Greedy Design Techniques

Contents

- The basic paradigm
- The greedy control abstraction
- Elements of greedy strategy
- Characteristics
  - Some optimization problems
- Applications
  - Greedy technique to various problems

Introduction

- The movie *Wall Street*, Michael Douglas quotes
  - Greed is good…Greed is right…Greed works !!
- We want to verify whether is it really so ?
- Take a number of computational problems
  - investigate the pros & cons of short-sighted greed
- How to we define a greedy algorithm ?

Basic paradigm

- Two important components
  - builds up the small solution in small steps
  - choose a decision myopically
  - to optimize some underlying criterion
- choose what is best for the moment
- typically works in stages
  - a decision made in one stage can’t be changed later
  - the choice must lead to the feasible solution
    - expected to be an optimal solution

Basic paradigm (contd)

- When a greedy algorithm succeeds
  - it typically implies something interesting about the structure of a problem
  - true even if the algorithm produces suboptimal solutions
- To prove greedy algorithms work
  - stays ahead
    - either establish that the algorithm succeeds at each step
  - exchange argument
    - proceed with solution and gradually transform it to optimal solution

Terminologies

- Greedy criterion
- optimal solution
- feasible solution
- suboptimal solution
- constraints
- optimization problems
- heuristics
  - bounded performance
    - approximation algorithms

The Thirsty Baby problem

- An intelligent baby wants to quench her thirst....!!!
- Input: n, t, s_i, a_i; $1 \le i \le n$; n integer
- Output: Real nos x_i; $1 \le i \le n$
  such that $\sum_{i=1}^{n} s_i x_i$ is maximum optimization func
- Constraints: $\sum_{i=1}^{n} x_i = t$ is true and $0 \le x_i \le a_i$; $1 \le i \le n$
- What is meant by irrevocability in this case ?

The control abstration

```
Algorithm Greedy(Type a[], int n)
solution=EMPTY;
i=1;
for i=1 to n
   do
       Type x = select(a);
       if feasible(solution, x)
             solution=solution U x;
   done
return solution
```

Greedy algorithms

- Optimization problems
  - solved through a sequence of choices that are:
    - feasible
    - locally optimal
  - irrevocable
- Not all optimization problems can be approached in this manner!

Applications

- Optimal solutions
  - simple scheduling problems
  - change making
  - Minimum Spanning Tree (MST)
  - Single-source shortest paths
  - Huffman codes
- Approximations
  - Traveling Salesman Problem (TSP)
  - Knapsack problem
  - other combinatorial optimization problems

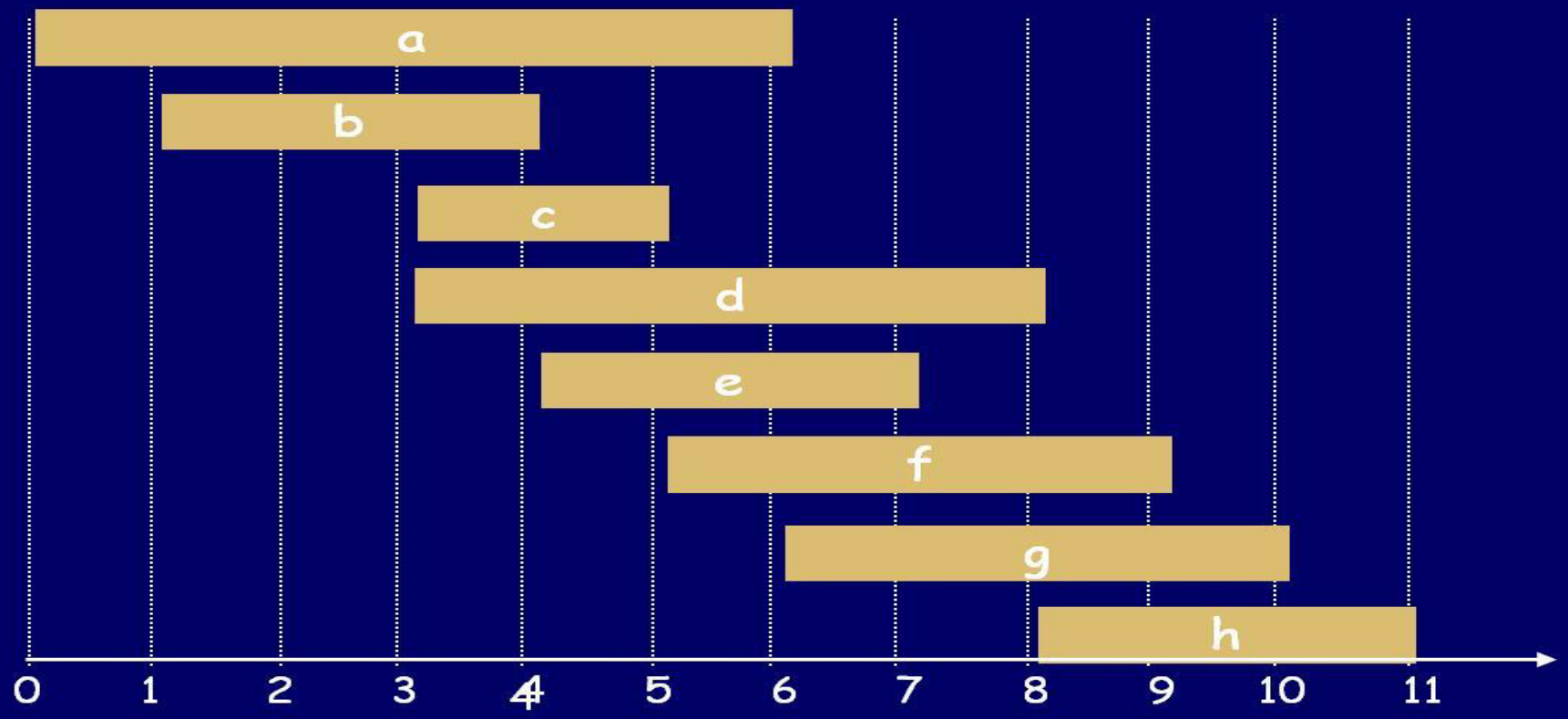
Simple Activity Selection

- Simple Machine Scheduling
- Interval Scheduling
  - selecting maximal set of mutually compatible activities
    - simple & elegant method
  - mutually compatible activities
    - if each activity `i` occurs during the half open intveral `[si,fi)`
    - When do `[si,fi)` and `[sj,fj)` not overlap ?
    - `i<j or j<i`

Problem definition

- Interval scheduling.
  - Job j starts at s_j and finishes at f_j.
  - Two jobs are compatible if they don't overlap.
  - Goal: find maximum subset of mutually compatible jobs.

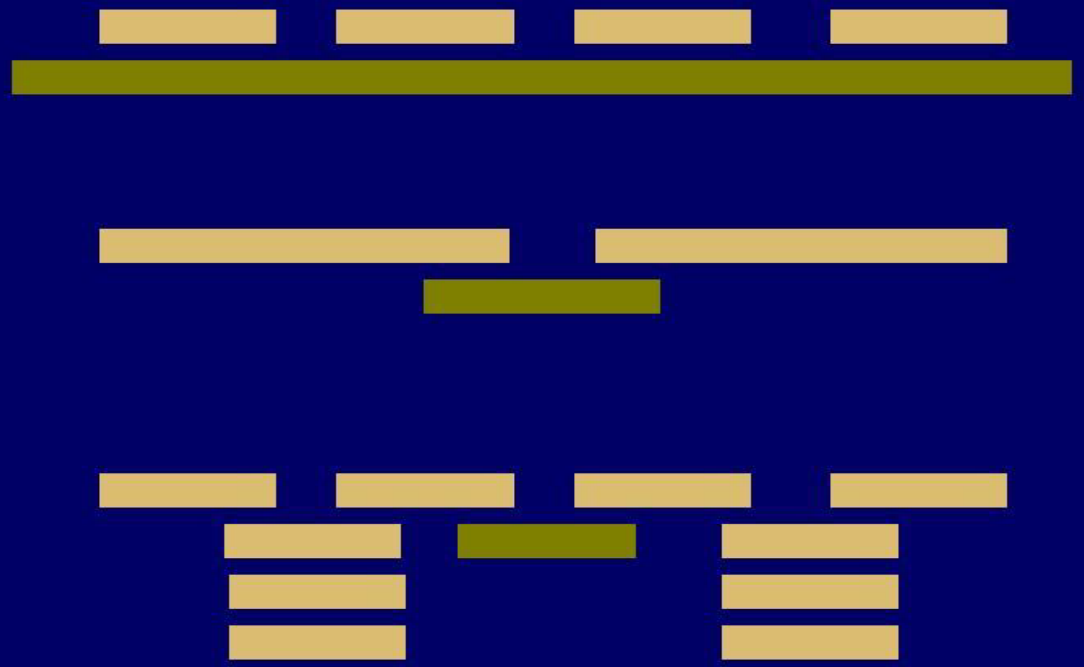
Illustrating



Variations

- Greedy template
  - Consider jobs in some order.
  - Take each job provided it's compatible with the ones already taken.
- [Earliest start time]
  - Consider jobs in ascending order of start time s_j.
- [Earliest finish time]
  - Consider jobs in ascending order of finish time f_j.
- [Shortest interval]
  - Consider jobs in ascending order of interval length $f_j - s_j$.
- [Fewest conflicts]
  - For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

Failure cases



breaks earliest start time

breaks shortest interval

breaks fewest conflicts

Solution approach

- Greedy algorithm
  - Consider jobs in increasing order of finish time.
  - Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that f1 ≤ f2 ≤
... ≤ fn.

      jobs selected
A ← ϕ
for j = 1 to n {
    if (job j compatible with A)
        A ← A ∪ {j}
}
return A
```

- What is the time taken by the algorithm to execute?
  - O(n log n)
    - Remember job j* that was added last to A.
    - Job j is compatible with A if $s_j \geq f_{j*}$.

Machine Scheduling Problem

Minimize average completion time

Second variation

- Jobs $j_1, j_2, j_3 \ldots j_n$ with running times $t_1, t_2, t_3, \ldots t_n$
  - to be scheduled on a single processor
  - such as to minimize the avg completion time

j_1	j_2	j_3	j_4
15	8	3	10

Schedule 1:

j_1	j_2	j_3	j_4
15	8	3	10

- What is the avg completion time?
- Is it optimal assignment?

Second variation (contd)

- Jobs $j_1, j_2, j_3 \ldots j_n$ with running times $t_1, t_2, t_3, \ldots t_n$
  - to be scheduled on a single processor
  - such as to minimize the avg completion time

j_1	j_2	j_3	j_4
15	8	3	10

Schedule 2:

j_3	j_2	j_4	j_1
3	8	10	15

- What is the average completion time?
- Is it optimal assignment?

Second variation (contd)

- Prove that the shortest job first assignment for
  - Jobs $j_1, j_2, j_3 \ldots j_n$ with running times $t_1, t_2, t_3, \ldots t_n$
  - to be scheduled on a single processor
  - such as to minimize the avg completion time

    is an optimal assignment.

- How to implement the above algorithm ?
- What is the time complexity?

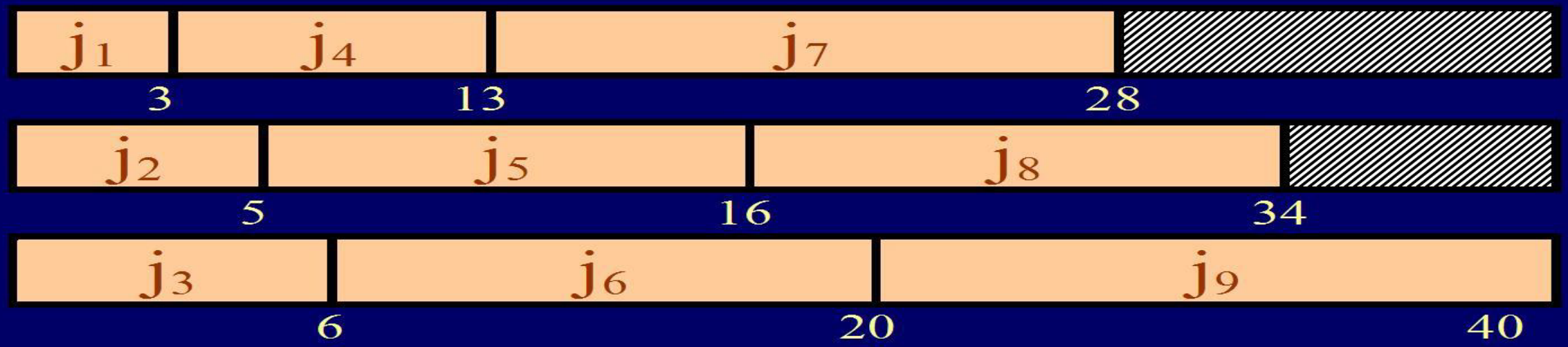
Third variation

- Jobs $j_1, j_2, j_3 \ldots j_n$ with running times $t_1, t_2, t_3, \ldots t_n$
  - to be scheduled on multiprocessors
  - such as to minimize the average completion time

j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9
3	5	6	10	11	14	15	18	20

- A typical schedule and time completion of each job

A typical schedule



M_1	M_2	M_3	M_1	M_2	M_3	M_1	M_2	M_3
j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9
3	5	6	10	11	14	15	18	20
3	5	6	13	16	20	28	34	40

Fourth Variation

- Processes
  - $j_1, j_2, j_3 \ldots j_m$ with running times $t_1, t_2, t_3, \ldots t_o$
  - to be scheduled on specified m no of machines $m_1, m_2, m_3, \ldots\ldots m_m$ such that
    - no machine processes more than one process at a time
    - no process is executed by more than one machine
    - there is non-preemptive scheduling
    - the final completion time is minimized.

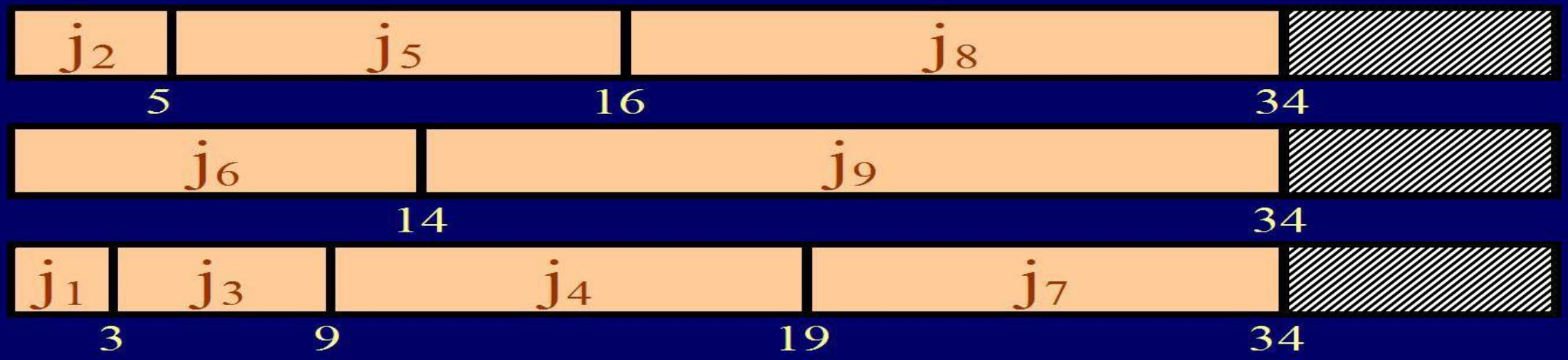
Fourth Variation – Illustration

- Jobs $j_1, j_2, j_3 \ldots j_n$ with running times $t_1, t_2, t_3, \ldots t_n$
  - to be scheduled on multiprocessors
  - such as to minimize the final completion time

j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9
3	5	6	10	11	14	15	18	20

- A typical schedule and time completion of each job

Fourth Variation – Typical Schedule



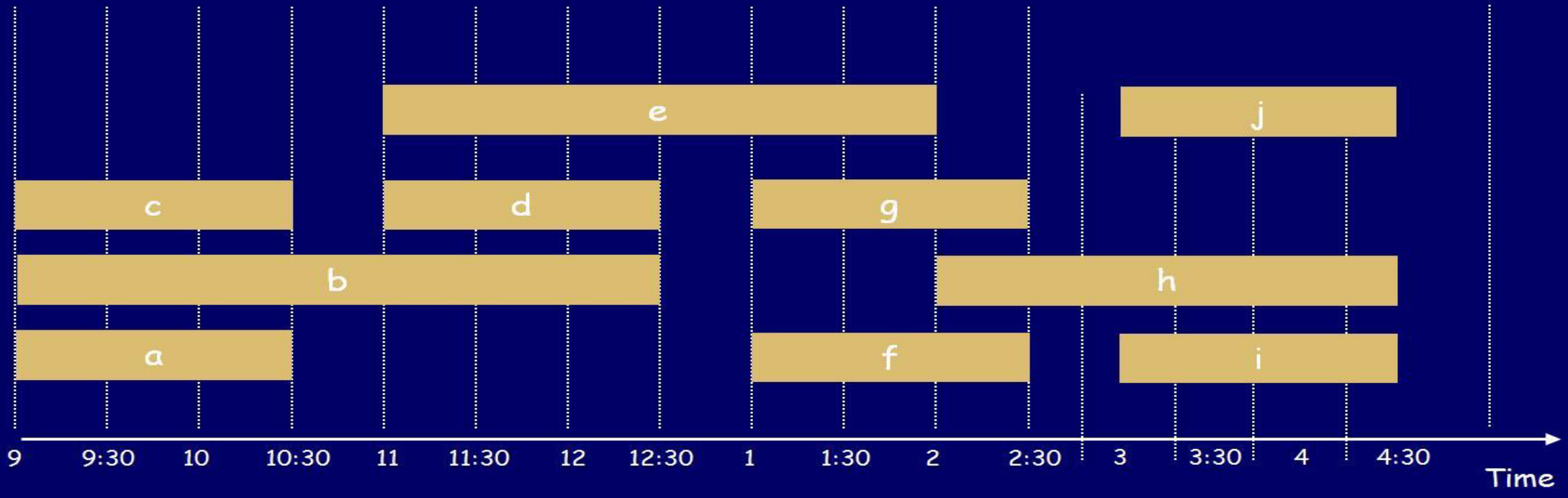
M_3	M_1	M_3	M_3	M_1	M_2	M_3	M_1	M_2
j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9
3	5	6	10	11	14	15	18	20

Summary of variations

- Before we proceed further
  - First…. Activity Selection problem/Interval Scheduling Problem……
    - maximize the compatible number of activities/intervals selected
  - Second…Machine Scheduling Problem…
    - Multiple jobs, Single machine……
    - minimize the average completion time
  - Third…Machine Scheduling Problem…
    - Multiple jobs, Multiple machines……
    - minimize the average completion time
  - Fourth…Machine Scheduling Problem…
    - Multiple jobs, Multiple machines……
    - minimize the final completion time
  - Fifth…Interval Partitioning/Interval Coloring Problem…
    - Multiple partitions (jobs), Multiple resources (machines)……
    - for compatible partitions, minimize the number of resources (machines) used
  - Sixth…Deadline Scheduling..
    - Multiple jobs, Multiple machines……
    - scheduling to meet the deadline

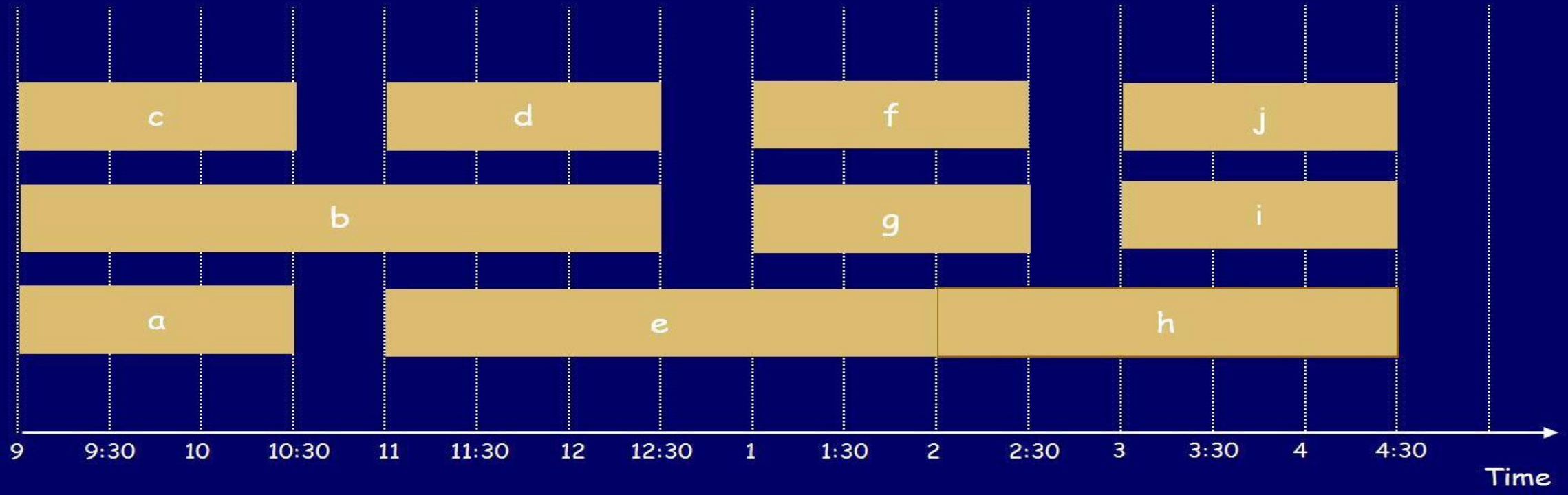
Interval Partitioning

- Lecture j starts at s_j and finishes at f_j
  - find minimum number of classrooms to schedule all lectures
  - so that no two occur at the same time in the same room.
- This schedule uses 4 classrooms to schedule 10 lectures



Interval Partitioning

- Lecture j starts at s_j and finishes at f_j.
- This schedule uses only 3.



Interval Partitioning

- One solution strategy
  - Arrange the lectures in their increasing order of start times
  - Keep track of availability times of classrooms.
    - i.e. availability time of a classroom M_1,
      - assigned lecture l_1, completing at time t_1 – has availability time t_1 when scheduling the next lecture l_2.

A Typical Schedule

task	start_ time	finish_ time	time required	availability time now on respective classroom	scheduling order
A	0	2	2	2(M1)	1
B	3	7	4	7(M1)	3
C	4	7	3	7(M3)	4
D	9	11	2	11(M3)	7
E	7	10	3	10(M1)	6
F	1	5	4	5(M2)	2
G	6	8	2	8(M2)	5

IP: Lower Bound on Optimal Solution

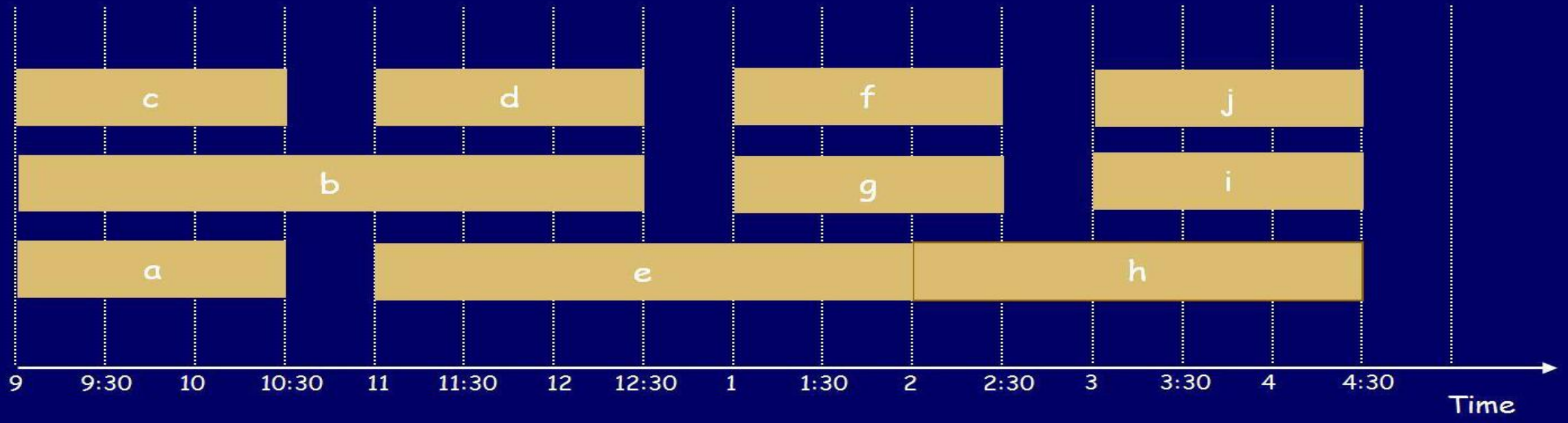
- def:
  - The depth of a set of open intervals is the maximum number of the intervals contained, at a unique time
- Key observation
  - Prove that the number of resources (classrooms) needed is at least the depth.

IP: Lower Bound on Optimal Solution

- e.g.
  - Depth of schedule below = 3 $\Rightarrow$ schedule below is optimal.

a, b, c all contain 9:30

- Does there always exist a schedule equal to depth of intervals?



IP: Greedy Algorithm

- Consider lectures in increasing order of start time
  - assign lecture to any compatible classroom.
- Implementation.
  - For each classroom k, maintain the finish time of the last job added.
  - Keep the classrooms in a priority queue.

IP: Implementation

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ...
  ≤ sn.

d ← 0  ← number of allocated classrooms

for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

- O(n log n).

IP: Greedy Analysis

- Theorem:
  - IF we use the greedy algorithm above, every lecture will be assigned a classroom and no two overlapping lectures will receive the same classroom

The Optimal tape storage problem

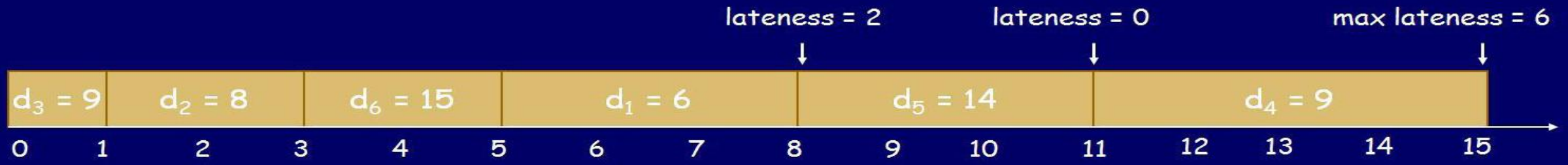
- Given n files of length m_1, m_2, m_3, m_4.... m_n find the best order in which the files can be stored on a sequential storage device.
  - e.g. if n=3 and m_1=5, m_2=10 and m_3=3
  - There can be 3! possible ordering.
  - Which one is the best ?
    - The optimal ordering necessitates only 29 records to be read.

4.2 Scheduling to Minimize Lateness

Scheduling to Minimizing Lateness

- Minimizing lateness problem – EDF scheduling
  - Single resource processes one job at a time.
  - Job j requires t_j units of processing time and is due at time d_j.
  - If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
  - Lateness: $\ell_j = \max \{ 0, \ f_j - d_j \}$.
  - Goal: schedule all jobs to minimize the maximum lateness $L = \max \ell_j$.
- e.g.

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness

- Greedy template. Consider jobs in some order.
  - [Shortest processing time first] Consider jobs in ascending order of processing time t_j.
  - [Earliest deadline first] Consider jobs in ascending order of deadline d_j.
  - [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

Minimizing Lateness

- Greedy template. Consider jobs in some order.
  - [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

	1	2
tj	1	10
dj	100	10

counterexample

  - Can we and if so how can one achieve lateness 0 above?

  - [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

counterexample

  - Can we and if so how can one achieve lateness 0 above?

Minimizing Lateness

Earliest deadline first.

