Greedy Method
Minimum Cost Spanning Trees &Single source shortest path

Dr. K.RAGHAVA RAO

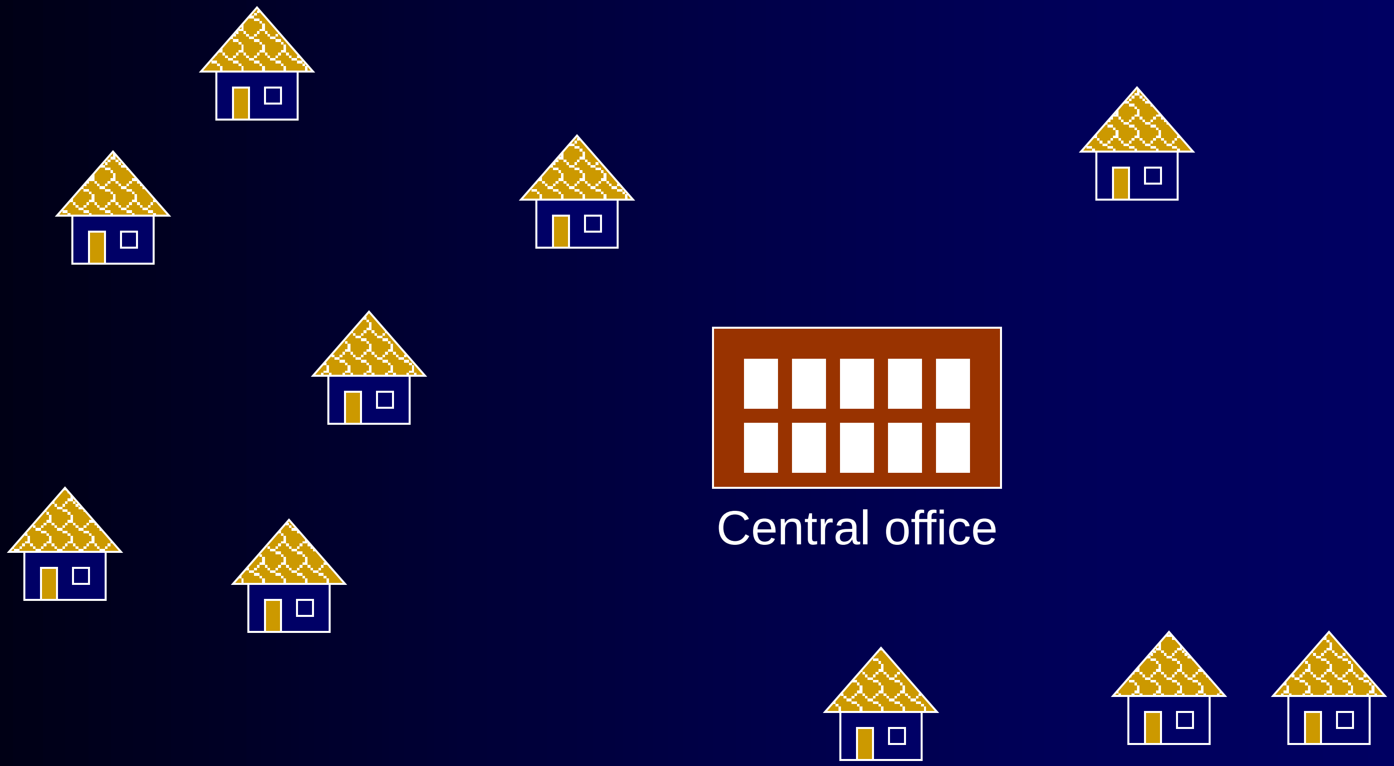
Professor in CSE

KL University

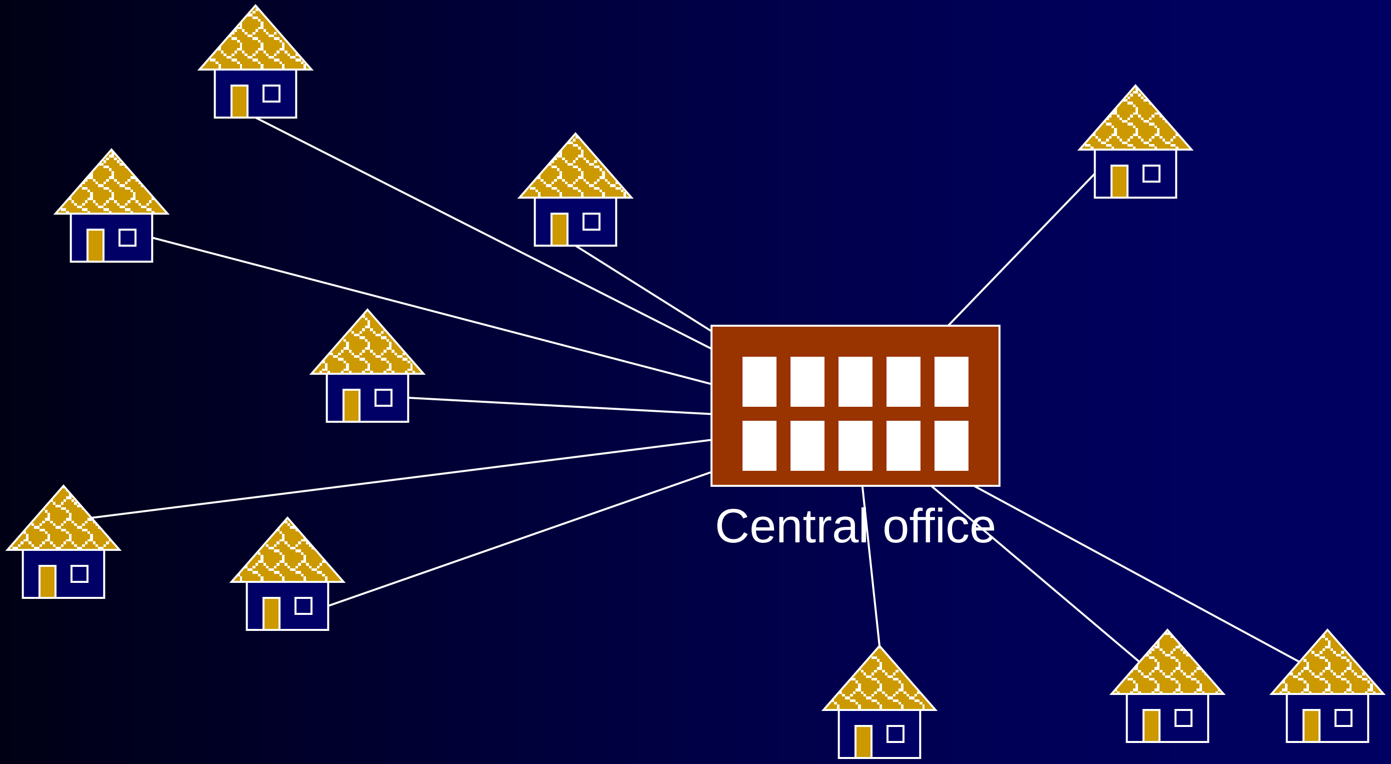
krraocse@gmail.com

http://mcadaa.blog.com

Problem: Laying Telephone Wire

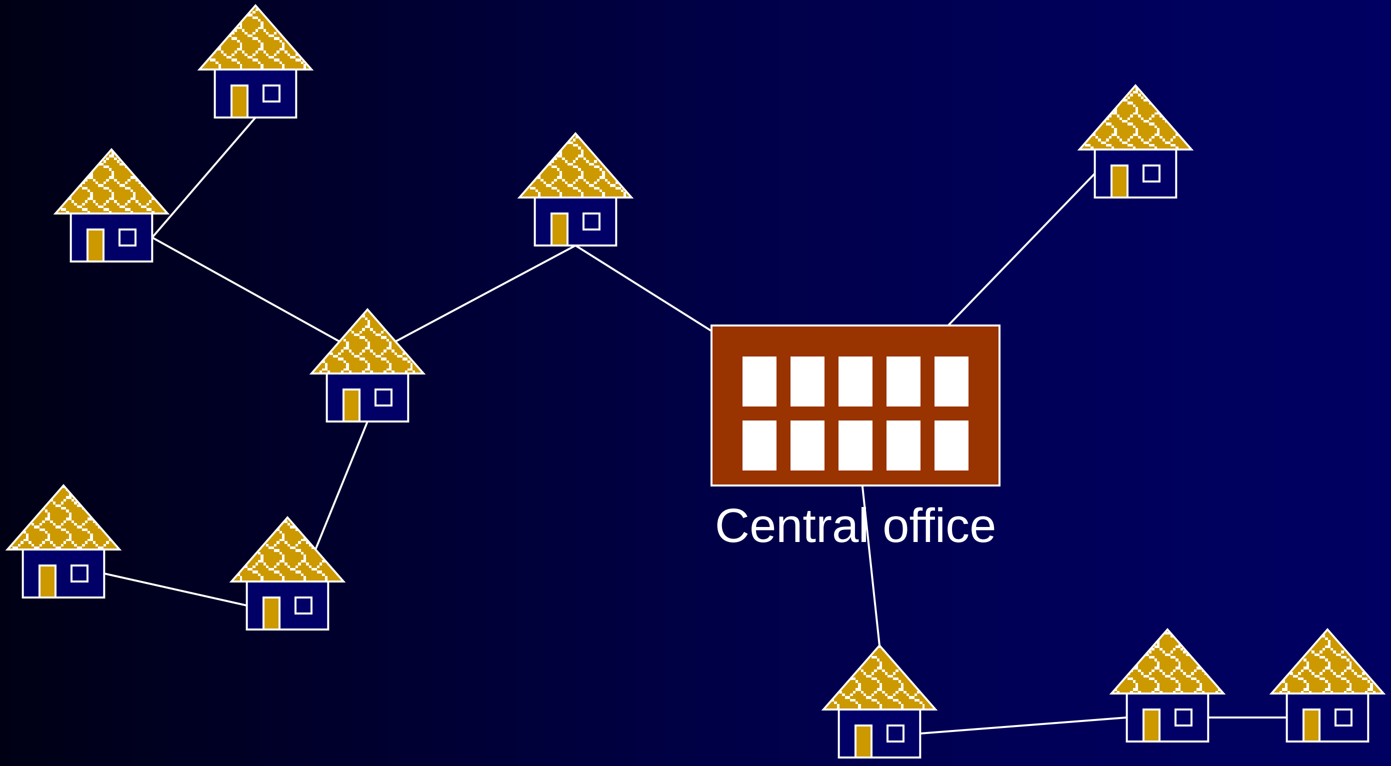


Wiring: Naïve Approach



Expensive!

