# DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

Module 4

Radhika Chapaneri

## Greedy Method

- Greedy and Dynamic Programming are methods for solving optimization problems.
- Many real-world problems are *optimization* problems in that they attempt to find an optimal solution among many possible *candidate/feasible* solutions.

# Greedy Method

- A feasible solution is a solution that satisfies the constraints.
- We need to find a feasible solution that either minimizes or maximizes a given objective function
- An optimal solution is a feasible solution that optimizes the objective/optimization function

# Greedy Method

- **Characteristics of greedy algorithms:**
- Make a sequence of choices
- Each choice is the one that seems best so far, only depends on what's been done so far
- A decision, once made, is (usually) not changed later.
- Choice produces a smaller problem to be solved

# Greedy method

- Greedy algorithm works in stages considering one input at a time.
- At each stage, a decision is made regarding whether a particular input is in an optimal solution.
- This is done by considering the inputs in an order determined by some selection procedure.
- If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution.
- Otherwise it is added

# Greedy method control abstraction

```
Algorithm :Greedy(a,n)
//(a[1:n]) contains the n inputs
{
  Solution = 0 // Initialize solution
  for i = 1 to n do
{
     X = SELECT(a)
     If FEASIBLE( solution, x) then
     then solution = UNION( solution, x)
   }
return  (solution)
}
```

# Greedy method control abstraction

- The function SELECT selects an input from a[ ] and removes it.
- The selected input's value is assigned to x
- FEASIBLE is boolean valued function that determines whether x can be included into the solution vector.
- The function UNION combines x with the solution and updates the objective function.

# Knapsack Problem

- There are *n* different items in a store
- Item *i* :
  - weighs $w_i$ pounds
  - worth Pi
- A thief breaks in
- Can carry up to *M* pounds in his knapsack
- What should he take to maximize the value of his haul?

# 0-1 vs. Fractional Knapsack

- 0-1 Knapsack Problem:
  - the items cannot be divided
  - thief must take entire item or leave it behind
- Fractional Knapsack Problem:
  - thief can take partial items
  - for instance, items are liquids or powders
  - solvable with a greedy algorithm…

# Fractional Knapsack

- Let $x_i$ denote the fraction of the object $i$ to be included in the knapsack, $0 \le x_i \le 1$, for $1 \le i \le n$.
- Objective: To fill the knapsack to maximize the total profit earned.
- Maximize Subject to

$$\sum_{1 \le i \le n} p_i x_i \qquad \text{----------(1)}$$

$$\sum_{1 \le i \le n} w_i x_i \le M \qquad \text{---------- (2)}$$

$$0 \le x_i \le 1,\ 1 \le i \le n \qquad \text{----------(3)}$$

# Fractional Knapsack

- A feasible solution is any set(x1,…….xn) satisfying equation (2) and (3)
- Optimal solution is a feasible solution for which (1) is maximized.

# Fractional knapsack example

- Example
- M = 20,
- $(P_1, P_2, P_3)$ =(25,24,15)
- $(W_1, W_2, W_3)$ = (18, 15, 10)
- Suppose these are four feasible solutions

| | $(X_1, X_2, X_3)$ | $\Sigma W_i X_i$ | $\Sigma P_i X$ |
|---|---|---|---|
| 1. | (1/2,1/3,1/4) | 16.5 | 24.25 |
| 2. | (1,2/15,0) | 20 | 28.2 |
| 3. | (0, 2/3, 1) | 20 | 31 |
| 4. | (0, 1, 1/2) | 20 | 31.5 |

- Here Solution 4 is optimal

# Fractional knapsack example

- **2nd Approach : Greedy by profit / unit**
- This balances between the rate at which profit increases and the rate at which capacity is used.
- Example
- M = 20,
- $(P_1, P_2, P_3)$ =(25,24,15)
- $(W_1, W_2, W_3)$ = (18, 15, 10)
- **Step1: calculate pi / wi**
- $p_1/w_1$ = 25/18 = 1.38,$p_2/w_2$ = 24/15 = 1.6,$p_3/w_3$ = 15/10 = 1.5
- **Step 2: Sort pi / wi in descending order**
- P2/w2> p3/w3 > p1w1
-

# Fractional knapsack example

- Knapsack capacity =20
- Step3:

| Selection of item(xi) | wixi | Pixi | Remaining weight |
|---|---|---|---|
| X2 =1 | 15*1 | 24 *1 = 24 | 20 -15 =5 |
| X3 =5/10 =1/2 | 10 *1/2 =5 | 15*1/2 = 7.5 | 5-5 =0 |
| **Total Profit** | | **31.5** | |

- Solution is {0, 1, 1/2}.
- This is optimal solution.

# Fractional Knapsack Algorithm

**The Optimal Knapsack Algorithm:**

**Input:** an integer $n$, positive values $w_i$ and $p_i$, for $1 \le i \le n$, and another positive value $M$.

**Output:** $n$ values $x_i$ such that $0 \le x_i \le 1$ and

$$\sum_{i=1}^{n} x_i w_i \le M \text{ and } \sum_{i=1}^{n} x_i p_i \text{ is maximized.}$$

# Algorithm

Greedy-Fractional knapsack(w,p,M)

(1) Sort the $n$ objects from large to small based on the ratios $p_i/w_i$. We assume the arrays $w[1..n]$ and $p[1..n]$ store the respective weights and values after sorting.

(2) initialize array $x[1..n]$ to zeros.

