Introduction to Algorithm Design

Lecture Notes 10

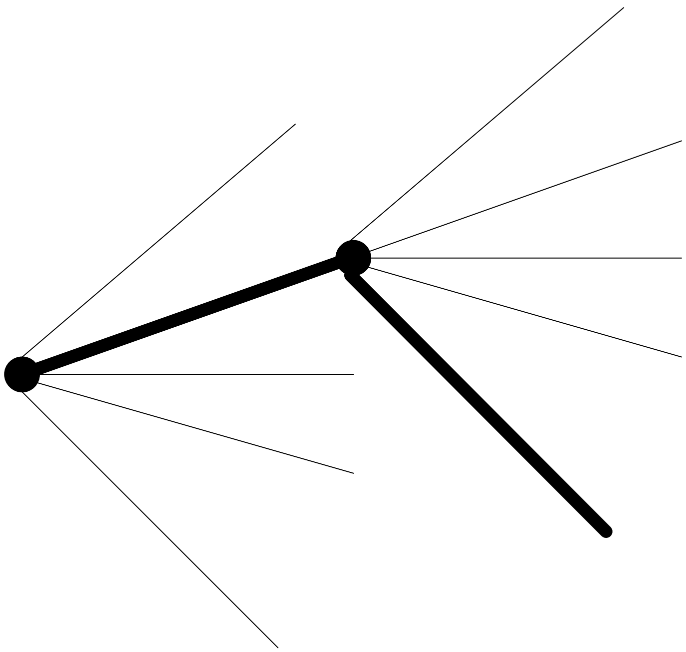
ROAD MAP

- **Greedy Technique**
  - **Knapsack Problem**
  - Minimum Spanning Tree Problem
    - Prim’s Algorithm
    - Kruskal’s Algorithm
  - Single Source Shortest Paths
    - Dijkstra’s Algorithm
  - Job Sequencing With Deadlines
  - Huffman Trees
  - Activity Selection Problem

Greedy Technique

- Used for solving *optimization problems*
  - such as engineering problems
- Construct a solution through a sequence of *decision steps*
  - Each expanding a partially constructed solution
  - Until a complete solution is reached
- Similar to dynamic programming
  - but, not all possible solutions are explored

Greedy Technique



On each decision step the choice should be

- **Feasible**
  - has to satisfy the problem's constraints
- **Locally optimal**
  - has to be the best local choice
- **Irrevocable**
  - once made, it can not be changed

Greedy Technique

```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

Greedy Technique

- In each step, greedy technique suggests a *greedy* selection of the best alternative avaliable
  - Feasible decision
  - Locally optimal decision
  - Hope to yield a globally optimal solution
- Greedy technique ***does not*** give the optimal solution for all problems
- There are problems for which a sequence of locally optimal choices does not yield an optimal solution
  - EX: TSP, Graph coloring
  - Produces approximate solution

Fractional Knapsack Problem

- ***Given :***

  w_i : weight of object *i*

  m : capacity of knapsack

  p_i : profit of all of *i* is taken

- ***Find:***

  x_i : fraction of *i* taken

- ***Feasibility:***

$$\sum_{i=1}^{n} x_i w_i \le m$$

- ***Optimality:***

$$\text{maximize} \quad \sum_{i=1}^{n} x_i p_i$$

Greedy Technique

```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

Knapsack Problem

```
Algorithm Knapsack (m,n)
  for i = 1 to n
      x(i) = 0
  for i = 1 to n
      select the object (j) with largest unit value
      if (w[j] < m)
            x[j] = 1.0
            m = m - w[j]
      else
            x[j] = m/w[j]
            break
```

- <u>Example :</u>
  M = 20 n = 3
  p = (25, 24, 15) w = (18, 15, 10)

- **Proof of Optimality**

  G is greedy solution

  $G = x_1, x_2, \ldots, x_n \qquad 0 \le x_i \le 1$

  let $x_k \ne 1 \qquad$ k: least index

  $x_i = 1 \qquad 1 \le i < k$

  $x_i = 0 \qquad k < i \le n$

  O is optimal solution

  $O = y_1, y_2, \ldots, y_n$

  let $x_j \ne y_j \qquad$ j is the least index

  $y_j < x_j$

G

		x_j				

O

		y_j			y_s	

$y_j \leftarrow x_j$

$y_j \leftarrow y_j + x_j - y_j$

$y_s \leftarrow y'_s$

O'

		X_j			y'_s	

$$\sum_{O} profit \leq \sum_{O'} profit$$

$O \rightarrow O'$

$\vdots$

$O \rightarrow O' \rightarrow O'' \rightarrow \ldots \rightarrow G$

Minimum Spanning Tree (MST)

- Problem Instance:
  - *A weighted, connected, undirected graph G (V, E)*
- Definition:
  - *A spanning tree* of a connected graph is its connected acyclic subgraph
  - A *minimum spanning tree* of a weighted connected graph is its spanning tree of the smallest weight
    - *weight* of a tree is defined as the sum of the weights on all its edges
- Feasible Solution:
  - *A spanning tree G' of G*

$$G' = (V, E') \qquad E' \subseteq E$$

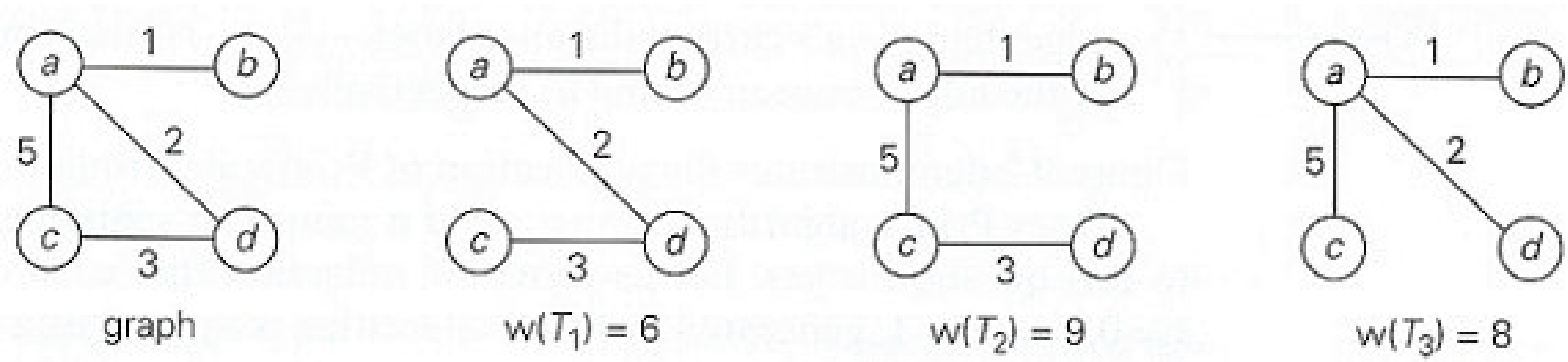
Minimum Spanning Tree

- Objective function :
  - Sum of all edge costs in G'

