



**WOLLO UNIVERSITY**  
**KOMBOLCHA INSTITUTES OF TECHNOLOGY**  
College of Informatics

## **Analysis of Algorithms**

---

### **Chapter 3**

### **Greedy Model**

Belachew N.  
nbelay2112@gmail.com

# Outline

- ✓ Introduction
- ✓ Job sequencing with deadlines
- ✓ Optimal merge pattern
- ✓ Minimum spanning trees
- ✓ Single source shortest pattern



# Introduction

- ✓ The greedy method is perhaps the most straightforward design technique and it can be applied to a wide variety of problems.
- ✓ Most of these problems have  $n$  inputs and require us to obtain a subset that satisfies some constraints.
- ✓ Any subset that satisfies these constraints is called a **feasible solution**.
- ✓ We are required to find a feasible solution that either maximizes or minimizes a given objective function.
- ✓ A feasible solution that does this is called an **optimal solution**.
- ✓ There is usually an obvious way to determine a feasible solution, but not necessarily an optimal solution.

## ...cont'd

- ✓ The greedy method suggests that one can devise an algorithm which works in stages, considering one input at a time.
- ✓ At each stage, a decision is made regarding whether or not 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.
- ✓ The selection procedure itself is based on some optimization measure.
- ✓ This measure may or may not be the objective function.
- ✓ In fact, several different optimization measures may be plausible for a given problem.
- ✓ Most of these, however, will result in algorithms that generate suboptimal solutions.

```

procedure GREEDY( $A, n$ )
  //  $A(1:n)$  contains the  $n$  inputs//
  solution  $\leftarrow \phi$  //initialize the solution to empty//
  for  $i \leftarrow 1$  to  $n$  do
     $x \leftarrow \text{SELECT}(A)$ 
    if FEASIBLE(solution,  $x$ )
      then solution  $\leftarrow \text{UNION}(\textit{solution}, x)$ 
    endif
  repeat
  return (solution)
end GREEDY

```

Algorithm 3.1 Greedy method



# Job sequencing with deadlines

✓ The problem is stated as below.

- There are  $n$  jobs to be processed on a machine.
- Each job  $i$  has a deadline  $d_i > 0$  and profit  $p_i > 0$ .
- $p_i$  is earned iff the job is completed by 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.

## ...cont'd

- ✓ Consider the following 5 jobs and their associated deadline and profit.

<b>index</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
<b>JOB</b>	<b>j1</b>	<b>j2</b>	<b>j3</b>	<b>j4</b>	<b>j5</b>
<b>DEADLINE</b>	<b>2</b>	<b>1</b>	<b>3</b>	<b>2</b>	<b>1</b>
<b>PROFIT</b>	<b>60</b>	<b>100</b>	<b>20</b>	<b>40</b>	<b>20</b>

## ...cont'd

- ✓ Sort the jobs according to their profit in descending order.
- ✓ Note: If two or more jobs are having the same profit then sort them as per their entry in the job list.

index	1	2	3	4	5
JOB	j2	j1	j4	j3	j5
DEADLINE	1	2	2	3	1
PROFIT	100	60	40	20	20



## ...cont'd

- ✓ Find the maximum deadline value
- ✓ Looking at the jobs we can say the max deadline value is 3. So,  $d_{\max} = 3$
- ✓ As  $d_{\max} = 3$  so we will have THREE slots to keep track of free time slots.
- ✓ Set the time slot status to EMPTY .

Time slot	1	2	3
Status	Empty	Empty	Empty

## ...cont'd

- ✓ If we look at job j2, it has a deadline 1. This means we have to complete job j2 in time slot 1 if we want to earn its profit.
- ✓ Similarly, if we look at job j1 it has a deadline 2.
- ✓ This means we have to complete job j1 on or before time slot 2 in order to earn its profit.
- ✓ Similarly, if we look at job j3 it has a deadline 3. This means we have to complete job j3 on or before time slot 3 in order to earn its profit.
- ✓ Our objective is to select jobs that will give us higher profit

Time slot	1	2	3
Status	j2	j1	j3

## ...cont'd

### Algorithm GreedyJob ( $d, J, n$ )

//  $J$  is a set of jobs that can be completed by their deadlines.

{

$J := \{1\};$

for  $i := 2$  to  $n$  do

{

if (all jobs in  $J \cup \{i\}$  can be completed by their dead lines) then  $J := J \cup \{i\};$

}

}

### Time complexity

The time complexity of this problem is  $O(n^2)$ .



# Optimal merge pattern

- ✓ When more than two sorted files are to be merged together the merge can be accomplished by repeatedly merging sorted files in pairs.
- ✓ Thus, if files  $X_1$ ,  $X_2$ ,  $X_3$  and  $X_4$  are to be merged we could first merge  $X_1$  and  $X_2$  to get a file  $Y_1$ .
- ✓ Then we could merge  $Y_1$  and  $X_3$  to get  $Y_2$ .
- ✓ Finally,  $Y_2$  and  $X_4$  could be merged to obtain the desired sorted file.
- ✓ Alternatively, we could first merge  $X_1$  and  $X_2$  getting  $Y_1$ , then merge  $X_3$  and  $X_4$  getting  $Y_2$  and finally  $Y_1$  and  $Y_2$  getting the desired sorted file.
- ✓ The problem we shall address ourselves to now is that of determining an optimal way to pairwise merge  $n$  sorted files together.