(3) weight = 0; $i$ = 1

(4) while ($i \le n$ and weight < $M$) do
- (4.1) if weight + $w[i] \le M$ then $x[i]$ = 1
- (4.2) else $x[i]$ = ($M$ – weight) / w[$i$]
- (4.3) weight = weight + x[$i$] * w[$i$]
- (4.4) $i$++

(5) Return x

# Complexity

- If items are already sorted into decreasing order of pi / wi then the while loop takes O(n)
- Therefore total time including sort is O(nlogn)

# 0-1 Knapsack

- 3 items:
  - item 1 weighs 10 lbs, worth $60 ($6/lb)
  - item 2 weighs 20 lbs, worth $100 ($5/lb)
  - item 3 weighs 30 lbs, worth $120 ($4/lb)
  - knapsack can hold 50 lbs

P1/w1 = 6 P2/w2 =5, p3/w3 = 4

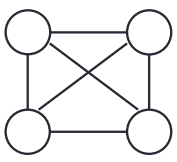
- greedy strategy:
  - take item 1
  - take item 2
  - no room for item 3

# 0-1 Knapsack Problem

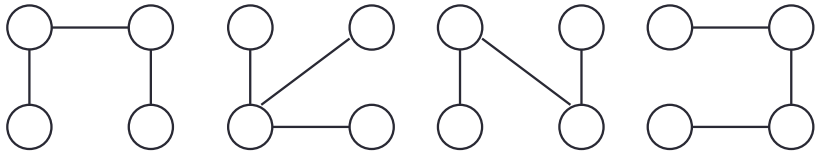
- Taking item 1 is a big mistake globally although looks good locally
- Optimal solution includes item 2 and 3
- Use dynamic programming to solve problem

# 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
- A spanning tree is a tree that contains all of the vertices in the graph and contains no cycles
-



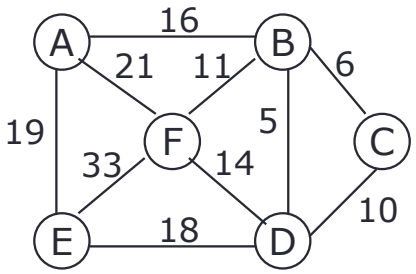
A connected, undirected graph



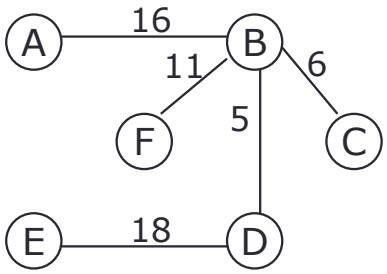
Four of the spanning trees of the graph

# 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 of a spanning tree
- A minimum-cost spanning tree is a spanning tree that has the lowest cost

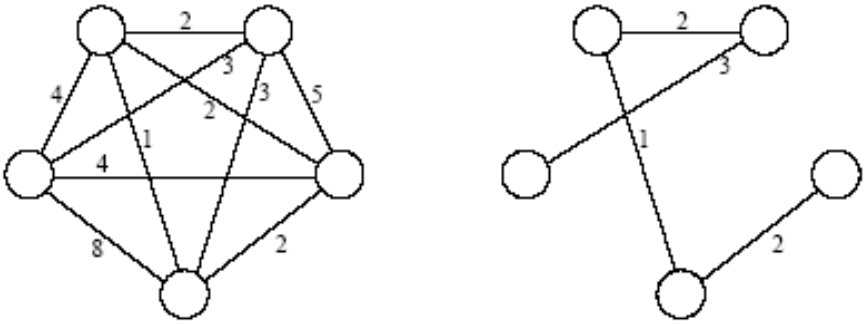



A connected, undirected graph




A minimum-cost spanning tree

# Minimum Spanning Tree



An undirected graph and its minimum spanning tree.

# More about Multiple MSTs

- Multiple MSTs can exist within a given undirected graph
- However, if there are weighted edges and all weighted edges are unique, only one MST will exist

# Application of minimum spanning tree

- Suppose you want to supply a set of houses (say, in a new subdivision) with:
  - electric power
  - water
  - sewage lines
  - telephone lines
- To reduce costs you could connect the houses with a minimum-cost spanning tree
- It can be used in computer network, for example we could create a tree that connects all of the computers. The minimum spanning tree gives us the shortest length of cable that can be used to connect all the computers together while ensuring that there is a path between any two computers.
- It can also be used in electrical circuits.

# Finding minimum spanning trees

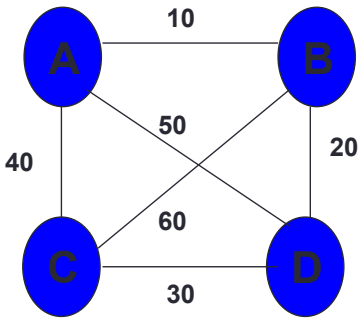
- Given a connected undirected graph with edge weights, find subset of edges that spans all the nodes, creates no cycle, and minimizes sum of weights
- There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms

- **Kruskal's algorithm**
- **Prim's algorithm**

# Prim's Algorithm

1. Select any vertex
2. Select the shortest edge connected to that vertex
3. Select the shortest edge connected to any vertex already connected
4. Repeat step 3 until all vertices have been connected

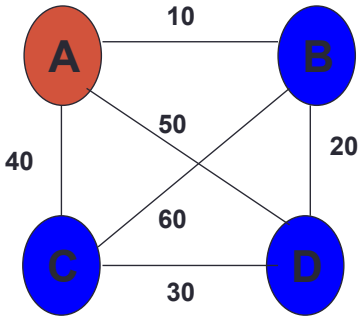
# Prim's Algorithm




Initialize array

| | *visited* | *d* | *p* |
|---|---|---|---|
| **A** | 0 | ∞ | 0 |
| **B** | 0 | ∞ | 0 |
| **C** | 0 | ∞ | 0 |
| **D** | 0 | ∞ | 0 |

