CSC 344 – Algorithms and Complexity

Lecture #6 – Greedy Algorithms

Optimization Problems

- An ***optimization problem*** is the problem of finding the ***best*** solution from all feasible solutions
- Shortest path is an example of an optimization problem: we wish to find the path with lowest weight.
- What is the general character of an optimization problem?

Optimization Problems

- Ingredients:
  - Instances: The possible inputs to the problem.
  - Solutions for Instance: Each instance has an exponentially large set of valid solutions.
  - Cost of Solution: Each solution has an easy-to-compute cost or value.
- Specification
  - Preconditions: The input is one instance.
  - Postconditions: A valid solution with optimal cost. (minimum or maximum)

Greedy Solutions to Optimization Problems

- Every two-year-old knows the greedy algorithm.
- In order to get what you want, just start grabbing what looks best.
- Surprisingly, many important and practical optimization problems can be solved this way.

Greedy Algorithms

- A greedy algorithm always makes the choice that looks best at the moment
- My everyday examples:
  - Driving in Los Angeles, New York, or Boston.
  - Playing cards
  - Invest on stocks
  - Choose a university

A Simple Example

- Problem: Pick k numbers out of n numbers such that the sum of these k numbers is the largest.
- Algorithm:

FOR i = 1 to k

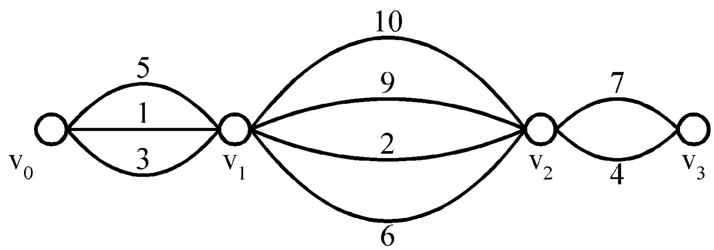
    Pick out the largest number and

    Delete this number from the input.

ENDFOR

Shortest Paths On A Special Graph

- Problem: Find a shortest path from v0 to v3.
- The ***greedy*** method can solve this problem.
- The shortest path: 1 + 2 + 4 = 7.



Greedy Algorithms

- <u>The hope</u> - a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
  - Greedy algorithms tend to be easier to code

Example - Making Change

- Problem - Find the minimum # of quarters, dimes, nickels, and pennies that total to a given amount.
- Commit to the object that looks the "best."
- Must prove that this locally greedy choice does not have negative global consequences.

Making Change

- **Instance** - A drawer full of coins and an amount of change to return
- **Solutions for Instance** - A subset of the coins in the drawer that total the amount

Amount = 92¢

25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

Making Change (continued)

- Cost of Solution:
  - The number of coins in the solution = 14
- Solutions for Instance:
  - A subset of the coins that total the amount.

Amount = 92¢

25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢
10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢
5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢
1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

A Hard "Making Change" Example

- **Problem** - Find the minimum # of 4-, 3-, and 1-cent coins to make up 6 cents.
- **Greedy Choice** - Start by grabbing a 4-cent coin.
- **Consequences**:

  4+1+1 = 6 mistake

  3+3=6 better
- Greedy Algorithm does not work!

When Do Greedy Algorithms Work?

- Greedy Algorithms are easy to understand and to code, but do they work?
- For most optimization problems, all greedy algorithms tried do not work (i.e. yield sub-optimal solutions)
- But some problems can be solved optimally by a greedy algorithm.
- The proof that they work, however, is subtle.

Minimum Spanning Trees

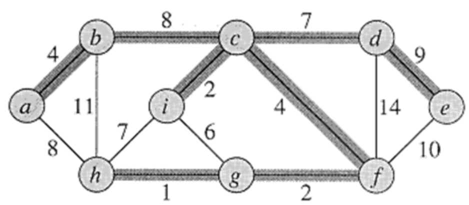
- **Example Problem:**
  - You are planning a new terrestrial telecommunications network to connect a number of remote mountain villages in a developing country.
  - The cost of building a link between pairs of neighboring villages (u,v) has been estimated: w(u,v).
  - You seek the minimum cost design that ensures each village is connected to the network.
  - The solution is called a ***minimum spanning tree*** (MST).

Minimum Spanning Trees

- The problem is defined for any undirected, connected, weighted graph.
- The weight of a subset T of a weighted graph is defined as:

$$w(T) = \sum_{(u,\, v) \in T} w(u, v)$$

- Thus the MST is the spanning tree T that minimizes w(T)

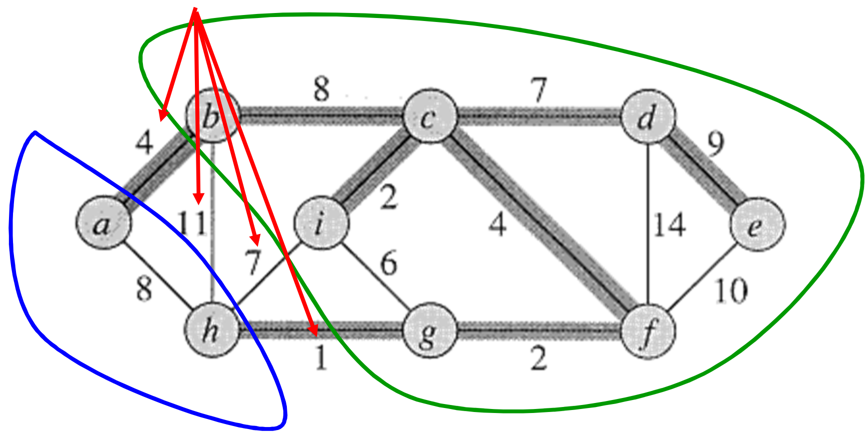


Building the Minimum Spanning Tree

- Iteratively construct the set of edges *A* in the MST.
- Initialize *A* to { }
- As we add edges to *A*, maintain a Loop Invariant:
  - *A* is a subset of some MST
  - Maintain loop invariant and make progress by only adding ***safe*** edges.
  - An edge (u,v) is called ***safe*** for *A* iff A∪({u,v}) is also a subset of some MST.

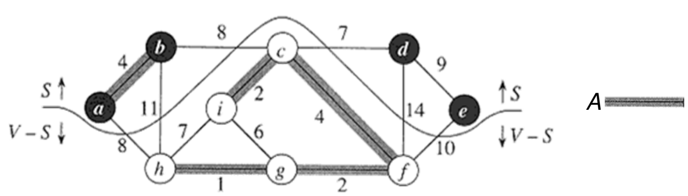
Finding A Safe Edge

- Idea: Every 2 disjoint subsets of vertices must be connected by at least one edge.
- *Which one should we choose?*



Some Definitions

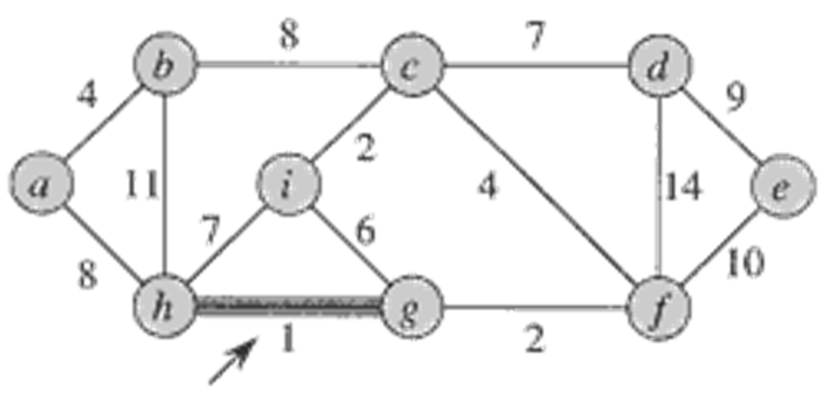
- A ***cut*** (S,V-S) is a partition of vertices into disjoint sets S and V-S.
- Edge (u,v)∈E ***crosses*** cut (S, V-S) if one endpoint is in S and the other is in V-S.
- A cut ***respects*** a set of edges A iff no edge in A crosses the cut.
- An edge is a ***light*** edge crossing a cut iff its weight is minimum over all edges crossing the cut.