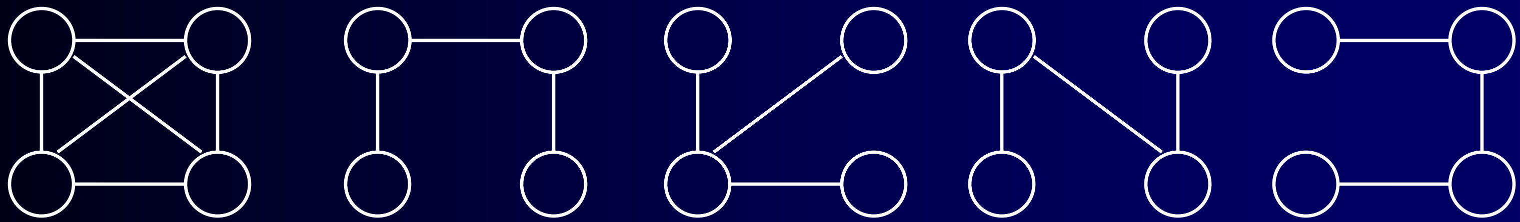
Wiring: Better Approach



Minimize the total length of wire connecting the customers

Spanning trees

- Suppose you have a connected undirected graph
  - Connected: every node is reachable from every other node
  - Undirected: edges do not have an associated direction
- ...then a spanning tree of the graph is a connected subgraph in which there are no cycles

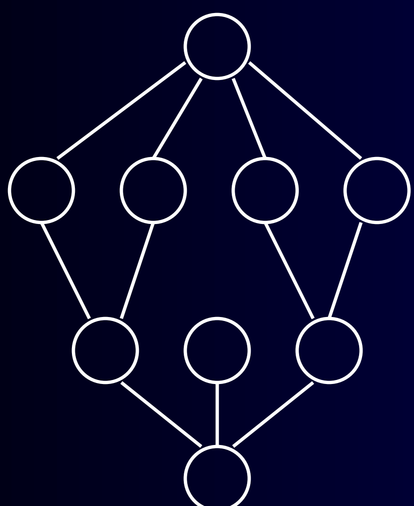


A connected, undirected graph

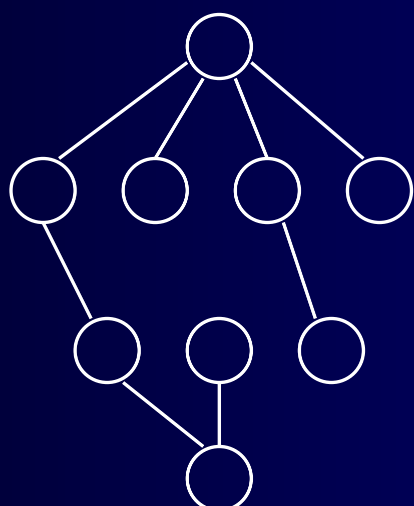
Four of the spanning trees of the graph

Finding a spanning tree

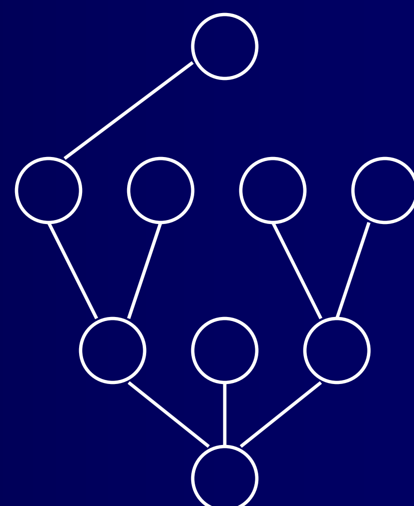
- To find a spanning tree of a graph,
  - pick an initial node and call it part of the spanning tree
  - do a search from the initial node:
    - each time you find a node that is not in the spanning tree, add to the spanning tree both the new node *and* the edge you followed to get to it



An undirected graph



Result of a BFS starting from top



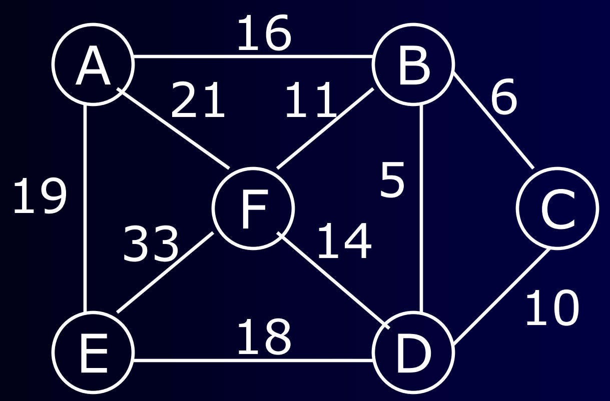
Result of a DFS starting from top

Minimizing costs

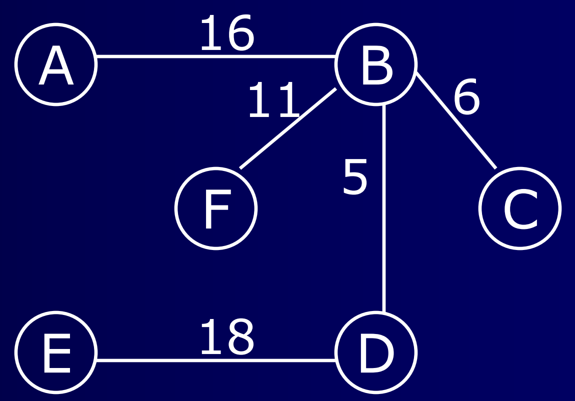
- Suppose you want to supply a set of houses (say, in a new subdivision) with:
  - electric power
  - water
  - sewage lines
  - telephone lines
- To keep costs down, you could connect these houses with a spanning tree (of, for example, power lines)
  - However, the houses are not all equal distances apart
- To reduce costs even further, you could connect the houses with a *minimum-cost* spanning tree

Minimum-cost spanning trees

- Suppose you have a connected undirected graph with a weight (or cost) associated with each edge
- The cost of a spanning tree would be the sum of the costs of its edges
- A minimum-cost spanning tree is a spanning tree that has the lowest cost



A connected, undirected graph

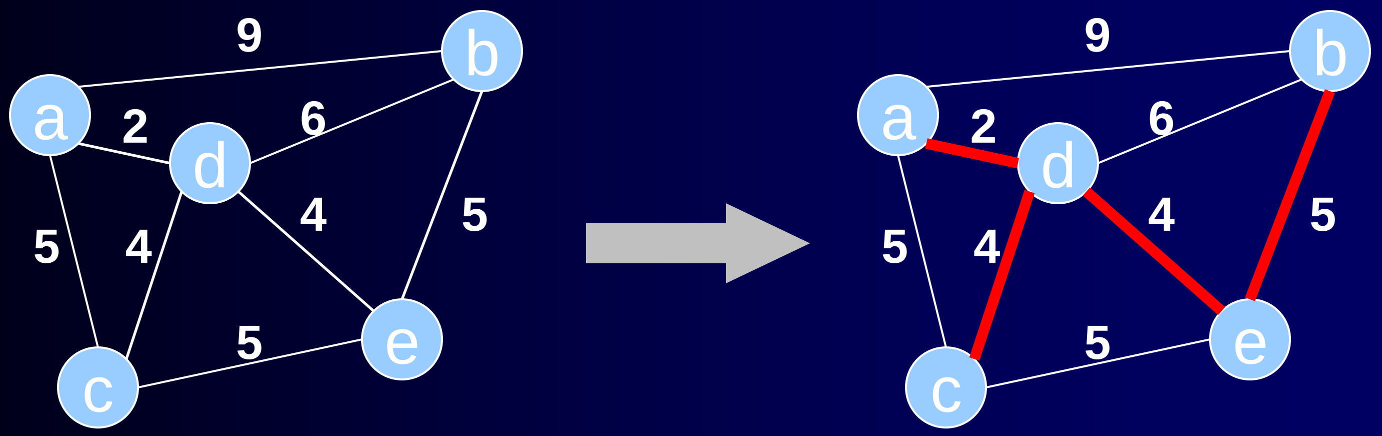


A minimum-cost spanning tree

Greedy Approach

- Both Prim's and Kruskal's algorithms are **greedy algorithms**
- The greedy approach works for the MST problem; however, **it does not work for many other problems**!

How Can We Generate a MST?