$$C(G') = \sum_{e \in G'} C(e)$$

- Optimum Solution :
  - Minimum cost spanning tree

Minimum Spanning Tree



T_1 is the minimum spanning tree

Greedy Technique

```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

Prim's Algorithm

- Prim's algorithm constructs a MST through a sequence of expanding subtrees
- Greedy choice :
  - Choose minimum cost edge add it to the subgraph

Prim's Algorithm

ALGORITHM *Prim*(G)

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

Prim's Algorithm

Approach :

1. Each vertex *j* keeps near[j] Є T (current tree)

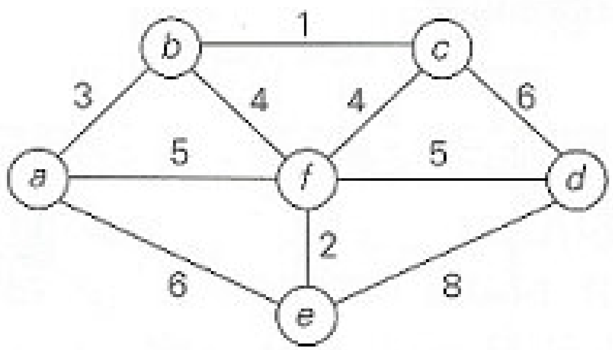
    where **`cost(j,near[j])`** is minimum

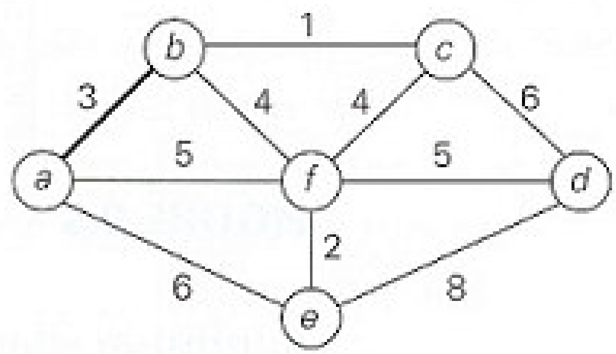
2. **`near[j] = 0 if j Є T`**

    = ∞ if there is no egde between *j* and *T*

3. Use a heap to select minimum of all edges

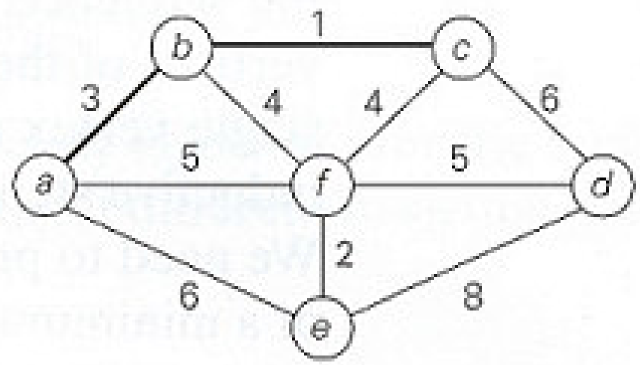
Prim's Algorithm Example



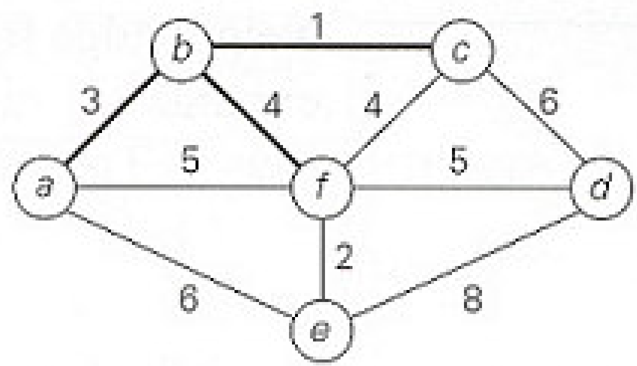
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	

Prim's Algorithm Example

b(a, 3) **c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)

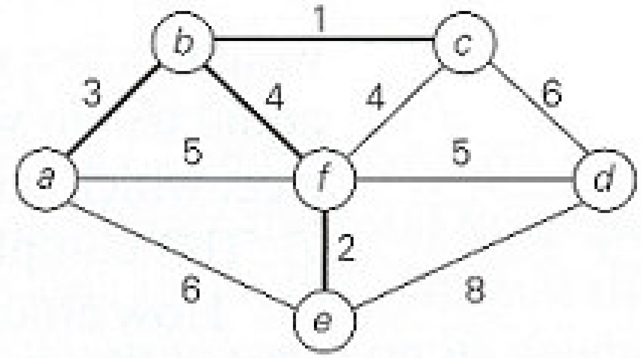


c(b, 1) d(c, 6) e(a, 6) **f(b, 4)**

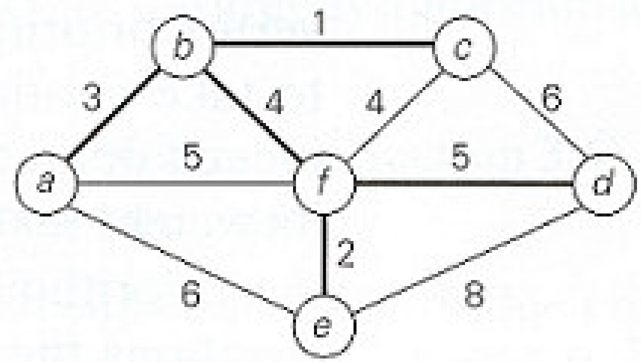


Prim's Algorithm Example

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



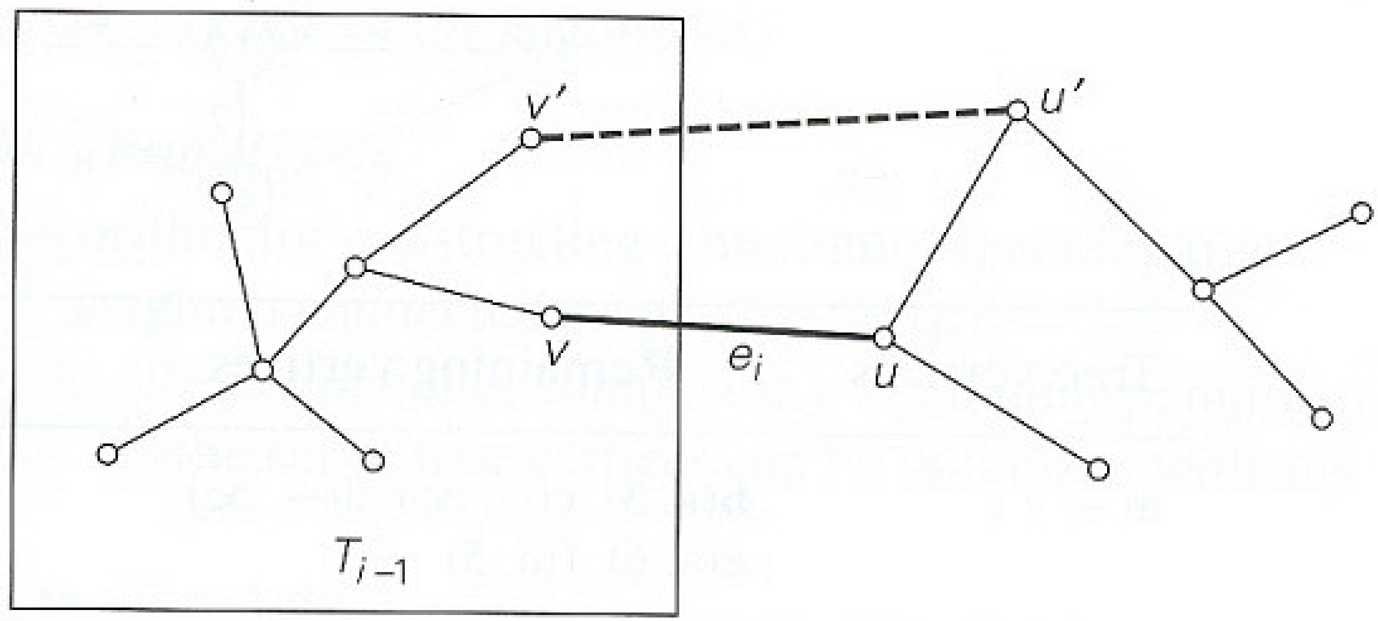
e(f, 2) **d(f, 5)**



d(f, 5)

Correctness proof of Prim's Algorithm

- Prim's algorithm always yield a MST
- We can prove it by induction
  - T_0 is a part of any MST
    - consists of a single vertex
  - Assume that T_{i-1} is a part of MST
  - We need to prove that T_i, generated by T_{i-1} by Prim's algorithm is a part of a MST
  - We prove it by contradiction
    - Assume that no MST of the graph can contain T_i



Let e_i=(u,v) be minimum weight edge from a vertex in T_{i-1} to a vertex not in T_{i-1} used by Prim's algorithm to expant T_{i-1} to T_i

By our assumption, if we add e_i to T (MST), a cycle must be formed.

In addition to e_i, cycle must contain another edge (v', u').
If we delete the edge e_k, (v', u'), we obtain another spanning tree

$w_{ei} \leq w_{ek}$ So weight of new spanning tree is less or equal to T. Since T is a MST, weight of new spanning tree can not be less. So, they are equal.

New spanning tree is a minimum spanning tree which contradicts the assumption that no minimum spanning tree contains T_i

Prim's Algorithm