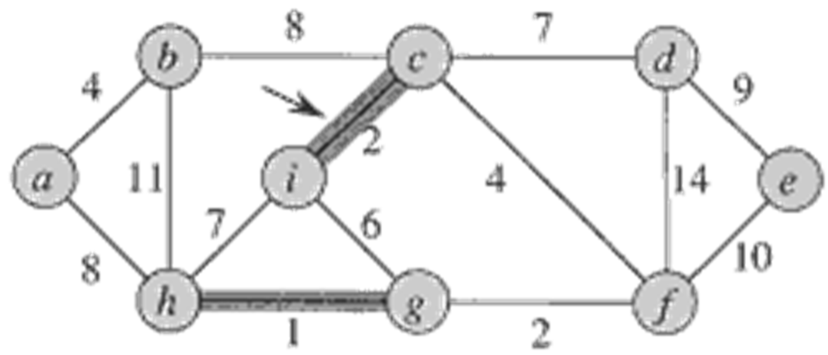
Kruskal's Algorithm for computing MST

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that crosses the cut between them.
- Scans the set of edges in monotonically increasing order by weight (***greedy***).

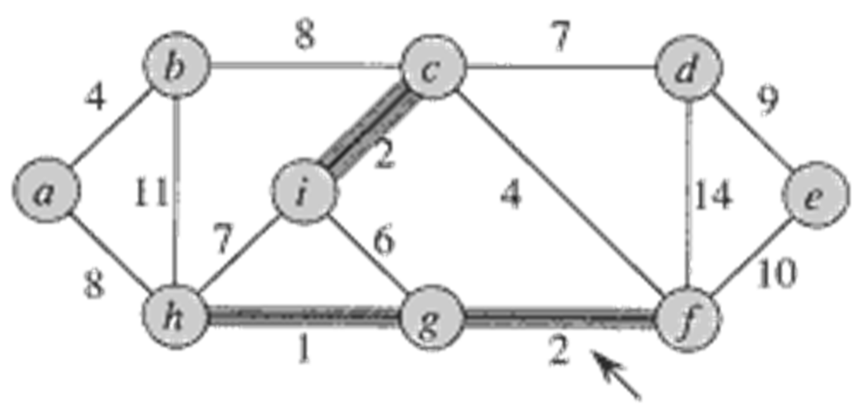
Kruskal's Algorithm: Example



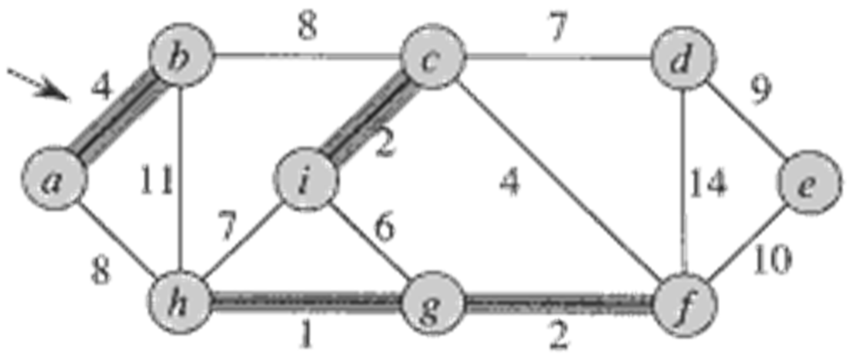
Kruskal's Algorithm: Example



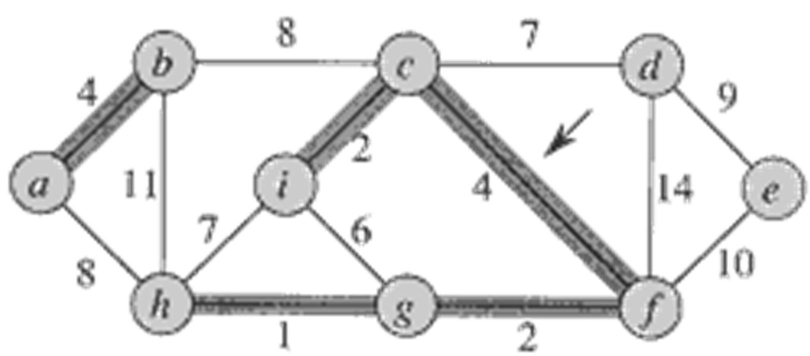
Kruskal's Algorithm: Example



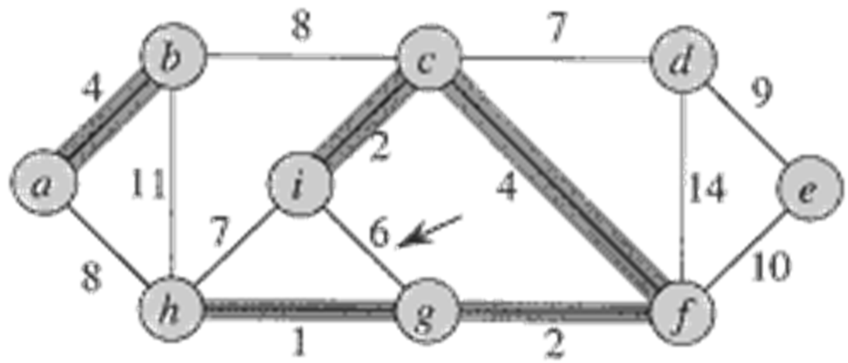
Kruskal's Algorithm: Example



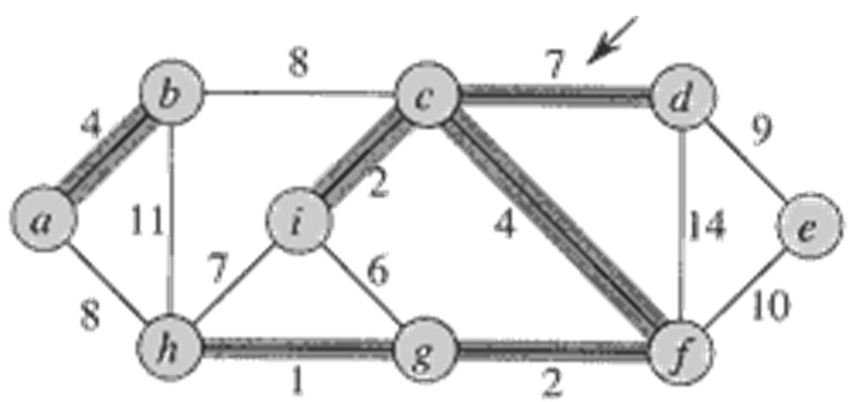
Kruskal's Algorithm: Example