Prim's algorithm

```
T = a spanning tree containing a single node s;
E = set of edges adjacent to s;
while T does not contain all the nodes {
    remove an edge (v, w) of lowest cost from E
    if w is already in T then discard edge (v, w)
    else {
        add edge (v, w) and node w to T
        add to E the edges adjacent to w
    }
}
```

- An edge of lowest cost can be found with a priority queue
- Testing for a cycle is automatic

Prim's Algorithm

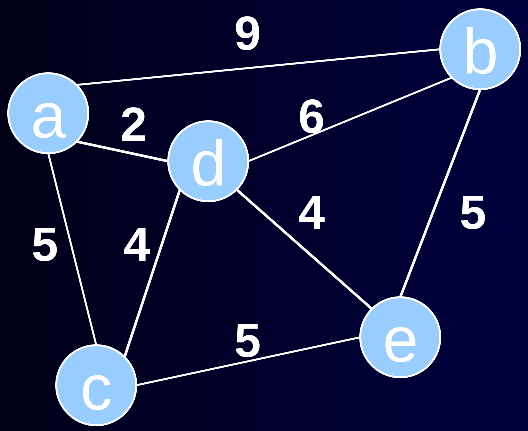
Initialization

a. Pick a vertex r to be the root

b. Set $D(r) = 0$, $parent(r) = null$

c. For all vertices $v \in V$, $v \neq r$, set $D(v) = \infty$

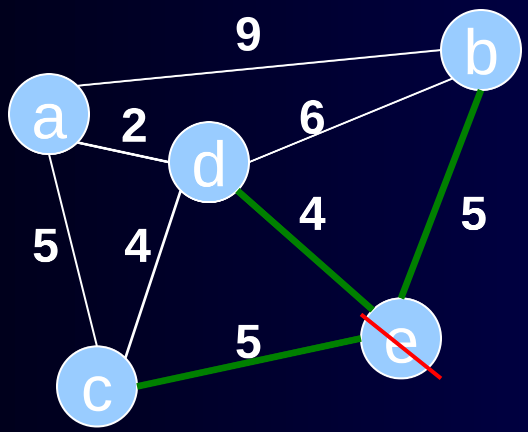
d. Insert all vertices into priority queue P, using distances as the keys



e	a	b	c	d
0	∞	∞	∞	∞

Vertex	Parent
e	-

Prim's algorithm



e	d	b	c	a
0	∞	∞	∞	∞

Vertex	Parent
e	-
b	-
c	-
d	-

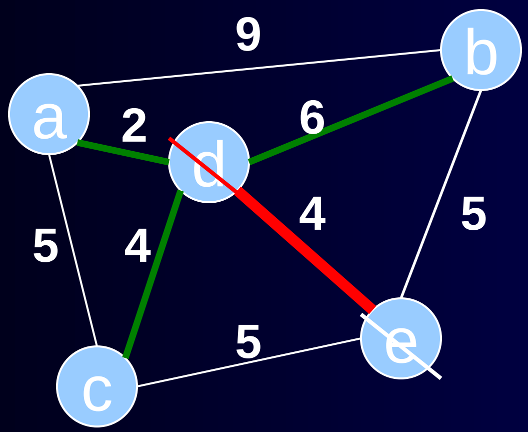
---

d	b	c	a
4	**5**	**5**	∞

Vertex	Parent
e	-
b	e
c	e
d	e

The MST initially consists of the vertex ***e***, and we update the distances and parent for its adjacent vertices

Prim's algorithm



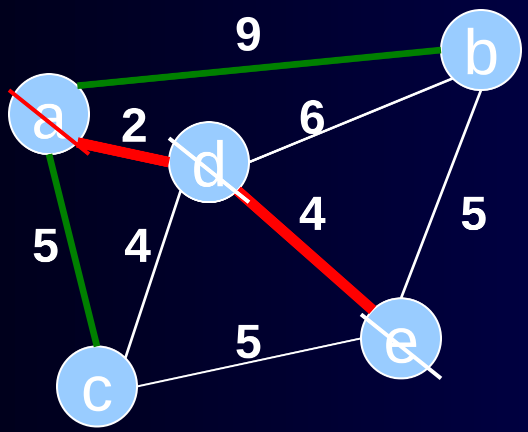
d	b	c	a
4	**5**	**5**	∞

Vertex	Parent
e	-
b	e
c	e
d	e

a	c	b
2	**4**	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim's algorithm



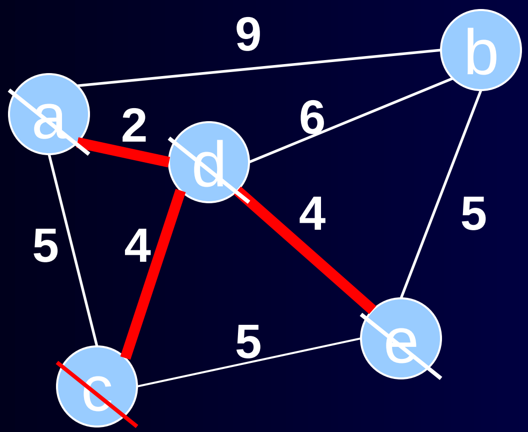
a	c	b
2	**4**	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

c	b
4	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim's algorithm



c	b
4	**5**

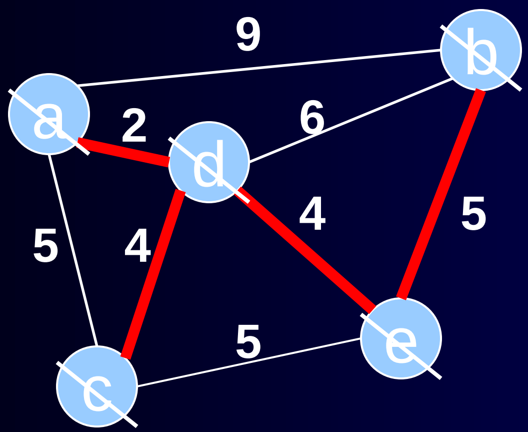
Vertex	Parent
e	-
b	e
c	d
d	e
a	d

---

b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim's algorithm



b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

The final minimum spanning tree

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Running time of Prim's algorithm (without heaps)

Initialization of priority queue (array): O($|V|$)

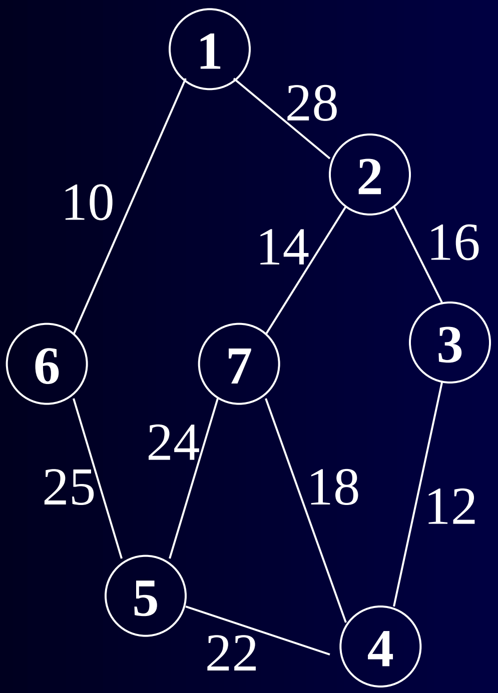
Update loop: $|V|$ calls

- Choosing vertex with minimum cost edge: O($|V|$)
- Updating distance values of unconnected vertices: each edge is considered only **once** during entire execution, for a **total** of O($|E|$) updates

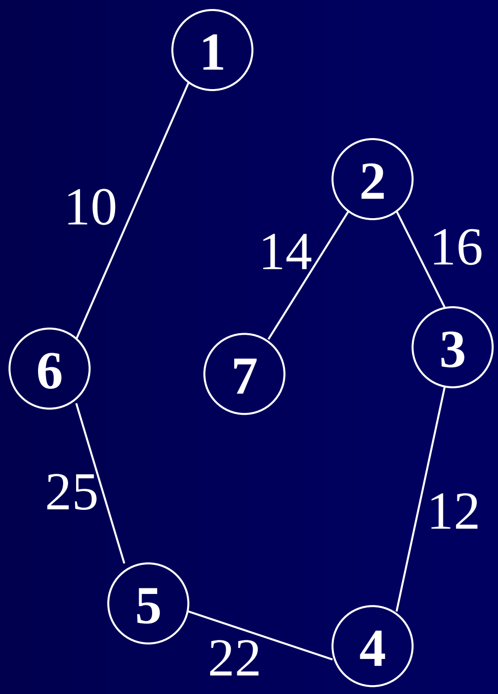
Overall cost without heaps: $\boxed{O(|E| + |V|^2)}$

Minimum-cost Spanning Trees

- Example of MCST
  - Finding a spanning tree of G with minimum cost



(a)



(b)

Prim's Algorithm

- Example

Prim’s Algorithm Invariant

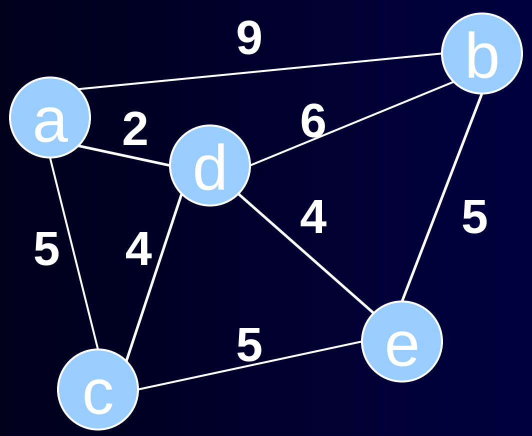
- At each step, we add the edge (***u,v***) s.t. the weight of (***u,v***) is **minimum** among all edges where ***u*** is in the tree and ***v*** is not in the tree

- Each step maintains a minimum spanning tree of the vertices that have been included thus far

- When all vertices have been included, we have a MST for the graph!