```
Initialize S with the start vertex, s, and V–S with the remaining vertices
    for all v in V – S
        if there is an edge (s, v)
            Set cost[v] to w(s, v)
            Set next[v] to s
        else
            Set cost[v] to ∞
            Set next[v] to NULL
    while V – S is not empty
        for all u in V – S, find the smallest cost[u]
        Remove u from V – S and add it to S
        Insert the edge (u, next[u]) into the spanning tree.
        for all v adjacent to u in V – S
            if w(u, v) < cost[v]
                Set cost[v] to w(u, v)
                Set next[v] to u.
```

Prim's Algorithm

Analysis :

- How efficient is Prim's algorithm ?
  - It depends on the data structure chosen
  - running time is $\Theta(|V|^2)$ If
    - graph is represented by its weight matrix
    - unordered array is used
  - running time of is $O(|E| \log |V|)$ If
    - graph is represented by adjacency list
    - priority queue such as a min-heap is used

Kruskal's Algorithm

- Another algorithm to construct MST
- Expands a subgraph
  - initially contains all the vertices but no edges
- Generates a sequence of subgraphs
  - always acyclic
  - not necessarily connected
- Resulting graph is connected and acyclic (i.e., tree)

Greedy choice :

- Choose minimum cost edge
  - Connecting two disconnected subgraphs
- It always yields an optimal solution

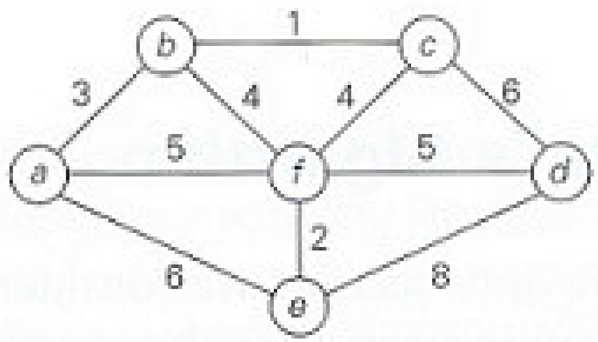
Greedy Technique

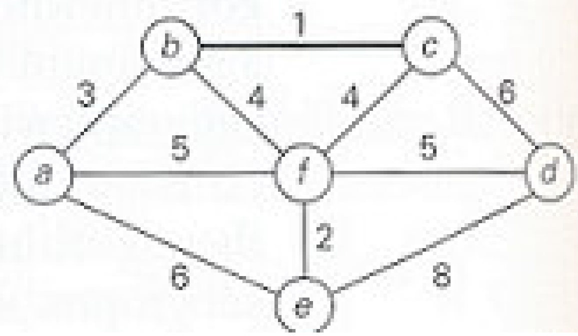
```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
      x = select (a)
      if feasible ( solution, x )
           solution = solution U {x}
  return solution
}
```

Kruskal's Algorithm