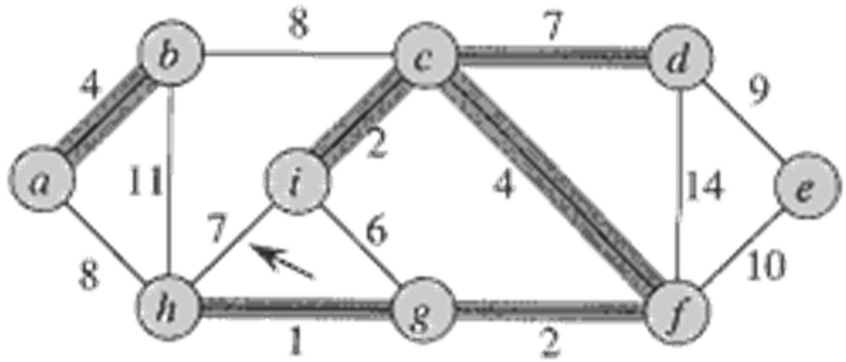
Kruskal's Algorithm: Example



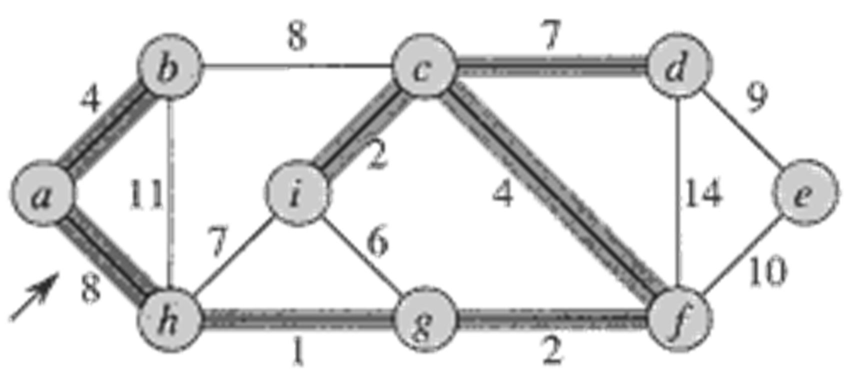
Kruskal's Algorithm: Example



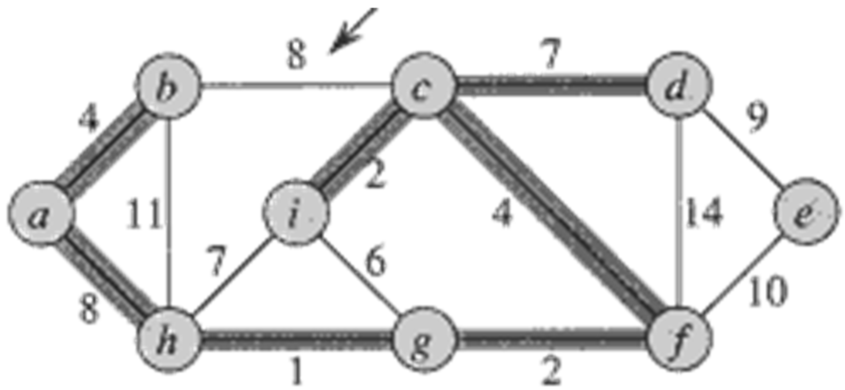
Kruskal's Algorithm: Example



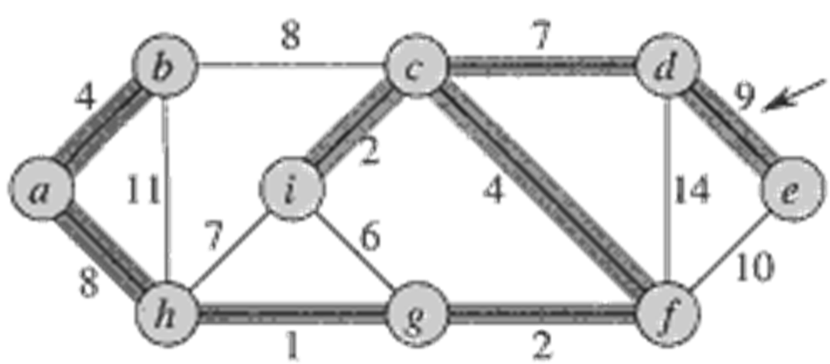
Kruskal's Algorithm: Example



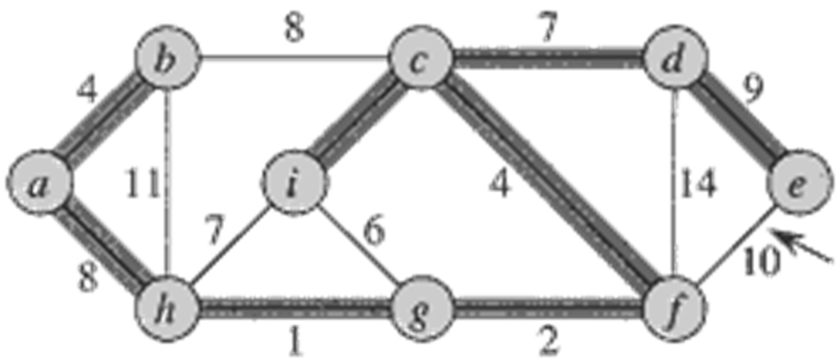
Kruskal's Algorithm: Example



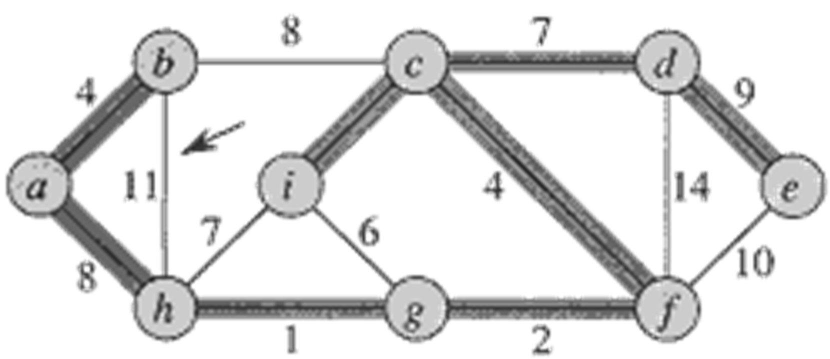
Kruskal's Algorithm: Example



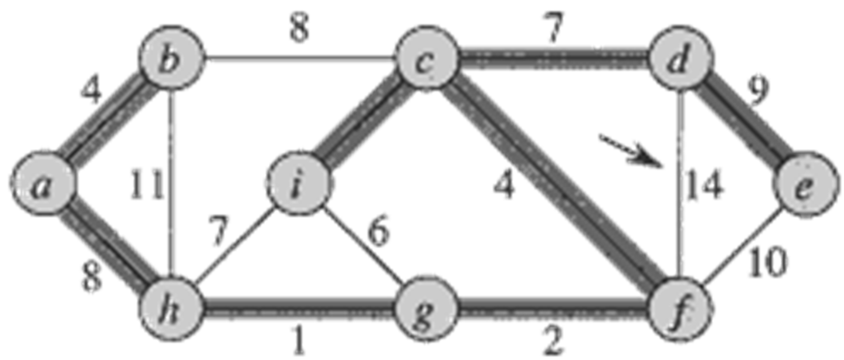
Kruskal's Algorithm: Example



Kruskal's Algorithm: Example



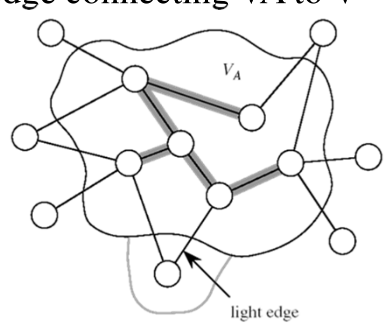
Kruskal's Algorithm: Example



Finished!