```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ … ≤ d_n

t ← 0

for j = 1 to n
    Assign job j to interval [t, t + t_j]
    s_j ← t, f_j ← t + t_j
    t ← t + t_j

output intervals [s_j, f_j]
```

Greedy Analysis Strategies

- Greedy algorithm stays ahead.
  - Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.
- Exchange argument
  - Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.
- Structural
  - Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

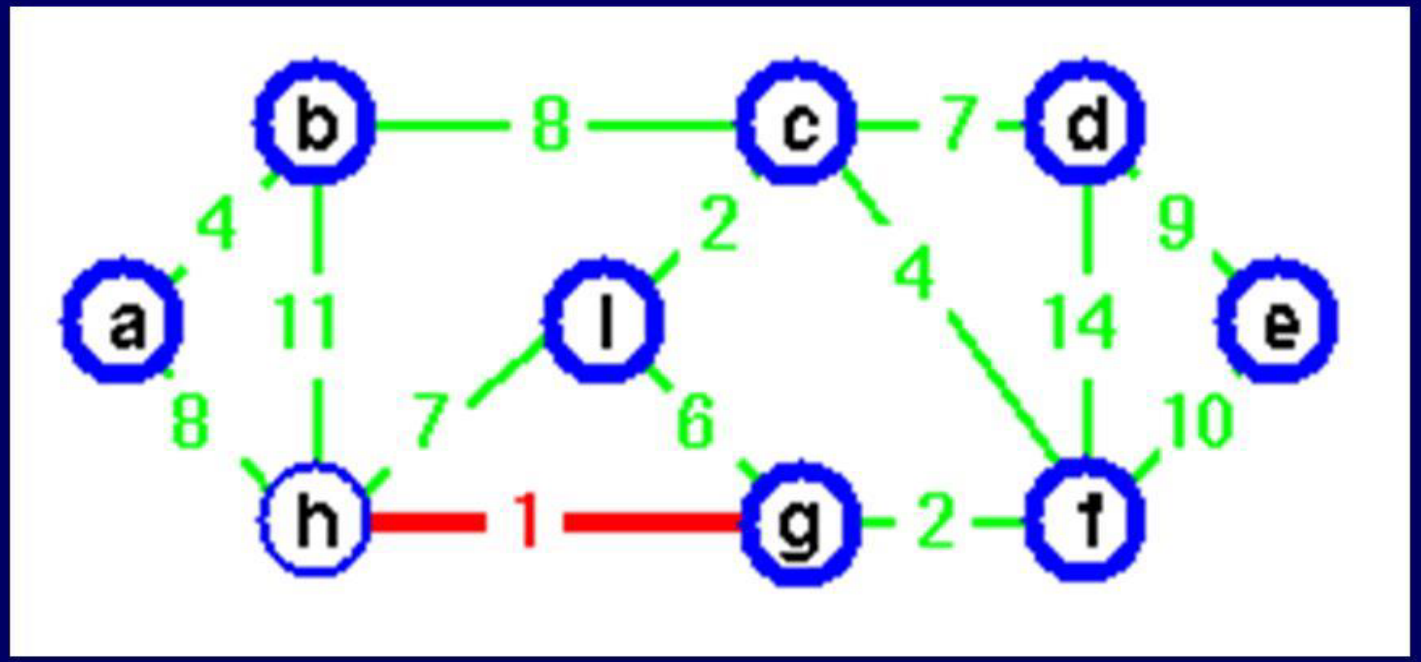
Greedy Algorithm Strategies

- Heuristic Strategies followed
  - solution which is the best for the moment.
- Can we always know whether
  - a greedy algorithm will solve a particular optimization problem or not ?
  - Two vital properties
    - Greedy-choice property
      - a globally optimal solution can be arrived from a locally optimal choice.
      - algorithm proceeds in a top down fashion – reducing the given problem instance into smaller ones
    - Optimal Sub-structure property
      - an optimal solution to a problem contains within it other optimal solutions to smaller subproblems

Minimum Spanning Trees

Graphs - Definitions

- Graph
  - a set of vertices (nodes) and edges connecting them
  - we say G = (V, E) *where*
    - V is a set of vertices: $V = \{ v_i \}$
    - An edge connects two vertices: $e = (v_i , v_j)$
    - E is a set of edges: $E = \{ (v_i , v_j) \}$

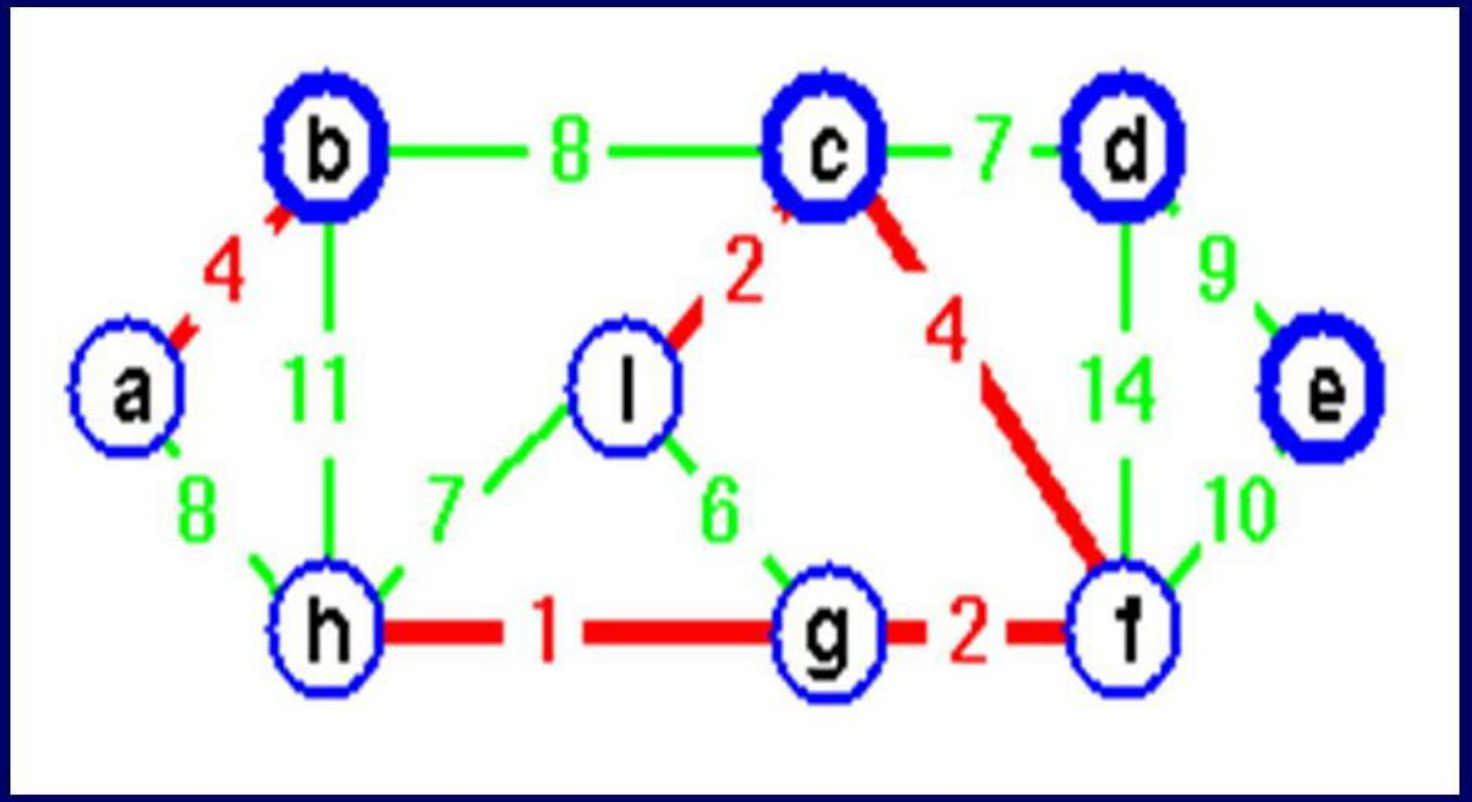


Vertices

Edges

Graphs - Definitions

- Path – p of length k, is a sequence of connected vertices
  - $p = \langle v_0, v_1, \ldots, v_k \rangle$ *where* $(v_i, v_{i+1}) \in E$



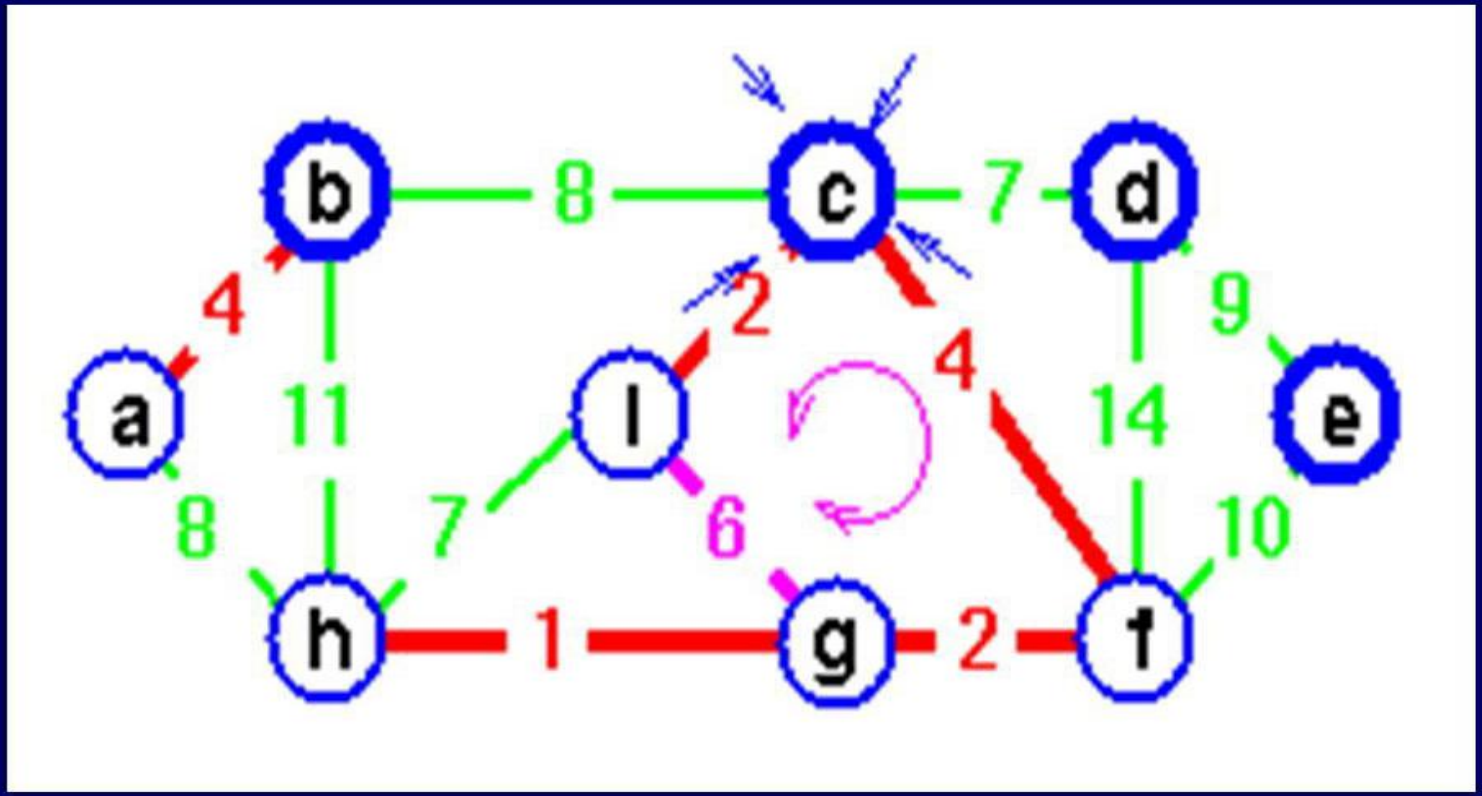
< i, c, f, g, h >
Path of length 5

< a, b >
Path of length 2

Graphs - Definitions

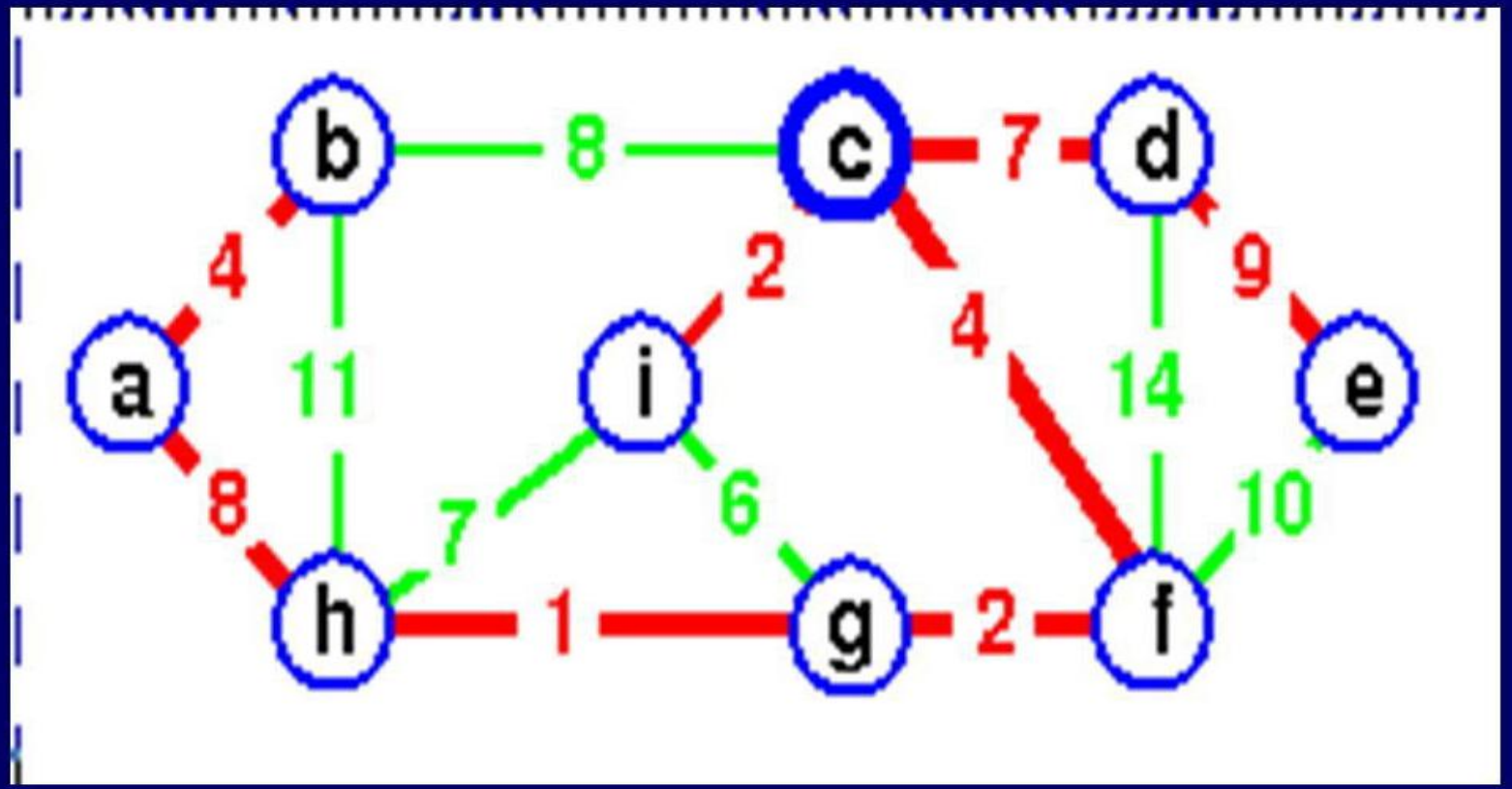
- Cycle
  - A graph contains no cycles if there is *no* path
  - $p = \langle v_0, v_1, \ldots, v_k \rangle$ *such that* $v_0 = v_k$

< i, c, f, g, i >
is a cycle



Graphs - Definitions

- Spanning Tree
  - A spanning tree is a set of $|V|-1$ edges that connect all the vertices of a graph

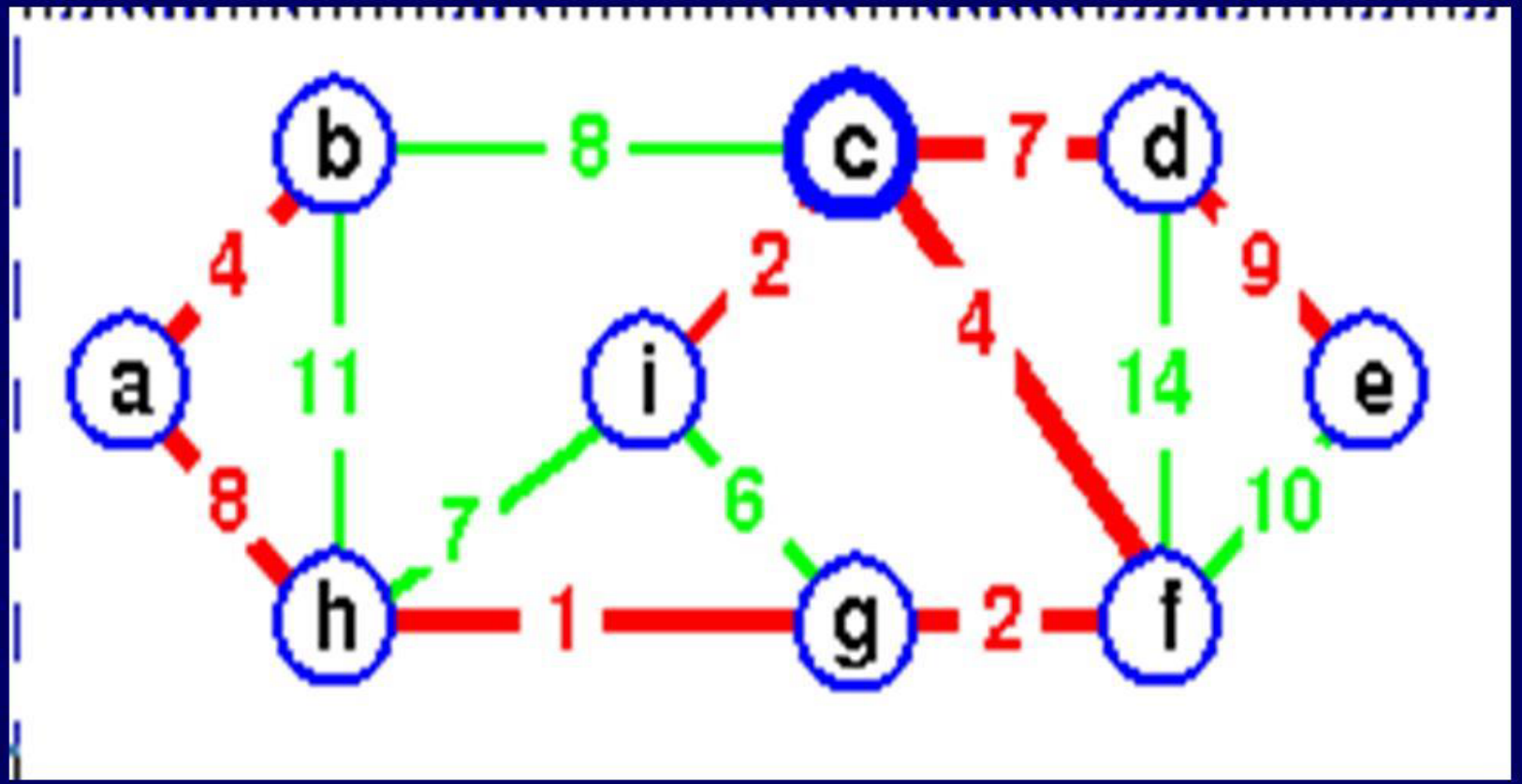


The red path connects all vertices, so it's a spanning tree

Graphs - Minimum Spanning Tree

- Generally there is more than one spanning tree
- Formal definition

Other ST's can be formed ..
- **Replace 2 with 7**
- **Replace 4 with 11**



The red tree is the Min ST

Cut and Light Edge

- A cut **(S, V − S)** of an undirected graph G = (V, E) is a partition of the nodes of the graph V in such a way that
  - an edge (u,v) ∈ E crosses the cut (S, V-S) if one of its endpoints is in S and the other endpoint is in V-S.
  - we say, a cut respects a set A of edges if no edge in A crosses the cut.
- Light edge
  - an edge is said to be a light edge, if while it is crossing a cut, its weight is minimum of any edge crossing the cut.
- There can be more than one light edge crossing a cut in case of ties.

Concept of a safe edge

- Let
  - G = (V, E) be a connected, undirected graph with a real-valued weight function w defined on E.
  - A be a subset of E that is included in some MST for G;
  - (S, V-S) be any cut of G that respects A and
  - (u,v) be a light edge crossing (S, V-S).
- Then, the edge (u,v) is the safe edge for A.

Generic MST

- Loop invariant
  - Prior to each iteration, A is a subset of some MST.
- Add the safe edge
  - At each step, investigate (u,v) that can be added to A without violating the constraint.