```
ALGORITHM  Kruskal(G)
    //Kruskal's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph G = ⟨V, E⟩
    //Output: E_T, the set of edges composing a minimum spanning tree of G
    sort E in nondecreasing order of the edge weights w(e_i1) ≤ ... ≤ w(e_i|E|)
    E_T ← ∅;   ecounter ← 0    //initialize the set of tree edges and its size
    k ← 0                      //initialize the number of processed edges
    while ecounter < |V| − 1 do
        k ← k + 1
        if E_T ∪ {e_ik} is acyclic
            E_T ← E_T ∪ {e_ik};   ecounter ← ecounter + 1
    return E_T
```

Kruskal’s Algorithm Example

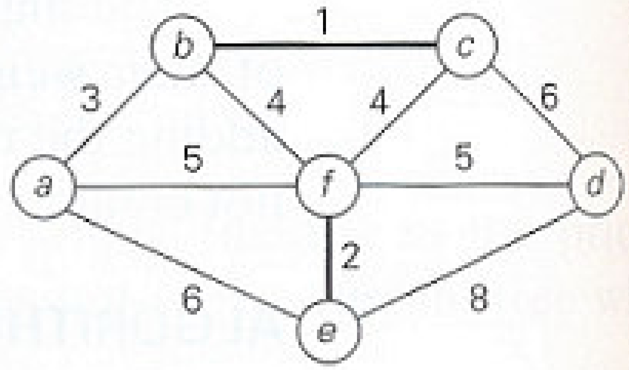


Tree edges	Sorted list of edges	Illustration
	bc cf ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	

Kruskal’s Algorithm Example

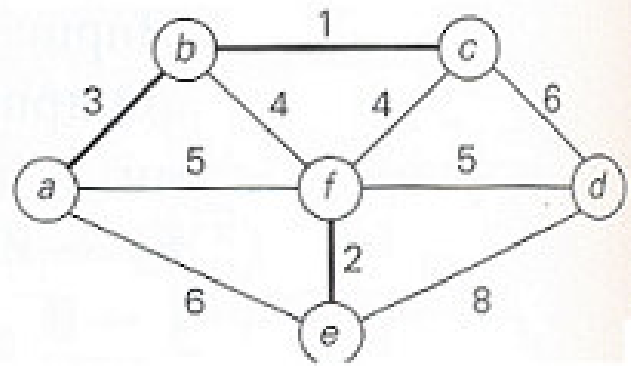
bc
1

bc	**ef**	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



ef
2

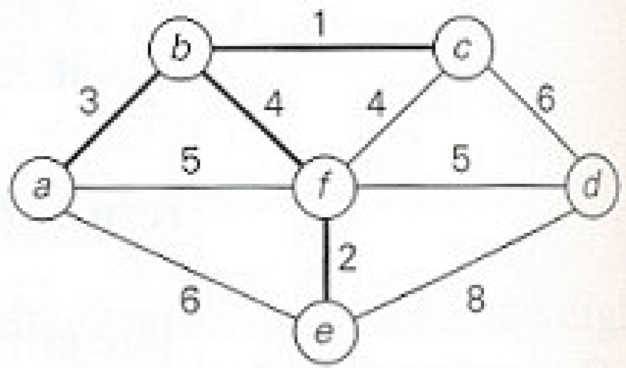
bc	ef	**ab**	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



Kruskal’s Algorithm Example

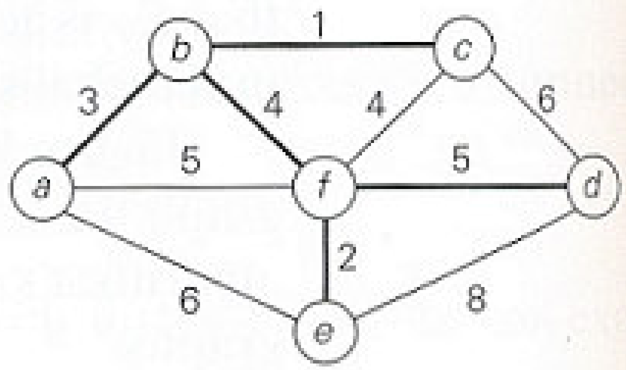
ab
3

bc	ef	ab	**bf**	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



bf
4

bc	ef	ab	bf	cf	af	**df**	ae	cd	de
1	2	3	4	4	5	5	6	6	8



df
5

Proof of Optimality

Algorithm → T not optimal

$e_1, e_2, \ldots, e_{n-1}$ $c(e_1) < c(e_2) < \ldots < c(e_{n-1})$

O is optimal

$e \in T$ $e \notin O$ O U {e} forms a cycle

The cycle contains an edge $e^* \notin T$

O'=O U {e} – {e*} forms another spanning tree

Cost(O') = Cost(O) + c(e) - c(e*)

Greedy choice → c(e) ≤ c(e*)

Cost(O') ≤ Cost(O)

$O \rightarrow O' \rightarrow \ldots \rightarrow O^{(k)} = T$

Disjoint Subsets and Union-Find Algorithms

- Some applications (such as Kruskal's algorithm) requires a dynamic partition of some *n*-element set S into a collection of disjoint subsets S_1, S_2, …, S_k
- After being initialized as a collection *n one*-element subsets, each containing a different element of S, the collection is subjected to a sequence of intermixed union and find operations

Disjoint Subsets and Union-Find Algorithms

- We are dealing with an abstract data type of a collection of disjoint subsets of a finite set with operations :
  - *makeset(x)* :
    - creates one-element set {x}