Another Approach

- Create a forest of trees from the vertices
- Repeatedly merge trees by adding "**safe edges**" until only one tree remains
- A "safe edge" is an edge of minimum weight which does not create a cycle



forest: {a}, {b}, {c}, {d}, {e}

Kruskal's algorithm

```
T = empty spanning tree;
E = set of edges;
N = number of nodes in graph;
while T has fewer than N - 1 edges {
    remove an edge (v, w) of lowest cost from E
    if adding (v, w) to T would create a cycle
        then discard (v, w)
        else add (v, w) to T
}
```

- Finding an edge of lowest cost can be done just by sorting the edges
- Running time bounded by sorting (or findMin)
- $O(|E|\log|E|)$, or equivalently, $O(|E|\log|V|)$

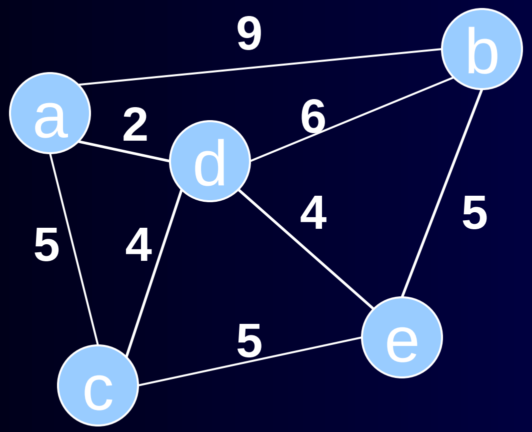
Kruskal's algorithm

Initialization

a. Create a set for each vertex $v \in V$

b. Initialize the set of "safe edges" A comprising the MST to the empty set

c. Sort edges by increasing weight

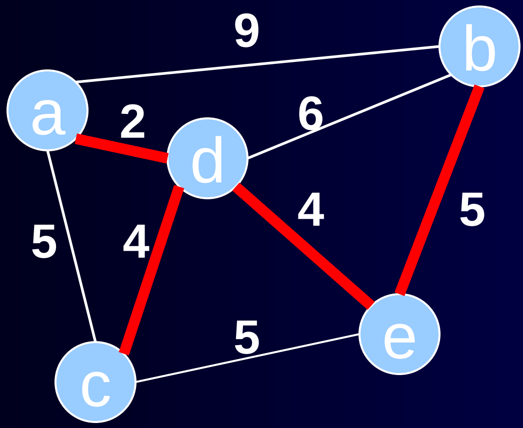


F = {a}, {b}, {c}, {d}, {e}

$A = \varnothing$

E = {(a,d), (c,d), (d,e), (a,c), (b,e), (c,e), (b,d), (a,b)}

Kruskal's algorithm



E = {~~(a,d)~~, ~~(c,d)~~, ~~(d,e)~~, ~~(a,c)~~,
~~(b,e)~~, (c,e), (b,d), (a,b)}

Forest	*A*
{a}, {b}, {c}, {d}, {e}	$\varnothing$
{a,d}, {b}, {c}, {e}	{(a,d)}
{a,d,c}, {b}, {e}	{(a,d), (c,d)}
{a,d,c,e}, {b}	{(a,d), (c,d), (d,e)}
{a,d,c,e,b}	{(a,d), (c,d), (d,e), (b,e)}

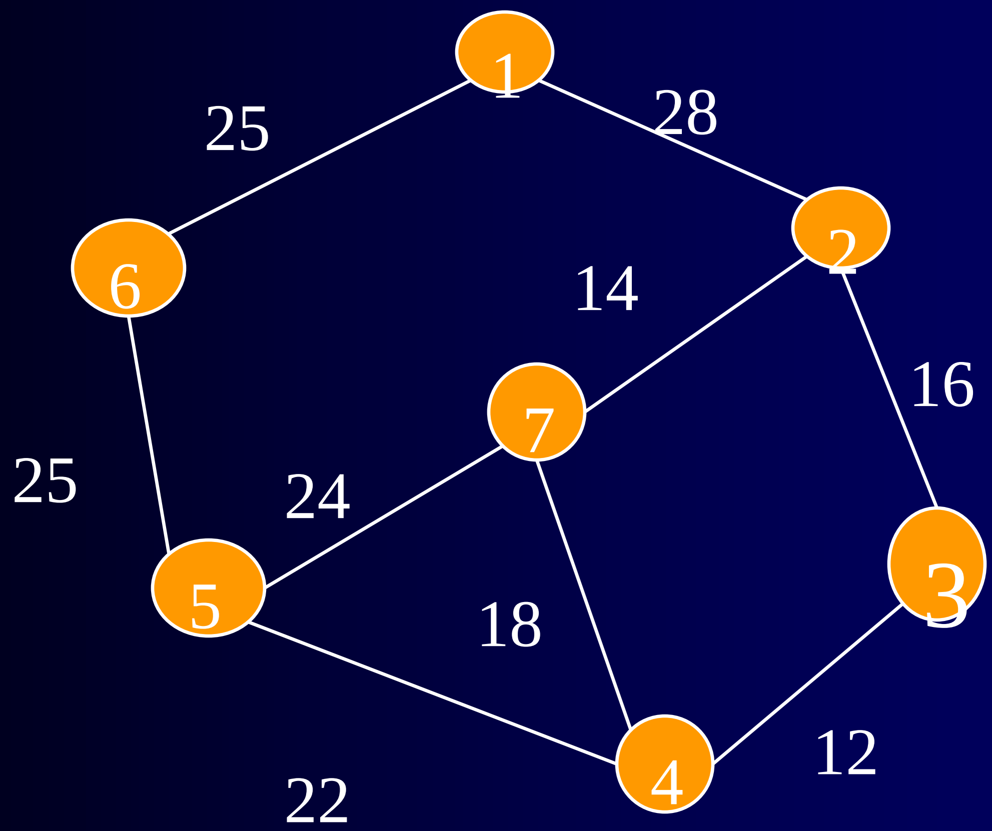
Kruskal's algorithm **Invariant**

- After each iteration, every tree in the forest is a MST of the vertices it connects
- Algorithm terminates when all vertices are connected into one tree

Kruskal’s Algorithm

- Example

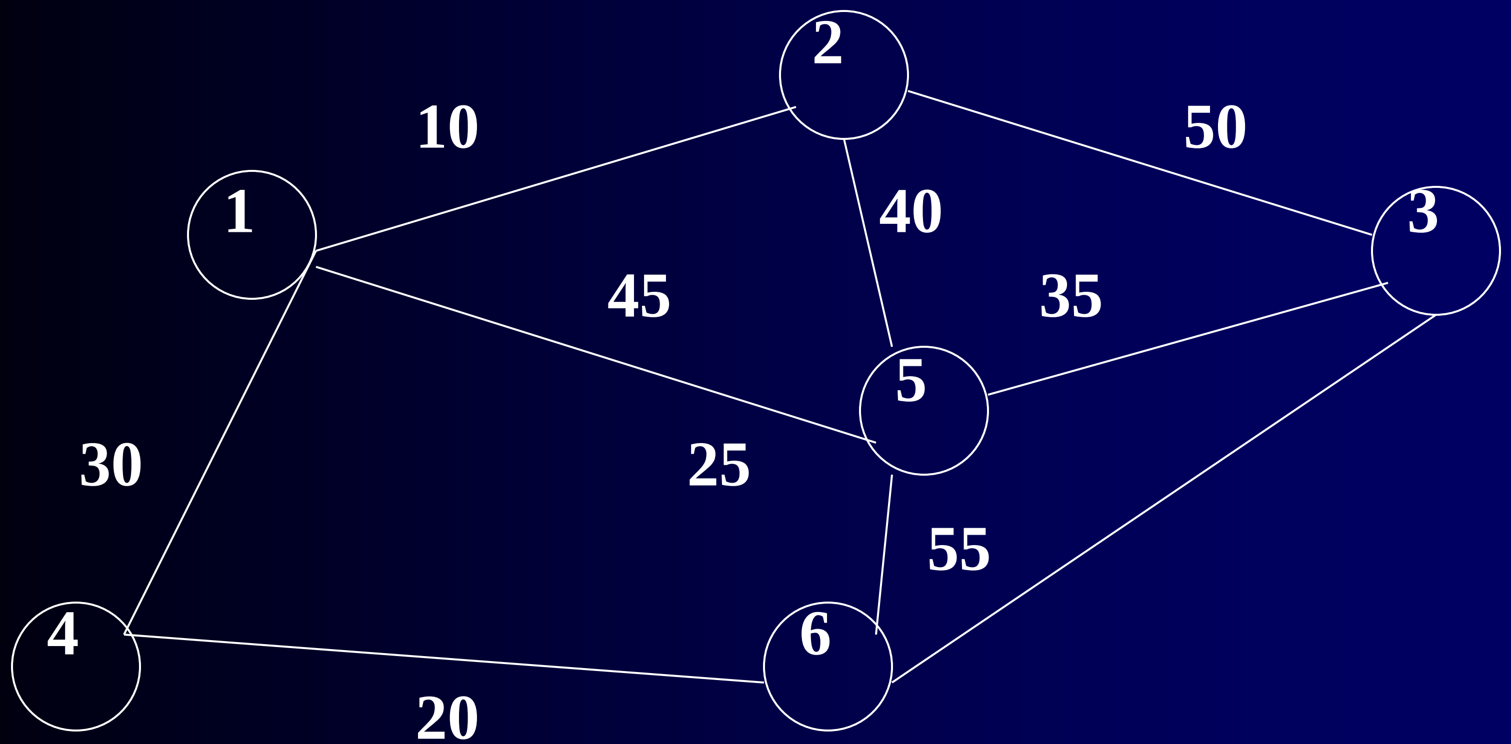
Exercise-1: compute MST for this graph using prim's and kruskal's algorithm