```
GENERIC-MST(G, w)
1. A ← Φ
2. while A does not form a spanning tree
3.    do find an edge (u,v) that is safe for A
4.       A ← A U {(u,v)}
5.   return A
```

Graphs - Kruskal's Algorithm

- How to find the safe edge ?!!!
- Calculate the minimum spanning tree
  - Put all the vertices into single node trees by themselves
  - Put all the edges in a priority queue
  - Repeat until we've constructed a spanning tree
    - *Extract the cheapest edge*
    - If it forms a cycle, ignore it,
      else add it to the forest of trees
      (it will join two trees into a larger tree)
  - Return the spanning tree

Graphs - Kruskal's Algorithm

- How to find the safe edge ?!!!
- Calculate the minimum spanning tree
  - Put all the vertices into single node trees by themselves
  - Put all the edges in a priority queue
  - Repeat until we've constructed a spanning tree
    - *Extract the cheapest edge*
    - If it forms a cycle, ignore it,
      else add it to the forest of trees
      (it will join two trees into a larger tree)
  - Return the spanning tree

Note that this algorithm makes no attempt

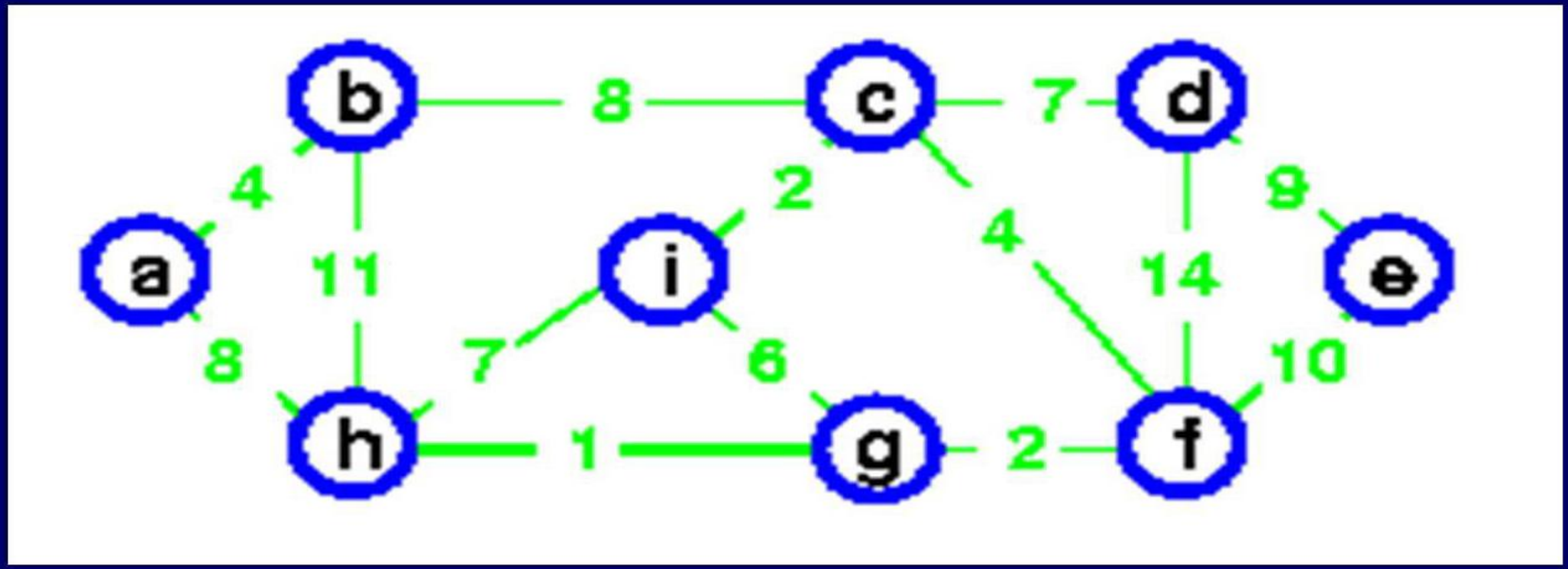
- **to be clever**
- **to make any sophisticated choice of the next edge**
- **it just tries the cheapest one!**

Graphs - Kruskal's Algorithm (contd)

