighted
- Both are greedy algorithms that produce optimal solutions.

Greedy Algoritms :Selecting Breakpoints

- Input: a planned route with n + 1 gas stations $b0$, … , bn; the car can go at most C after refueling at a breakpoint
- Output: a refueling schedule ($b0$→bn) that minimizes the number of stops.

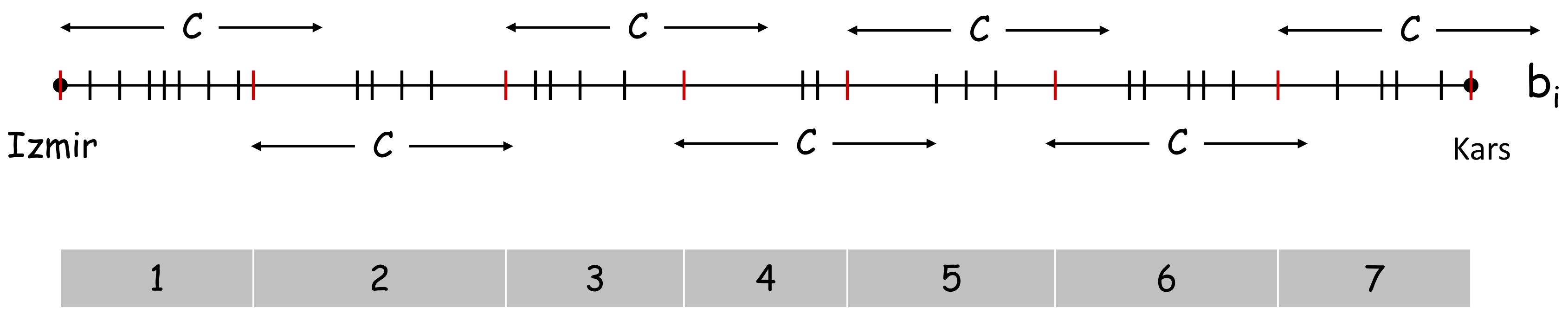
Greedy choice: go as far as you can before refueling (select bj)

Selecting Breakpoints

Selecting breakpoints:

- Example : Road trip from Izmir to Kars along a fixed route.
- Refueling stations at certain points along the way.
- Fuel capacity = C   (C is distance !)
- Goal: make as few refueling stops as possible.

Greedy algorithm. Go as far as you can before refueling.
→Truck Driwer's Algorithm



Selecting Breakpoints: Greedy Algorithm

"Truck Driver's" algorithm.

```
BP_Select(C,b)
Sort breakpoints:b₀ < b₁ < b₂ < ... < bₙ
S ← {0}          ← breakpoints selected
x ← 0            ← current location
while (x ≠ bₙ)
   let p be largest integer such that bₚ ≤ (x + C)
   if (bₚ = x)
      return "no solution"
   x ← bₚ
   S ← S ∪ {p}
return S
```

Complexity : $(n \log n) + n = O(n \log n)$

Unfeasible Greedy Solutions

For some problems, it may be possible that not even a feasible greedy solution can be found.

Example1:Solving Sudoku

- Consider the following greedy algorithm for solving Sudoku:
  - For each empty square, starting at the top-left corner and going across:
  - Fill that square with the smallest number which does not violate any of our conditions
  - All feasible solutions have equal weight.

Unfeasible Greedy Example

Let’s try this algorithm on the previously seen Sudoku square:

8			6					2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Unfeasible Greedy Example

Neither 1 nor 2 fits into the first empty square, so fill it with 3

8	3		6					2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Unfeasible Greedy Example

The second empty square may be filled with 1

8	3	1	6					2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Unfeasible Greedy Example

And the 3rd empty square may be filled with 4

8	3	1	6	4				2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Unfeasible Greedy Example

At this point, we try to fill in the 4th empty square

8	3	1	6	4	?			2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Unfeasible Greedy Example

Unfortunately, all nine numbers 1 – 9 already appear in such a way to block it from appearing in that square !

→There is no known greedy algorithm which finds the one feasible solution

8	3	1	6	4	?			2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Sub-Optimal Greey Solution :TSP

The Traveling Salesman Problem(TSP): You want to cycle through *n* cities without visiting the same city twice.

- Assumption : It is possible to go from any one city to another.

Use The *Nearest Neighbor* Algorithm which is greedy:

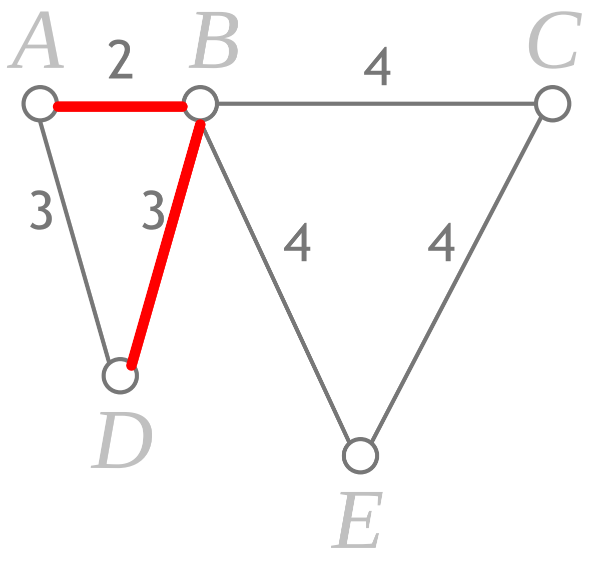
→Always go to the closest city which has not yet been visited.

Is this solution feasible ? YES !

Is it optimal ? NO,it is unlikely to be optimal!

Traveling salesman: Suboptimal Greedy Solution

Greedy algorithm: He goes to the next nearest city from wherever he is



From ***A*** he goes to ***B***
From ***B*** he goes to ***D***

....

This is *not* going to result in a shortest path!

The best result he can get now will be ***ABDBCE***, at a cost of 16

An actual least-cost path from ***A*** is ***ADBCE***, at a cost of 14

Greedy Algorithms : Summary

- Choose the best possible option at each step
- This decision usually (but not always) leads to the best overall solution.
- Greedy Choice :A globally optimal solution is derived from a locally optimal (greedy) choice.

→ When choices are considered, the choice that looks best in the current problem is chosen, without considering results from subproblems

Optimal Substructure :A problem has optimal substructure if an optimal solution to the problem is composed of optimal solutions to subproblems.