## ...cont'd

- ✓ Merging an  $n$  record file and an  $m$  record file requires possibly  $n + m$  records moves, the obvious choice for a selection criterion is: at each step merge the two smallest size files together.
- ✓ Thus, if we have five files ( $F1, \dots, F5$ ) with sizes  $(20, 30, 10, 5, 30)$  our greedy rule would generate the following merge pattern:
  - ✓ Merge  $F4$  and  $F3$  to get  $Z1$  ( $|Z1| = 15$ );
  - ✓ Merge  $Z1$  and  $F1$  to get  $Z2$  ( $|Z2| = 35$ );
  - ✓ Merge  $F2$  and  $F5$  to get  $Z3$  ( $|Z3| = 60$ );
  - ✓ Merge  $Z2$  and  $Z3$  to get the answer  $Z4$ .
- ✓ The total number of record moves is 205.

...cont'd

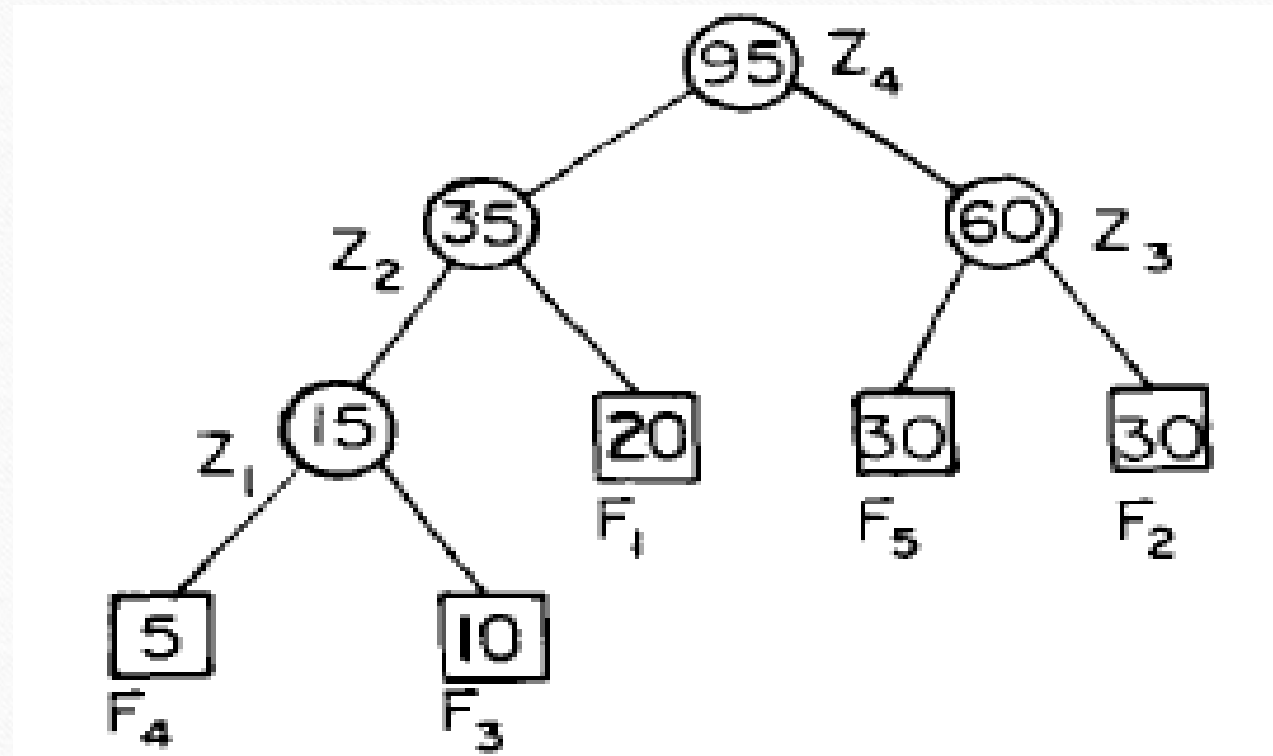


Fig 3.2 Binary merge tree representing a merge pattern



## ...cont'd

Algorithm: TREE (n)

for i := 1 to n-1 do // declare new node

node.leftchild := least (list)

node.rightchild := least (list)

node.weight := ((node.leftchild).weight) + ((node.rightchild).weight)

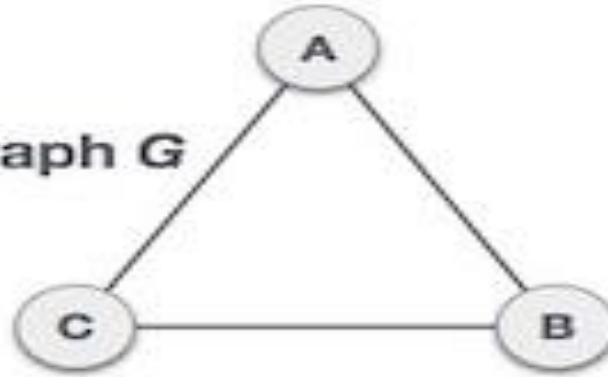
insert (list, node);

return least (list);