```
Algorithm MinimumSpanningTree(Graph G(V,E),
   Weight_function w)
for each vertex v ∈ V[G]
    Construct a single-vertex forest from G
Sort the edges of E into nondecreasing order by weight
   w ∈ E,
for each edge (u,v)
   do extract the cheapest edge
      add it to A
   while (it does not form a Cycle)
return A;
```

Kruskal's Algorithm in operation

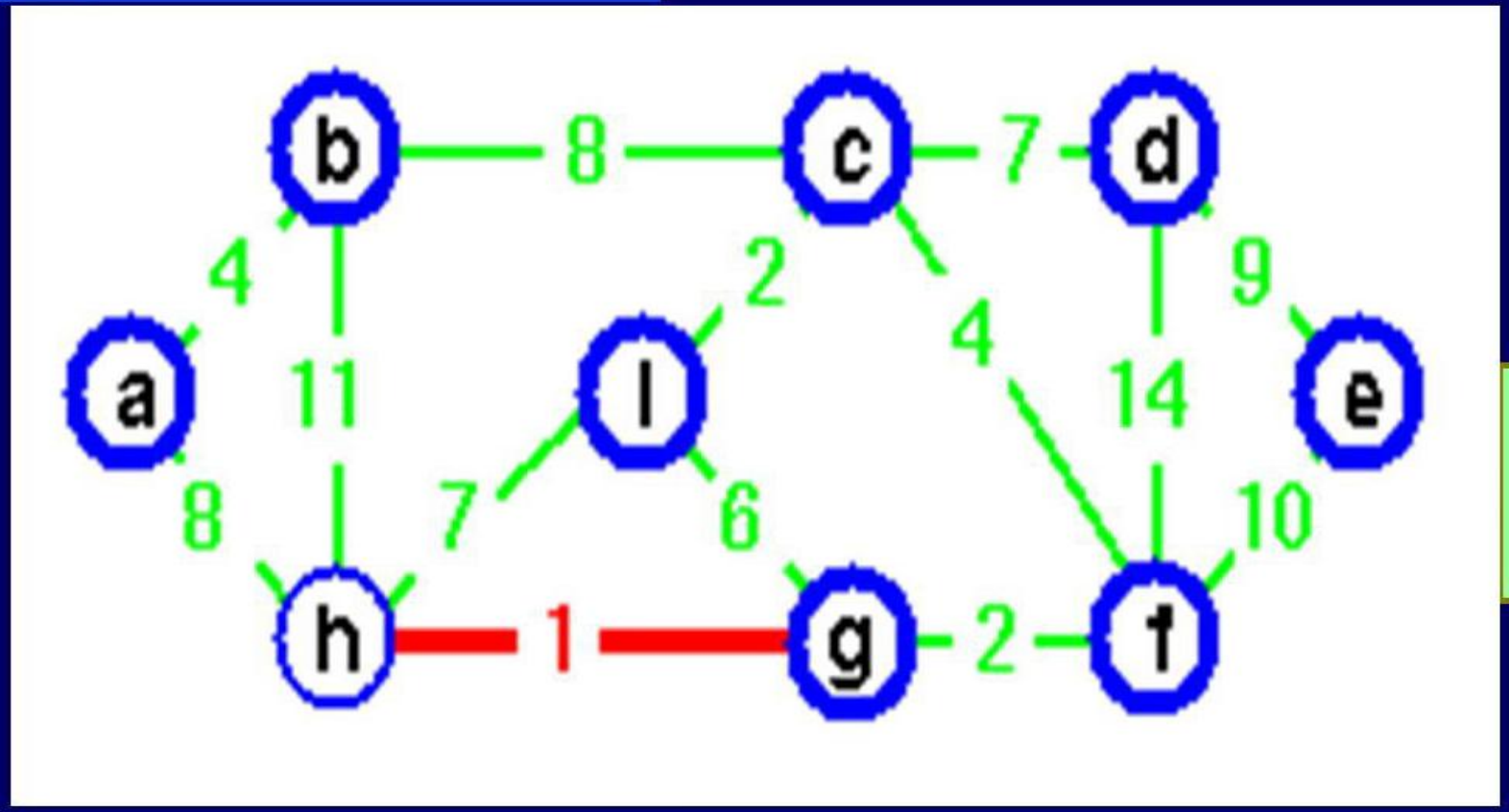
All the vertices are in single element trees



Each vertex is its own representative

Kruskal's Algorithm in operation

All the vertices are in single element trees

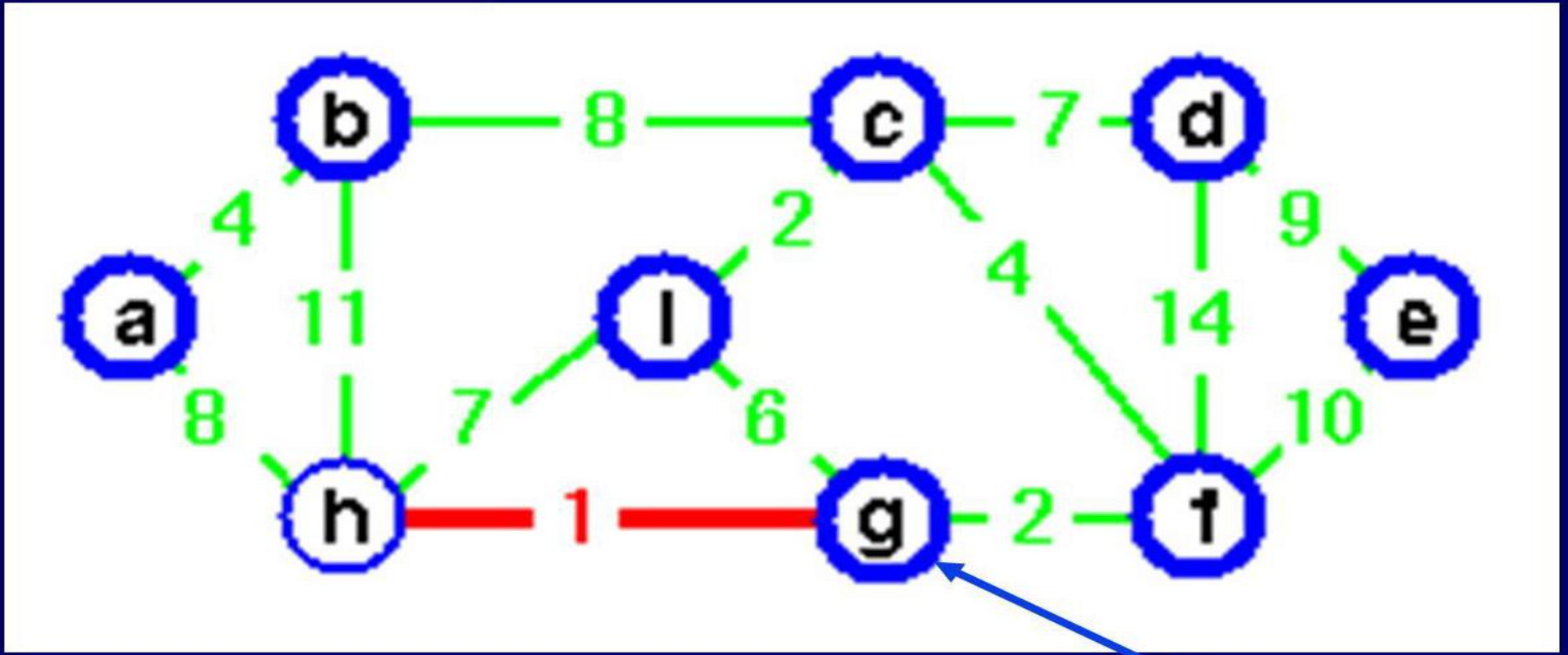


Add it to the forest, joining h and g into a 2-element tree

The cheapest edge is h-g

Kruskal's Algorithm in operation

The cheapest edge is h-g



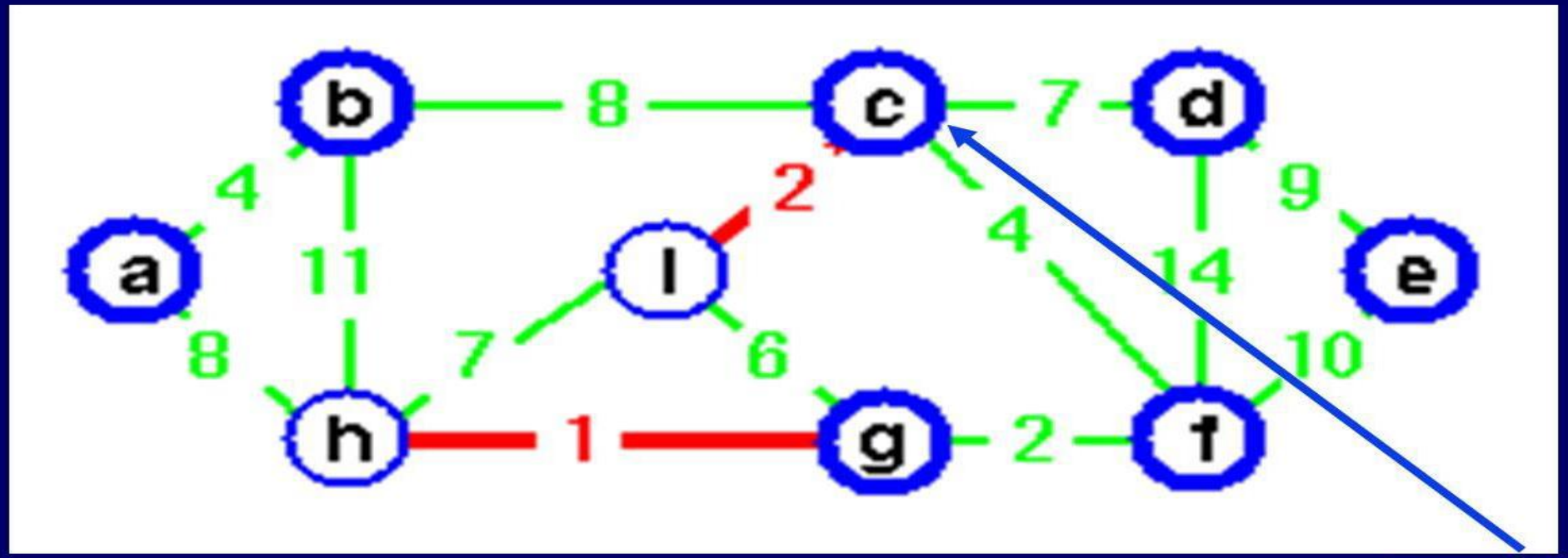
Add it to the forest, joining h and g into a 2-element tree

Choose g as its representative

Kruskal's Algorithm in operation

The next cheapest edge is c-i

Add it to the forest, joining c and i into a 2-element tree



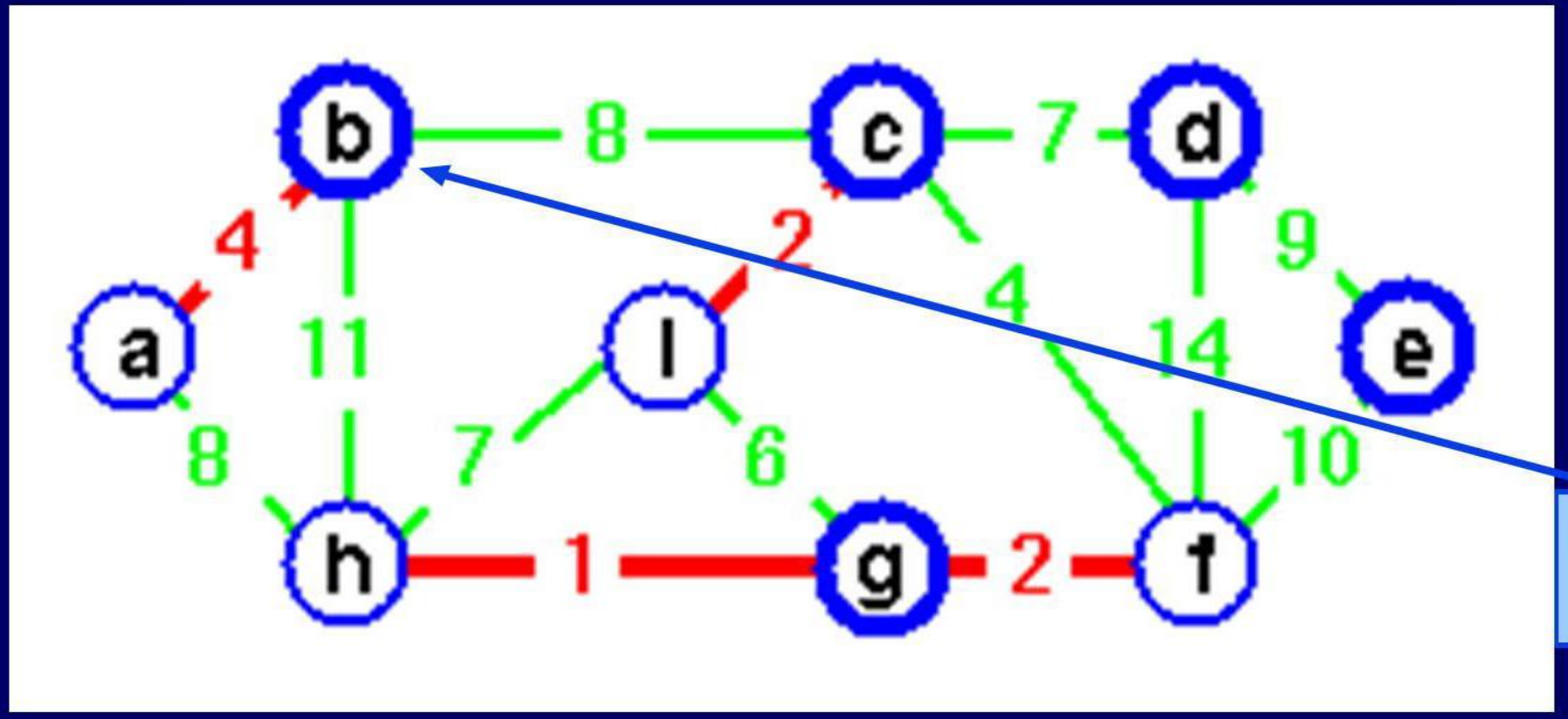
Choose c as its representative

Our forest now has 2 two-element trees and 5 single vertex ones

Kruskal's Algorithm in operation

The next cheapest edge is a-b

Add it to the forest, joining a and b into a 2-element tree

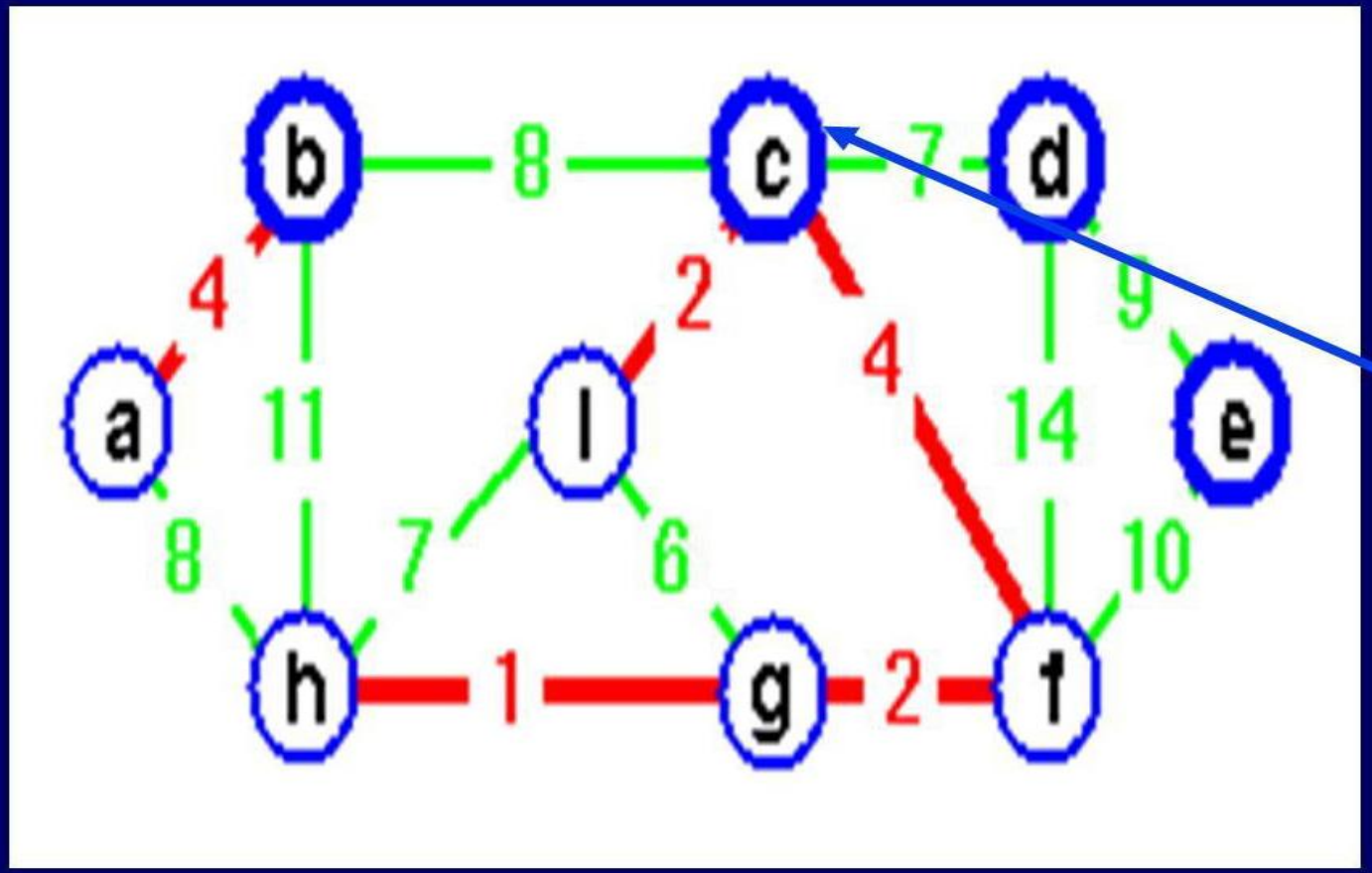


Choose b as its representative

Our forest now has 3 two-element trees and 4 single vertex ones

Kruskal's Algorithm in operation

The next cheapest edge is c-f

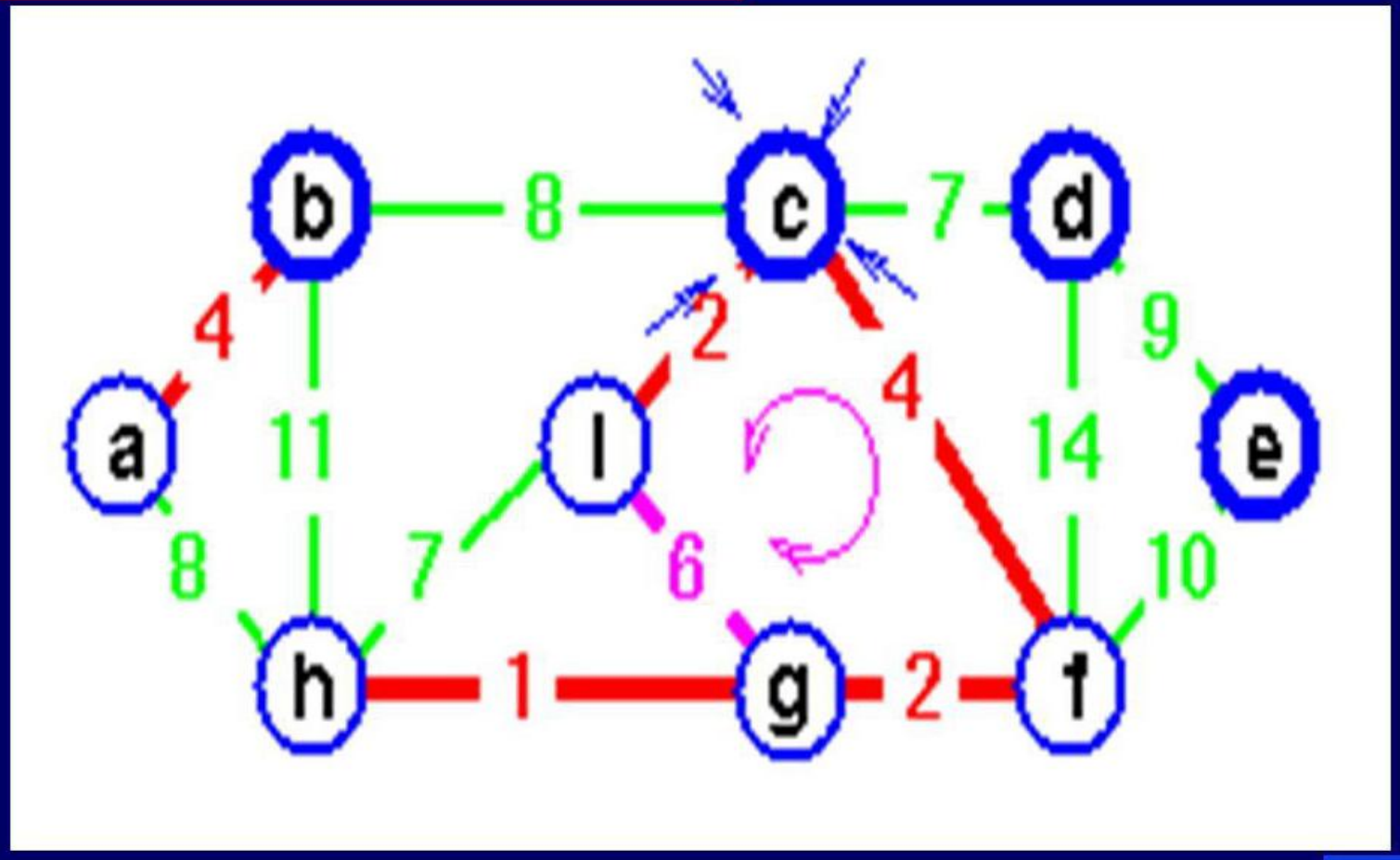


Add it to the forest, *merging* two 2-element trees

Choose the rep of one as its representative

Kruskal's Algorithm in operation

The next cheapest edge is g-i



The rep of g is c

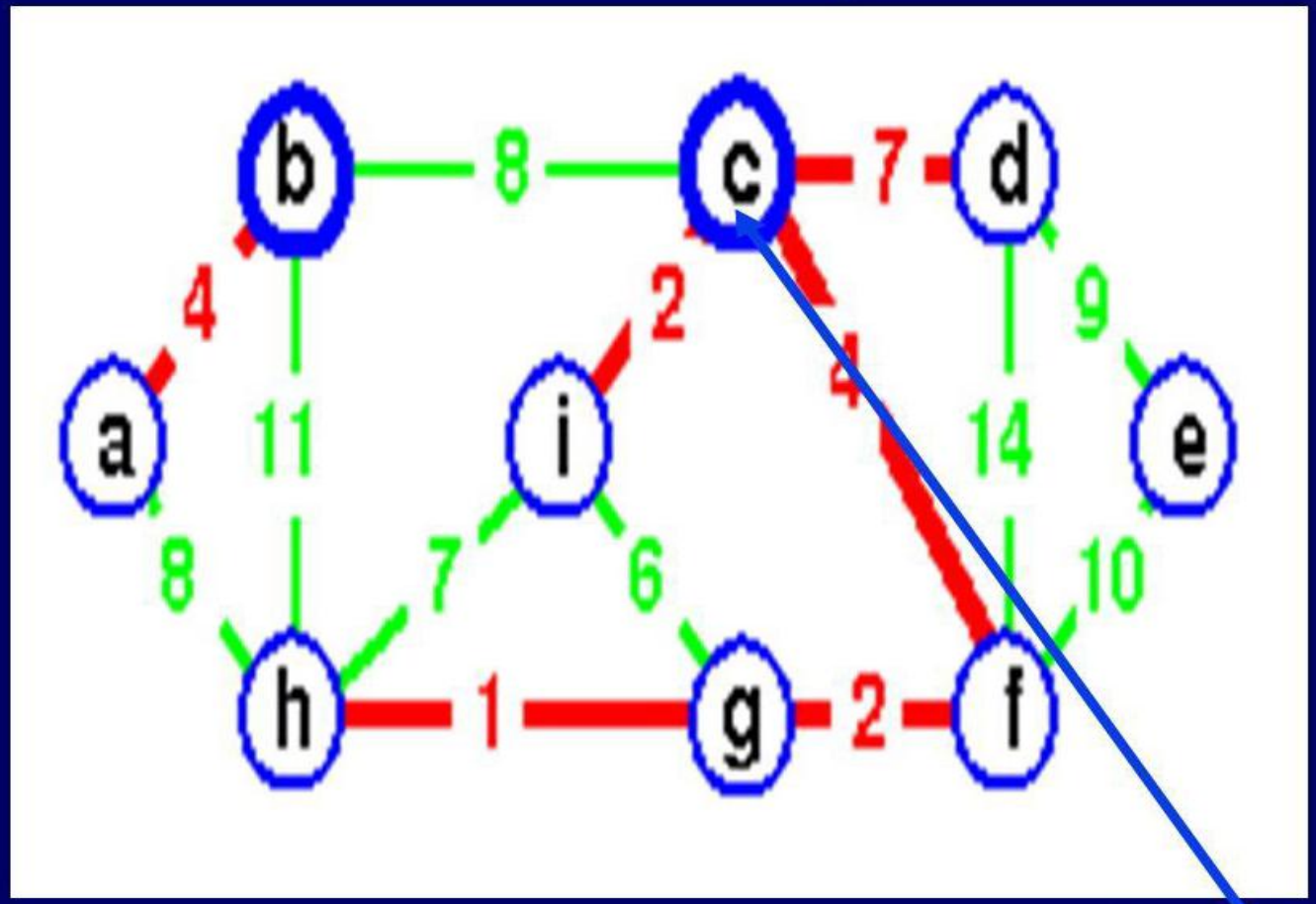
The rep of i is also c

∴ g-i forms a cycle

It's clearly not needed!

Kruskal's Algorithm in operation

The next cheapest edge is c-d



The rep of c is c

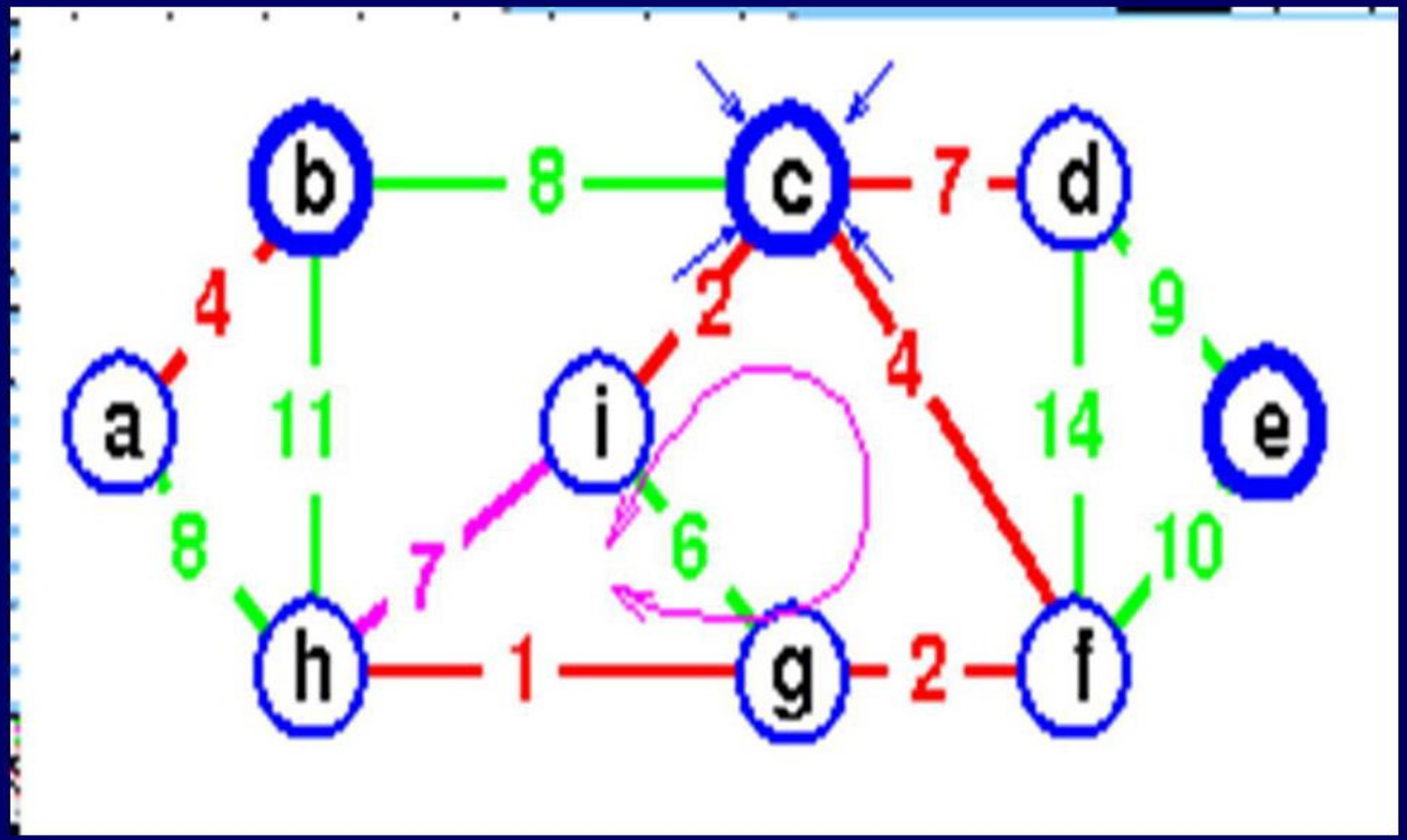
The rep of d is d

∴ c-d joins two trees, so we add it

.. and keep c as the representative

Kruskal's Algorithm in operation

The next cheapest edge is h-i



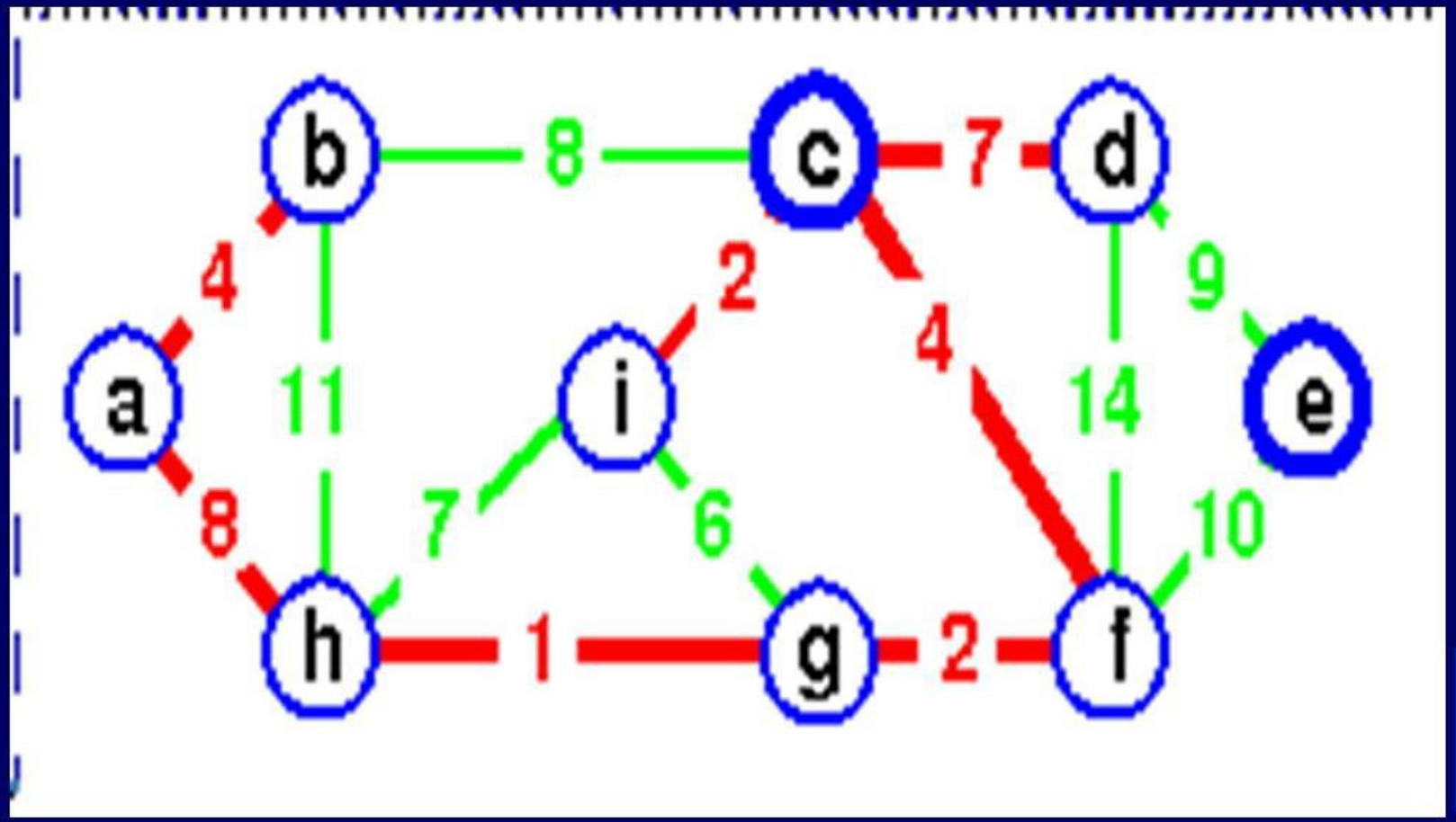
The rep of h is c

The rep of i is c

∴ h-i forms a cycle, so we skip it

Kruskal's Algorithm in operation

The next cheapest edge is a-h



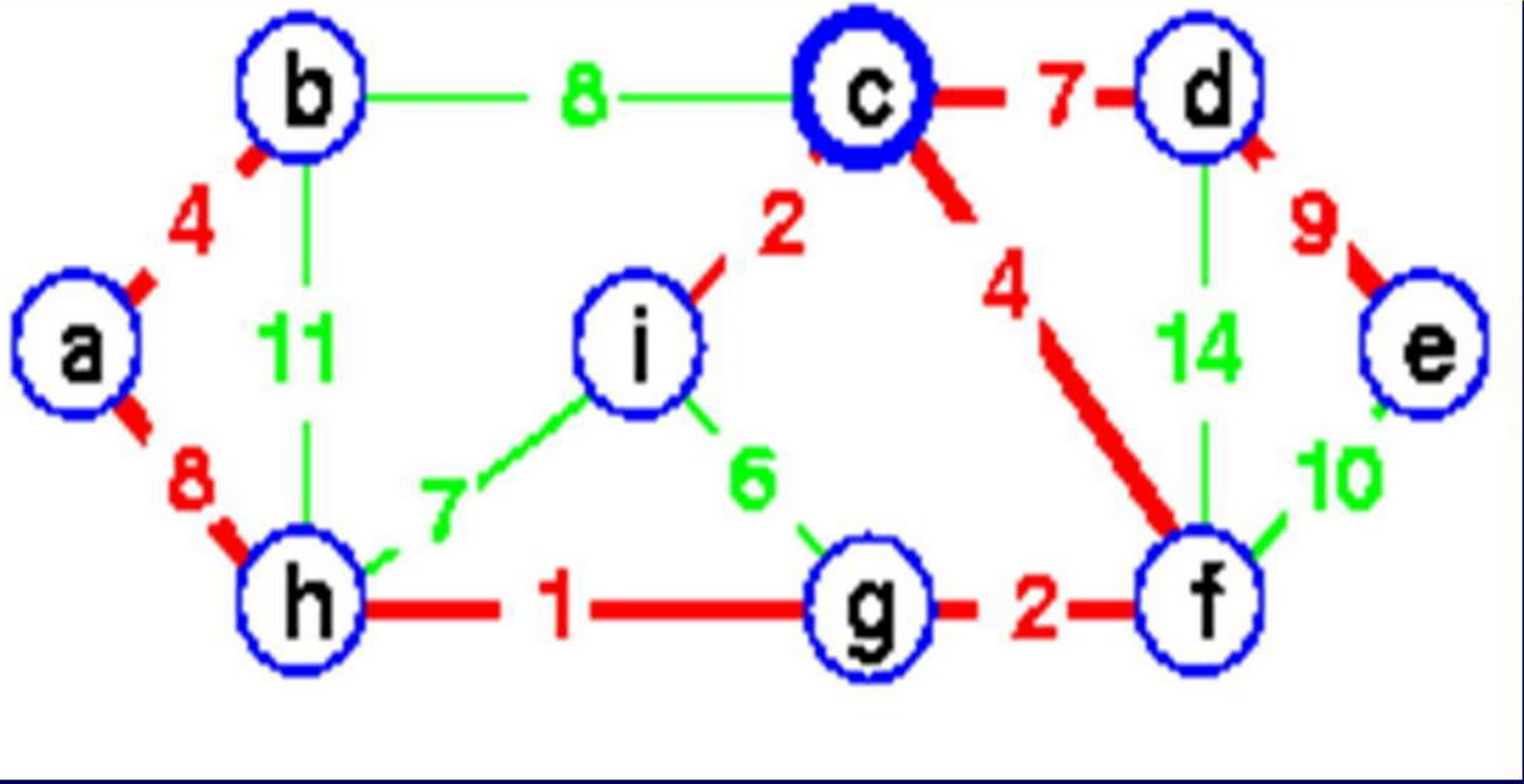
The rep of a is b

The rep of h is c

∴ a-h joins two trees, and we add it

Kruskal's Algorithm in operation

The next cheapest edge is b-c



But b-c forms a cycle

So add d-e instead

... and we now have a spanning tree

Kruskal's Algorithm