    - it is assumed that this operation can be applied to each of the element of set S only once
  - find(x) :
    - returns a subset containing x
  - *union (x,y) :*
    - constructs the union of disjoint subsets S_x, S_y containing x and y, respectively
    - Adds it to the collection to replace S_x and S_y
      - which are deleted from it

Disjoint Subsets and Union-Find Algorithms

- Example :
  - S = { 1, 2, 3, 4, 5, 6 }
    - make(i) creates the set(i)
    - applying this operation six times initializes the structure to the collection of six singleton sets :

      {1}, {2}, {3}, {4}, {5}, {6}

    - performing union (1,4) and union (5,2) yields

      {1, 4}, {5, 2}, {3}, {6}

    - if followed by union (4,5) and then by union (3,6)

      {1, 4, 5, 2}, {3, 6}

Disjoint Subsets and Union-Find Algorithms

- There are two principal alternatives for implementing this data structure

1. quick find

- optimizes the time efficiency of the find operation

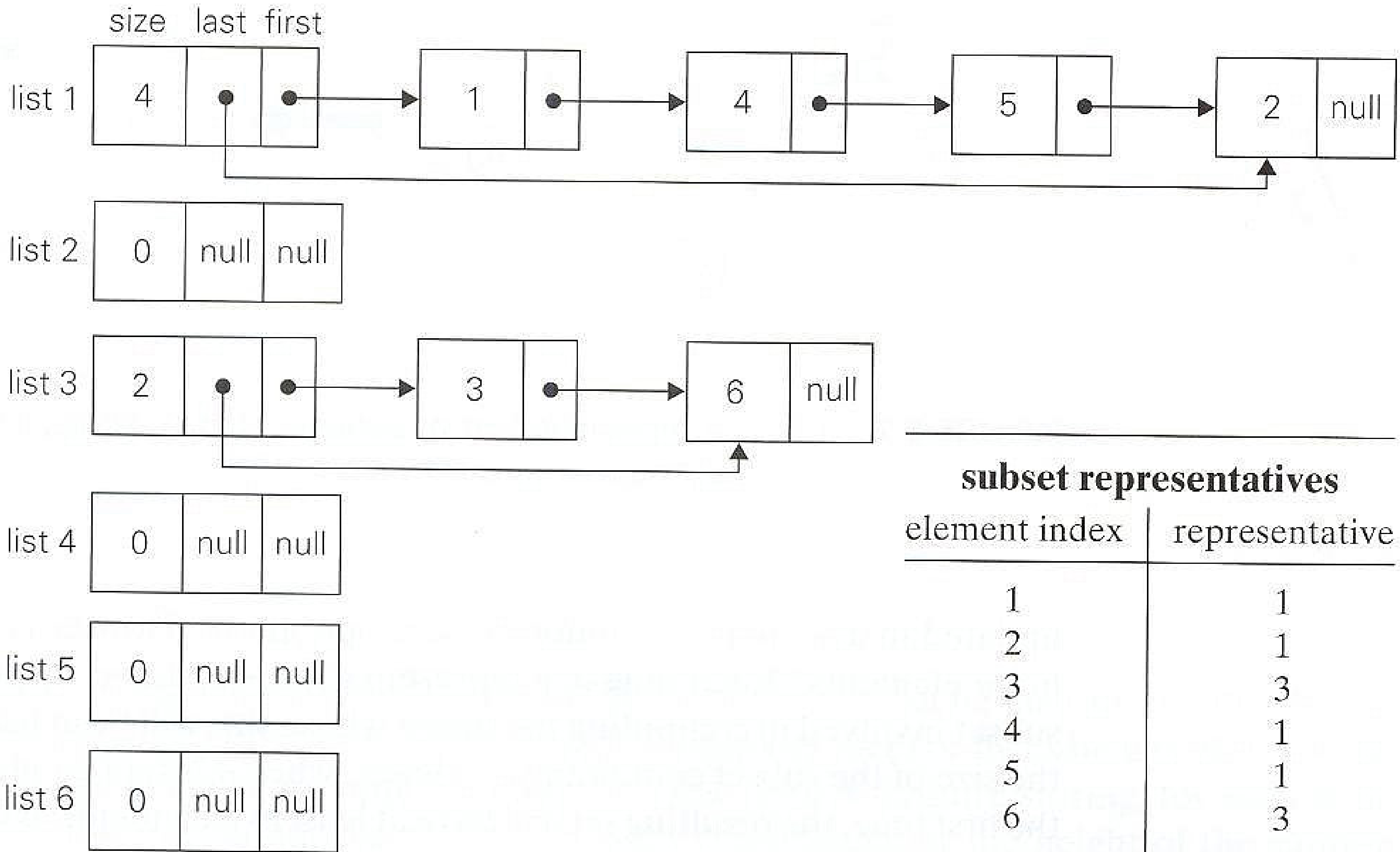
2. quick union

- optimizes the union operation

Disjoint Subsets and Union-Find Algorithms

1. quick find

- optimizes the time efficiency of the find operation
- uses an array indexed by the elements of the underlying set S
- each subset is implemented as a linked list whose header contains the pointers to the first and last elements of the list



subset representatives

element index	representative
1	1
2	1
3	3
4	1
5	1
6	3

Linked list representation of subsets {1, 4, 5 ,2} and {3, 6} obtained by quick-find after performing union (1,4), union (5,2), union (4,5) and union (3,6)

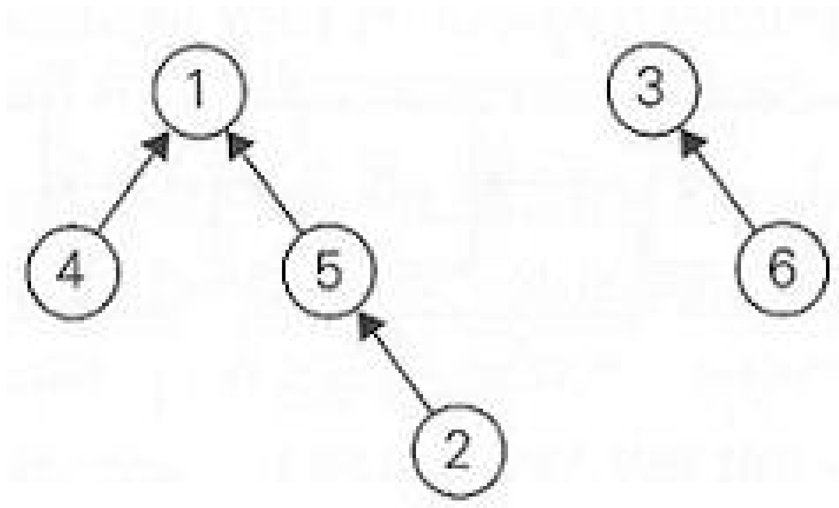
Disjoint Subsets and Union-Find Algorithms

- Time efficency of makeset(x) is $\Theta(1)$, hence initialization of *n* singleton subsets is $\Theta(n)$
- Time efficency of find(x) is $\Theta(1)$
- Executing union(x,y) takes longer,
  - $\Theta(n^2)$ for a sequence of n union operations
  - A simple way to improve the overall efficiency is to append the shorter of the two lists to the longer one
  - This modification is called union-by-size
  - But it does not improve the worst case efficiency

Disjoint Subsets and Union-Find Algorithms

2. quick union

- Represents each subsets by a rooted tree
- The nodes of the tree contain the subset's elements (one per node)