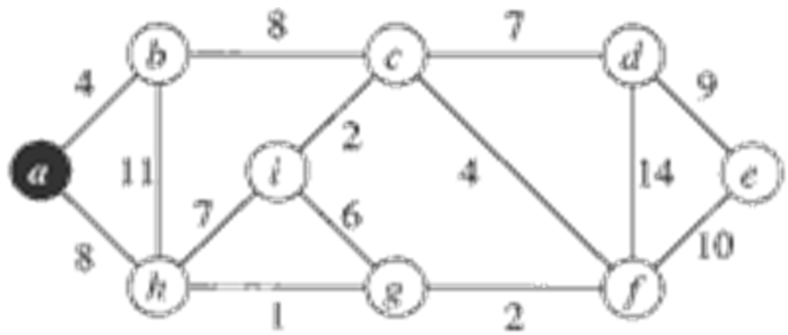
Prim's Algorithm for Computing MST

- Build one tree A
- Start from arbitrary root r
- At each step, add light edge connecting VA to V- VA (***greedy***)

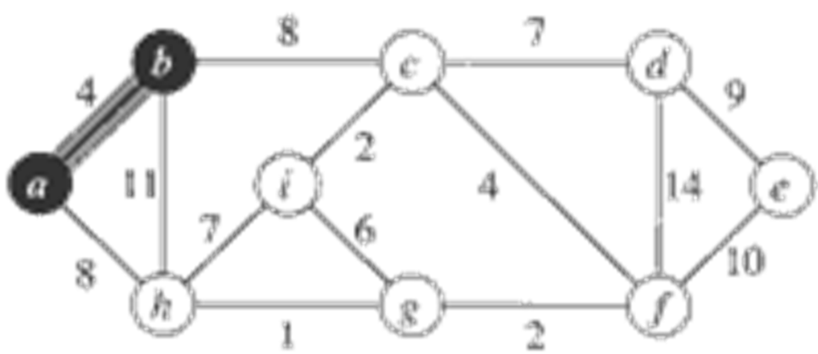


[Edges of A are shaded.]