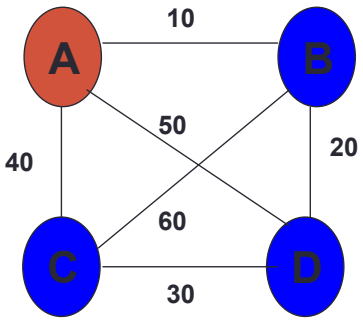
# Prim's Algorithm




Start with any vertex, suppose A

| | *visited* | *d* | *p* |
|---|---|---|---|
| **A** | 1 | 0 | 0 |
| **B** | 0 | ∞ | 0 |
| **C** | 0 | ∞ | 0 |
| **D** | 0 | ∞ | 0 |

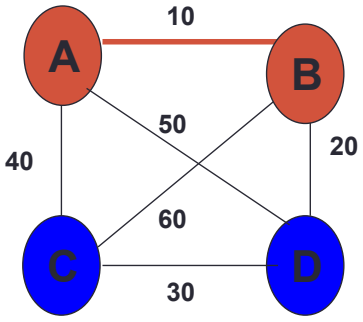
# Prim's Algorithm




Update distances of adjacent, unselected nodes

| | *visited* | *d* | *p* |
|---|---|---|---|
| **A** | 1 | 0 | 0 |
| **B** | 0 | 10 | A |
| **C** | 0 | 40 | A |
| **D** | 0 | 50 | A |

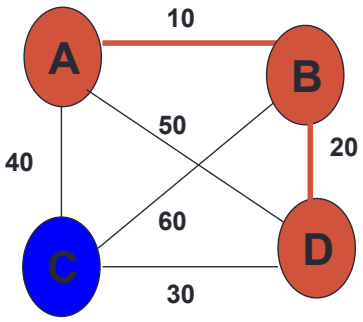
# Prim's Algorithm




- Select node with minimum distance and mark it visited
- Update distances of adjacent, unselected nodes(ie. If visited =0
- Update only if the value of the distance is less than previous value.

| | *visited* | *d* | *p* |
|---|---|---|---|
| **A** | 1 | 0 | 0 |
| **B** | 1 | 10 | A |
| **C** | 0 | 40 | A |
| **D** | 0 | 20 | B |

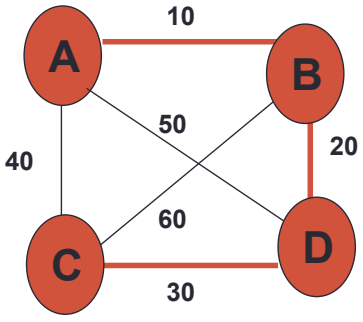
# Prim's Algorithm




- Select next unvisited node(i.e visted = 0) with minimum distance and mark it visited (make visited =1)
- Update distances of adjacent, unselected nodes(ie. nodes with visited =0)
- Update d value for that node only if the value of the distance is less than previous value.

| | *visited* | *d* | *p* |
|---|---|---|---|
| **A** | 1 | 0 | 0 |
| **B** | 1 | 10 | A |
| **C** | 0 | 30 | D |
| **D** | 1 | 20 | B |

# Prim's Algorithm



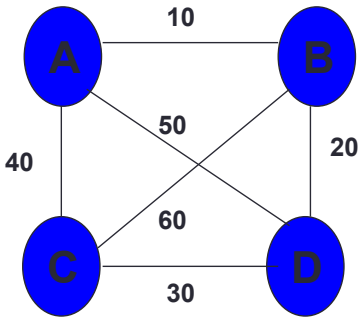

- Stop when all vertices are visited i.e visited =1 for all vertices
- Resulting tree is minimum spanning tree

| | *visited* | *d* | *p* |
|---|---|---|---|
| **A** | 1 | 0 | 0 |
| **B** | 1 | 10 | A |
| **C** | 1 | 30 | D |
| **D** | 1 | 20 | B |