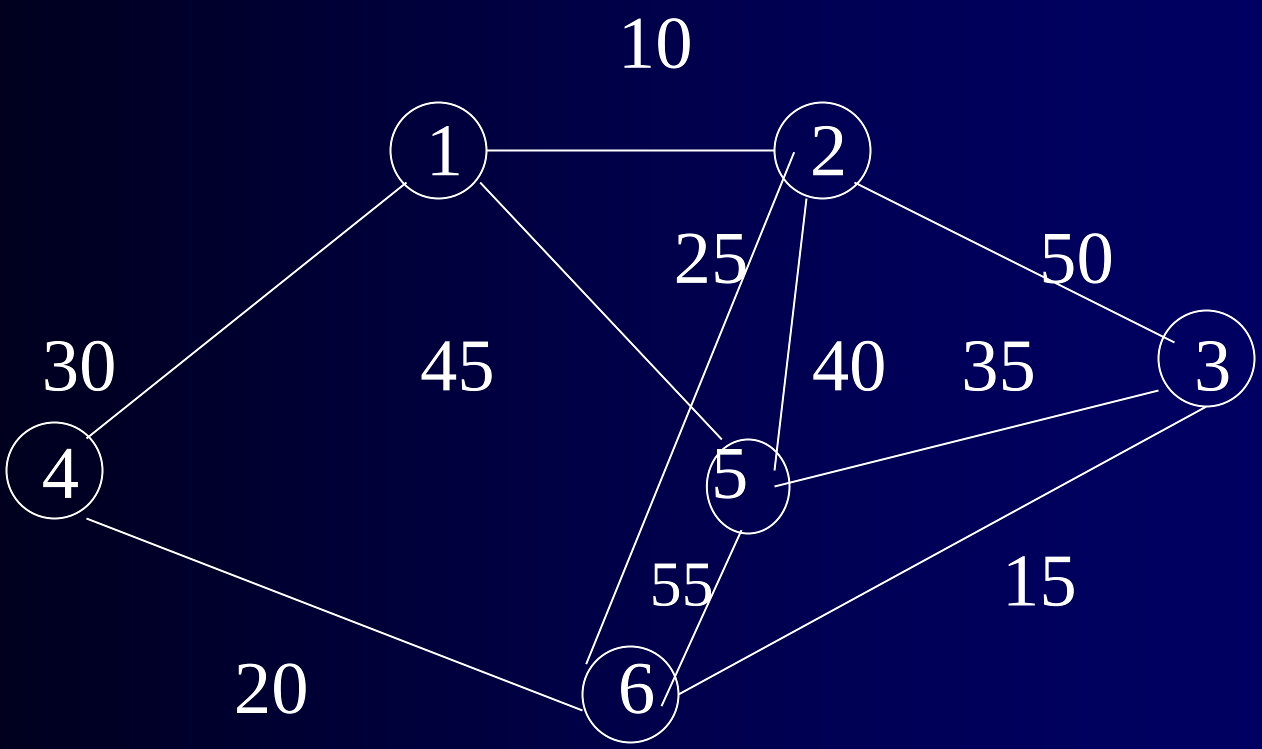
Mst cost is 25 pair 5,6

1-6-5-4-3-2-7

Exercise-2: compute MST for this graph using prim's and kruskal's algorithm



Exercise-3: compute MST for this graph using prim's and kruskal's algorithm

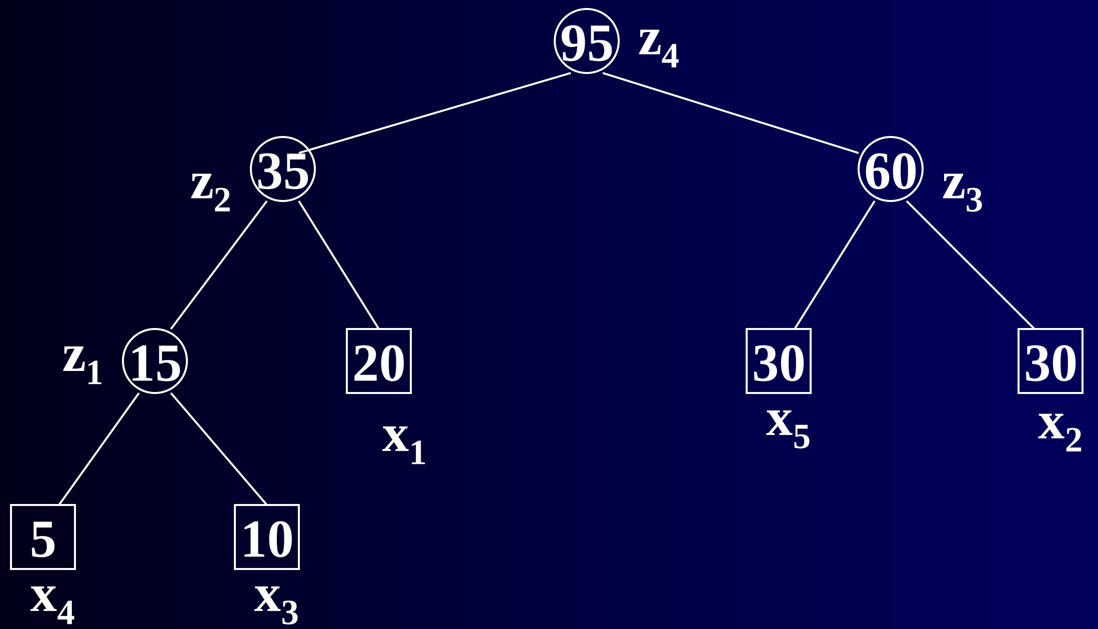


Optimal Merge Patterns

- Problem
  - Given n sorted files, find an optimal way (i.e., requiring the fewest comparisons or record moves) to pairwise merge them into one sorted file
  - It fits ordering paradigm
- Example
  - Three sorted files (x_1,x_2,x_3) with lengths (30, 20, 10)
  - Solution 1: merging x_1 and x_2 (50 record moves), merging the result with x_3 (60 moves) → total 110 moves
  - Solution 2: merging x_2 and x_3 (30 moves), merging the result with x_1 (60 moves) → total 90 moves
  - The solution 2 is better

Optimal Merge Patterns

- A greedy method (for 2-way merge problem)
  - At each step, merge the two smallest files
  - e.g., five files with lengths (20,30,10,5,30) (Figure 4.11)



$\sum_{i=1}^{n} d_i q_i$

Total number of record moves = weighted external path length

The optimal 2-way merge pattern = binary merge tree with minimum weighted external path length