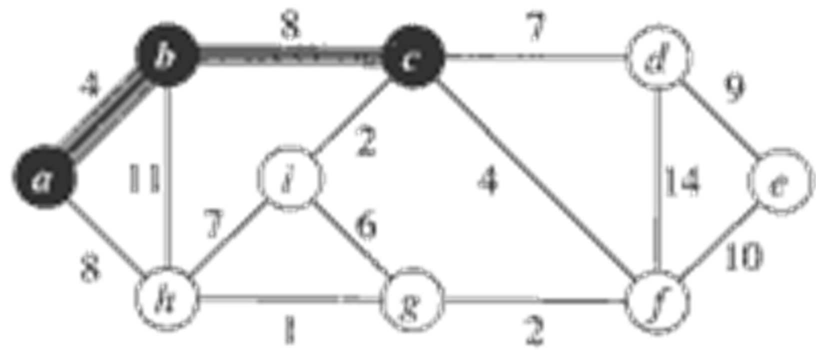
Prim's Algorithm: Example



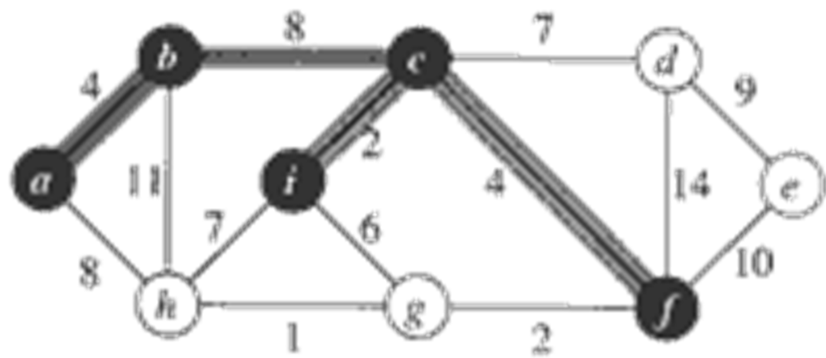
Prim's Algorithm: Example



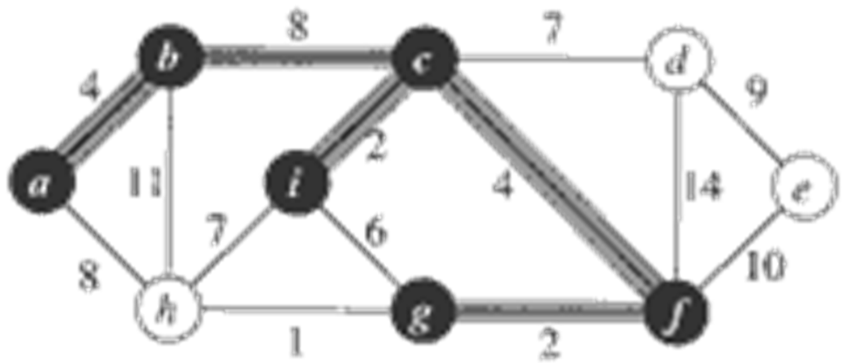
Prim's Algorithm: Example



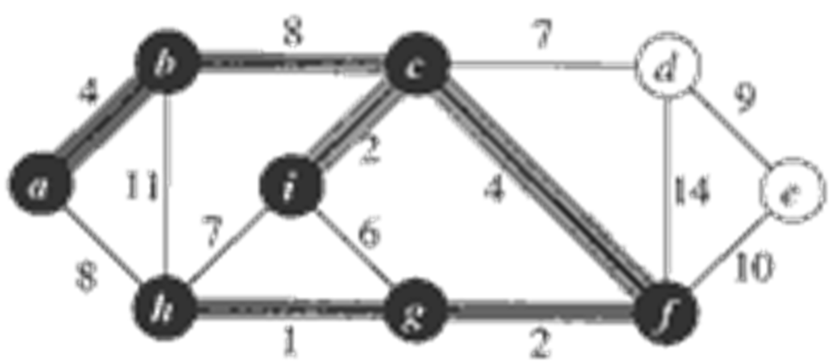
Prim's Algorithm: Example



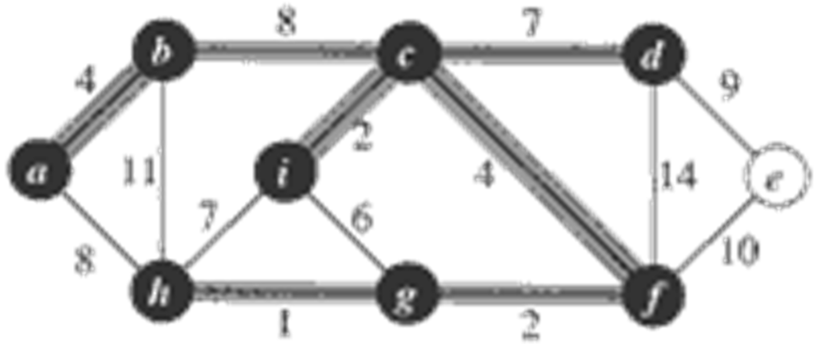
Prim's Algorithm: Example



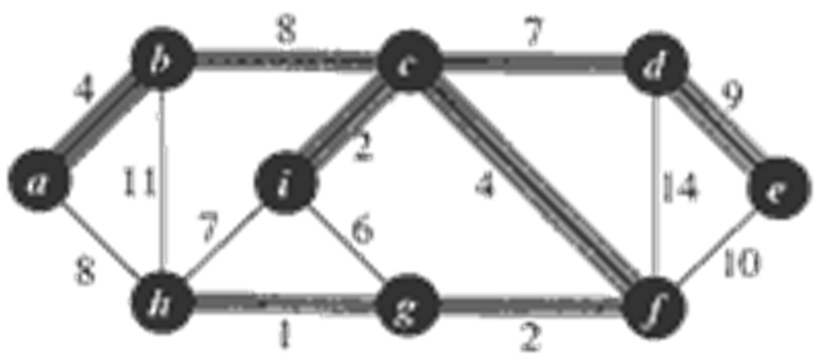
Prim's Algorithm: Example



Prim's Algorithm: Example



Prim's Algorithm: Example



Finished!

An Activity Selection Problem (Conference Scheduling Problem)

- **Input: A set of activities S = {a_1,…, a_n}**
- Each activity has start time and a finish time
  - $a_i=(s_i, f_i)$
- Two activities are compatible if and only if their interval does not overlap
- **Output: a maximum-size subset of mutually compatible activities**

The Activity Selection Problem

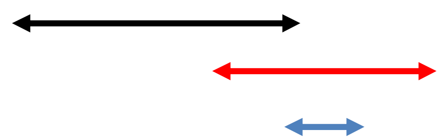
- Here are a set of start and finish times

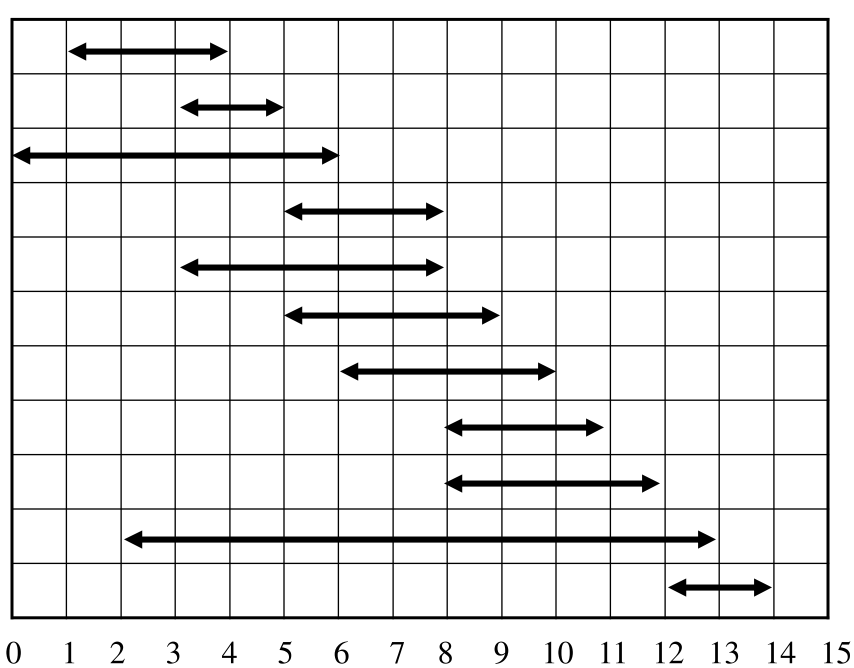
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

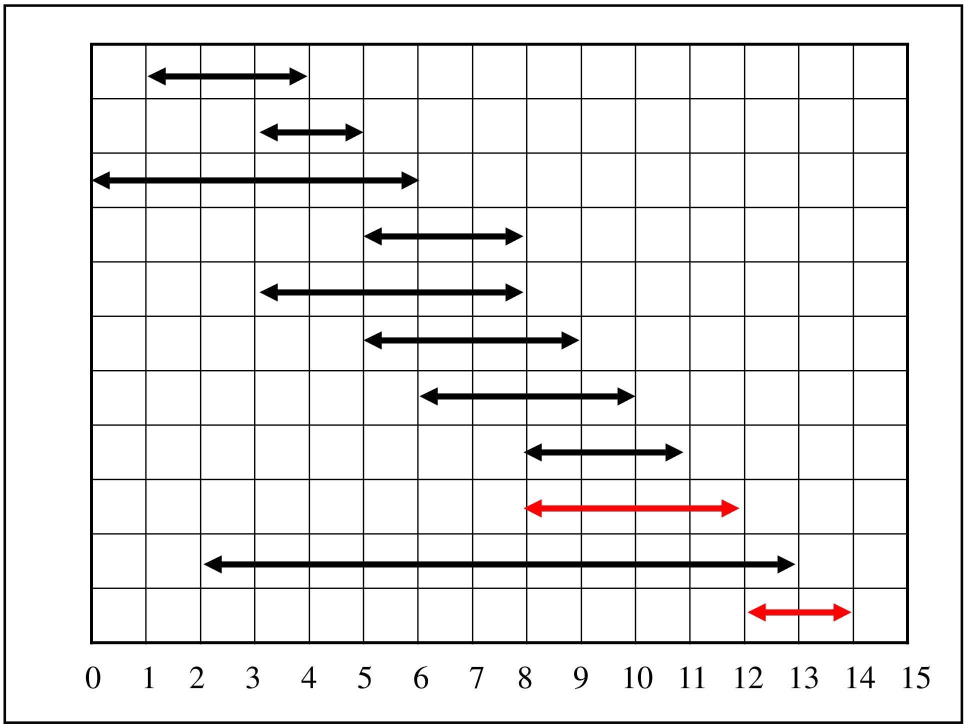
- What is the maximum number of activities that can be completed?
  - {a_3, a_9, a_{11}} can be completed
  - But so can {a_1, a_4, a_8, a_{11}} which is a larger set
  - But it is not unique, consider {a_2, a_4, a_9, a_{11}}