```
Algorithm Kruskal(Graph G(V,E),
   Weight_function w)
A←Φ
for each vertex v ∈ V[G]
     do MAKE-SET(v)
Sort the edges of E into nondecreasing order
   by weight w
for each edge (u,v) ∈ E,
    do if FIND-SET(u)≠ FIND-SET(u)
    then A ← A U {(u,v)}
    UNION(u,v)
return A
```

Greedy Algorithms

- *At no stage did we attempt to "look ahead"*
- We simply made the naïve choice
  - Choose the cheapest edge!
- Therefore, MST is an example of a greedy algorithm

MST - Time complexity

- Steps
  - Initialise forest $O(|V|)$
  - Sort edges $O(|E|\log|E|)$
    - Check edge for cycles $O(|V|)$ x
    - Number of edges $O(|V|)$ $O(|V|^2)$
  - Total $O(|V|+|E|\log|E|+|V|^2)$
  - Since $|E| = O(|V|^2)$ $O(|V|^2 \log|V|)$

MST - Time complexity

- Thus we would class MST as $O(n^2 \log n)$ for a graph with n vertices
- This is an *upper bound*, some improvements on this are known ...
  - Prim's Algorithm can be $O(|E| + |V| \log |V|)$ using Fibonacci heaps
  - even better variants are known for restricted cases, such as sparse graphs ($|E| \approx |V|$)

Prim's Algorithm

- Follows the natural greedy approach
  - starting with the source vertex to create the spanning tree,
  - add an edge to the tree that is attached at exactly one end to the tree & has minimum weight among all such edges.
- Prim's algorithm starts from one vertex and
  - grows the rest of the tree an edge at a time.
- As a greedy algorithm, which edge should we pick?

Prim's Algorithm

- In Kruskal,
  - the selection function uses a greedy approach…. chooses an edge that is minimum weighted edge …. then in increasing order
  - does not worry too much about their connection to the previously chosen edges…except cycles.
  - the result is a sort of forest of trees that grows haphazardly and later merge into a tree.
- Here, the MST grows in a natural manner… staring from an arbitrary root.
- At each, a new edge is added to a tree already constructed.

Prim's algorithm

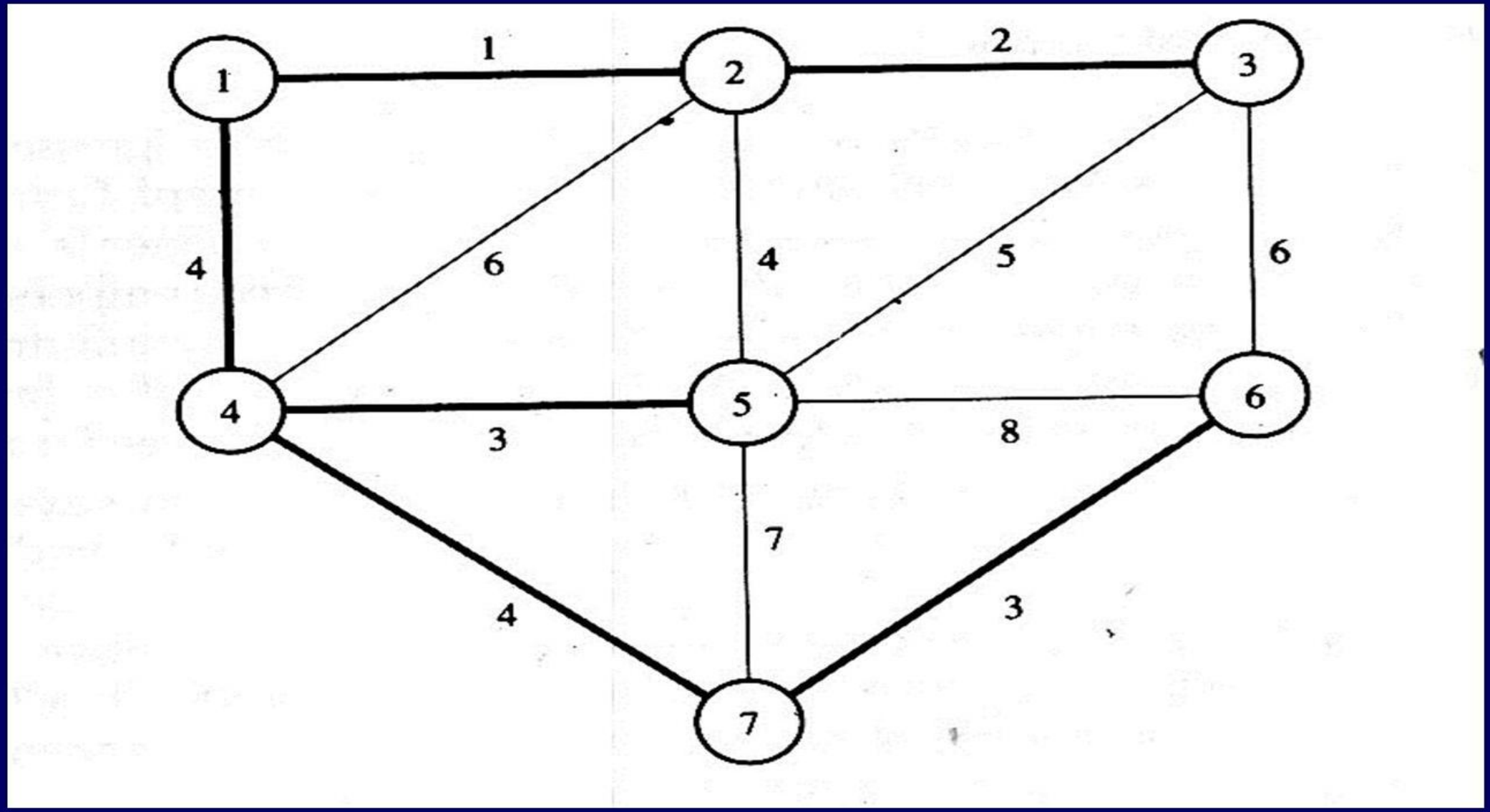
- Consider
  - B - aset of nodes , T – a set of edges
  - initially B contains a single arbitrary node
  - at each step, Prim's algorithm looks for the shortest edge {u,v} such that `u ∈ B and v ∈ N\B.`
  - then it addes `v to B` and `{u, v}` to `T.`
  - in this way, in T at any instant an MST for the nodes in B is formed.

Algorithm

Algorithm Prim(G=(V, E), weight)