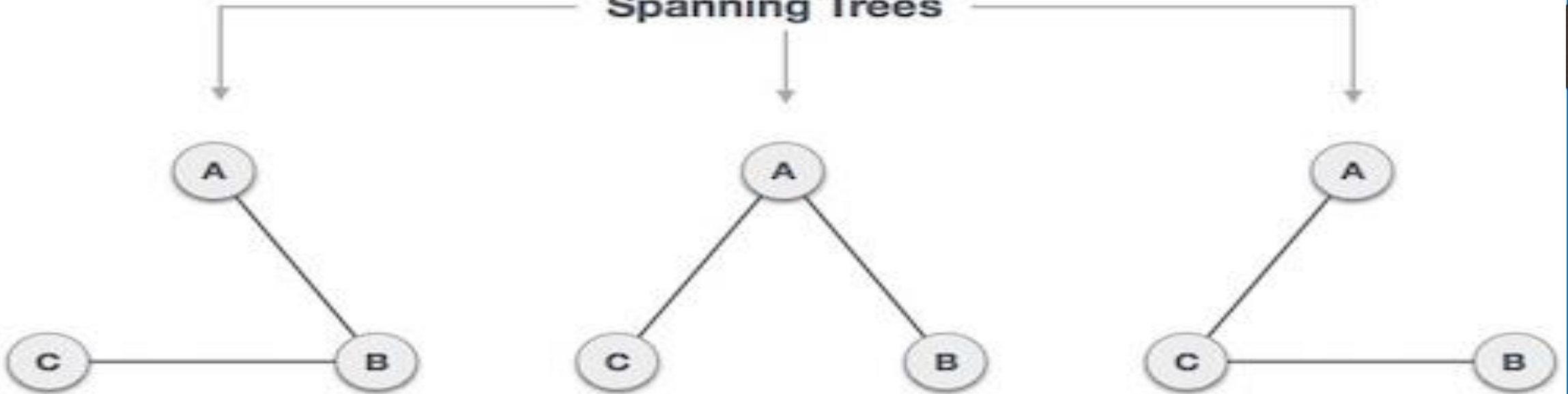
# Minimum Spanning Trees

- ✓ First let's define a tree, a spanning tree, and a minimum spanning tree:
- ✓ **Tree:** A connected graph without cycles.
- ✓ A cycle is a path that starts and ends at the same vertex.
- ✓ A **spanning tree** is a subset of Graph  $G$ , which has all the vertices covered with minimum possible number of edges.
- ✓ Hence, a spanning tree does not have cycles and it cannot be disconnected.
- ✓ By this definition, we can draw a conclusion that every connected and undirected Graph  $G$  has at least one spanning tree.

**Graph G**



**Spanning Trees**





# General Properties of Spanning Tree

- ✓ A connected graph  $G$  can have more than one spanning tree.
- ✓ All possible spanning trees of graph  $G$ , have the same number of edges and vertices.
- ✓ The spanning tree does not have any cycle (loops).
- ✓ Removing one edge from the spanning tree will make the graph disconnected.
- ✓ Adding one edge to the spanning tree will create a circuit or loop.
- ✓ **Mathematical Properties of Spanning Tree**
  - Spanning tree has  $n-1$  edges, where  $n$  is the number of nodes (vertices).
  - From a complete graph, by removing maximum  $e - n + 1$  edges, we can construct a spanning tree.
  - A complete undirected graph can have maximum  $n^{n-2}$  number of spanning trees.

## ...cont'd

- ✓ In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.
- ✓ In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.
- ✓ Minimum Spanning-Tree Algorithm
  - Kruskal's Algorithm
  - Prim's Algorithm

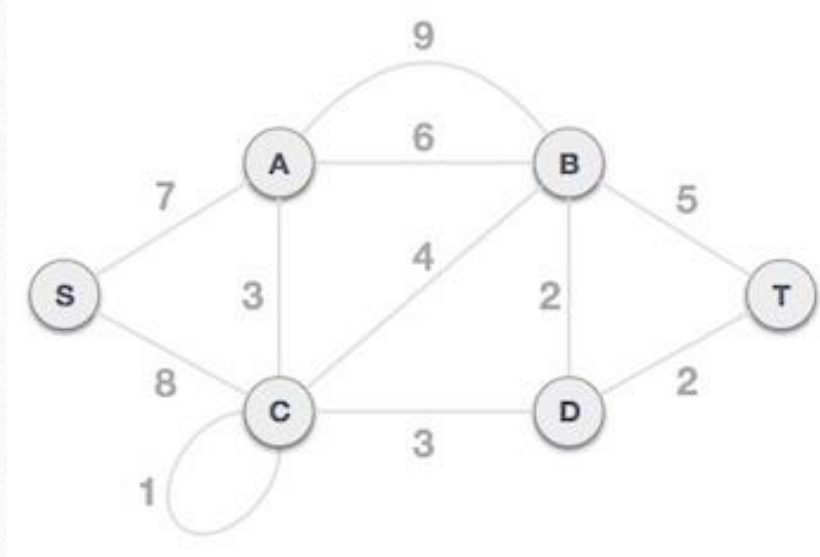
# Kruskal's Algorithm

- ✓ Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph.
- ✓ The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph.
- ✓ Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.



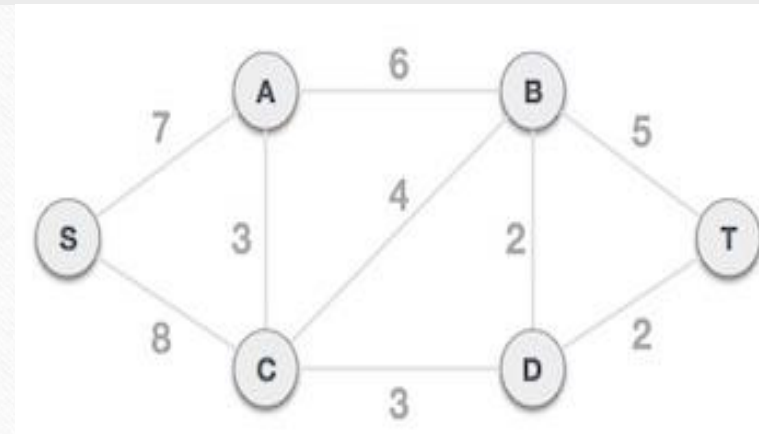
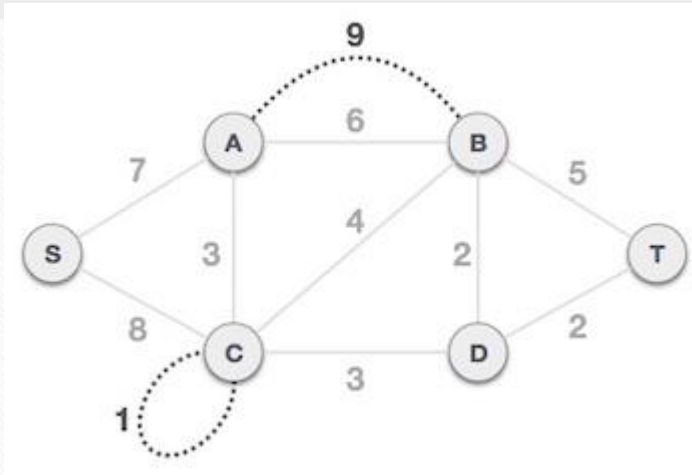
## ...cont'd