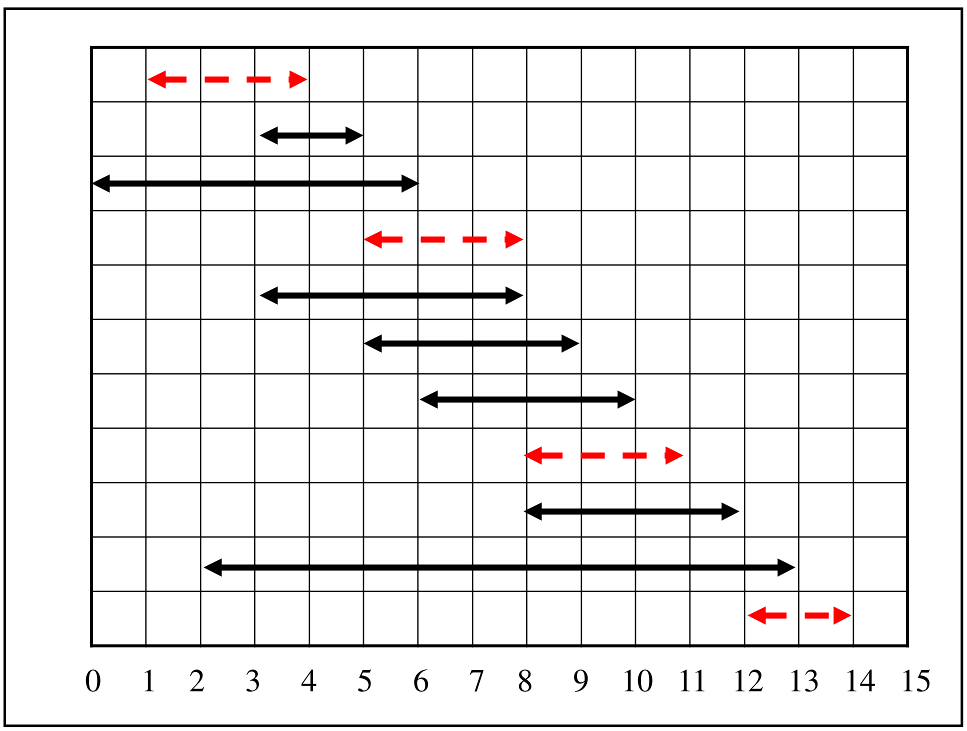
Interval Representation

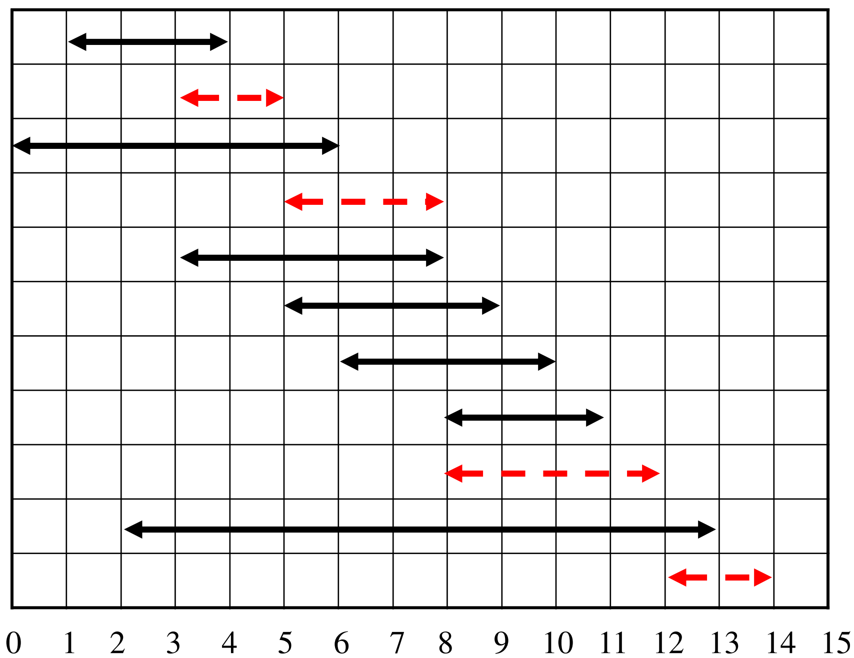
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14