The cost of minimum spanning tree = 60

# Minimum Spanning Tree: Prim's Algorithm

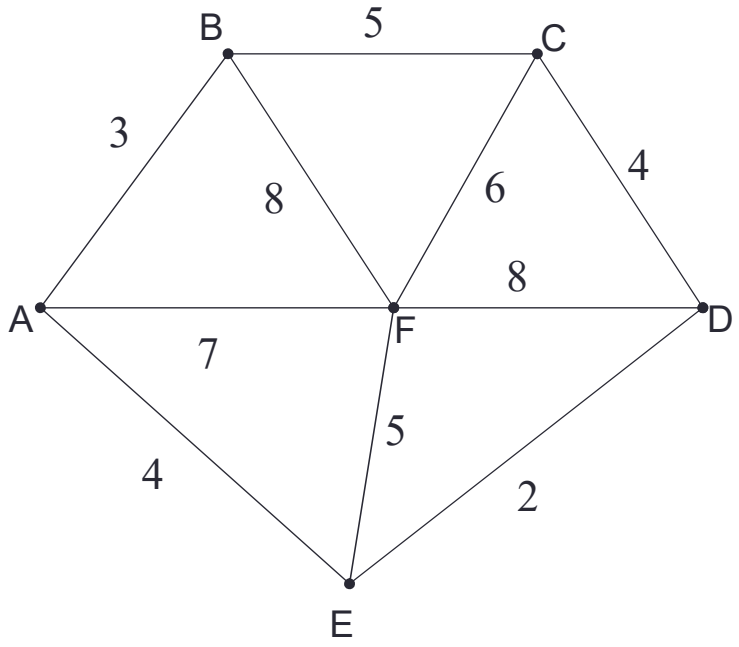
```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```




# Kruskal ‘s Algorithm steps

1. Select the shortest edge in a network
2. Select the next shortest edge which does not create a cycle
3. Repeat step 2 until all vertices have been connected
4. The edges included so far is a minimum spanning tree. calculate the total cost of resulting minimum spanning tree.

# Example



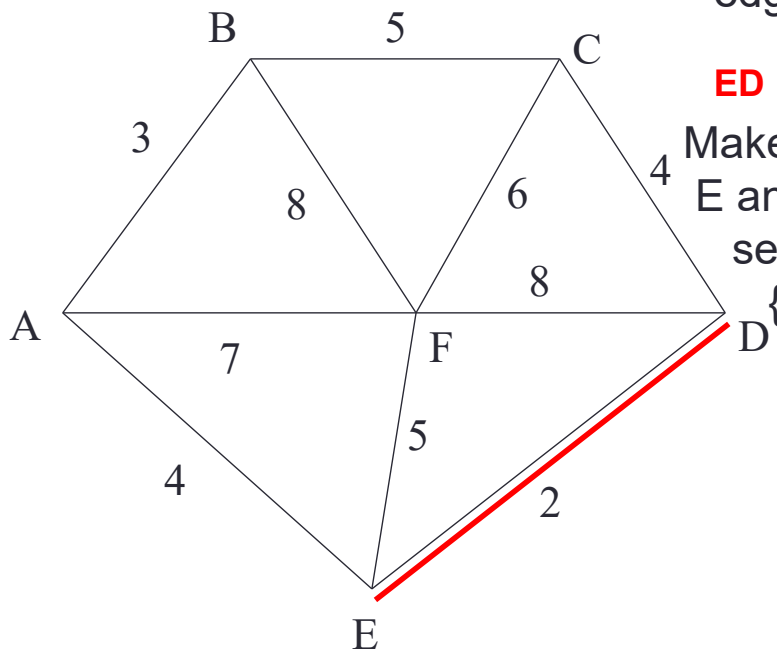

# Example

Make set of every vertices
{A} {B} {C} {D} {E} {F}

Sort the edges in order of size:

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

## Kruskal's Algorithm



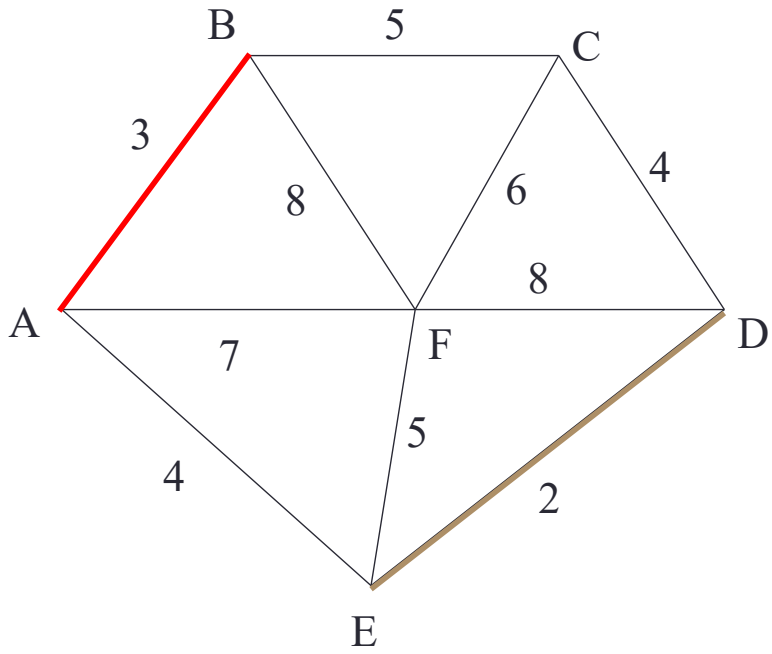

Select the shortest edge in the network

**ED 2**

Make union of set containing E and D if they are in different set

{A} {B} {C} {DE} {F}

## Kruskal's Algorithm



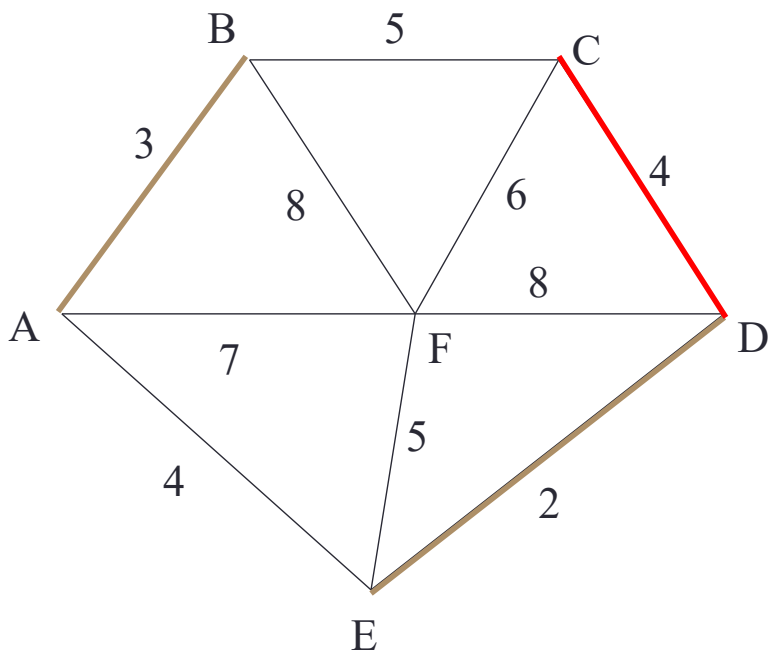

Select the next shortest edge which does not create a cycle