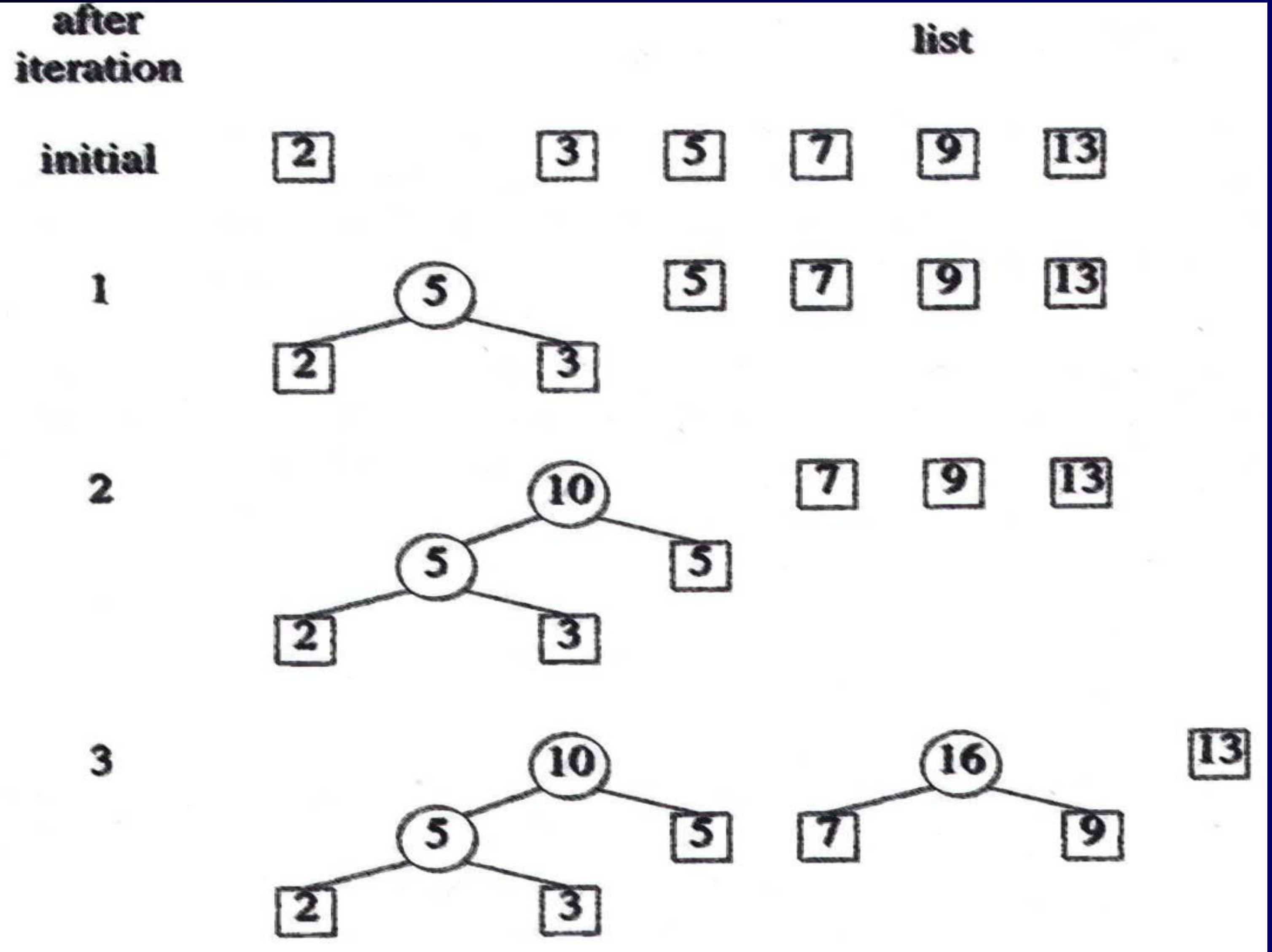
Optimal Merge Patterns

Algorithm

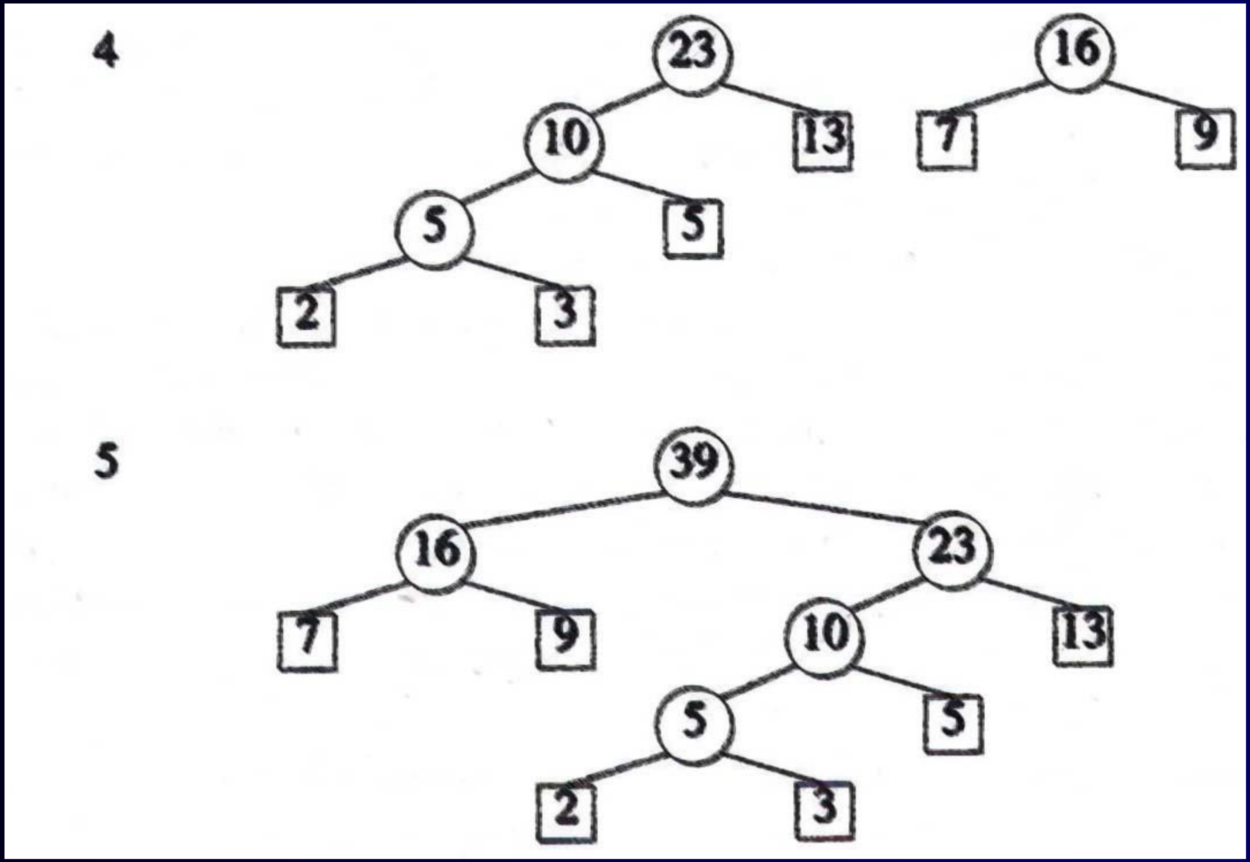
```
struct treenode {
        struct treenode *lchild, *rchild;
        int weight;
};
typedef struct treenode Type;

Type *Tree(int n)
//      list is a global list of n single node
//      binary trees as described above.
{
        for (int i=1; i<n; i++) {
            Type *pt = new Type;
            // Get a new tree node.
            pt -> lchild = Least(list); // Merge two trees with
            pt -> rchild = Least(list); // smallest lengths.
            pt -> weight = (pt->lchild)->weight
                    + (pt->rchild)->weight;
            Insert(list, *pt);
        }
       return (Least(list)); // Tree left in l is the merge tree.
}
```

Optimal Merge Patterns

- Example



Optimal Merge Patterns



Time

- If list is kept in nondecreasing order: $O(n^2)$
- If list is represented as a minheap: $O(n \log n)$

Optimal Merge Patterns

- Exercise;
- Let n=3 and (l1,l2,l3)=(5,10,3), There are n!=6 possible orderings. Find optimal ordering .

Optimal Storage on Tapes

- There are n programs that are to be stored on a computer tape of length L. Associated with each program i is a length L_i.
- Assume the tape is initially positioned at the front. If the programs are stored in the order I = $i_1, i_2, \ldots, i_n$, the time t_j needed to retrieve program i_j

$$t_j = \sum_{k=1}^{j} L_{i_k}$$

Optimal Storage on Tapes

- If all programs are retrieved equally often, then the mean retrieval time (MRT) = $$\frac{1}{n}\sum_{j=1}^{n} t_j$$

- This problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing

$$d(I) = \sum_{j=1}^{n}\sum_{k=1}^{j} L_{i_k}$$

Optimal Storage on Tapes

Example-1. n=3 (l1,l2,l3)=(5,10,3) 3!=6 total combinations

L1 l2 l3 $= \frac{l1+(l1+l2)+(l1+l2+l3)}{n} = \frac{5+15+18}{3} = 38/3=12.6$

L1 l3 l2 $= \frac{l1+(l1+l3)+(l1+l2+l3)}{n} = \frac{5+8+18}{3} = 31/3=10.3$

L2 l1 l3 $= \frac{l2+(l2+l1)+(l2+l1+l3)}{n} = \frac{10+15+18}{3} = 43/3=14.3$

L2 l3 l1 $= \frac{10+13+18}{3} = 41/3=13.6$

L3 l1 l2 $= \frac{3+8+18}{3} = 29/3=9.6$ min

L3 l2 l1 $= \frac{3+13+18}{3} = 34/3=11.3$ min

permutation at (3,1,2)

- $n = 4$, $(p_1, p_2, p_3, p_4) = (100,10,15,27)$
  $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

	Feasible solution	Processing sequence	value
1	(1,2)	2,1	110
2	(1,3)	1,3 or 3, 1	115
3	(1,4)	4, 1	127
4	(2,3)	2, 3	25
5	(3,4)	4,3	42
6	(1)	1	100
7	(2)	2	10
8	(3)	3	15
9	(4)	4	27

Example

- Let $n = 3$, $(L_1,L_2,L_3) = (5,10,3)$. 6 possible orderings. The optimal is 3,1,2

Ordering I	d(I)
1,2,3	5+5+10+5+10+3 = 38
1,3,2	5+5+3+5+3+10 = 31
2,1,3	10+10+5+10+5+3 = 43
2,3,1	10+10+3+10+3+5 = 41
3,1,2	3+3+5+3+5+10 = 29
3,2,1,	3+3+10+3+10+5 = 34

Optimal Storage on Tapes

- Exercise-1
- N=4 (l1,l2,l3,l4)=(2,4,6,8) . Find optimal storage on tapes.
- Answer permutation is at (1,2,3,4)

TVSP(Tree Vertex Splitting Problem)

Let T=(V,E,W) be a directed tree.

A weighted tree can be used to model a distribution network in which electrical signals are transmitted.

Nodes in the tree correspond to receiving stations & edges correspond to transmission lines.

In the process of transmission some loss is occurred. Each edge in the tree is labeled with the loss that occurs in traversing that edge.

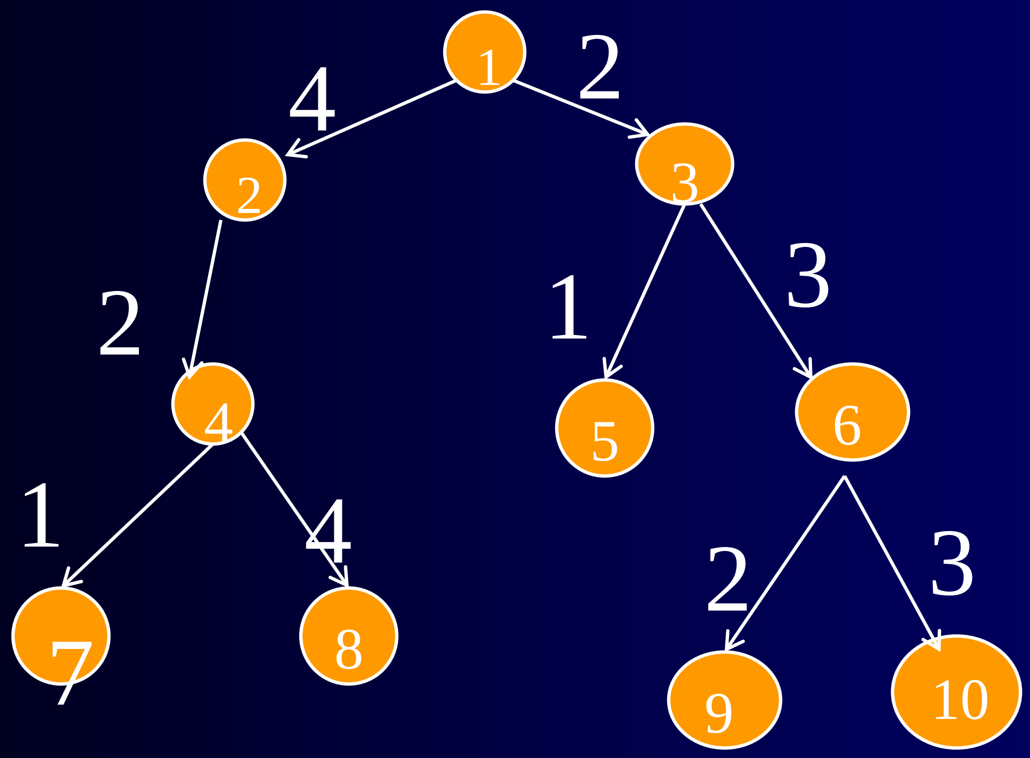
The network model may not able tolerate losses beyond a certain level. In places where the loss exceeds the tolerance value boosters have to be placed.