- Tree's edges are directed from children to their parents

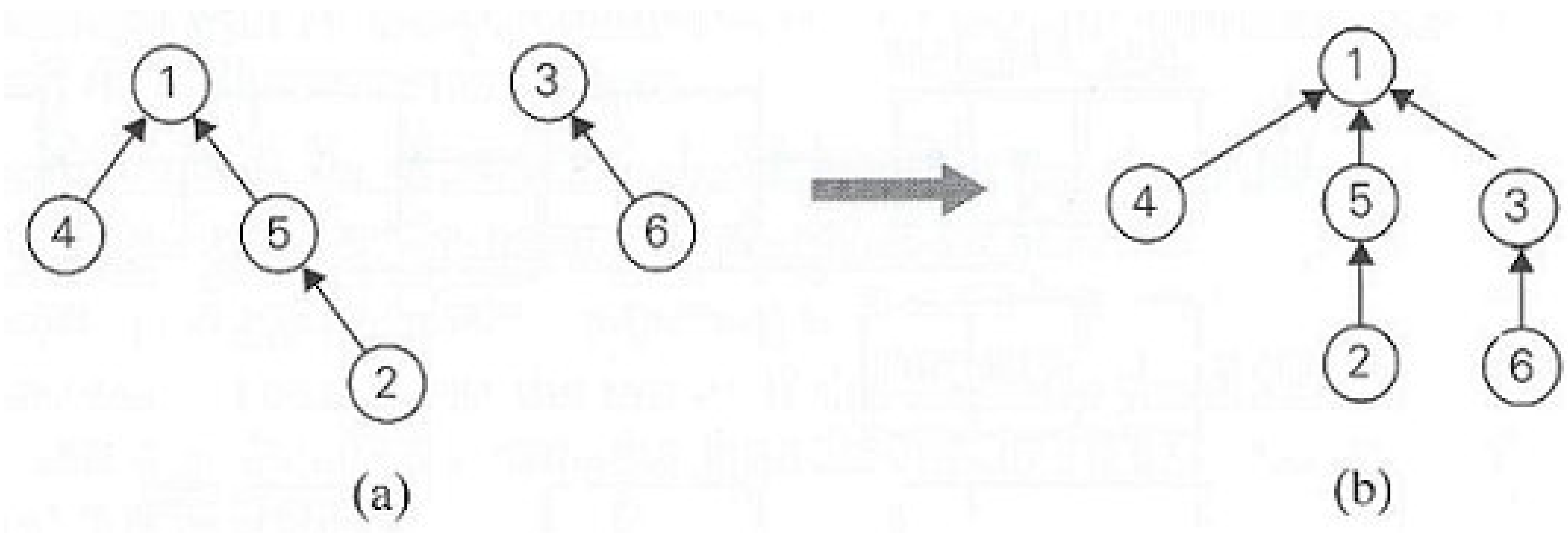


Disjoint Subsets and Union-Find Algorithms

2. quick union

- Time efficency of makeset(x) is Θ(1), hence initialization of *n* singleton subsets is Θ(n)
- Time efficency of find(x) is Θ(n)
  - A tree representing a subset can degenerate into a linked list with *n* nodes
  - A find is performed by following the pointer chain from the node containing x to the tree's root
- Executing union (x,y) takes Θ(1)
  - It is implemented by attaching the root of the y's tree to the root of the x's tree

Disjoint Subsets and Union-Find Algorithms

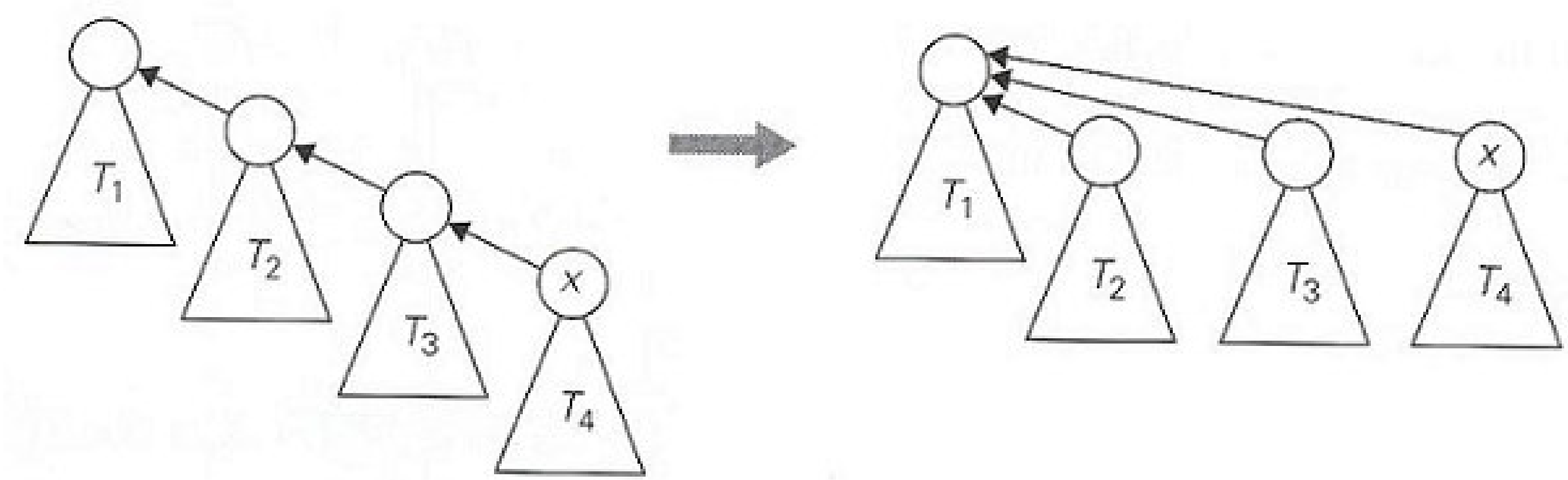


a) forest representation of subsets {1, 4, 5, 2} & {3, 6} used by quick union

b) result of union (5,6)

Disjoint Subsets and Union-Find Algorithms

- A better efficiency can be obtained by using ***path compression***
  - Every node encountered during the execution of a find operation are made to point to the tree's root



- A sequence of *n-1* unions and *m* finds takes only slightly worse than linear

Kruskal's Algorithm

ALGORITHM *Kruskal*(G)

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

Algorithm requires at most *n-1* unions and 2*m* finds
Analysis → O(|E| log|E|)

ROAD MAP

- **Greedy Technique**
  - Knapsack Problem
  - Job Sequencing With Deadlines
  - Minimum Spanning Tree Problem
    - Prim’s Algorithm
    - Kruskal’s Algorithm
  - **Single Source Shortest Paths**
    - **Dijkstra’s Algorithm**
  - Huffman Trees
  - Activity Selection Problem

Single Source Shortest Paths

- Definition:
  - For a given vertex called ***source*** in a *weighted* connected graph, find shortest paths to all other vertices in the graph

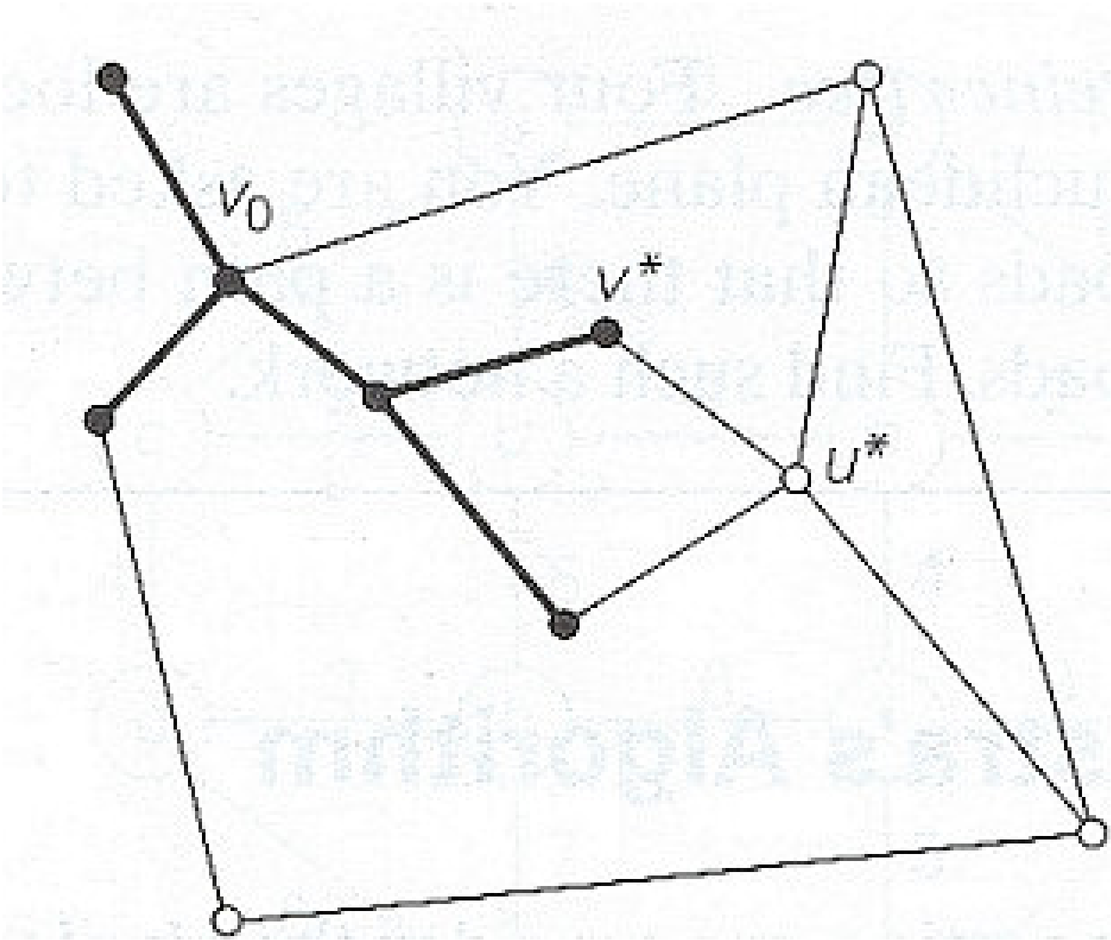
Dijkstra's Algorithm

- Idea :
  - Incrementally add nodes to an empty tree
  - Each time add a node that has the smallest path length

- Approach :

  1. S = { }
  2. Initialize *dist [v]* for all *v*
  3. Insert *v* with min *dist[v]* in *T*
  4. Update *dist[w]* for all $w \notin S$

Dijkstra’s Algorithm



Idea of Dijkstra’s algorithm

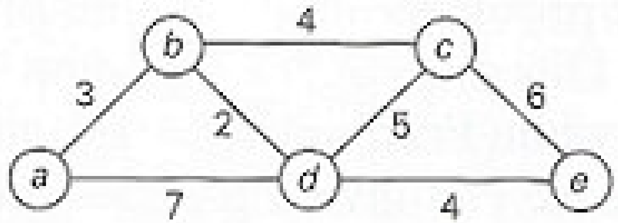
Greedy Technique

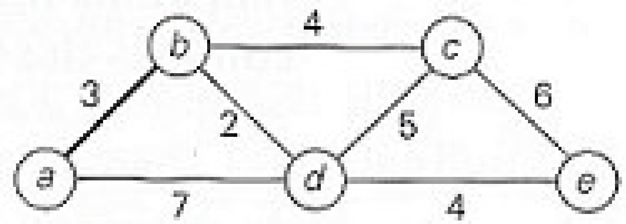
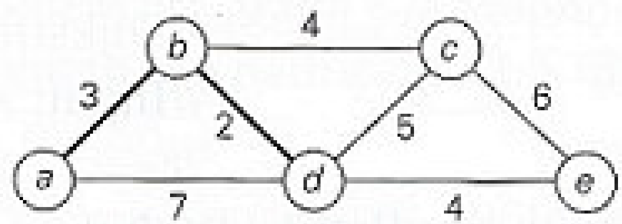
```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