**ED 2**
**AB 3**

Make union of set containing A and B if they are in different set

{AB} {C} {DE} {F}

## Kruskal's Algorithm

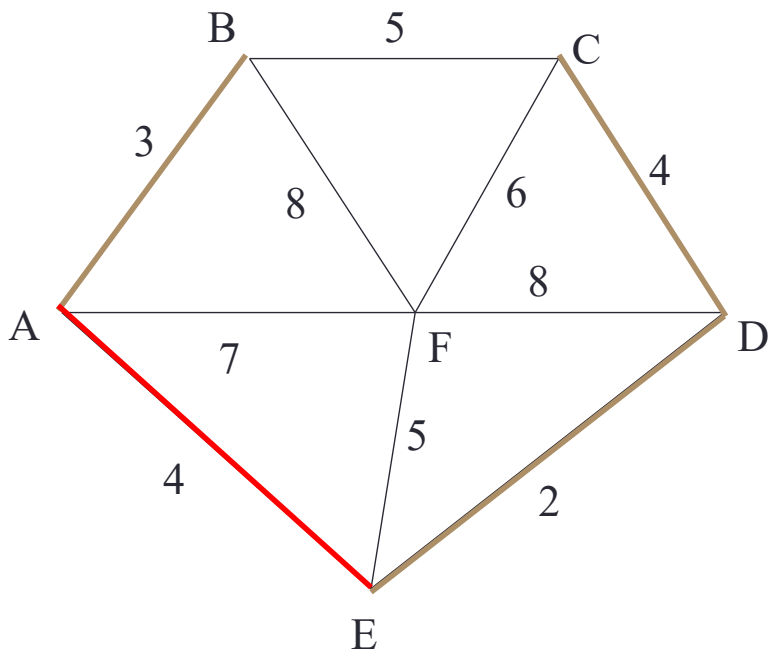



Select the next shortest edge which does not create a cycle

**ED 2**
**AB 3**
**CD 4 (or AE 4)**

Make union of set containing C and D if they are in different set
{AB} {CDE} {F}

## Kruskal's Algorithm

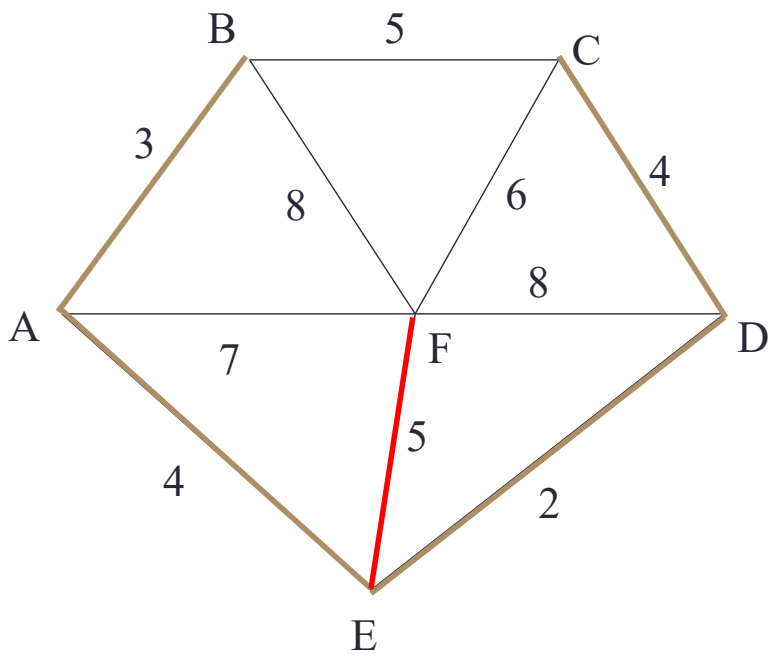



Select the next shortest edge which does not create a cycle

**ED 2**
**AB 3**
**CD 4**
**AE 4**

Make union of set containing A and E if they are in different set
{ABCDE} {F}

## Kruskal's Algorithm

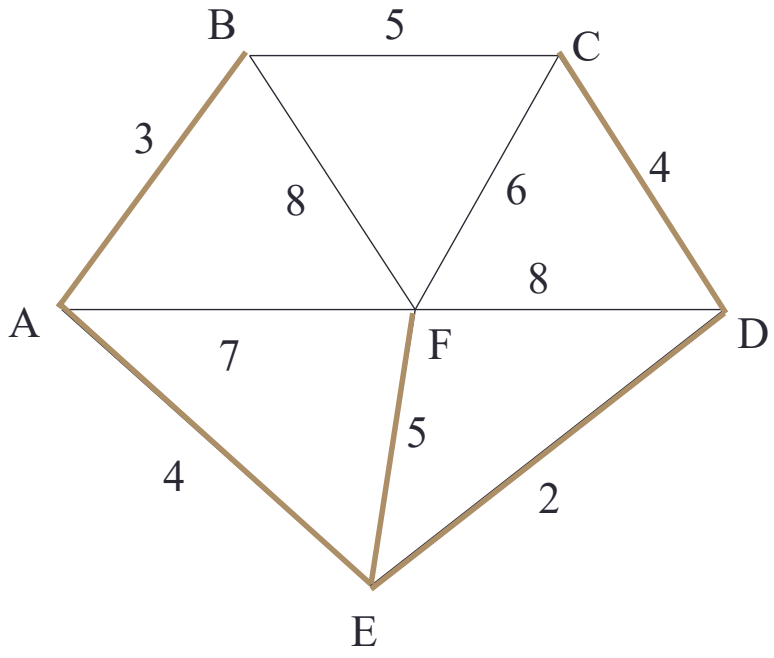



Select the next shortest edge which does not create a cycle

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

BC are in same set hence forms a cycle
Make union of set containing
E and F
if they are in different set {ABCDEF}

## Kruskal's Algorithm




All vertices have been connected.

The solution is

**ED 2**
**AB 3**
**CD 4**
**AE 4**
**EF 5**

Total cost of Minimum spanning tree: 18

# Kruskal's Algorithm