```
{initialization}
T ← ∅
B ← {an arbitrary member of V}
while B ≠ N do
     find E = {u, v} of minimum
                        weight such that
     u ∈ B and v ∈ V-B
     T ← T U {e}
     B ← B U {v}
     return T
```

Running Kruskal



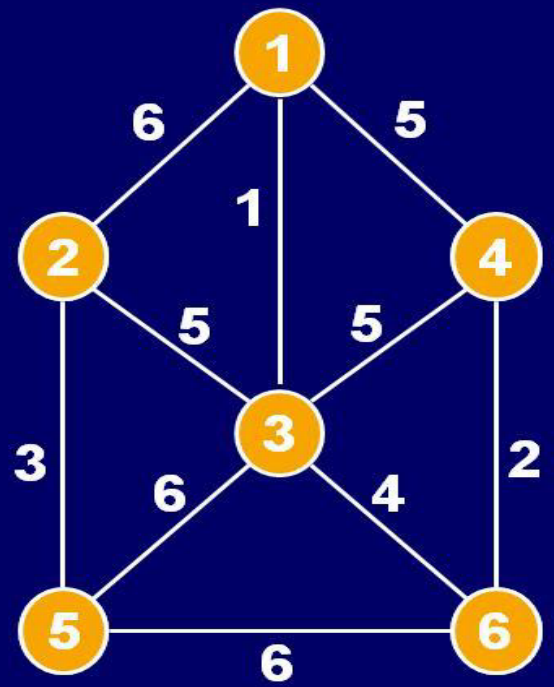
Running Kruskal (contd)

Step	Edge consi	Connected components
Init	-	{1} {2} {3} {4} {5} {6} {7}
1	{1 2}	{1 2} {3} {4} {5} {6} {7}
2	{2 3}	{1 2 3} {4} {5} {6} {7}
3	{4 5}	{1 2 3} {4 5} {6} {7}
4	{6 7}	{1 2 3} {4 5} {6 7}
5	{1 4}	{1 2 3 4 5} {6 7}
6	{2 5}	Rejected
7	{4 7}	{1 2 3 4 5 6 7}

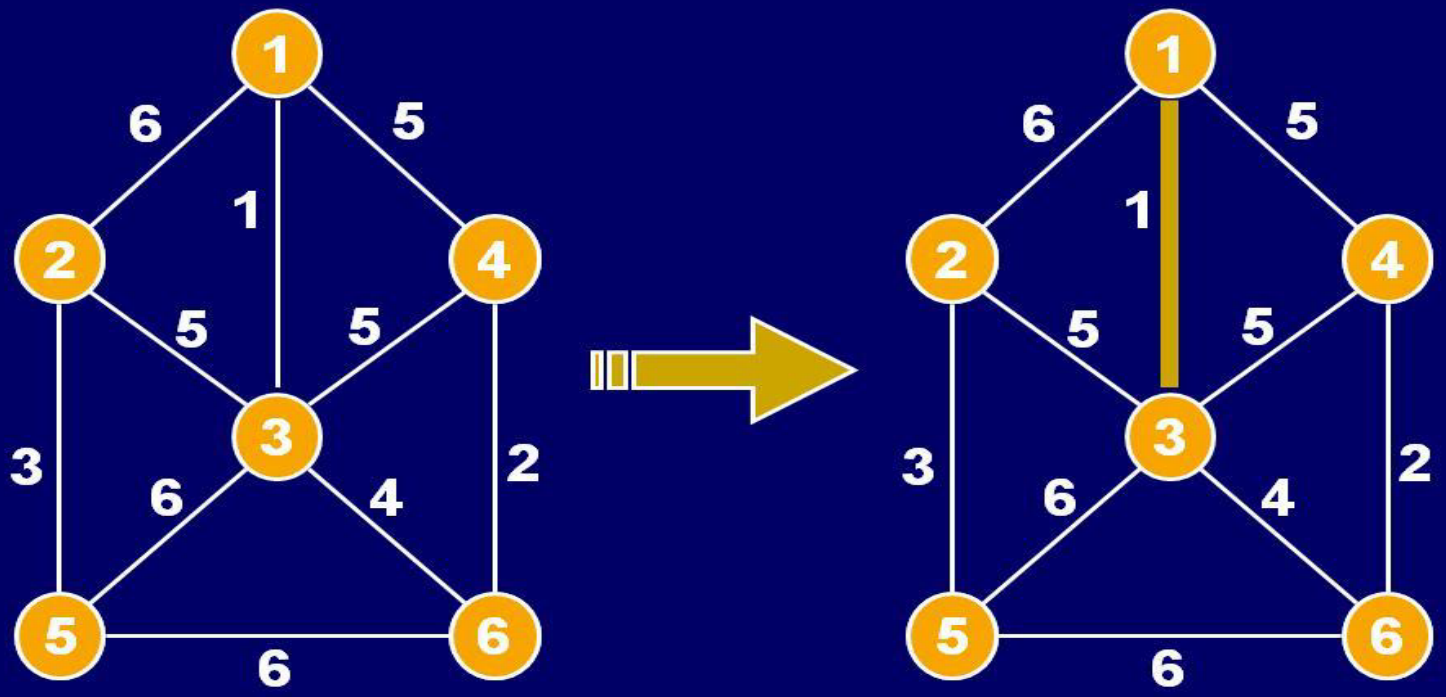
The Algorithm – Prim's Algorithm

- A vertex-based greedy approach.
- Starts at any vertex.
- Finds the least cost subsequent vertices without causing cycles in the tree.
- Perform above steps until all the vertices are connected.

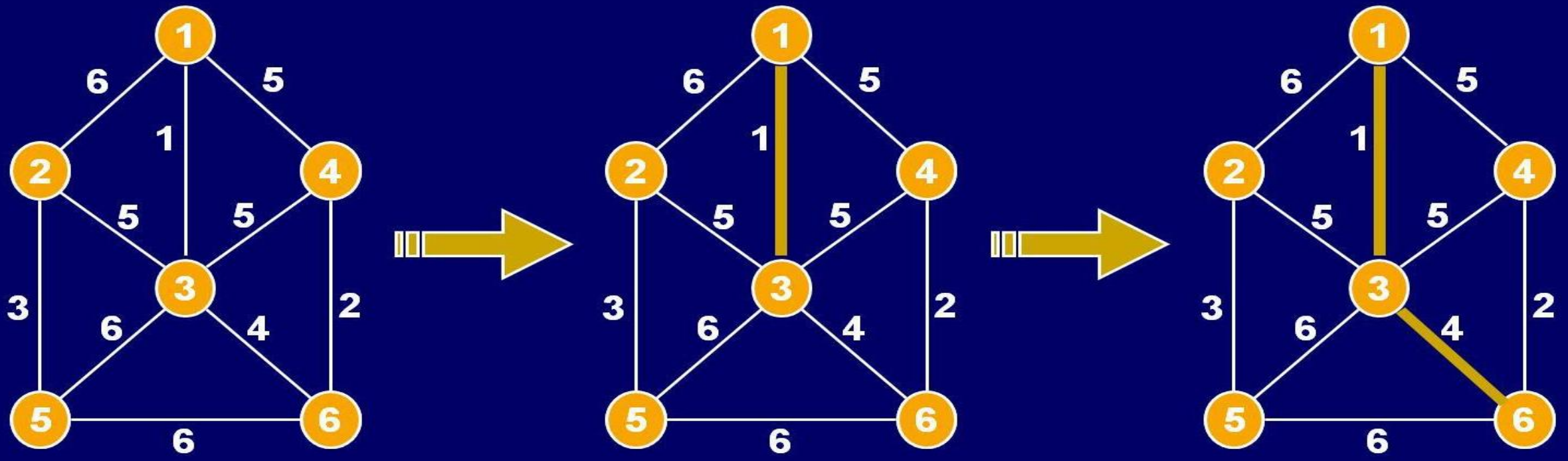
Example – Prim's Algorithm



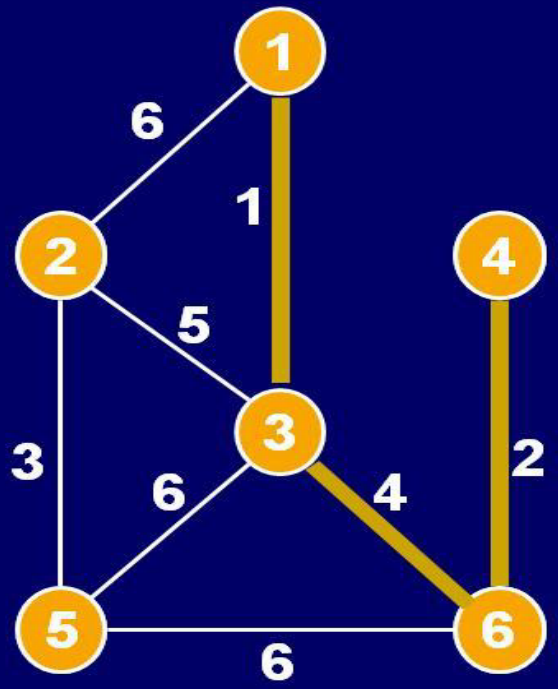
Example – Prim's Algorithm



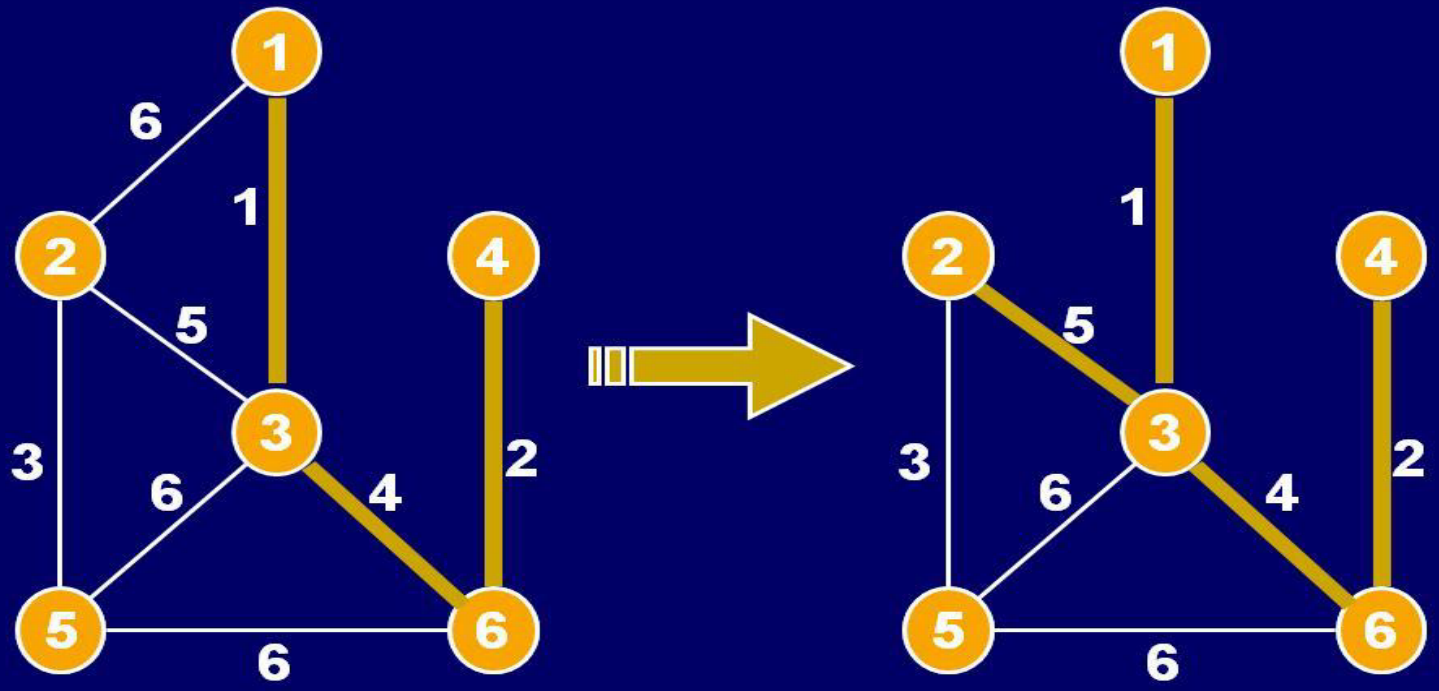
Example – Prim's Algorithm



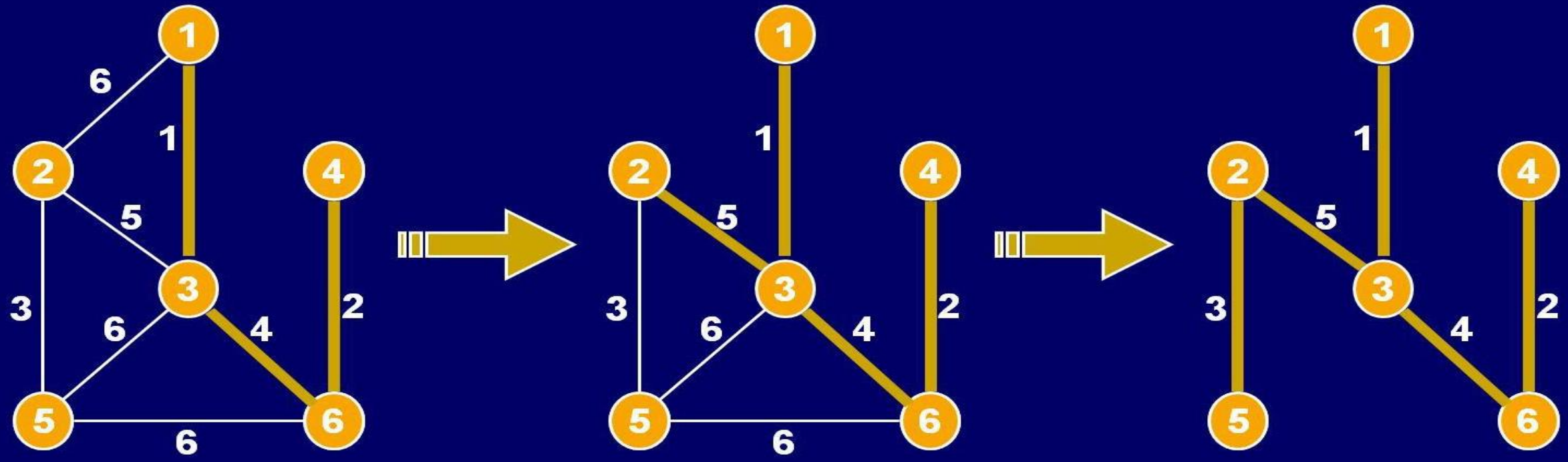
Example – Prim's Algorithm



Example – Prim's Algorithm

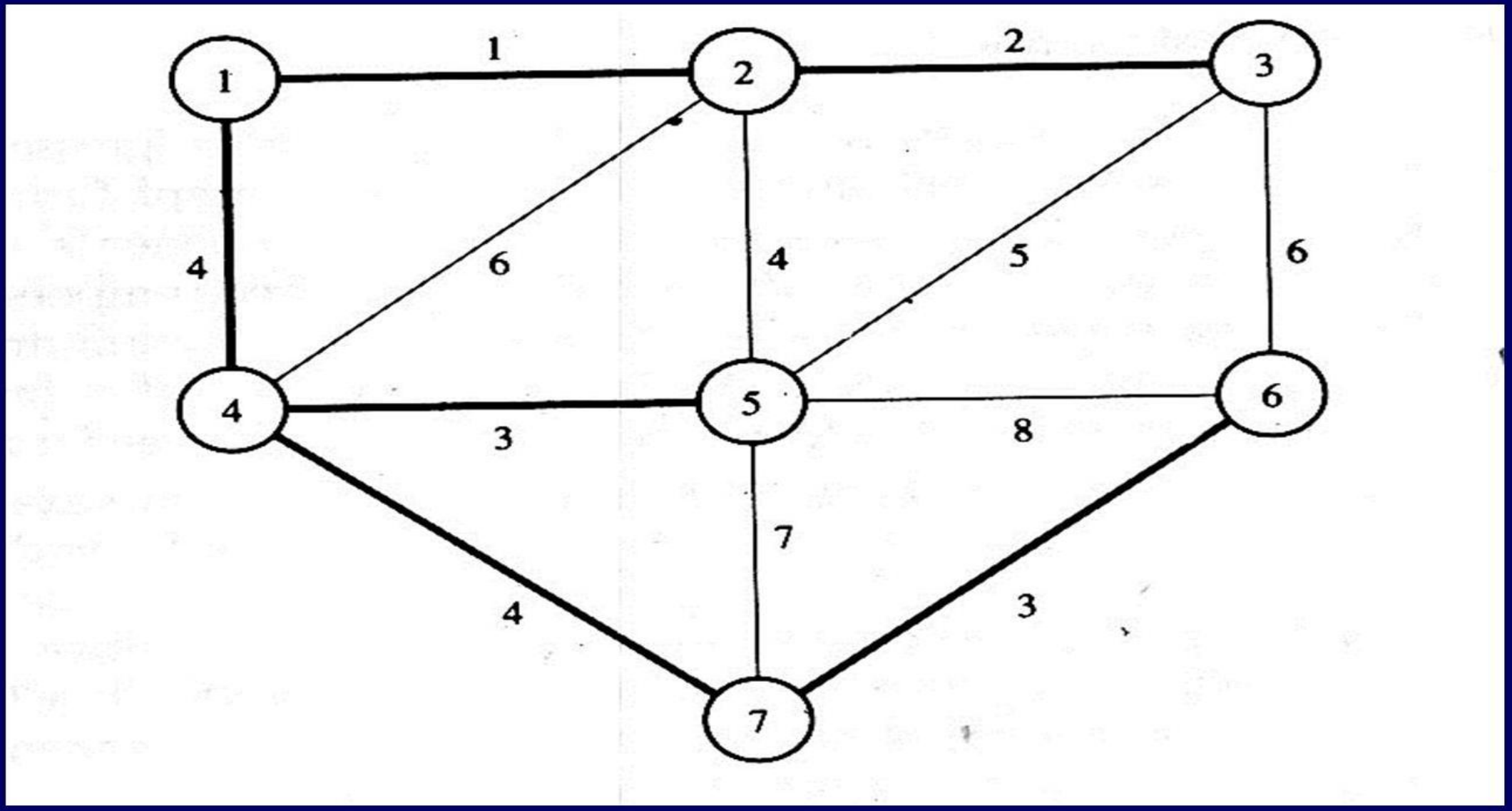


Example – Prim's Algorithm



- Minimum Spanning Tree Cost = 15

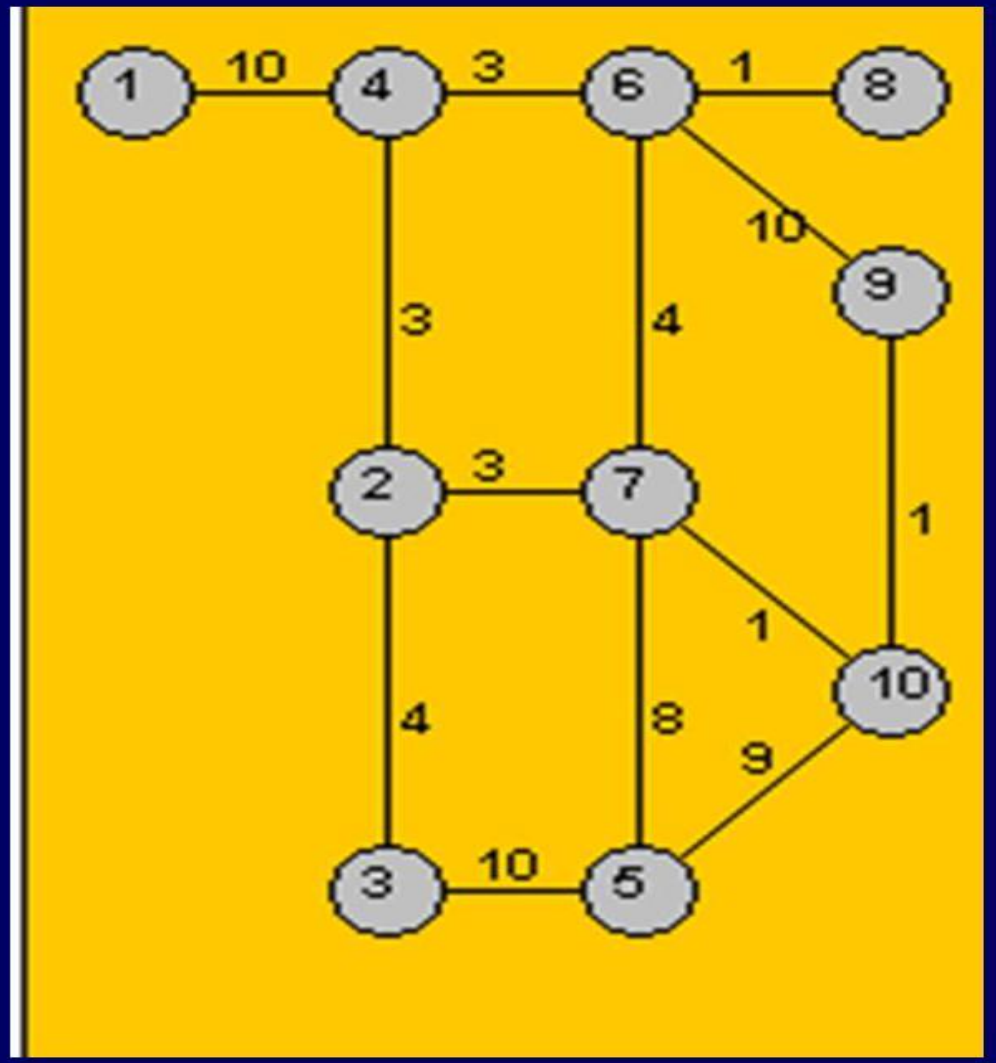
Running Prim

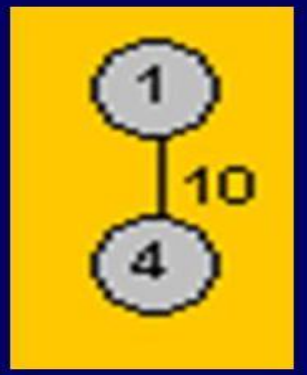


Prim's algorithm run

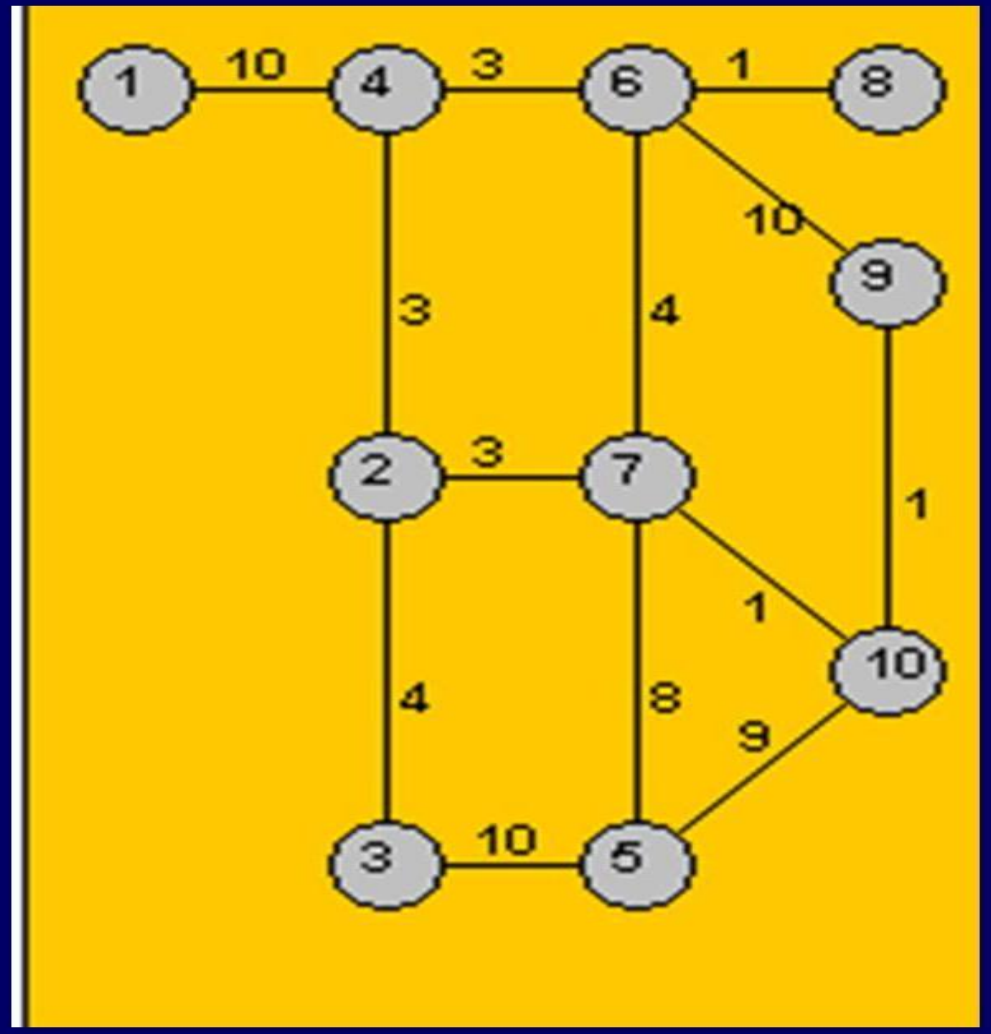
Step	{u v}	B
Init	-	{1}
1	{1 2}	{1 2}
2	{2 3}	{1 2 3}
3	{1 4}	{1 2 3 4}
4	{4 5}	{1 2 3 4 5}
5	{4 7}	{1 2 3 4 5 7}
6	{5}	rejected
7	{4 7}	{1 2 3 4 5 6 7}

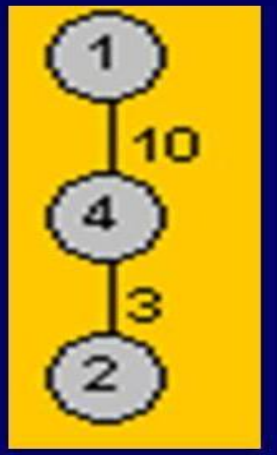
Prim's algorithm run



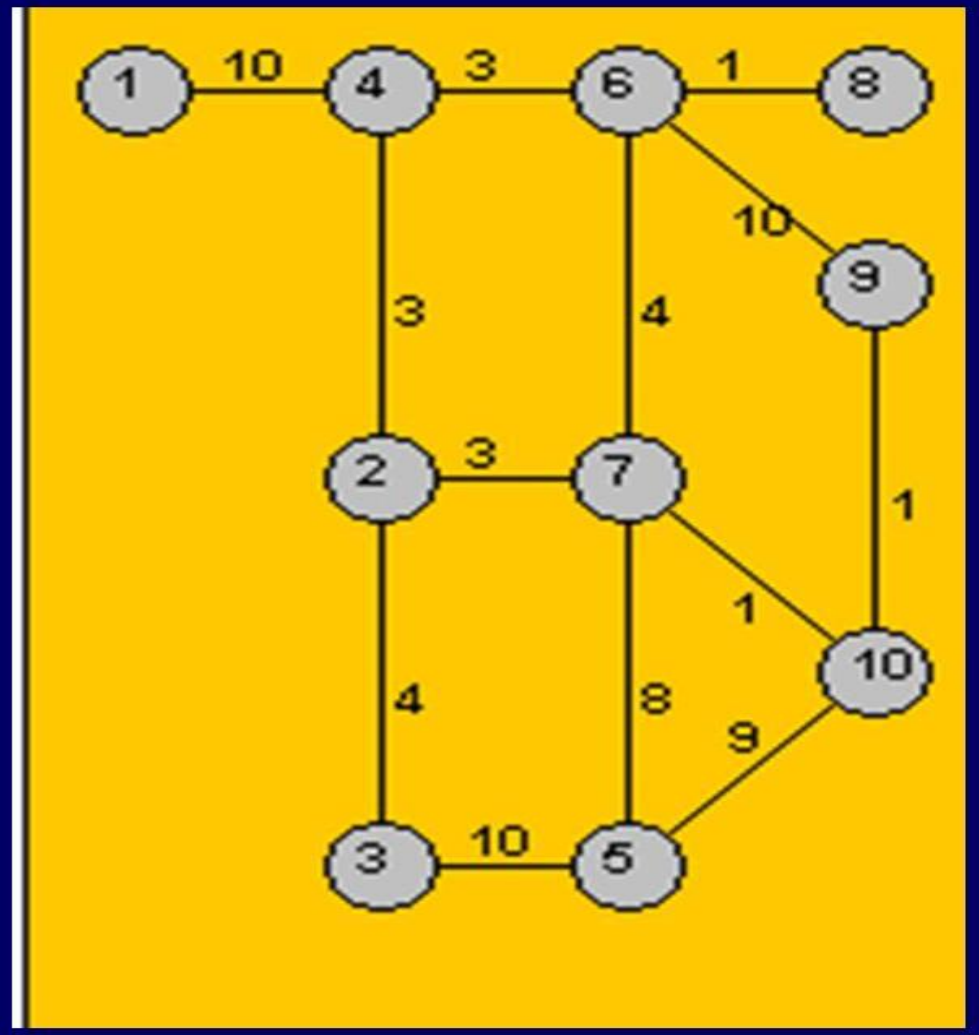


Prim's algorithm run

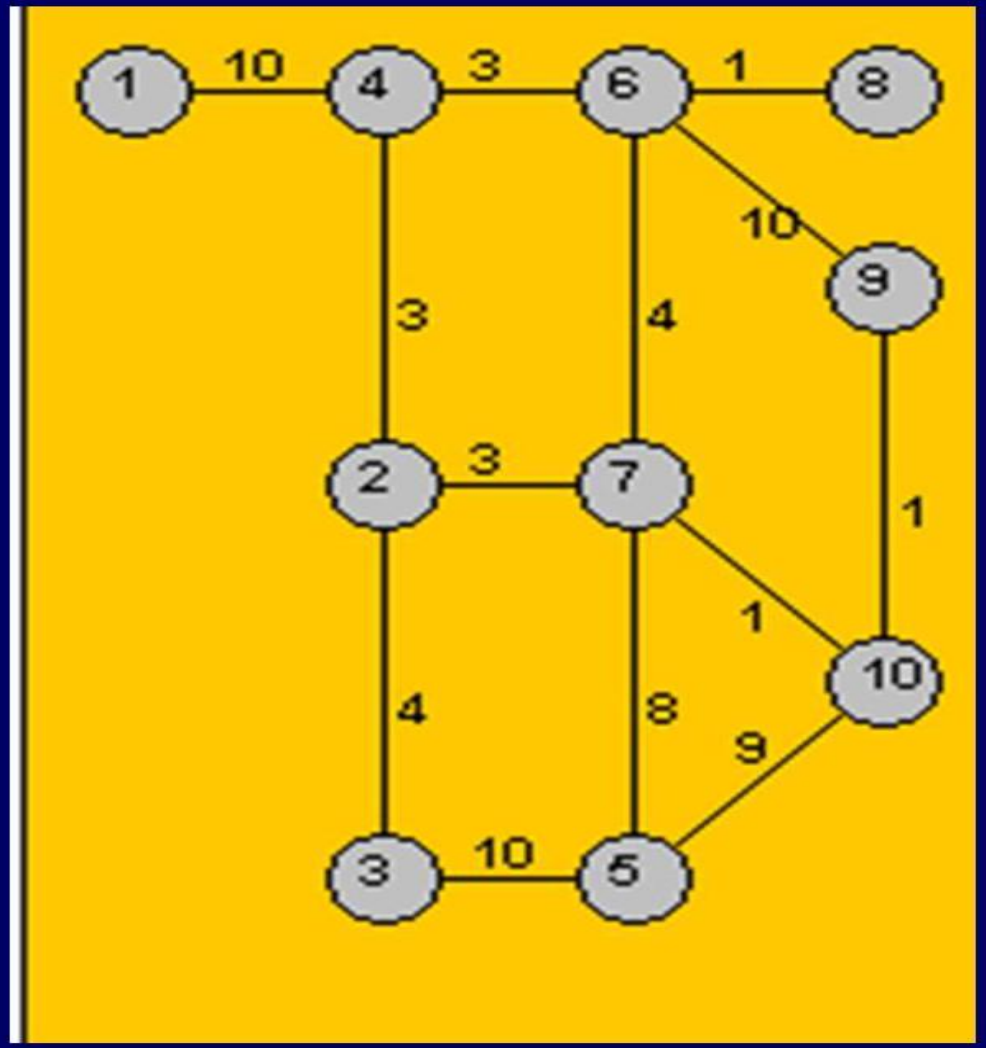


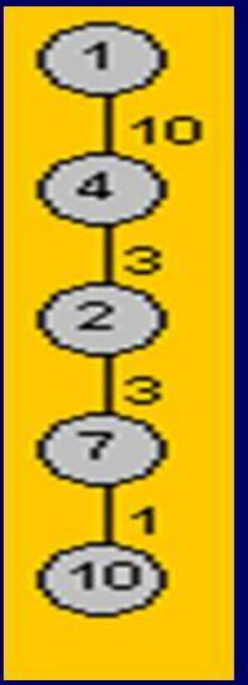


Prim's algorithm run

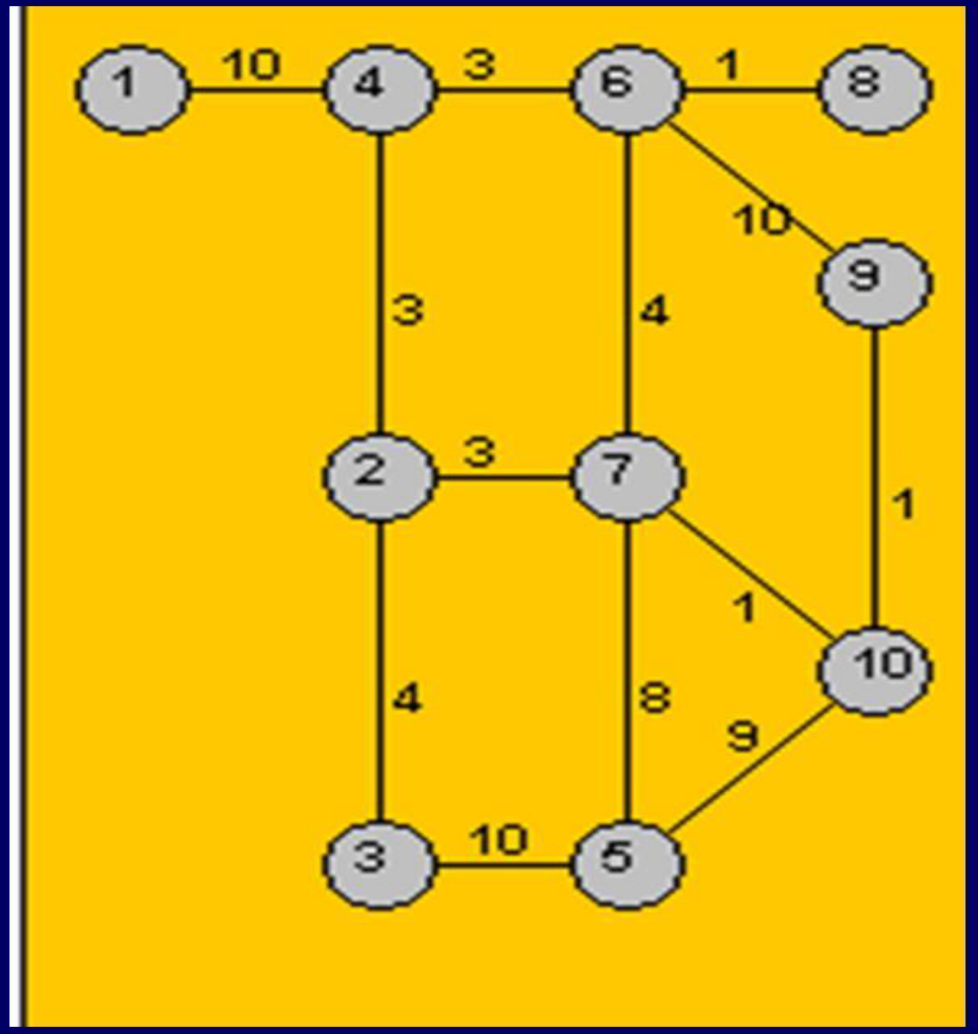


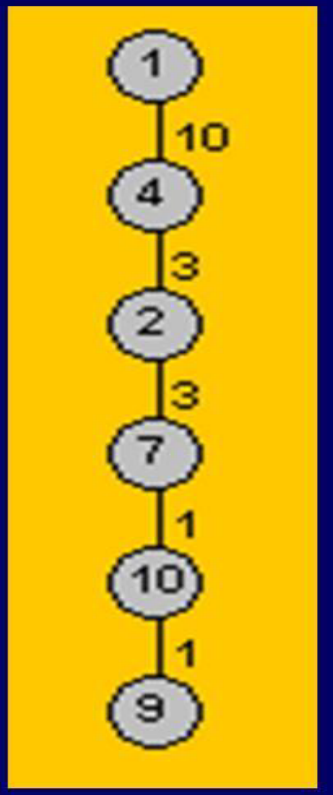
Prim's algorithm run



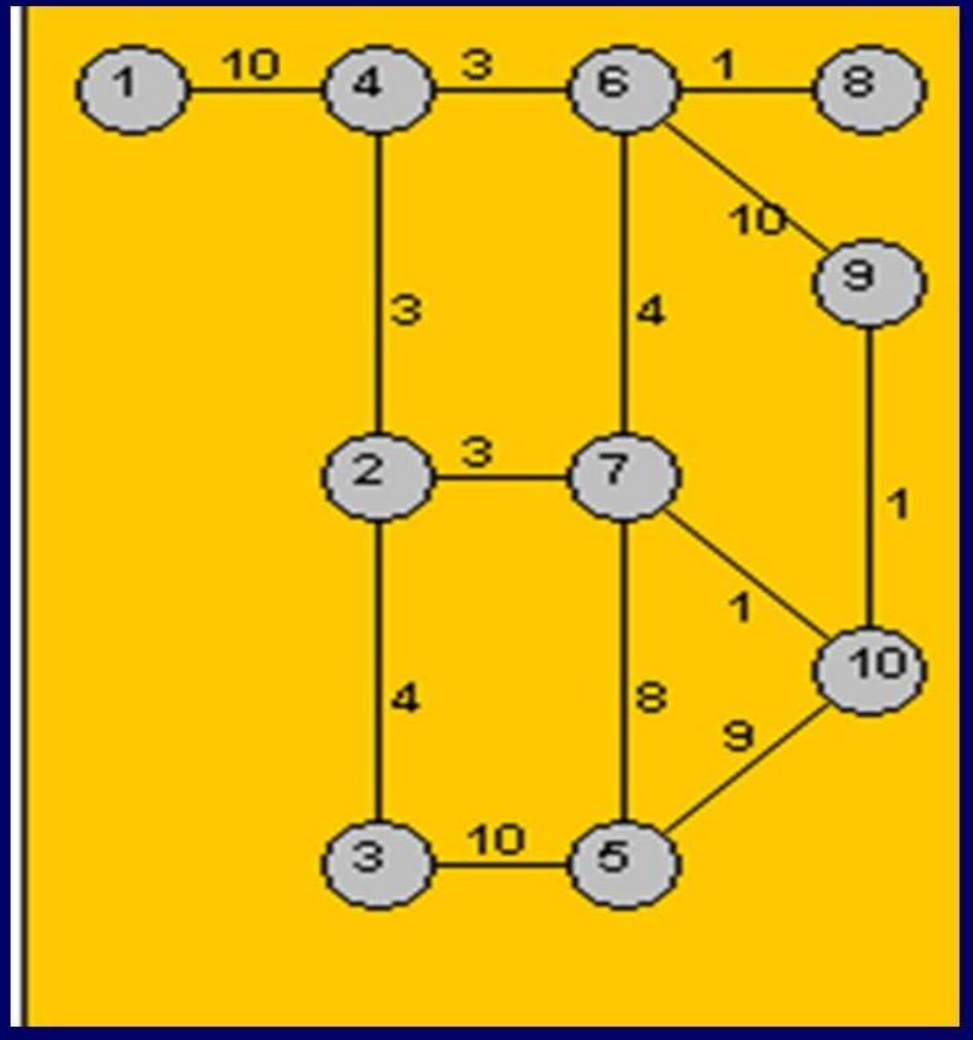


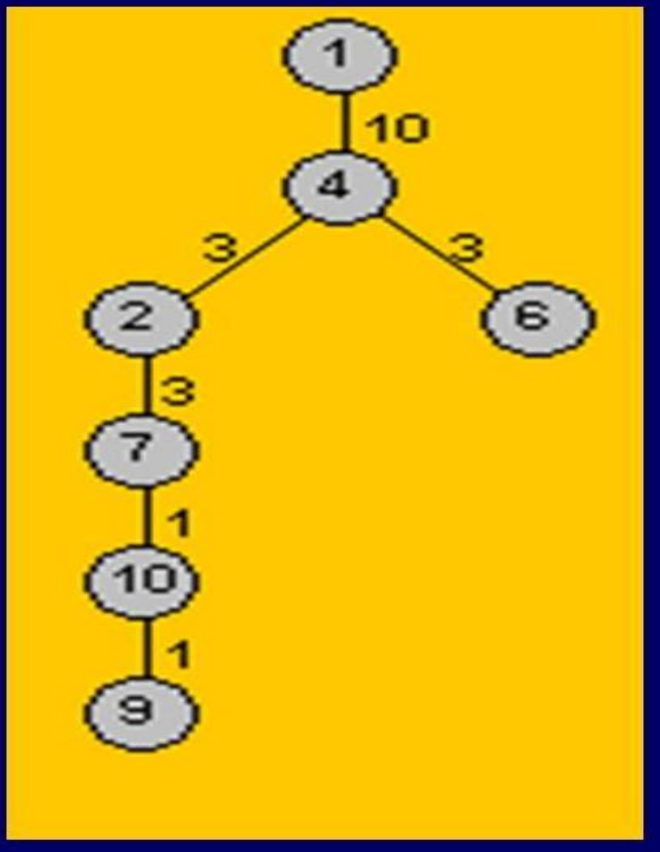
Prim's algorithm run



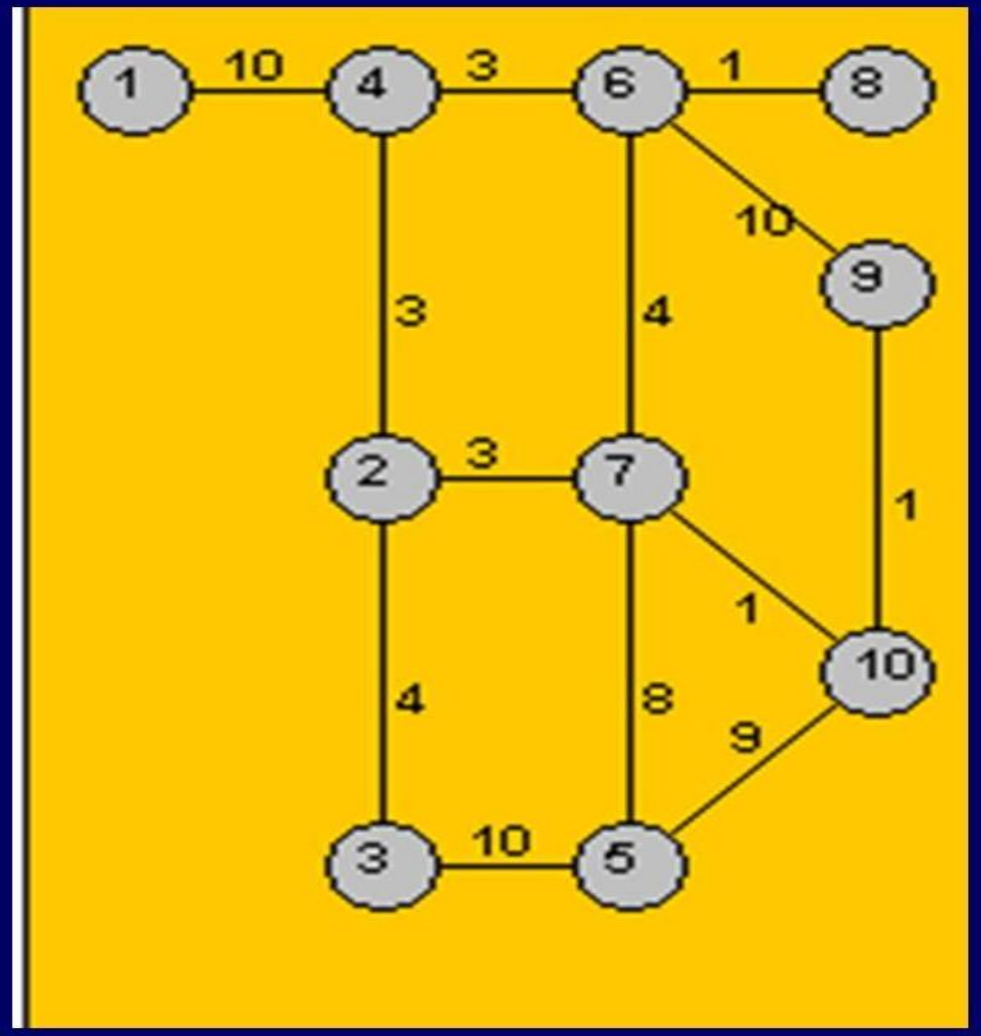


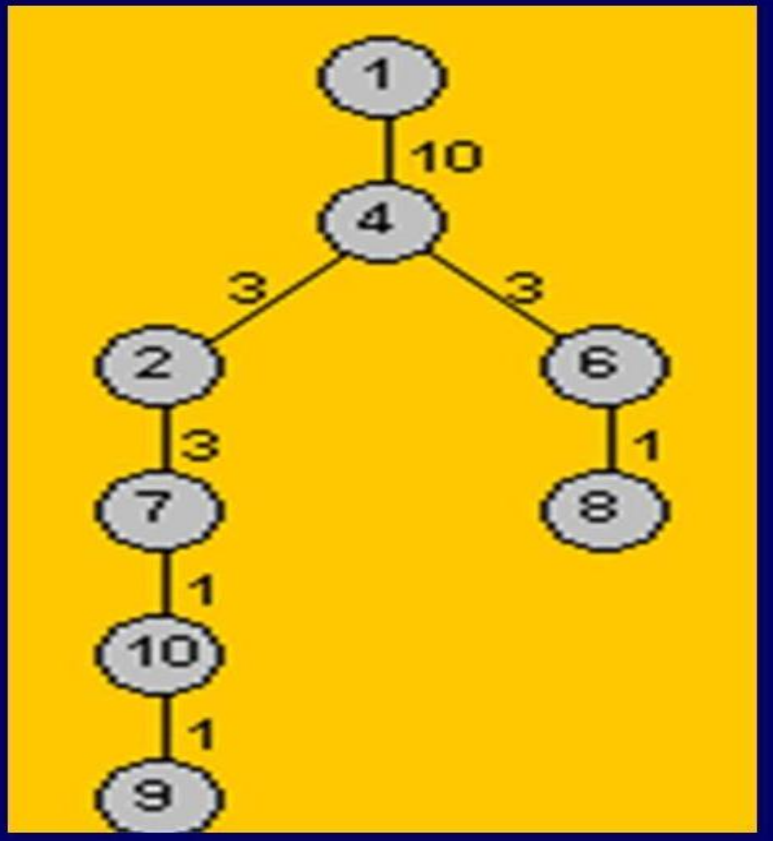
Prim's algorithm run



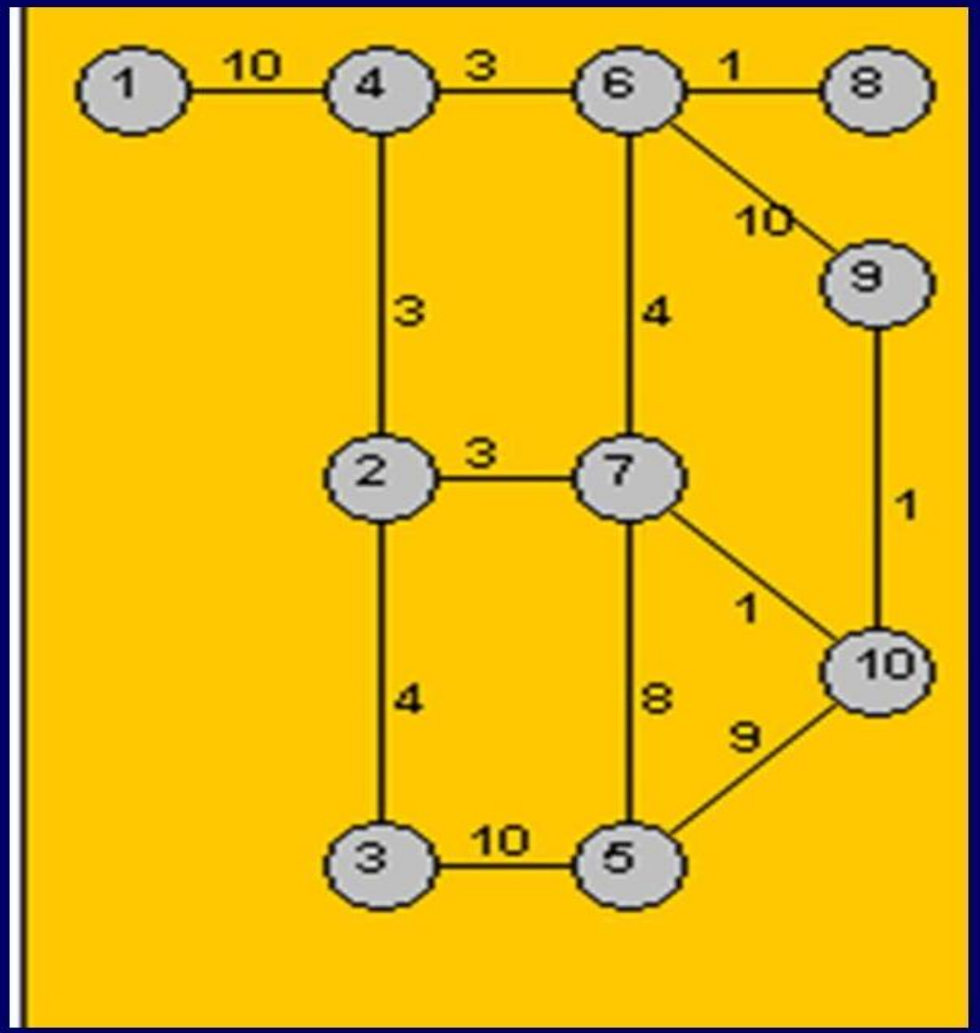


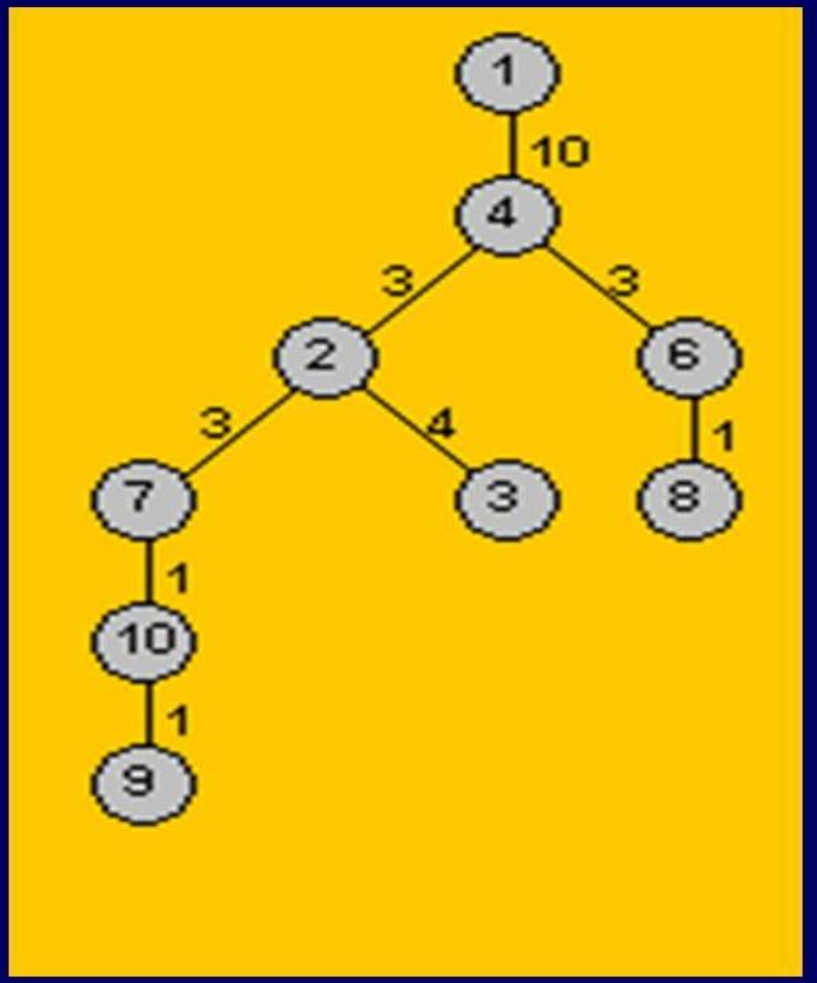
Prim's algorithm run



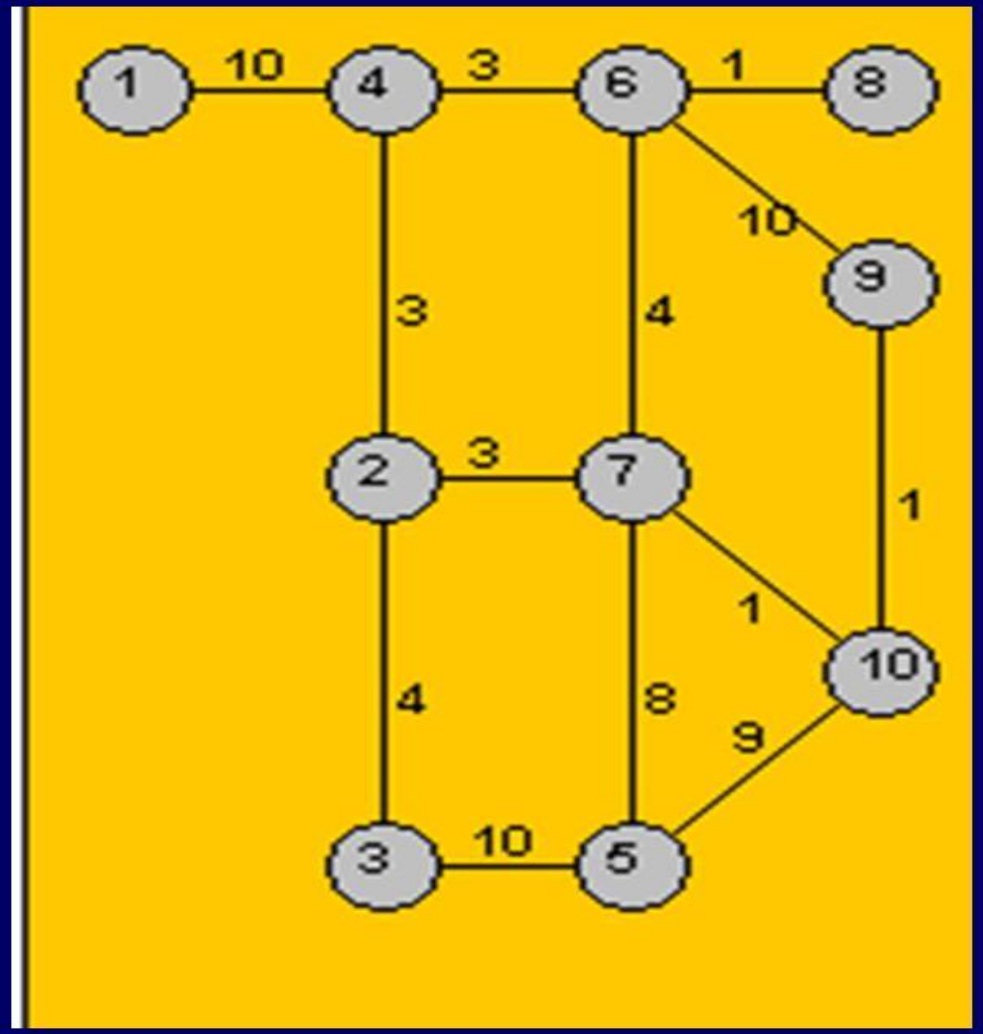


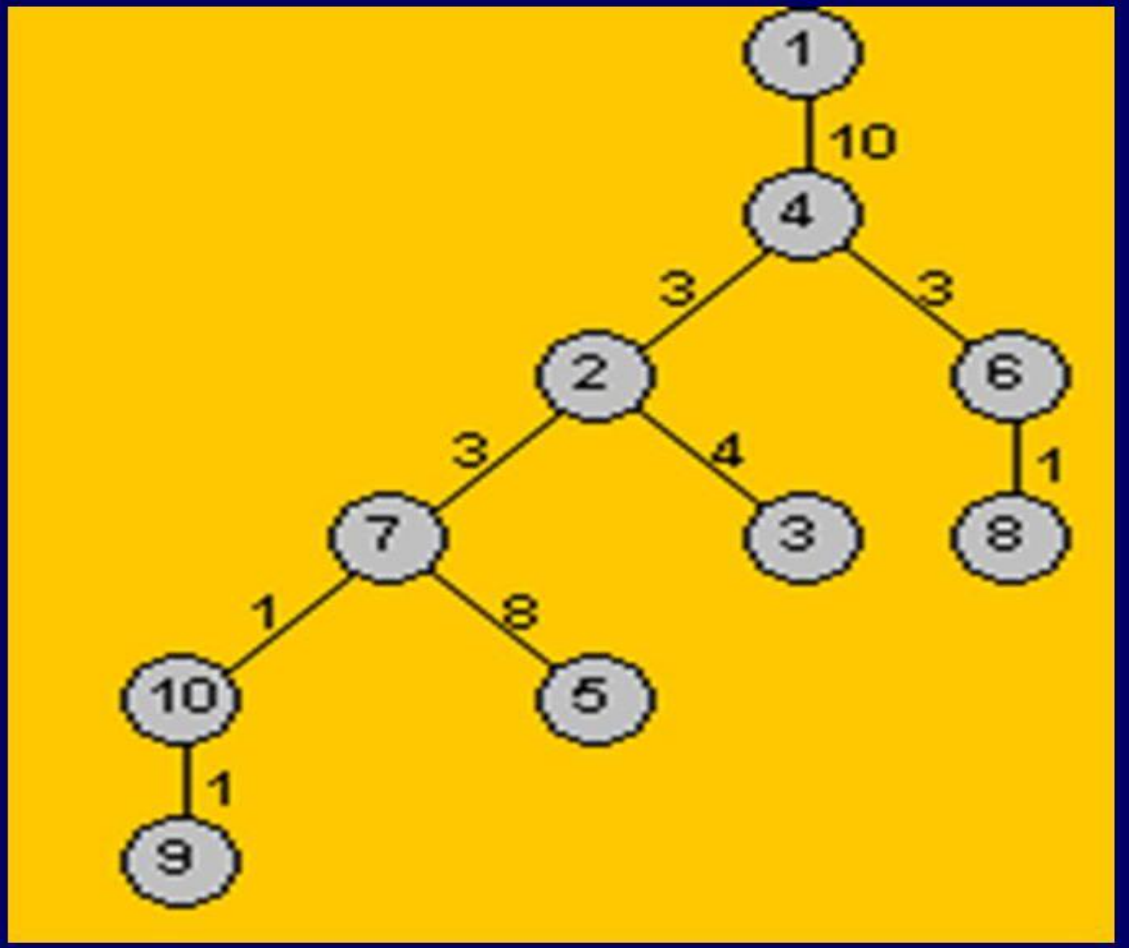
Prim's algorithm run



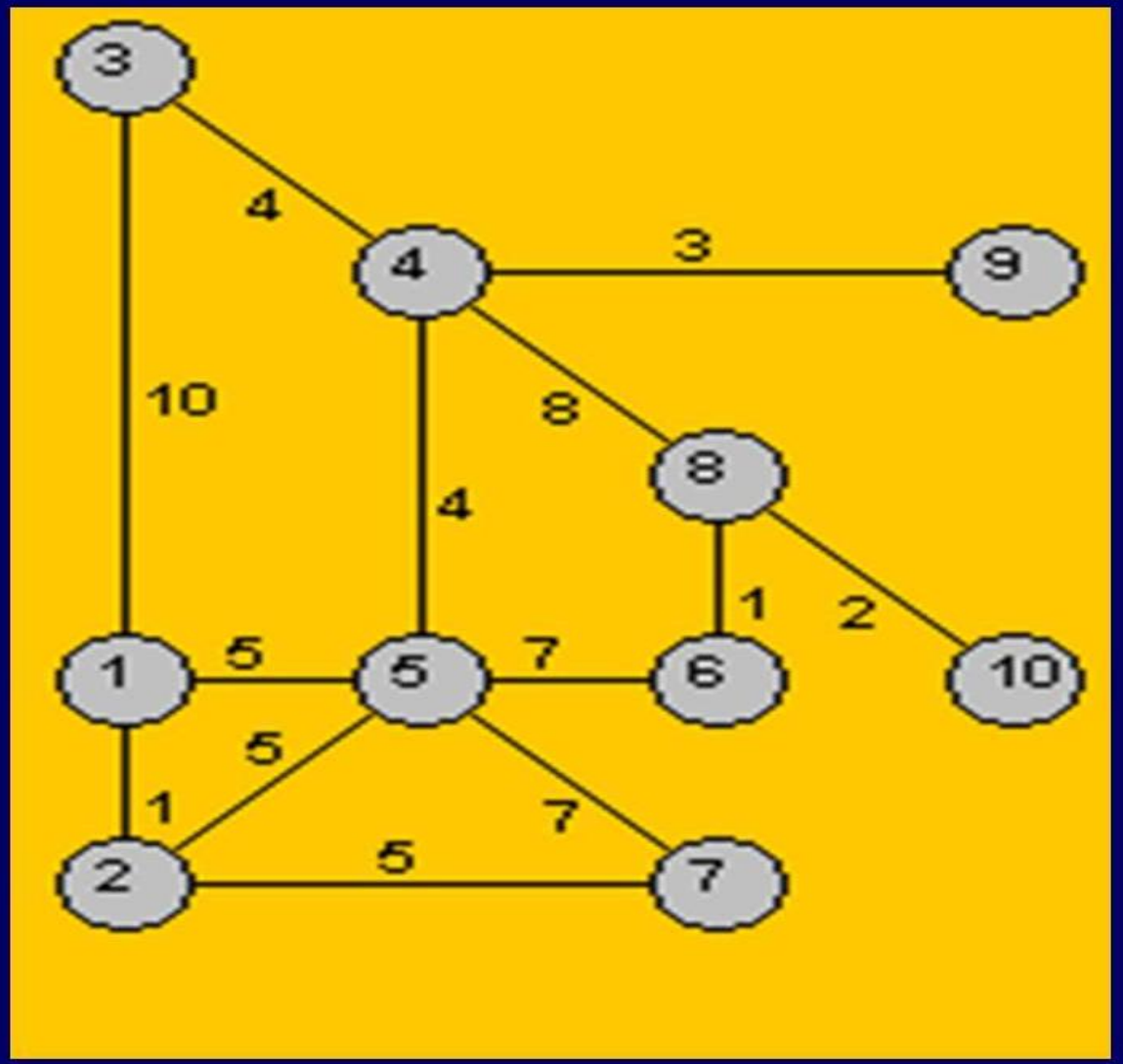


Prim's algorithm run





Tutorial #4 - Prim's algorithm run



MST Problem Extensions

- MST problem is only a particular formation of a broader network design goal
  - only a good way to connect a set of sites by installing edges between them such that
    - the TOTAL edge cost is the minimum
- But, does it ensure that
  - point-to-point distance between the edges it has, is also the minimum ?
  - sometimes, one is willing to pay more for the total cost but reduce the cost between specific nodes

Huffman's Coding

Huffman Coding

- Basically, a file compression technique
- 25% saving on large files while
  - 50-60% saving on many large files
- Fixed length codes like ASCII.
  - How many bits are required to encode C distinct characters ?
  - Consider an alphabet = <a, u, x, z> and an example string "aaxuaxz" ?
  - How many characters are required to encode using 8-bit code?
  - How many characters are required to encode using 2-bit code?
- Now consider a string of 1000 characters from the same alphabet…..
- How many bits are required for this string in
  - in 8-bit ASCII code ?
  - in 2-bit fixed length code ?

Alternate Schemes

- Are we exploiting any information
  - about the frequency of occurrence of a particular character in the input string in above ?
- Consider a variable length coding scheme with
  - a = 0, x =10, u = 110, z = 111
  - What is the interesting characteristic of the code selection, here?
- Again, consider an example string "aaxuaxz" ?
- What are the number of bits required ?
- How does it compare with ASCII and 2-bit fixed length code?

Sr No	ASCII code	2-bit Fixed Length code	2-bit variable length code
1	56 bits	48 + 14	48 + 13

- Obviously the advantage occurs for very large strings.

Prefix Coding

- For example, consider the frequency distribution as
  - a = 996 times, x = 2 times, u,z = 1 time each
- But, how to decode a string of variable length ??
- Prefix code first defined by Huffman
- definition
  - e.g. 1 and 01 are prefix codes
  - e.g. 1, 01, 001, 000 are prefix codes … … … … … … … … … … … (α)
  - e.g. 0, 10, 110, 111 are prefix codes

- Observations
  - A prefix code can always be represented as a full binary tree/trie
  - A binary tree representation always gives optimal code
  - Conclusion ??
- How would the string 1101001101000 be decoded with scheme α above.

Constructing the tree

- Need a suitable representation
  - Extended binary tree
  - Full binary tree or a trie
- Construct the full binary tree for the fixed length code seen before
- Construct the full binary tree for the variable length code seen before i.e.
  - a = 00, x = 01, u = 10, z = 11 and the frequency being 996,2,1,1 respectively
  - a = 0, x = 10, u = 110, z = 111 with the same frequency as above

Weighted External path length

- Cost of the tree:
  - Let C be the alphabet, |C| be the no of characters
  - Binary tree will have |C| leaves, |C| - 1 internal nodes
  - Then, no of bits required to encode a file
    - B(t) = $\sum_{c \in C} f(c) d_T(c)$ for every c $\in$ C
    - Cost of the tree T
    - Weighted External path length of a binary tree

Constructing the tree

- Given a prefix code, how do we compute its bit pattern for encoding ?
- Defining Huffman tree
- Construct the Huffman tree for the following

a	b	c	d	e	f
6	2	3	3	4	9

Huffman Coding Algorithm