ALGORITHM *Dijkstra*(G, s)

//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
// and its vertex s
//Output: The length d_v of a shortest path from s to v
// and its penultimate vertex p_v for every vertex v in V
Initialize(Q) //initialize vertex priority queue to empty
for every vertex v in V **do**
  $d_v \leftarrow \infty$; $p_v \leftarrow$ **null**
  Insert(Q, v, d_v) //initialize vertex priority in the priority queue
$d_s \leftarrow 0$; *Decrease*(Q, s, d_s) //update priority of s with d_s
$V_T \leftarrow \emptyset$
for $i \leftarrow 0$ **to** $|V| - 1$ **do**
  $u^* \leftarrow$ *DeleteMin*(Q) //delete the minimum priority element
  $V_T \leftarrow V_T \cup \{u^*\}$
  for every vertex u in $V - V_T$ that is adjacent to u^* **do**
    if $d_{u^*} + w(u^*, u) < d_u$
      $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
      Decrease(Q, u, d_u)

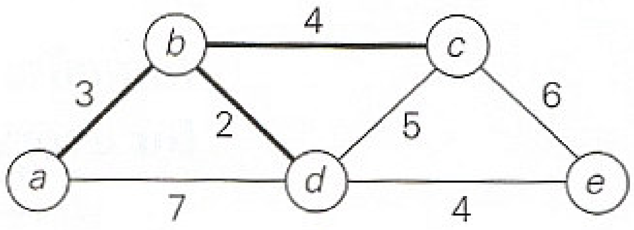
Dijkstra's Algorithm Example



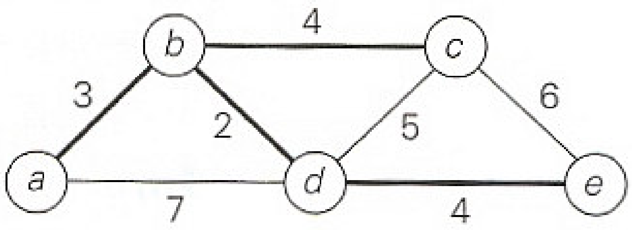
Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	

Dijkstra's Algorithm Example

d(b, 5) **c**(**b**, **7**) e(d, 5 + 4)



c(b, 7) **e**(**d**, **9**)



e(d, 9)

from a to b :	$a - b$	of length 3
from a to d :	$a - b - d$	of length 5
from a to c :	$a - b - c$	of length 7
from a to e :	$a - b - d - e$	of length 9

Dijkstra's Algorithm

- **Analysis :**
  - Time efficiency depends on the data structure used for priority queue and for representing an input graph itself
  - For graphs represented by their weight matrix and priority queue implemented as an unordered array, efficiency is in $\Theta(|V|^2)$
  - For graphs represented by their adjacency list and priority queue implemented as a min-heap efficiency is in $O(|E|\log|V|)$
  - A better upper bound for both Prim and Dijkstra's algorithm can be achieved, if *Fibonacci heap* is used

ROAD MAP

- **Greedy Technique**
  - Knapsack Problem
  - Minimum Spanning Tree Problem
    - Prim's Algorithm
    - Kruskal's Algorithm
  - Single Source Shortest Paths
    - Dijkstra's Algorithm
  - **Job Sequencing With Deadlines**
  - **Huffman Trees**
  - **Activity Selection Problem**

Job Sequencing With Deadlines

- ***Given :***

  n jobs 1, 2, … , n

  deadline $d_i > 0$ each job taken 1 unit time

  profit $p_i > 0$ 1 machine available

- ***Find***

$$J \subseteq \{1,2,\ldots,N\}$$

- ***Feasibility:***

  The jobs in *J* can be completed before their deadlines

- ***Optimality:***

$$\text{maximize} \sum_{i \in j} P_i$$

Job Sequencing With Deadlines

- Example :

  n = 4

  d_i = 2, 1, 2, 1

  p_i = 100, 10, 15, 27

$$\mathrm{J} = \{1,2\} \quad \Rightarrow \quad \sum p_i = 110$$

$$\mathrm{J} = \{1,3\} \quad \Rightarrow \quad \sum p_i = 115$$

$$\mathrm{J} = \{1,4\} \quad \Rightarrow \quad \sum p_i = 127 \quad \leftarrow \quad \text{optimal}$$

$\mathrm{J} = \{1,2,3\}$ is not feasible

Job Sequencing With Deadlines

- Greedy strategy?