```
Kruskal(G,w)
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm: Running Time

- To summarize:
  - Sort edges: O(E lg E)
  - O(V) MakeSet()'s
  - O(E) FindSet()'s
  - O(V) Union()'s
  - Overall thus O(E lg E)

# Huffman Algorithm

- **Fixed Length Character Encoding:**
- Computers usually use fixed-length character encodings.
- •ASCII uses 8 bits per character.
- example: “bat” is stored in a text file as the following sequence of bits:
- 01100010 01100001 01110100
- Fixed-length encodings are simple, because
- All character encodings have the same length
- A given character always has the same encoding

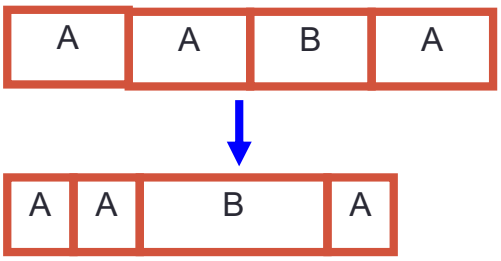


# Huffman Algorithm

- **Variable Length Character Encoding:**
- Problem: fixed-length encodings waste space.
- Solution: use a variable-length encoding.
- Assign shorter encodings to frequently occurring characters

# Huffman Code

- Approach
  - Variable length encoding of symbols
- Principle
  - Use fewer bits to represent frequent symbols
  - Use more bits to represent infrequent symbols
  - Obey prefix property.






# Huffman Code Example

| Symbol | A | B | C | D |
|---|---|---|---|---|
| Frequency | 13% | 25% | 50% | 12% |
| Original Encoding | 00 | 01 | 10 | 11 |
| | 2 bits | 2 bits | 2 bits | 2 bits |
| Huffman Encoding | 110 | 10 | 0 | 111 |
| | 3 bits | 2 bits | 1 bit | 3 bits |

- Expected size
  - Original $\Rightarrow$ 1/8×2 + 1/4×2 + 1/2×2 + 1/8×2 = 2 bits / symbol
  - Huffman $\Rightarrow$ 1/8×3 + 1/4×2 + 1/2×1 + 1/8×3 = 1.75 bits / symbol

# Prefix Property : Huffman Code

- **Prefix Property**: once a certain bit pattern has been assigned as the code of a symbol, no other codes should start with that pattern (the pattern cannot be the *prefix* of any other code).