- ✓ To understand Kruskal's algorithm let us consider the following example



- ✓ Step 1 - Remove all loops and Parallel Edges
  - Remove all loops and parallel edges from the given graph.
  - In case of parallel edges, keep the one which has the least cost associated and remove all others.

...cont'd



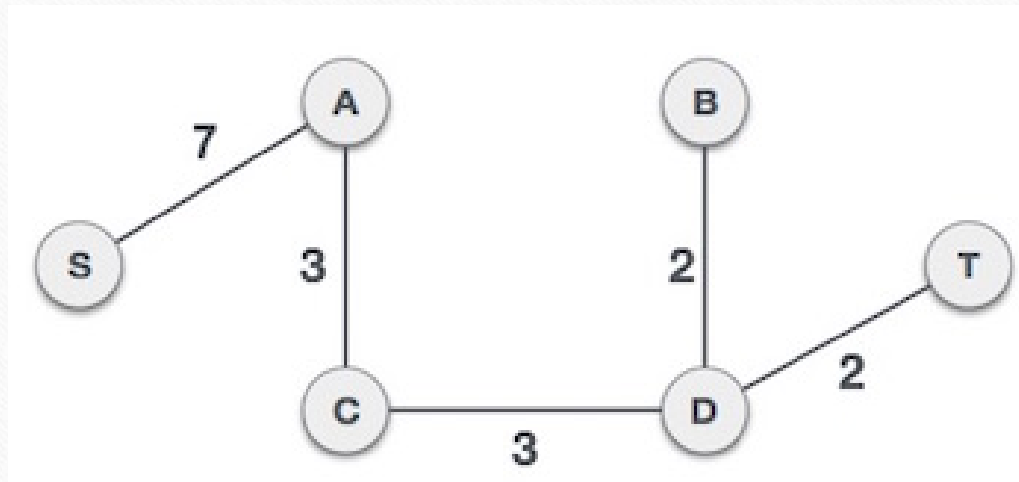
✓ Step 2 - Arrange all edges in their increasing order of weight

- The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

## ...cont'd

- ✓ Step 3 - Add the edge which has the least weightage
- Now we start adding edges to the graph beginning from the one which has the least weight.
  - Throughout, we shall keep checking that the spanning properties remain intact.
  - In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.





## ...cont'd

Algorithm Kruskal ( $E$ , cost,  $n$ )

```
{
     $T = 0$ ;    /* start with a Tree having no edges */
    While ( $T$  contains less than  $n-1$  edges and  $E$  is not empty)
    {
        choose edge  $(u,v)$  from  $E$  such that cost  $[u,v]$  is minimum
        delete     $(u,v)$  from  $E$ 
        if  $(u,v)$  does not create a cycle in  $T$ 
            add  $(u,v)$  to  $T$ 
        else
            discard  $(u,v)$ 
    }
    if ( $T$  contains fewer than  $n-1$  edges)
        Print "no spanning tree"
}
```

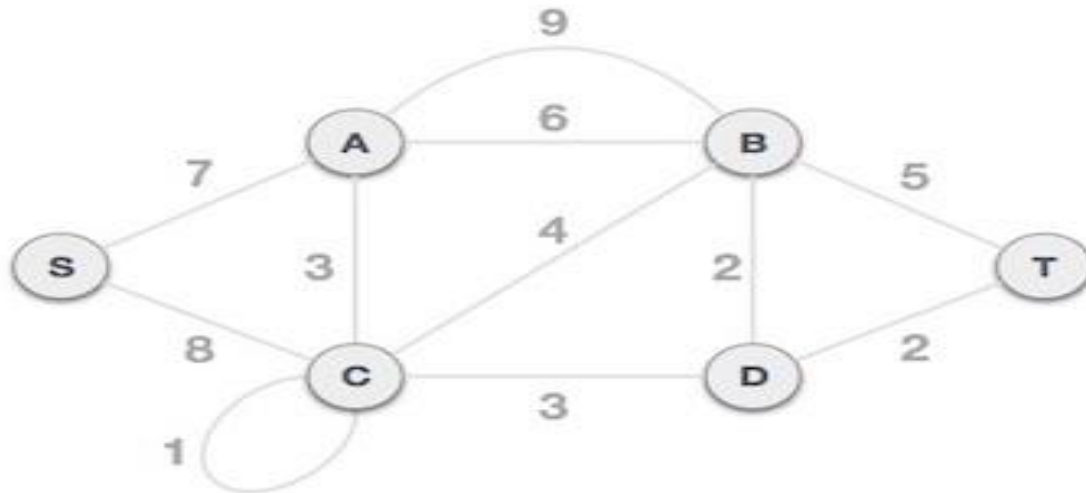
5/23/2020

## ...cont'd

- ✓ **Analysis:** Where  $E$  is the number of edges in the graph and  $V$  is the number of vertices, Kruskal's Algorithm can be shown to run in  $O(E \log E)$  time, or simply,  $O(E \log V)$  time, all with simple data structures.
  - These running times are equivalent because:
  - $E$  is at most  $V^2$  and  $\log V^2 = 2 \times \log V$  is  $O(\log V)$ .