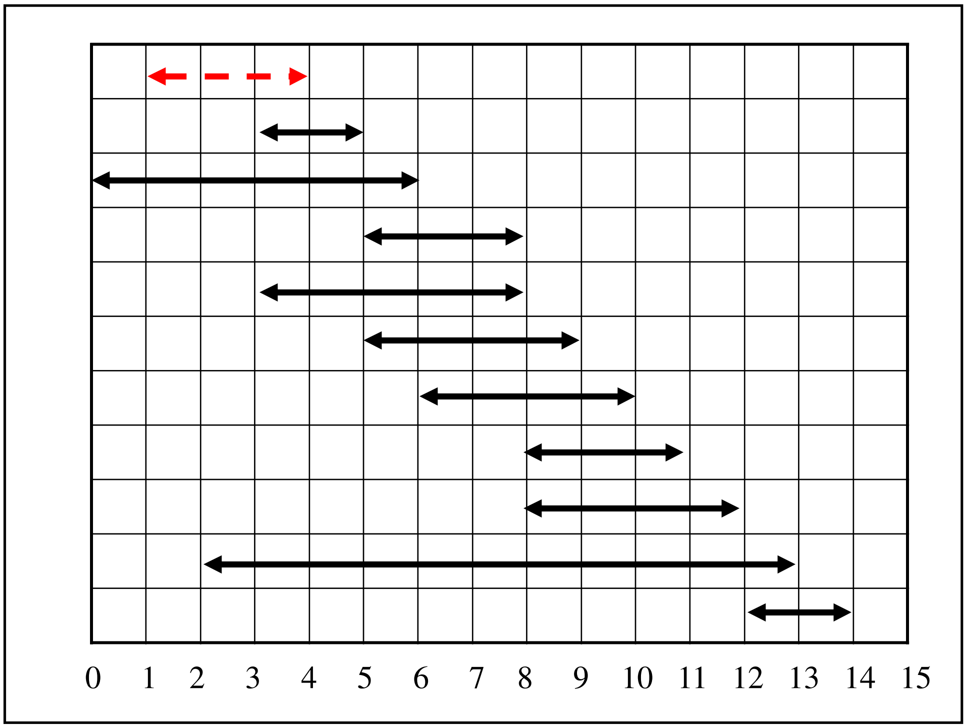

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

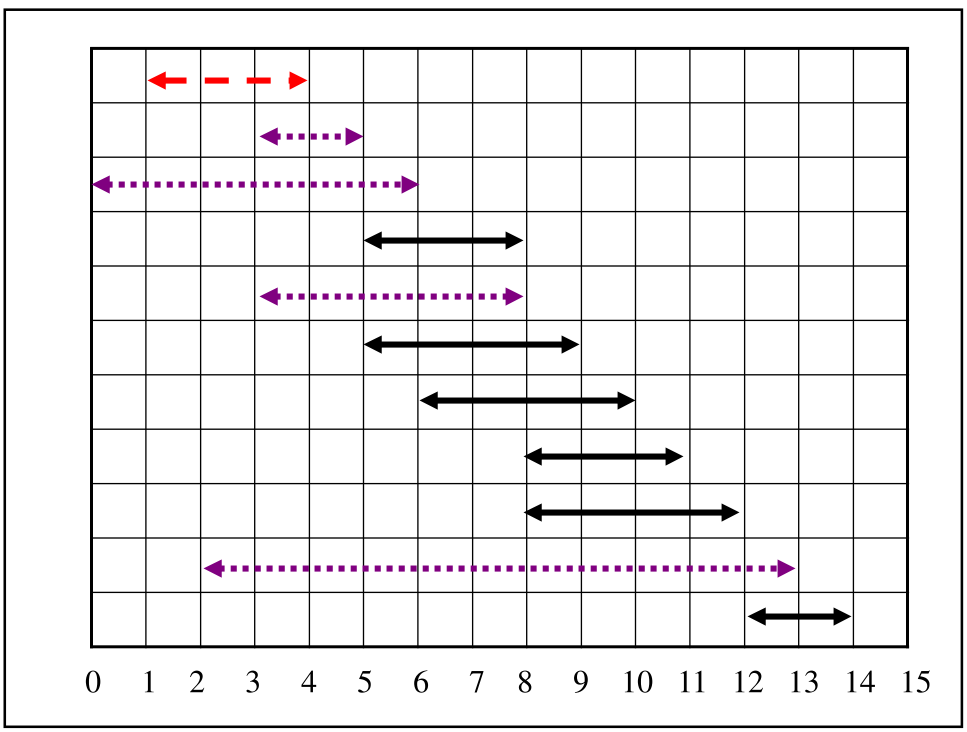

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

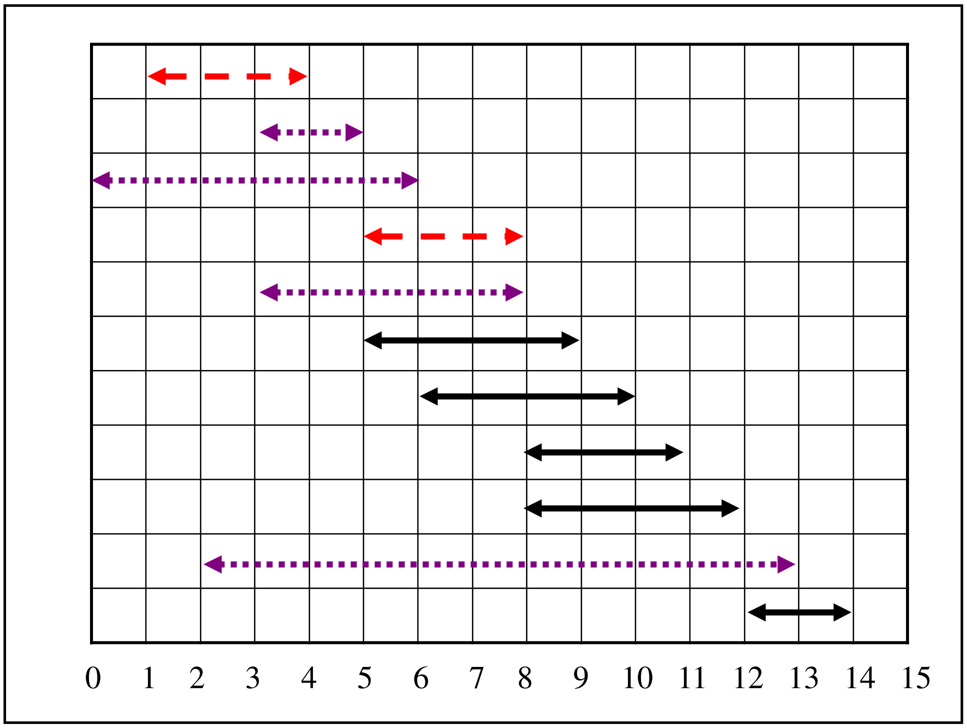


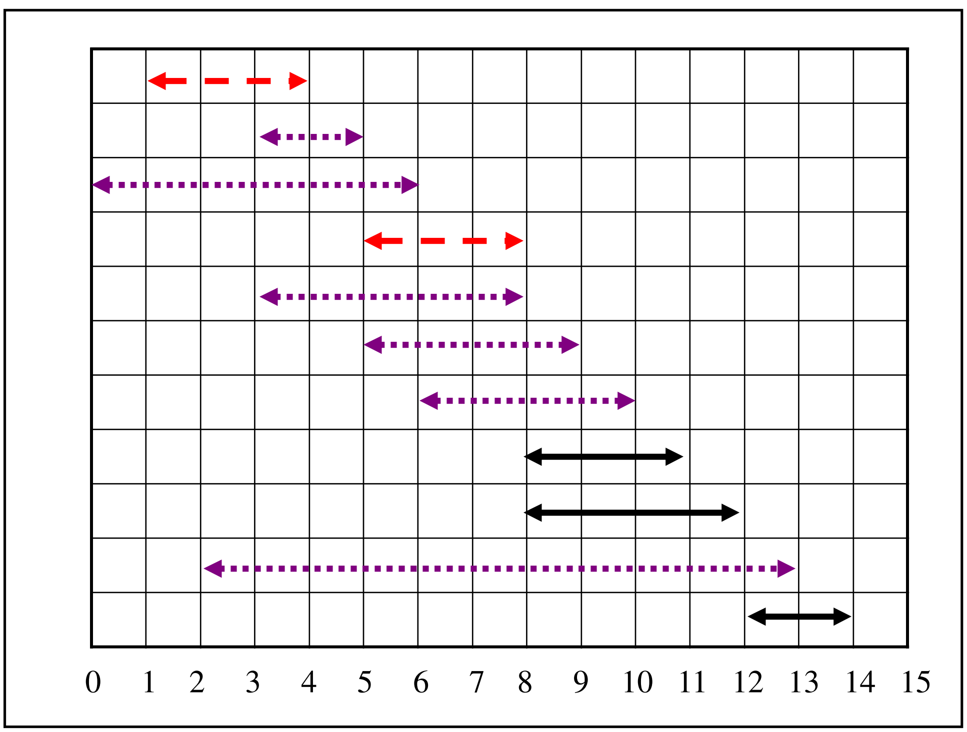
Early Finish Greedy

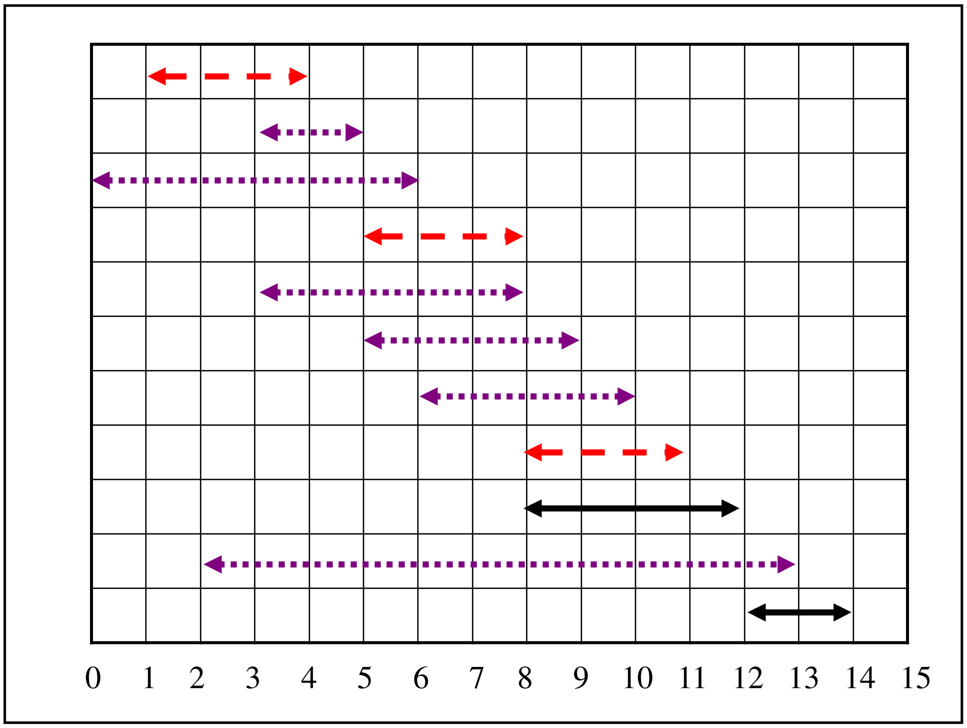
- Select the activity with the earliest finish
- Eliminate the activities that could not be scheduled
- Repeat!