```
Algorithm HUFFMAN(C)
1.  n = |C|
2.  for i = 1 to n-1
3.    do z = ALLOCATE_NODE()
4.     x = left(z) = EXTRACT_MIN(Q)
5.     y = right(z) = EXTRACT_MIN(Q)
6.     f[z] = f[x] + f[y]
7.    INSERT(Q,z).
8.return EXTRACT_MIN(Q)
```

Why n-1 ?

Why ?

The Container Loading Problem

- Let x_i be a boolean variable (1=container loaded , 0= container not loaded). The problem is to assign values x_i such that with
  - W_i - the weight of the container i and
  - C - the maximum cargo carrying capacity of a ship,

$$\sum_{i=1}^{n} x_i$$ is maximized

with respect to the constraints that

$$\sum_{i=1}^{n} w_i x_i \leq C$$

An example

- Given that
  - n = 8,
  - i = [1,2,3,4,5,6,7,8]
  - w_i = [100,200, 50, 90, 150, 50, 20, 80]
  - C= 400
- What is the order of loading ?
  - [x_1….x_8] = [1,0,1,1,0,1,1,1]
- What is the sigma x_i ?
- Implement the above problem
  - using Indirect Sorting viz. IndirectSort(w, t, n)
  - the two input arrays t_i and w_i as input

Algorithm Container Loading