  Example

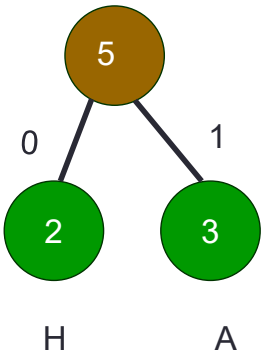
  Huffman("I") ⇒ 00

  Huffman("X") ⇒ 001 // not legal prefix code



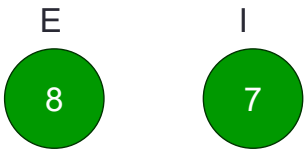
# Huffman Tree Construction 1

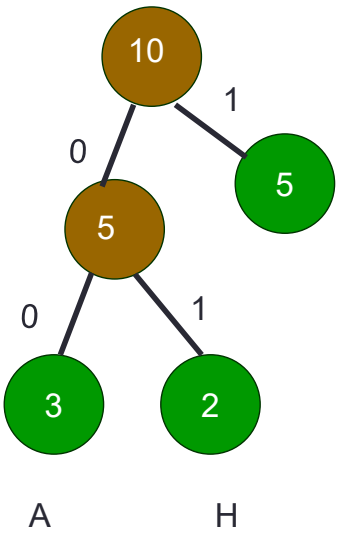
# Huffman Tree Construction 2

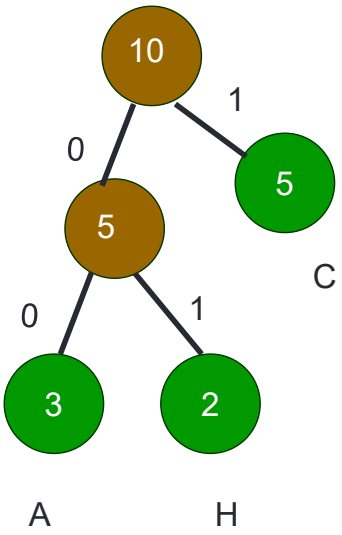


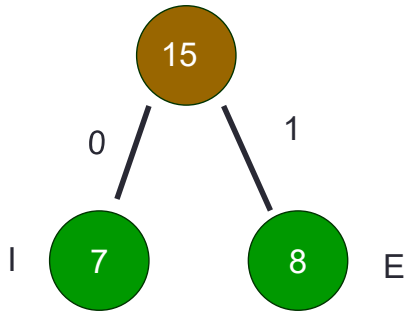

# Huffman Tree Construction 2



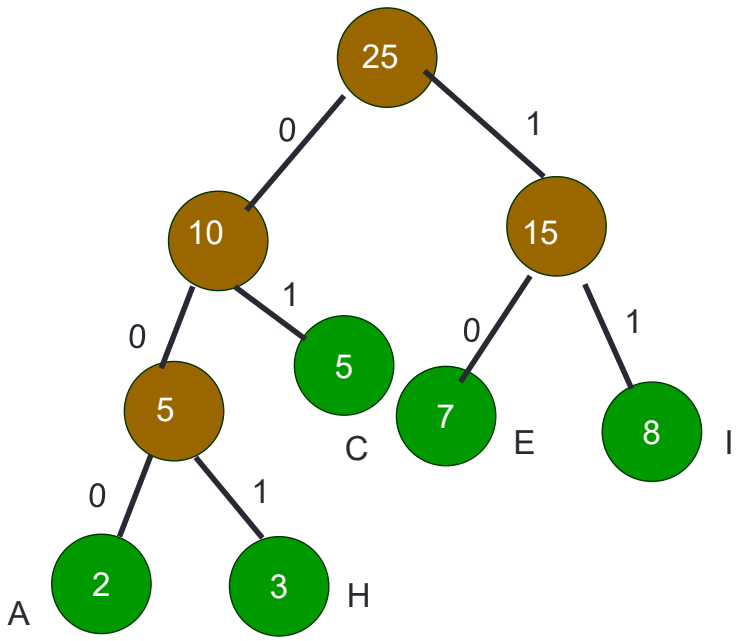
# Huffman Tree Construction 2






# Huffman Tree Construction




| | | |
|---|---|---|
| E | = | 10 |
| I | = | 11 |
| C | = | 01 |
| A | = | 000 |
| H | = | 001 |

# Huffman Coding Example

- Huffman code

| | | |
|---|---|---|
| E | = | 01 |
| I | = | 00 |
| C | = | 10 |
| A | = | 111 |
| H | = | 110 |

- Input
  - ACE
- Output
  - (111)(10)(01) = 1111001



# Huffman Tree Algorithm

- **Q- Minimum priority Queue**
- **huffman** (C)
- n = |C|
- Q = C
- **for** (i = 1 to n − 1) **do**
- **allocate** a new node z
- left[z] = x = **extractMin**(Q)
- right[z] = y = **extractMin**(Q)
- f[z] = f[x] + f[y]
- **insert**(Q, z)
- **return extractMin**(Q)

# Dijkstra's algorithm

**Dijkstra's algorithm** - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

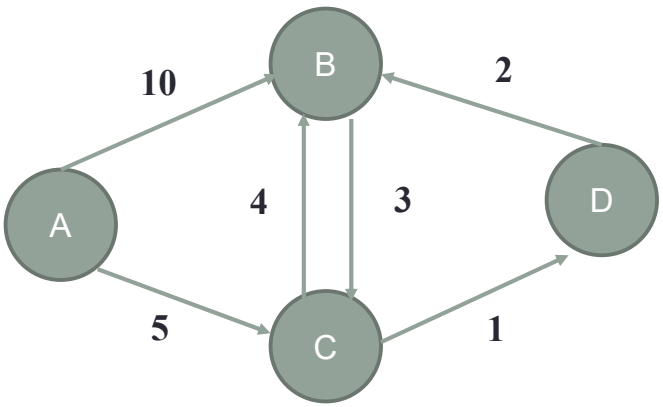
Approach: Greedy

Input: Weighted graph G={E,V} and source vertex $s \in V$, such that all edge weights are nonnegative

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

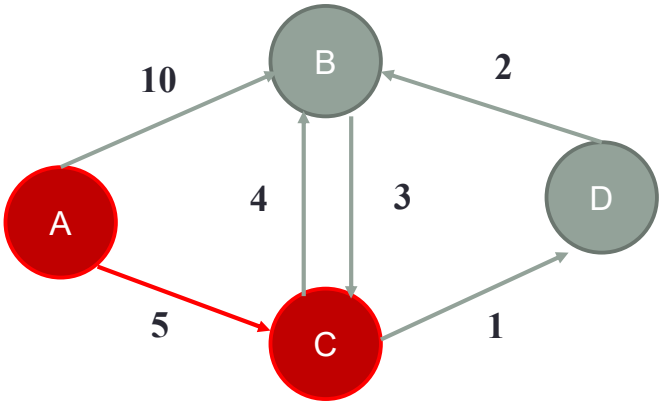
# Example

Consider A as a source vertex

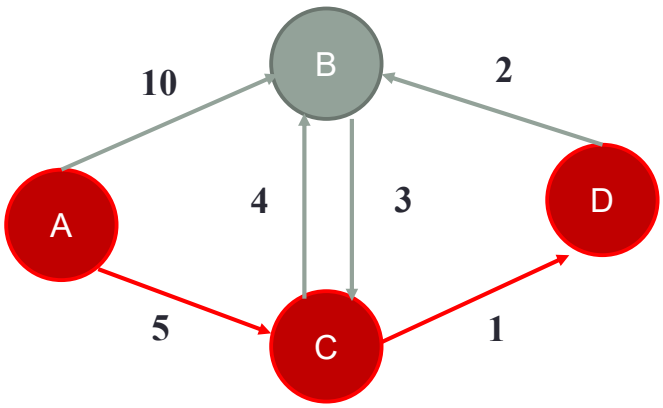



| | *visited* | *d* | *p* |
|---|---|---|---|
| **A** | 1 | 0 | 0 |
| **B** | 0 | ∞ | 0 |
| **C** | 0 | ∞ | 0 |
| **D** | 0 | ∞ | 0 |