Given a networks and tolerance value , the TVSP problem is to determine an optimal placement of boosters. The boosters can only placed at the nodes of the tree. $d(u) = \mathrm{Max}\ \{ d(v) + w(\mathrm{Parent}(u), u)\}$

d(u) – delay of node v-set of all edges & v belongs to child(u)

δ tolerance value

TVSP(Tree Vertex Splitting Problem)



TVSP(Tree Vertex Splitting Problem)

- If d(u)>= δ than place the booster.

d(7)= max{0+w(4,7)}=1

d(8)=max{0+w(4,8)}=4

d(9)= max{0+ w(6,9)}=2

d(10)= max{0+w(6,10)}=3   d(5)=max{0+e(3.3)}=1

d(4)= max{1+w(2,4), 4+w(2,4)}=max{1+2,4+3}=6> δ ->booster

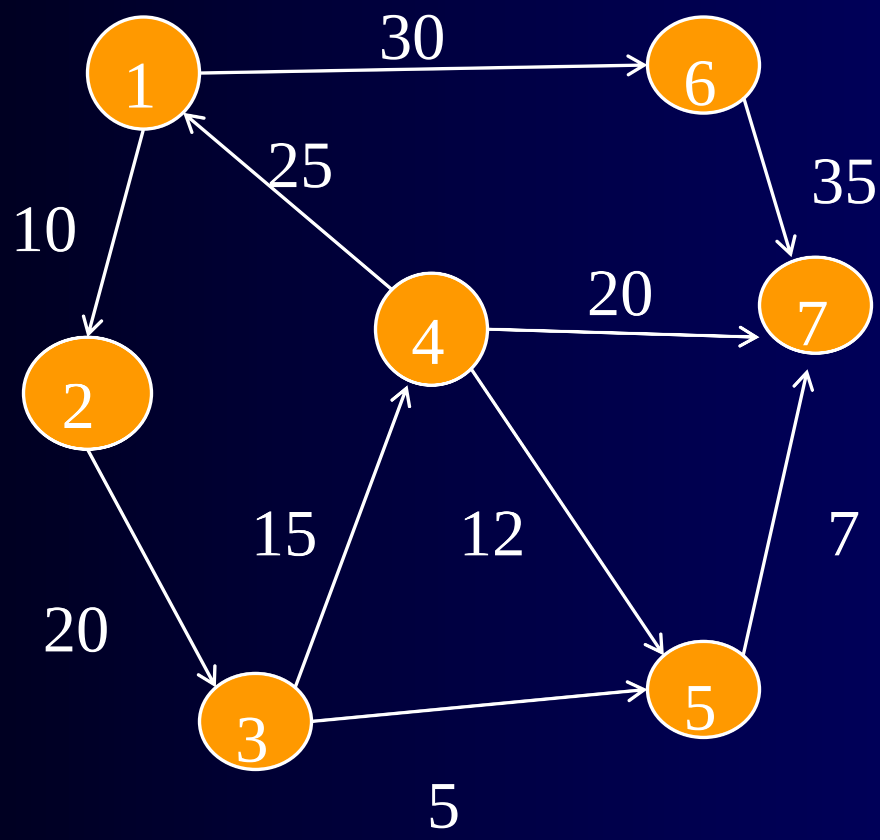
d(6)=max{2+w(3,6),3+w(3,6)}=max{2+3,3+3}=6> δ->booster

d(2)=max{6+w(1,2)}=max{6+4)=10> δ->booster

d(3)=max{1+w(1,3), 6+w(1,3)}=max{3,8}=8> δ ->booster

Note: No need to find tolerance value for node 1 because from source only power is transmitting

Single-source Shortest Paths



Single-source Shortest Paths

- Let G=(V,E) be a directed graph and a main function is C(e)(c=cost,e=edge) for the edges of graph ‘G’ and a source vertex it will represented with V_0 the vertices represents cities and weights represents distance between 2 cities.
- The objective of the problem find shortest path from source to destination.
- The length of path is defined to be sum of weights of edges on the path.
- S[i]=T if vertex i present in set ‘s’
- S[i]=F if vertex i is not present in set ‘s’
- Formula
- Min {distance[w],distance[u]+cost[u,w]}

u-recently visited node w-unvisited node

Single-source Shortest Paths

- Step-1 s[1]
- s[1]=T dist[2]=10
- s[2]=F dist[3]=α
- s[3]=F dist[4]= α
- s[4]=F dist[5]= α
- s[5]=F dist[6]= 30
- s[6]=F dist[7]= α
- S[7]=F
- Step-2 s[1,2] the visited nodes
- W={3,4,5,6,7} unvisited nodes
- U={2} recently visited node

Single-source Shortest Paths

- s[1]=T w=3
- s[2]=T dist[3]=α
- s[3]=F min {dist[w], dist[u]+cost(u,w)}
- s[4]=F min {dist[3], dist[2]+cost(2,3)}
- s[5]=F min{α, 10+20}= 30
- s[6]=F w=4 dist[4]= α
- S[7]=F min{dist(4),dist(2)+cost(2,4)}
- min{α,10+ α}= α

W=5 dist[5]= α min{dist(5),dist(2)+cost(2,5)}

min{α,10+ α}= α

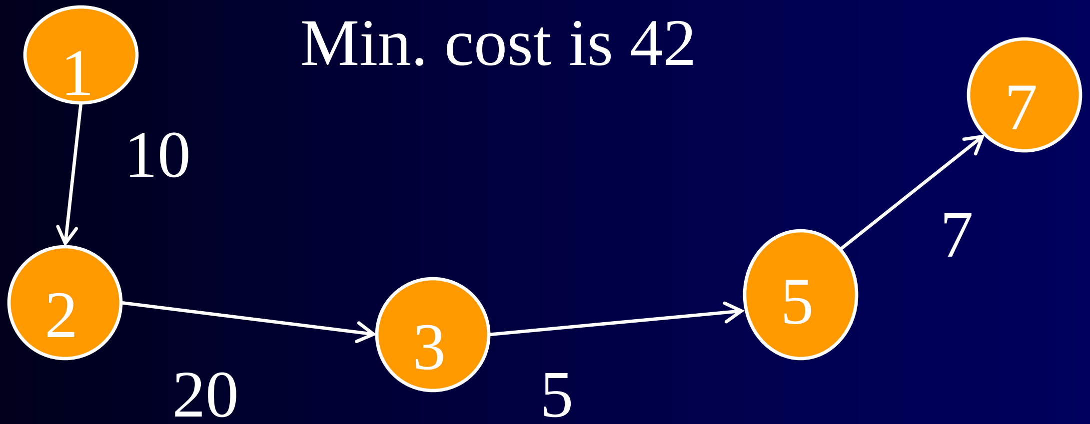
Single-source Shortest Paths

- W=6 dist[6]=30
- Min{dist(6), dist(2)+cost(2,6)}=min{30,10+ α}=30
- W=7, dist(7)= α min{dist(7),dist(2)+cost(2,7)}
- min{α,10+ α}= α let min. cost is 30 at both 3 and 6 but
- Recently visited node 2 have only direct way to 3, so consider 3 is min cost node from 2.
- Step-3 | w=4,5,6,7
- s[1]=T | s={1,2,3} w=4 ,dist[4]= α
- s[2]=T | min{dist[4],dist[3]+cost(3.4)}=min{α,30+15}=45
- s[3]=T | w=5, dist[5]= α min{dist(5), dist(3)+cost(3,5)}
- s[4]=F | min{α,30+5}=35 similarily we obtain
- s[5]=F | w=6, dist(6)=30 w=7 ,dist[7]= α so min cost is 30 at w=6 but
- s[6]=F | no path from 3 so we consider 5 node so visited nodes 1,2,3, 5
- S[7]=F |

Single-source Shortest Paths

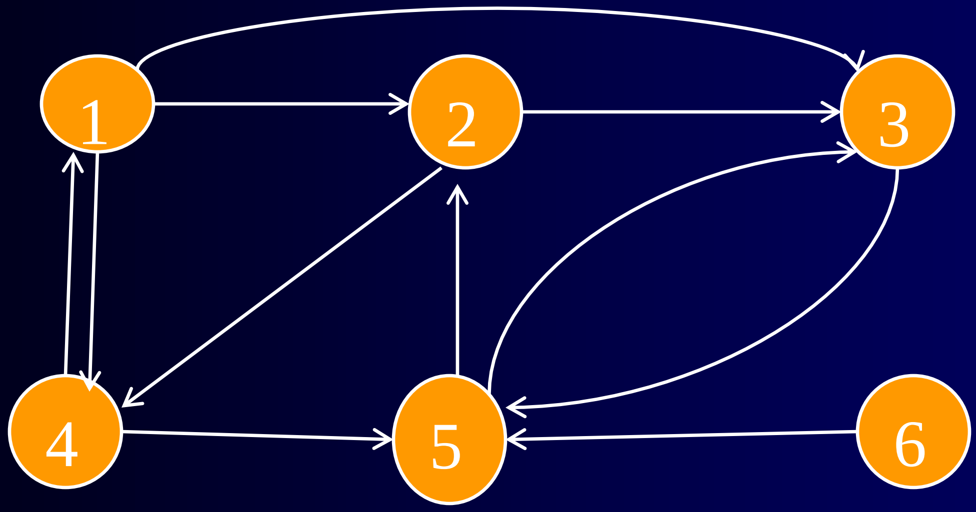
- Step-4 | w=4,6,7 s={1,2,3,5}
- s[1]=T | w=4, dist[4]=45 min {dist[4], dist[5]+cost(5,4)}
- s[2]=T | min{45,35+ α}=45
- s[3]=T | w=6,dist[6]=30 min{dist[6],dist[5]+cost(5,6)}
- s[4]=F | min{30, 35+ α}=30
- s[5]=T | w=7,dist[7]= α min{dist[7],dist[5]+cost(5,7)}
- s[6]=F | min{α, 35+7}=42
- S[7]=F | here min cost is 30 at 6 node but there is no path from 5 yo 6, so we consider 7 , 1,2,3,5,7 nodes visited.
- Therefore the graph traveled from source to destination
- Single source shortest path is drawn in next slide.