Job Sequencing With Deadlines

Greedy Choice :

- Take the job gives largest profit
- Process jobs in nonincreasing order of p_i's

Greedy Technique

```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
      x = select (a)
      if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

Job Sequencing With Deadlines

How to check feasibility ?

- Need to check all permutations
  - k jobs than k! permutations
- To check feasibility
  - If the jobs are scheduled as follows

    $$i_1, i_2, \ldots, i_k$$

    check whether

    $$d_{i_j} \geq j$$
- k jobs requires at least k! time
- What about using a greedy strategy to check feasibility?

Job Sequencing With Deadlines

- Order the jobs in nondecreasing order of deadlines

$$j = i_1, i_2, \ldots i_k$$

$$d_{i_1} \le d_{i_2} \le \ldots \le d_{i_k}$$

- We only need to check this permutation
  - The subset is feasible if and only if this permutation is feasible

Greedy Technique

```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

Job Sequencing With Deadlines

Analysis:

- Use presorting
  - Sorting and selection takes $O(n \log n)$ time in total
- Checking feasibility
  - Each check takes linear time in the worst case
  - $O(n^2)$ in total
- Total runtime is $O(n^2)$
  - Can we improve this?

Encoding Text

- Suppose we have to encode a text that comprises characters from some n-character alphabet by assigning to each of the text's characters some sequence of bits called ***codeword***
- We can use a fixed-encoding that assigns to each character
  - Good if each character has same frequency
  - What if some characters are more frequent than others

Encoding Text

- EX: The number of bits in the encoding of 100 characters long text

	a	b	c	d	e	f		
freq	45	13	12	16	9	5		
fixed word	000	...				101	=	300
variable word	0	101	100	111	1101	1100	=	224

Prefix Codes

- A codeword is not prefix of another codeword
  - Otherwise decoding is not easy and may not be possible
- Encoding
  - Change each character with its codeword
- Decoding
  - Start with the first bit
  - Find the codeword
    - A unique codeword can be found – prefix code
  - Continue with the bits following the codeword
- Codewords can be represented in a tree

Prefix Codes

- EX: Trees for the following codewords…

	a	b	c	d	e	f
fixed word	000	...				101
variable word	0	101	100	111	1101	1100

Huffman Codes

- Given: The characters and their frequencies
- Find: The coding tree
- Cost : Minimize the cost

$$Cost = \sum_{c \in C} f(c) \times d(c)$$

- *f(c) : frequency of c*
- *d(c) : depth of c*

Huffman Codes

- What is the greedy strategy?

Huffman Codes

- Approach :

1. Q = forest of one-node trees
   // initialize n one-node trees;
   // label the nodes with the characters
   // label the trees with the frequencies of the chars
2. for i=1 to n-1
3. x = select the least freq tree in Q & delete
4. y = select the least freq tree in Q & delete
5. z = new tree
6. z→left = x and z→right = y
7. f(z) = f(x) + f(y)
8. Insert z into Q

Greedy Technique

```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     is feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

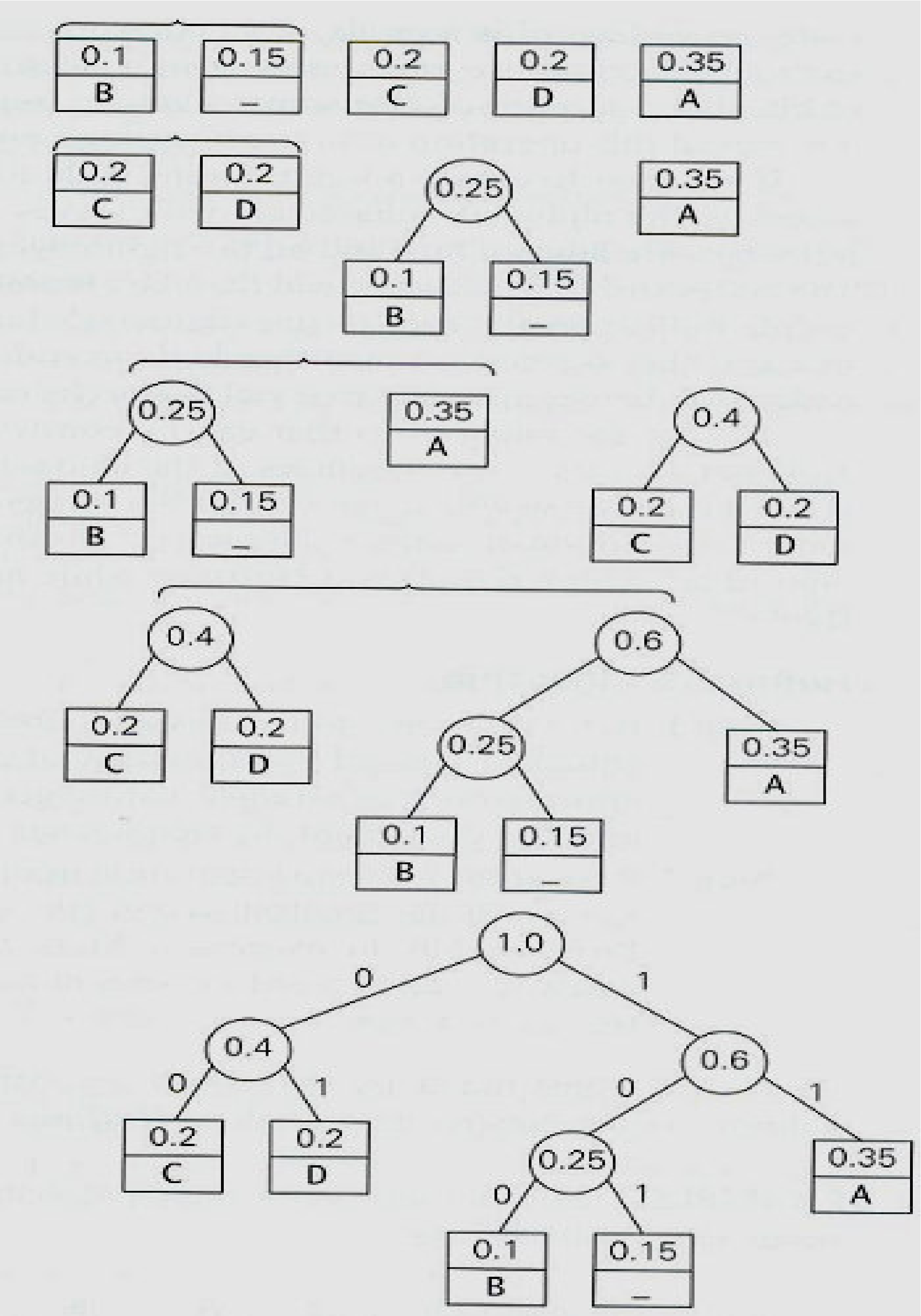
Huffman Codes Example

Consider five characters {A,B,C,D,-} with following occurrence probabilities

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is as follows

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101



0.1
B
0.15
_
0.2
C
0.2
D
0.35
A
0.25
0.4
0.6
1.0
0
1

Huffman Codes

- Optimality Proof :
  - Tree should be full
  - Two least frequent chars x and y must be two deepest nodes
  - Induction argument
    - After merge operation → new character set
      - Characters in roots with new frequencies

Huffman Codes

- Analysis :
  - Use priority queues for forest

$$O(|c| \log |c|)$$

$$\text{Buildheap} + (2|c| - 2)\ \text{DeleteMin}$$
$$+ (|c| - 1)\ \text{Insert}$$