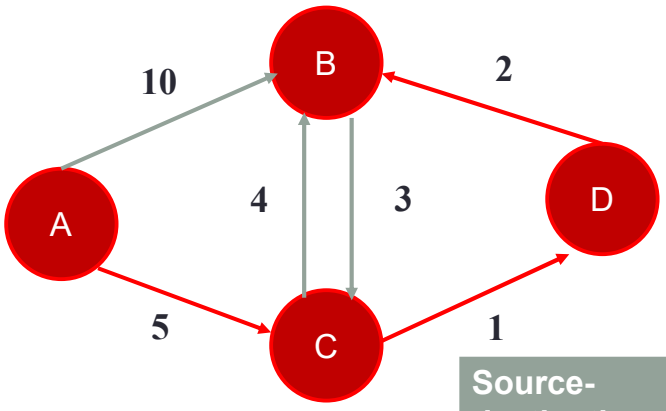
| | *visited* | *d* | *p* |
|---|---|---|---|
| **A** | 1 | 0 | 0 |
| **B** | 0 | 10 | A |
| **C** | 0 | 5 | A |
| **D** | 0 | ∞ | 0 |




| | *visited* | *d* | *p* |
|---|---|---|---|
| **A** | 1 | 0 | 0 |
| **B** | 0 | 9 | C |
| **C** | 1 | 5 | A |
| **D** | 0 | 6 | C |




| | *visited* | *d* | *p* |
|---|---|---|---|
| **A** | 1 | 0 | 0 |
| **B** | 0 | 8 | D |
| **C** | 1 | 5 | A |
| **D** | 1 | 6 | C |




| | *visited* | *d* | *p* |
|---|---|---|---|
| **A** | 1 | 0 | 0 |
| **B** | 1 | 8 | D |
| **C** | 1 | 5 | A |
| **D** | 1 | 6 | C |

| Source-destination | Shortest path | Shortest PathLength |
|---|---|---|
| A-B | A-C-D-B | 5+1+2 = 8 |
| A-C | A-C | 5 |
| A-D | A-C-D | 5+1=6 |

# Dijkstra's Algorithm

- Dijkstra(G=(V,E,w),s)

1. Let H = V- {s}
2. For every vertex v do
3. dist[v] = ∞, parent[v] = null
4. dist[s] = 0, parent[s] =none
5. update(s)
6. For i =1 to n-1 do
7. u = extract vertex from H of smallest weight
8. Update(u)
9. Return dist[]

# Dijkstra's Algorithm

- Update(u)
- 1. for every neighbour v of u(such that v in H)
  2. If dist[v] > dist[u] + w(u,v) then
  3. dist[v] = dist[u] + w(u,v)
  4. parent[v] = u
  5.

  Complexity of Dijkstra's algorithm: $O(n^2)$

# Job sequencing with deadlines

- There are n jobs to be processed on a machine.
- Each job i has an integer deadline $d_i \geq 0$ and profit $p_i \geq 0$.
- Pi is earned if the job is completed within its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine.

# Job sequencing with deadlines

- A feasible solution is a subset of jobs J such that each job is completed by its deadline.
- An optimal solution is a feasible solution with maximum profit value.

  **Example** : Let 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

| Sr.No. | Feasible Solution | Processing Sequence | Profit value |
|---|---|---|---|
| (i) | (1,2) | (2,1) | 110 |
| (ii) | (1,3) | (1,3) or (3,1) | 115 |
| (iii) | (1,4) | (4,1) | 127 is the optimal one |
| (iv) | (2,3) | (2,3) | 25 |
| (v) | (3,4) | (4,3) | 42 |
| (vi) | (1) | (1) | 100 |
| (vii) | (2) | (2) | 10 |
| (viii) | (3) | (3) | 15 |
| (ix) | (4) | (4) | 27 |

# Job Sequencing with deadline

**Example**

| Job | Profit | Deadline |
|---|---|---|
| 1 | 100 | 2 |
| 2 | 10 | 1 |
| 3 | 15 | 2 |
| 4 | 27 | 1 |

**Step1: Arrange according to descending order of profit**

| SNO | Job | Profit | Deadline |
|---|---|---|---|
| 1 | Job 1 | 100 | 2 |
| 2 | Job 4 | 27 | 1 |
| 3 | Job 3 | 15 | 2 |
| 4 | Job 2 | 10 | 1 |

# Job Sequencing with deadline

- Create an array J[ ] which stores the jobs. Initally J[ ] will be

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |

- Add ith job in array J[ ] at the index denoted by its deadline di
- First job is job1.The deadline for this job is 2. Hence insert job1 in the array J[ ] at index 2

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 0 | Job1 | 0 | 0 |

# Job Sequencing with deadline

- Next job is p4. its deadline is 1. So insert at index 1

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| Job4 | Job1 | 0 | 0 |

- Next job is Job 3 its deadline is 2 but its already occupied. So job 3 is not part of the solution
- Next job is job2, its deadline is 1 but its already occupied. So job 3 is not part of the solution
- So final optimum solution is Job 1 and Job 4 and processing sequence is **<4,1>** and the total profit earned is **127**

# 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 $L_i$ where $1 \leq i \leq n$
- All programs can be stored on the tape if and only if the sum of the lengths of the programs is at most L
- 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) =
- $$\frac{1}{n}\sum_{j=1}^{n} t_j$$
- In the optimal storage on tape problem, we are required to find a permutation for the n programs so that when they are stored on the tape in this order the MRT is minimized.
- Minimizing the MRT is equivalent to minimizing

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

# Example

- Let n = 3, $(L_1,L_2,L_3) = (5,10,3)$.
- There are n!=6 possible orderings.
- 

| Ordering I | d(I) |
|---|---|
| 1,2,3 | (5)+(5+10)+(5+10+3) = 38 |

-

# Example

- Let n = 3, $(L_1,L_2,L_3) = (5,10,3)$.
- There are n!=6 possible orderings.

| 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 |

The optimal is 3,1,2

# Optimal Storage on Tapes

- Algorithm:
- Opimalstorage on single tape(n,lengths)
- Make tape empty
- Sort in ascending order according to length of the files
- For I =1to n
- grab the next shortest on file
- put it next on the tape
- Return ordering which gives optimal solution

- Complexity of this Algorithm is 0(nlogn)