Single-source Shortest Paths

- 

Min. cost is 42

Exercise



V0=1

Vd=5

Single-source Shortest Paths

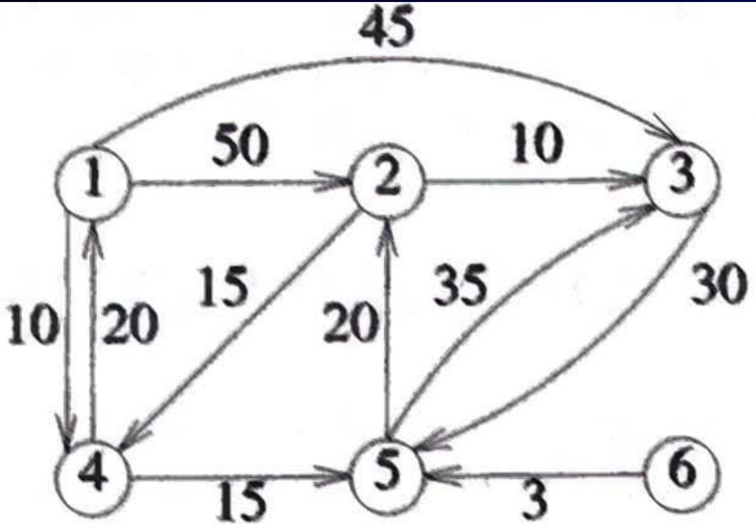
- Design of greedy algorithm
  - Building the shortest paths one by one, in nondecreasing order of path lengths
    - e.g., in next slide figure
      - 1→4: 10
      - 1→4→5: 25
      - …
    - We need to determine 1) the next vertex to which a shortest path must be generated and 2) a shortest path to this vertex.
  - Notations
    - S = set of vertices (including v_0) to which the shortest paths have already been generated
    - $dist(w)$ = length of shortest path starting from v_0, going through only those vertices that are in S, and ending at w

Single-source Shortest Paths

- Design of greedy algorithm (Continued)
  - Three observations
    - If the next shortest path is to vertex *u*, then the path begins at v_0, ends at *u*, and goes through only those vertices that are in *S*.
    - The destination of the next path generated must be that of vertex *u* which has the minimum distance, *dist(u)*, among all vertices not in *S*.
    - Having selected a vertex *u* as in observation 2 and generated the shortest v_0 to *u* path, vertex *u* becomes a member of *S*.

Single-source Shortest Paths

- Example



(a) Graph

	Path	*Length*
1)	1, 4	10
2)	1, 4, 5	25
3)	1, 4, 5, 2	45
4)	1, 3	45

(b) Shortest paths from 1

DIJKSTRA'S SHORTEST PATH ALGORITHM

Procedure SHORT-PATHS (v, cost, Dist, n)

// Dist (j) is the length of the shortest path from v to j in the //graph G with n vertices; Dist (v) = 0 //

Boolean S(1:n); real cost (1:n,1:n), Dist (1:n); integer u, v, n, num, i, w

// S(i) = 0 if i is not in S and s(i) =1 if it is in S//

// cost (i, j) = +α if edge (i, j) is not there//

// cost (i,j) = 0 if i = j; cost (i, j) = weight of < i, j > //

for i$\leftarrow$1 to do // initialize S to empty //

S(i) $\leftarrow$0; Dist (i)$\leftarrow$ cost(v, i)

repeat

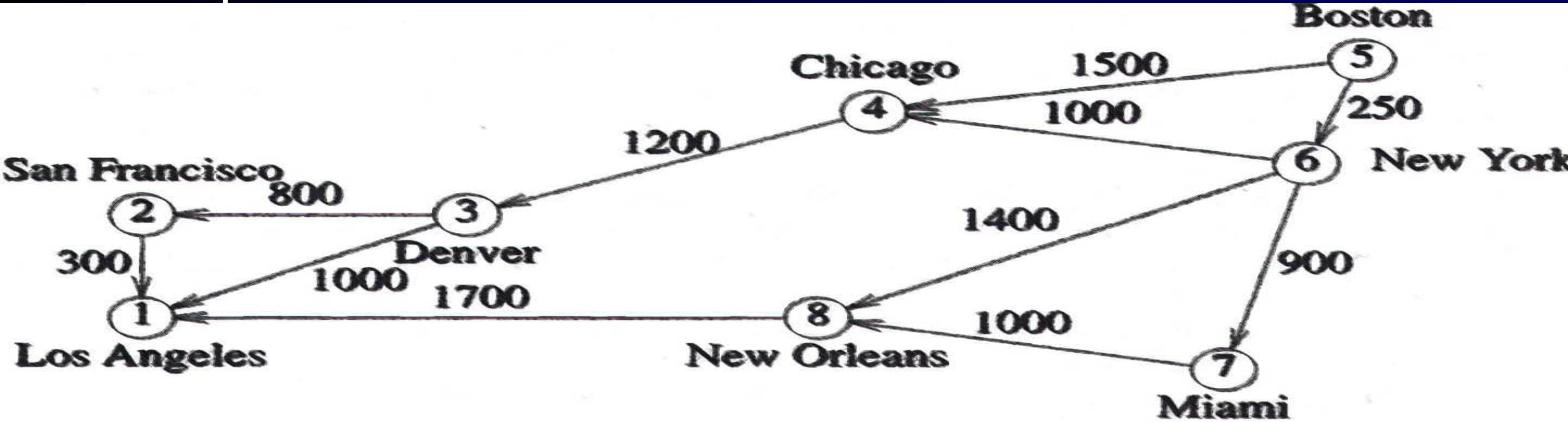
DIJKSTRA'S SHORTEST PATH ALGORITHM (Contd..)

// initially for no vertex shortest path is available//
S (v)←1; dist(v)←0// Put v in set S //
for num←2 to n-1 do // determine n-1 paths from// //vertex v //
choose u such that Dist (u)=min{dist(w)} and S(w)=0
S(u)←1 // Put vertex u in S //
Dist(w)←min (dist(w),Dist(u) + cost (u,w))
Repeat
repeat
end SHORT - PATHS

Overall run time of algorithm is O((n+|E|) log n)

Single-source Shortest Paths

- Example



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

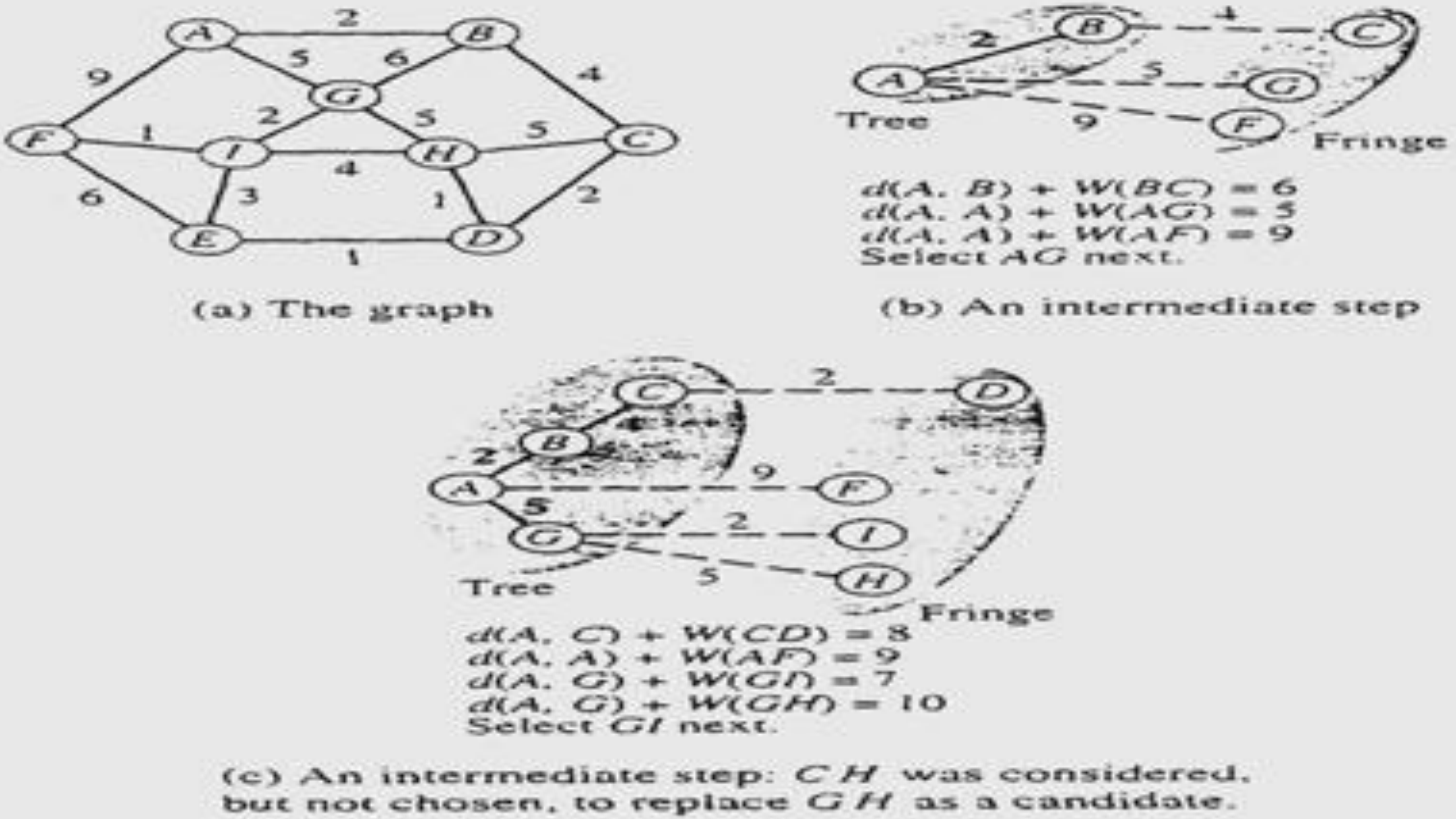
Single-source Shortest Paths

- Example

Iteration	S	Vertex selected	Distance							
			LA [1]	SF [2]	DEN [3]	CHI [4]	BOST [5]	NY [6]	MIA [7]	NO [8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

Single-source Shortest Paths

The Algorithm in action,



(a) The graph

(b) An intermediate step

(c) An intermediate step: CH was considered, but not chosen, to replace GH as a candidate.

Single-source Shortest Paths

Exercise

Write Trace execution of Dijkstra's algorithm on the graph below.