# Prim's Algorithm

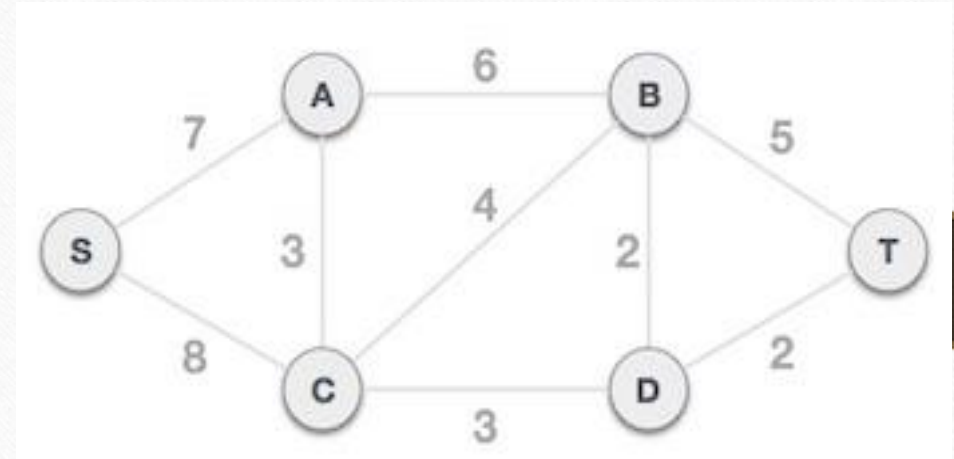
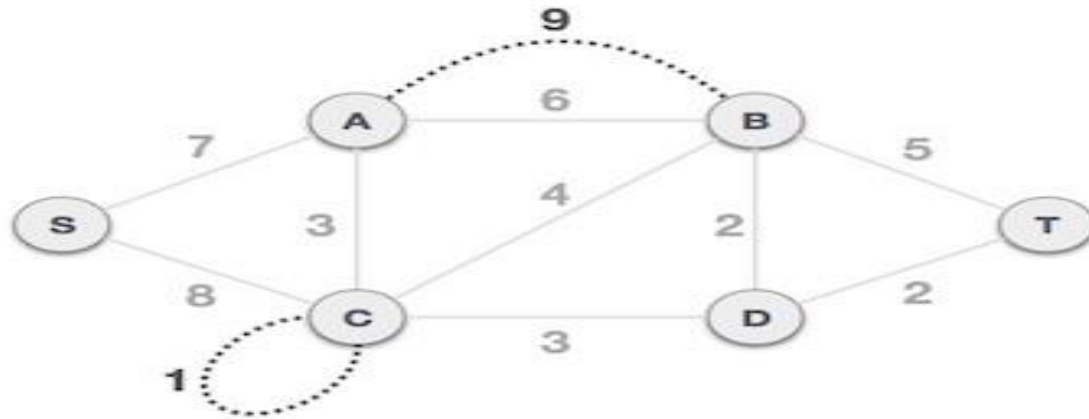
- ✓ Prim's algorithm to find minimum cost spanning tree uses the greedy approach.
- ✓ Prim's algorithm shares a similarity with the shortest path first algorithms.
- ✓ Prim's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.
- ✓ we shall see with example –





## ...cont'd

- ✓ Step 1 - Remove all loops and parallel edges



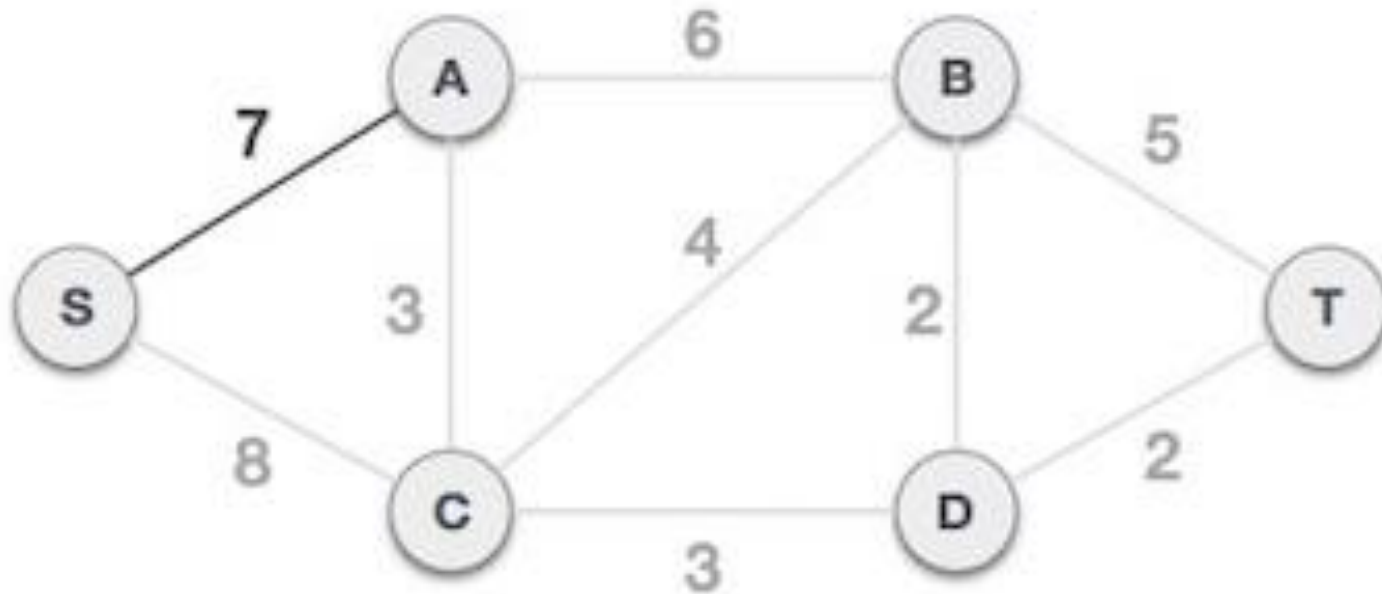
- ✓ Remove all loops and parallel edges from the given graph.
- ✓ In case of parallel edges, keep the one which has the least cost associated and remove all others.