```
1.  for i=1 to n
2.      do x[i] = 0
3.  t ← ALLOCATE_MEMORY(n)
4.  IndirectSort(w, t, n)
5.  while (i <=n) and w[t[i]] ≤ C
6.  do
7.      x[t[i]] = 1
8.      C = C – w[t[i]]
9.      i=i+1
10. delete t
```

The Knapsack problems

- are generalizations of the Container loading problem
- Given
  - n objects each with weight w_i and value v_i
  - a knapsack with maximum weight carrying capacity W
  - optimize the value $\sum_{i=1}^{n} x_i v_i$ 1 <= i <=n
  - such that

$$\sum_{i=1}^{n} w_i x_i \leq W$$

  - What could be the domain of the values x_i ?
  - How are the output optimization function and the W related ?
  - Illustrations……

Illustration

- Consider the case where
  - n = 5, W = 100 and
  - w[i] = 10 20 30 40 50
  - v[i] = 20 30 66 40 60
- What are the values of sigma(w[i]) and sigma(v[i]) for i = 1 to 5 – with *maximum value first* approach ?
- What could be the optimal answer ?
- What should be the correct approach to solve the problem ?
- The values
  - v[i]/w[i] = 2.0 1.5 2.2 1.0 1.2
- The optimal answer….

Illustration

- Consider the case where
  - n = 5, W = 100 and
  - w[i] = 10 20 30 40 50
  - v[i] = 20 30 66 40 60
- What are the values of sigma(w[i]) and sigma(v[i]) for i = 1 to 5 – with *maximum value first* approach ?
- What could be the optimal answer ?
- What should be the correct approach to solve the problem ?
- The values
  - v[i]/w[i] = 2.0 1.5 2.2 1.0 1.2
- The optimal answer….

Max Weigth First = 146
Min Weight First = 156
Max ValueDensity First = 164

The 0/1 knapsack problem

- The thirsty baby and the container loading problem are the generalizations of the fractional knapsack problem
- Different samples:
  - i= [1 2 3] v=[20 15 15] w=[100 10 10] and W=105
  - i= [1 2] v=[5 100] w=[10 20] and W=25
  - i= [1 2 3] v=[40 25 25] w=[20 15 15] and W=30
- Prepare a table that shows
  - all the approaches in three columns and all the samples in rows and tick which one works……and fourth column shows the actual optimal answer.
- Compare with the optimal solutions

The 0/1 knapsack problem

- The thirsty baby and the container loading problem are the generalizations of the fractional knapsack problem
- Different samples:
  - i= [1 2 3] v=[20 15 15] w=[100 10 10] and W=105 $V_{opt} = 30$
  - i= [1 2] v=[5 100] w=[10 20] and W=25 $V_{opt} = 100$
  - i= [1 2 3] v=[40 25 25] w=[20 15 15] and W=30 $V_{opt} = 50$
- Prepare a table that shows
  - all the approaches in three columns and all the samples in rows and tick which one works……and fourth column shows the actual optimal answer.
- Compare with the optimal solutions

Tutorial Problem #5

- Let F(I) be the value of the solution generated on knapsack problem instance I by GreedyKnapsack when the objects are processed in nonincreasing order of their profit values. Let F*(I) be the value of an optimal solution of this instance. How large can the ratio F*(I)/F(I) get ?

Tutorial Problem #6

- For a fractional knapsack problem, let n = 7, W=20 $(v_1, v_2, v_3, v_4, v_5, v_6, v_7) = (12, 8, 5, 11, 8, 12, 6)$ $(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (4, 5, 2, 8, 3, 6, 2)$. Find the optimal solution.

Tutorial Problem #7

- Consider the 0/1 knapsack problem with n objects whose profits and weights are v_i, w_i $1<=i<=n$, respectively. The capacity of the knapsack is m. IT is also given that all the objects have the same weight. Present an O(m) algorithm to solve this problem. Also, give its proof of correctness.

Tutorial #8

- Suppose a Consulting firm that does security consultancy needs to obtain licenses for n different pieces of cryptographic software. However, due to regulations, they can only buy these licenses at the rate of at most one license per month. Each license is currently selling at the rate of Rs 1000. However, the cost of the licenses are designed to increase following the monthly exponential growth, at the defined rates i.e. the cost of license j increases at the rate r_j (>1) every month, that grows exponentially. For example, if license j is purchased t months from now, it will cost 100 * r_j^t.We assume that all the price growth rates are distinct i.e. r_j != r_j for i != j.

  Given as the input, the n rates of price growth viz. r_1, r_2, r_3, …..r_n, determine the order in which the licenses must be bought so that the total amount of money spent is minimized.

Tutorial #9, #10

- Activity Selection Problem – Suppose that instead of selecting the first activity to finish, we instead select the last activity to start that is compatible to all the previous activities. Describe how this approach is a greedy algorithm and prove that it yields an optimal solution.
- Suppose that we have a set of activities to schedule amongst a large number of lecture halls. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

Tutorial #11, #12

- Prof Midas drives an automobile from Newark to Reno along Interstate 80. His car's gas tank, when full, holds enough gas to travel n miles and his map gives the distances between gas stations on his route. The professors wishes to make as few gas stops as possible along the way. Give an efficient method by which Professor Midas can determine at which gas stations should he stop and prove that your strategy yields an optimal solution.

- Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem and argue that your answer is correct.

Tutorial #13

- Show how to solve the fractional knapsack problem in O(n) time. Assume that you have a solution to the problem 9.2/CLRS.
  - For n distinct elements x_1, x_2, x_3, ……x_n with positive weights w_1, w_2, w_3, ….w_n such that $\sum_{i=1}^{n} w_i = 1$, the weighted lower median is the elements x_k satisfying

    $$\sum_{x_i<x_k} w_i < \frac{1}{2} \quad and \quad \sum_{x_i>x_k} w_i < \frac{1}{2}$$

  - This problem is solved using a $\theta(n)$ algorithm in chapter 9.

Tutorial #14

- Suppose you are given two sets A and B, each containing n positive integers. You can choose to order each set however you like. After reordering, let ai be the ith element of set A and let bi be the ith element of set B. You then receive a payoff of $\prod_{i=1}^{n} {a_i}^{bi}$. Give an algorithm that will maximize you payoff. Prove that your algorithm maximizes the payoff and state its running time.

Tutorial #15

- Let's consider a long, quiet country road with houses scattered very sparsely along it. That is, picture the orad as a line segment, with an eastern endpoint and a western endpoint). Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations (*'towers' as we say here*) at certain points along the road, so that every house is within four miles of one of the base stations.

  Give an efficient algorithm that achieves the goal, using as few base stations as is possible. Prove your algorithm , too.

Single Source Shortest Path Problem

Single Source Shortest Path Problem

- A directed graph G=(V,E) and the edge weight function w: E $\rightarrow$ R which maps edges to relative weights.
- A fixed vertex s.
- Defining δ(s,u) as the length of the shortest path from s to u.
- Given G, w, and a vertex S, compute δ(s,u) for all u in V.
- Also, find the path of shortest distance from u to s.
- What is the weight of path p = <v0,v1,v2,v3…..vk> ?

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Single Source Shortest Path (contd)

- What is the shortest path weight from u to v ?
  $\delta(u, v)=$
- A shortest path from vertex u to vertext v is then defined as any path p with weight $w(p) = \delta(u, v)$
- Problem : Finding the shortest distance from location 1 to location 2.
- How do we model this problem ?

The SSSP Problem (contd)

- What is the weight of path p = $\langle v_0, v_1, v_2, v_3 \ldots . v_k \rangle$?
  - $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- What is the shortest path weight from u to v ?
  - $\delta(u, v) =$
- A shortest path from vertex u to vertext v is then defined as any path p with weight $w(p) = \delta(u, v)$

The shortest path problem

- Finding the shortest distance from location 1 to location 2.
- How do we model this problem?
  - vertices to represent intersections
  - edges to represent road segments between intersections
  - edge weights to represent road distances.
  - goal is to find out the shortest distance from intersection 1 to intersection 2.
- Surprisingly ..
  - Edge weights can be metrics other than distances.
  - BFS problem – shortest path algorithm for un-weighted graphs. While finding the shortest path from a source to one destination,
  - we can find the shortest paths to all over destinations as well!

Several different versions

- single destination shortest path problem
  - How can this problem be solved?
- single pair shortest path problem
  - How can this problem be solved?
- all pairs shortest paths problem
  - How can this problem be solved?
- directed graphs v.s. undirected graphs
- G=(V, E) with negative weight-cycles reachable from the source s and the the shortest path weight $\delta(u, v)$
- Assume all edge weights are nonnegative i.e. $\delta(u, v)$= +ve only

Negative weight cycles

- Fig below shows the effect of the negative weights on shortest path weights

- We only discuss the simpler case: all weights are non-negative.

Representing the shortest paths

- Using a predecessor array $\pi[v]$ for each vertex $v \in V$ i.e.
  - given a graph $G = (V, E)$, we maintain for each vertex $v \in V$, a predecessor $\pi[v]$ that is either another vertex or is nil.
- How do we set the π attributes ?
  - the chain of predecessors originating at a vertex v runs backwards along a shortest path from s to v
  - Therefore to print a shortest path from s to v we use the PRINT_PATH routine.

Representing the shortest paths (contd)

```
1. PRINT-PATH(G, s, v)
2. if v = s
3.    then print s
4.    else if π[v] = NIL
5.           then print "no path from" s
   "to" v "exists"
6.           else PRINT-PATH(G, s, π[v])
7.           print v
```

- Running time of this routine?
- During the execution of the algorithm, do the π values necessarily indicate the shortest paths?

Representing the shortest paths (contd)

- The predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is obtained out of the induced π values during the run of the shortest-path algorithm, where,
  - $V_\pi = \{v \in V: \pi[v] \neq NIL\} \cup \{s\}$
  - $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$
- This will be at termination, a shortest path tree i.e. a rooted tree containing a shortest path from a source s to every vertex that is reachable from s.
- How is this shortest path tree different from the one produced by the BFS?
- Formal definition of the shortest path for $G = (V, E)$: a weighted directed graph with weight function $w: E \rightarrow R$.
- Shortest paths are not necessarily unique and neither are shortest paths trees as is shown below

Generic SSSPath Algorithm

```
1.  Assumption: all edge weights are nonnegative
2.  a set S of vertices; S contains the processed vertices
3.  a set VS of vertices
4.  a variable d(v) for  each vertex v; d(v) will  contain d(s,v) at the end of execution of algorithm
5.  a variable Π(v) for each vertex v; Π(v) is the parent of v, i.e., the vertex just before v in the shortest path from s
    to v
6.  S = {s}; d(s) = 0;
7.  for each vertex v of G do {d(v) = infinity}
8.        for each vertex v in Adj[s] do {
9.           d(v)=w(s, v)
          }

10.       while S does not contain all the vertices of G
11.       do {
12.        Let v be the vertex with minimum d[v] in set VS.
            S=S U {v} ;
            for each vertex x in Adj[v] do {
13.                 if x is in set VS {
14.                       if(d(x) > d(v) + w(v,x){    //Relaxation of
    edge(v,x)
15.                       d(x) = d(v) + w(v,x);
16.                              Π(x) = v;
17.                       }
18.                 }
19.       }
20. }
```

Dijkstra's Algorithm

- Dijkstra's Algorithm
  - is an efficient implementation of the generic single source shortest path algorithm
  - solves the single source shortest paths problem on a weighted, directed graph **G = (V, E)**.
  - all edge weights are nonnegative i.e. **w(u,v) >= 0** for each edge **w(u,v) ϵ E**.
  - keeps *two* sets of vertices:
  - S Vertices whose shortest paths have already been determined

    V-S Remainder – Maintained as a priority queue Q, where vertices are stored using their d values as the keys.
- Also
  - d Best estimates of shortest path to each vertex

    $\prod$ Predecessors for each vertex
  - for all vertices $v \in S$, $d[v] = \delta(s, v)$.
- The algorithm repeatedly selects the vertex $u \in V-S$ with the minimum shortest path estimate, inserts u into S and relaxes all edges leaving u.

Outline of Dijkstra's algorithm

- Initialise d and π
  - For each vertex, j, in V
    - dj = ¥
    - pj = nil
  - Source distance, ds = 0
- Set S to empty
- While V-S is not empty
  - Sort V-S based on d
  - Add u, the closest vertex in V-S, to S
  - Relax all the vertices still in V-S connected to u

Dijkstra's algorithm

```
1. d(s) = 0;
2. for each vertex v of G do { d(v) = infinity;
                                Q.insert (v); }
3. for each vertex v in Adj[s] do
                        { d(v) = w(s, v);Q.update(v);}
4. while Q is not empty do {
5.    v= Q.ExtractMin().
6.    for each vertex x in Adj[v] do {
7.             if x is in set VS {
8.                    if( d(x) > d(v) + w(v,x){ //Relaxation of
   edge(v,x)
9.                           d(x) = d(v) + w(v,x);
10.                             Q.update(x);
11.                             π(x) = v;
12.                          }
13.                   }
14.            }
15.     }
```

- Running Time = O(V2 + E) = O(V2)
- Which design technique is it based upon?

SSSP Computation with negative edge weights

- def: A vertex v is kreachable from s if there exists a shortest path from s to v consisting of exactly k+1 vertices.
- BellmanFord Algorithm
  - Given a weighted directed graph **G=(V,E)** with source s and weight function **w: E→R**, the Bellman-Ford algorithm returns a Boolean value.
    - indicating whether or not, there is a negative-weight cycle that is reachable from the source.
    - If there is such a cycle, the algorithm indicates that no solution exists.
    - If there is no such cycle, the algorithm produces the shortest paths and their weights.
- Algorithm given below is for directed graphs, but can be used for undirected graphs also by converting the input undirected graph into directed graph

BellmanFord Algorithm

```
BELLMAN-FORD(G, w, s)
1. S = {s};
2. d(s) = 0;
3. for each vertex v of GS do {d(v) = infinity}
4.      for i = 1 to n 1 do {
5.                      for each edge e =(u,v) of G do {
6.                                  if(d(v)>d(u) + w(u,v){
//Relaxation of edge (u,v)
7.                                  d(v) = d(u) + w(u,v);
8.                                          π(v) = u;
9.                                  }
10.             }
11.     }
12. for each edge e =(u,v) of G do {
13.             if d[v] > d[u] + w(u,v) then
{return False}
14. }
15. return True
```

- Running Time = O(V,E)
- Which design technique is it based upon?