## ...cont'd

- ✓ Step 2 - Choose any arbitrary node as root node
- ✓ In this case, we choose S node as the root node of Prim's spanning tree.
- ✓ This node is arbitrarily chosen, so any node can be the root node.
- ✓ One may wonder why any node can be a root node.
- ✓ So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

## ...cont'd

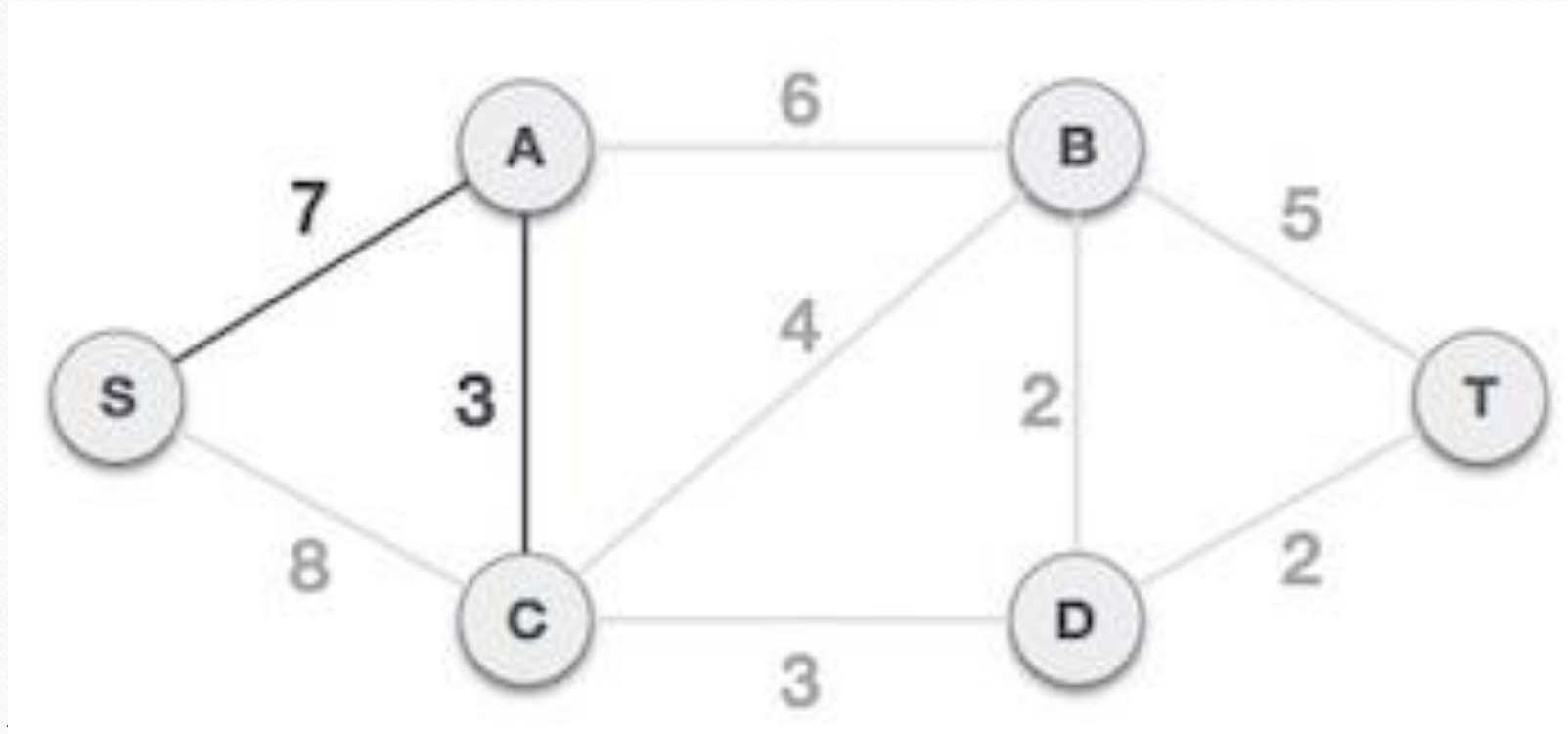
- ✓ Step 3 - Check outgoing edges and select the one with less cost
- ✓ After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.





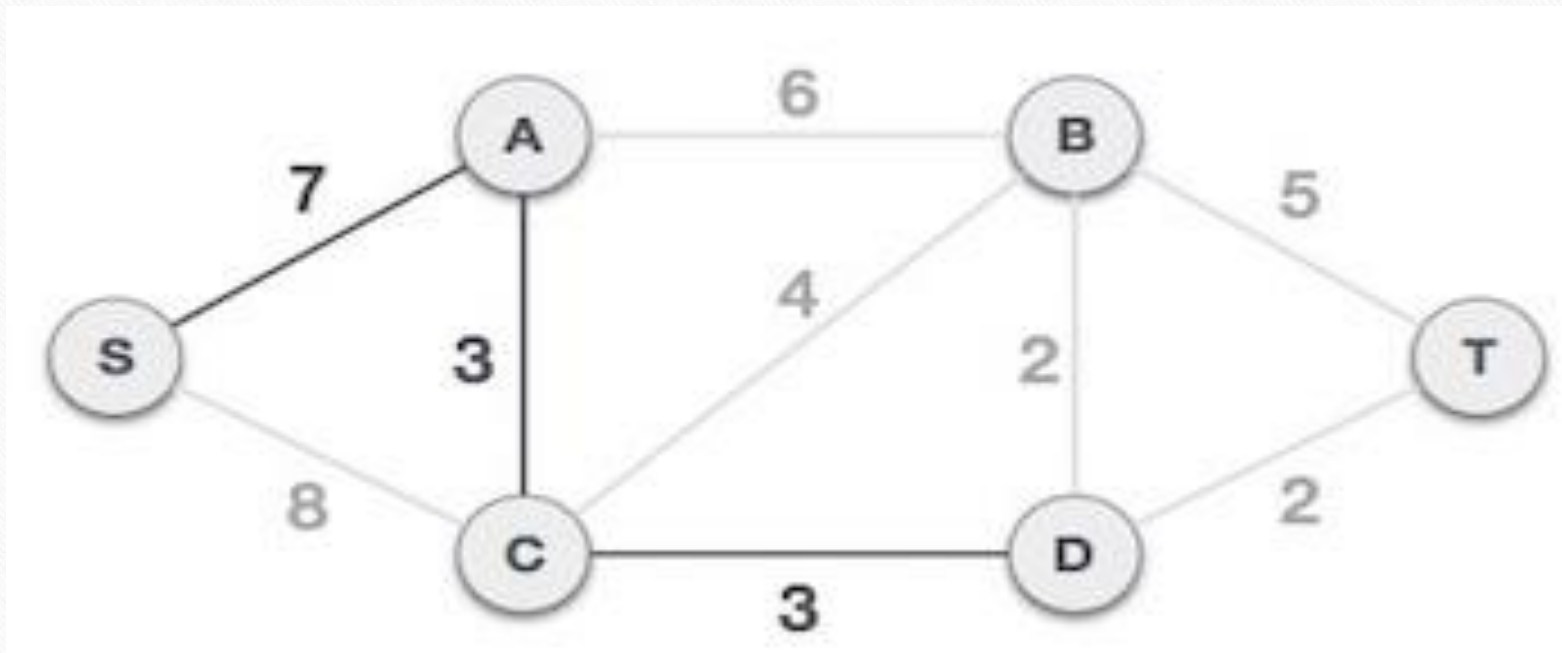
## ...cont'd

- ✓ Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



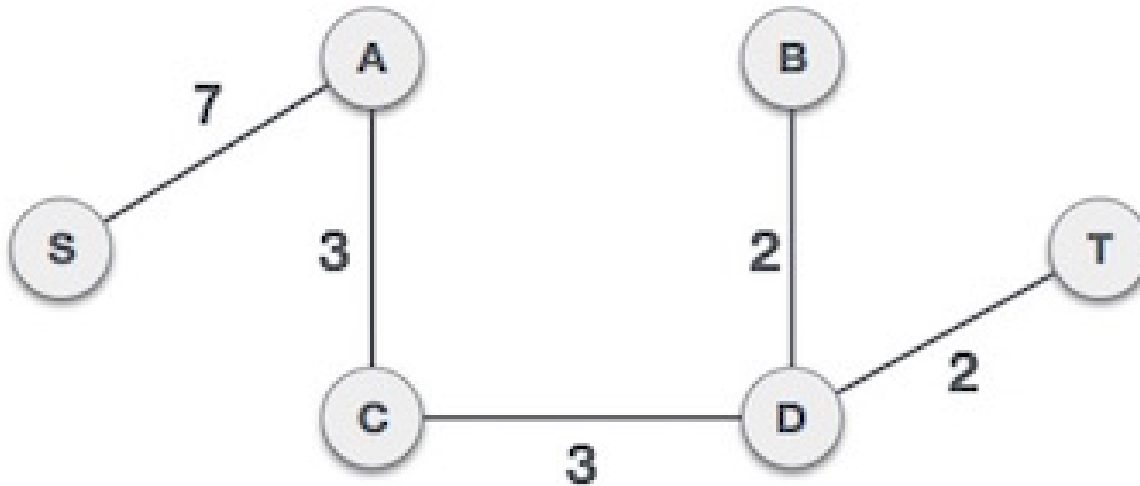
## ...cont'd

- ✓ After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



## ...cont'd

- ✓ After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B.
- ✓ Thus, we can add either one. But the next step will again yield edge 2 as the least cost.
- ✓ Hence, we are showing a spanning tree with both edges included.





Algorithm PRIMS (E, Cost, n)

{/\* E is the set of edges in G, Cost is the adjacency cost matrix, n are the number of vertices \*/

T = {0}; /\* Start with vertex 0 and no edges \*/

while (T contains less than n-1 edges)

{

  select (u,v) from E such that cost [u,v] is minimum and  $u \in T$  and  $v \notin T$

    If (u,v) is found then

      Add v to T

    else

      break;

  }

  if (T contains fewer than n-1 edges print - No spanning tree)

}

Prim:  $O(n \log n)$  search the least weight edge for every vertices

# Single Source Shortest Pattern

- ✓ In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.
- ✓ Given a directed graph, and a single node called the source.
- ✓ For each of the remaining nodes, find a shortest path connected from the source .

# Dijkstra's Algorithm

- ✓ Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph  $G=(V, E)$  for the case in which all edge weights are non-negative.
- ✓ We assume that  $w(u, v) \geq 0$  for each edge  $(u, v) \in E$ .
- ✓ It finds the shortest paths from some initial vertex, say  $v_s$  to all the other vertices one-by-one.
- ✓ The essential feature of Dijkstra's algorithm is the order in which the paths are determined:
- ✓ The paths are discovered in the order of their weighted lengths, starting with the shortest, proceeding to the longest.



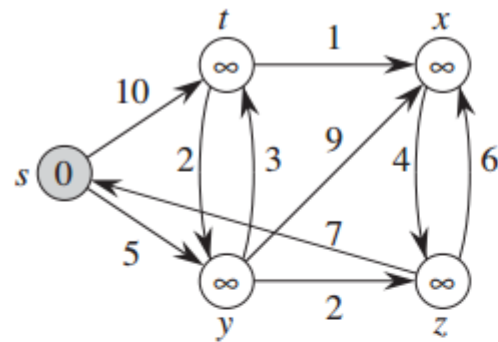
## ...cont'd

- ✓ Dijkstra's algorithm maintains a set  $S$  of vertices whose final shortest-path weights from the source  $s$  have already been determined.
- ✓ The algorithm repeatedly selects the vertex  $u \in V - S$  with the minimum shortest-path estimate, adds  $u$  to  $S$ , and relaxes all edges leaving  $u$ .
- ✓ In the following implementation, we use a min-priority queue  $Q$  of vertices, keyed by their  $d$  values.

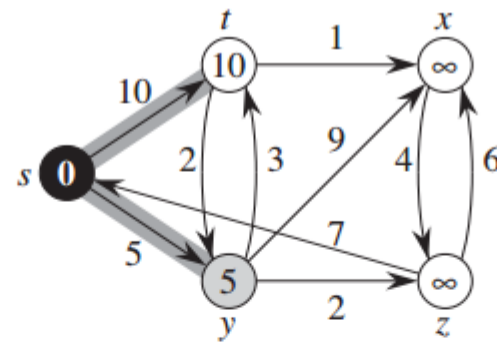
```
DIJKSTRA( $G, w, s$ )  
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )  
2   $S = \emptyset$   
3   $Q = G.V$   
4  while  $Q \neq \emptyset$   
5       $u = \text{EXTRACT-MIN}(Q)$   
6       $S = S \cup \{u\}$   
7      for each vertex  $v \in G.Adj[u]$   
8          RELAX( $u, v, w$ )
```

# ...cont'd

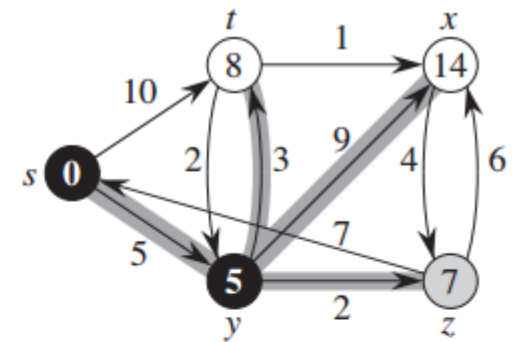
- ✓ The following figures are the execution of Dijkstra's algorithm.



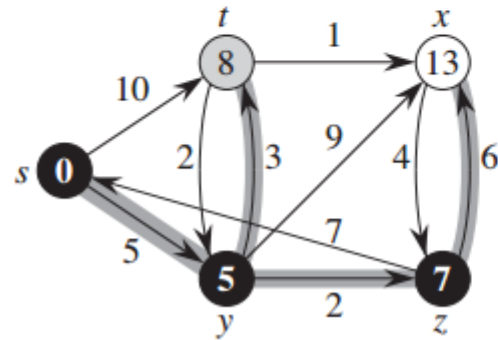
(a)



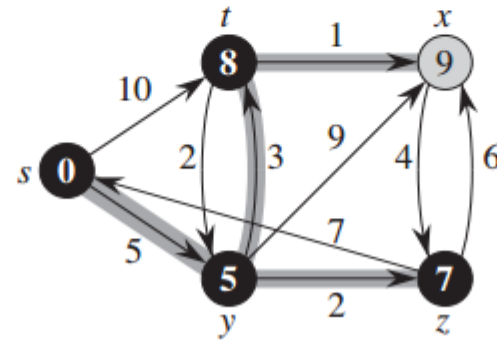
(b)



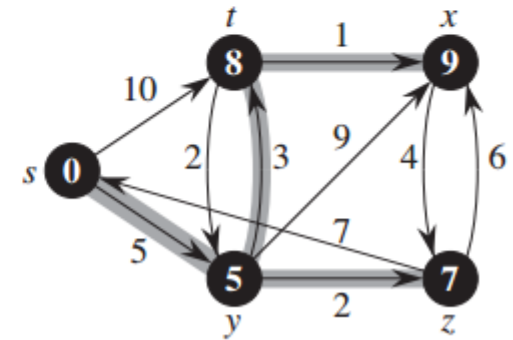
(c)



(d)



(e)



(f)

# Review Questions

1. Explain greedy method with it's advantage.
2. How job sequencing with deadlines works
3. Describe minimum spanning tree with its real world applications.
4. Compare Kruskal's and prim's algorithms.
5. How do you solve the single-source, shortest-path problem on an weighted directed graph. Explain with examples.



# End of Ch.3

---

Questions, Ambiguities, Doubts, ... ???