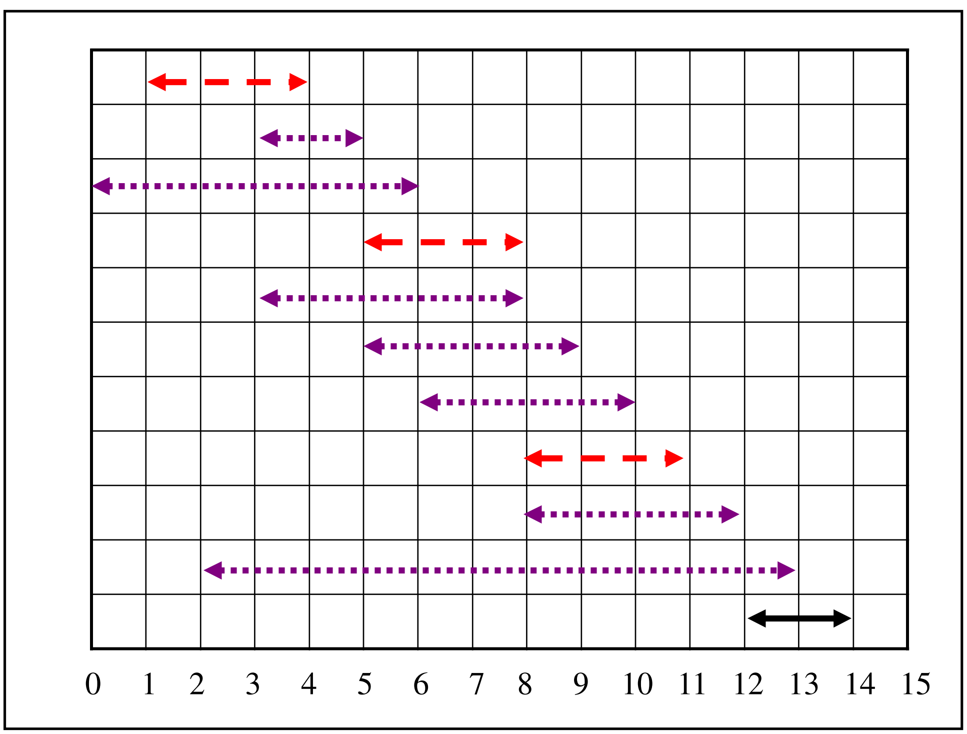

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

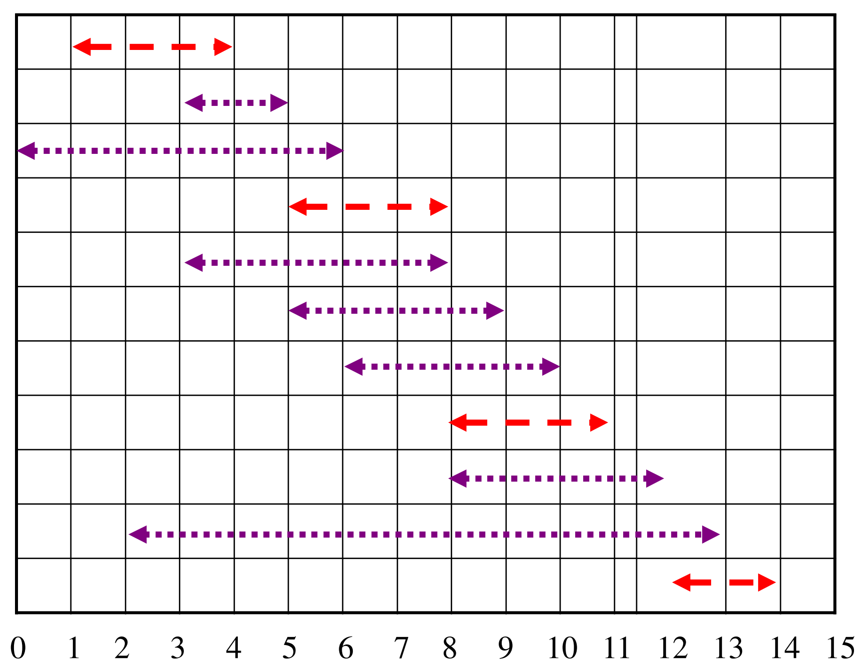

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



Why It Is Greedy?

- Greedy in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled.
- The greedy choice is the one that maximizes the amount of unscheduled time remaining.

Knapsack Problem

- There are n different items in a store
- Item i :
  - weighs w_i pounds
  - worth $\$v_i$
- A thief breaks in
- Can carry up to W 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…

Greedy Fractional Knapsack Algorithm

- Sort items in decreasing order of value per pound
- While still room in the knapsack (limit of W pounds) do
  - Consider next item in sorted list
  - Take as much as possible (all there is or as much as will fit)
- $O(n \log n)$ running time (for the sort)

Greedy 0-1 Knapsack Algorithm?

- 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
- Greedy strategy:
  - Take item 1
  - Take item 2
  - No room for item 3

suboptimal!

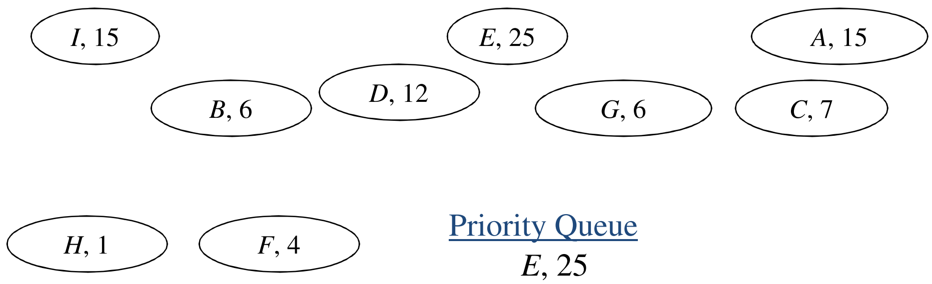
0-1 Knapsack Problem

- Taking item 1 is a big mistake globally although looks good locally
- Use dynamic programming to solve this in pseudo-polynomial time

Huffman’s Algorithm

- Huffman’s algorithm places the characters on a priority queue, removing the two least frequently appearing characters (or combination of characters), merging them and placing this node on the priority tree.
- The node is linked to the two nodes from which it came.

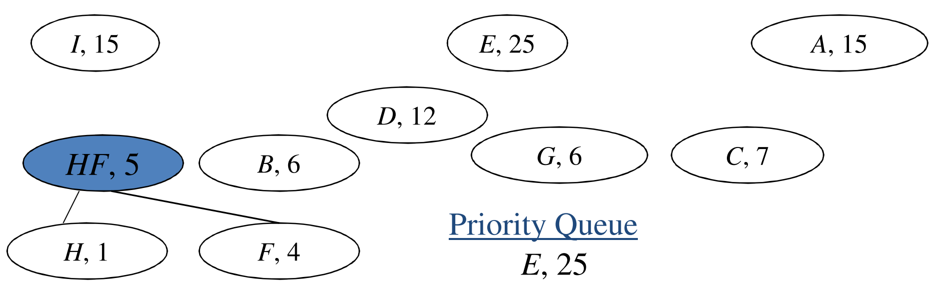
Huffman Tree



Priority Queue

E, 25
I, 15
A, 15
D, 12
C, 7
G, 6
B, 6
F, 4
H, 1

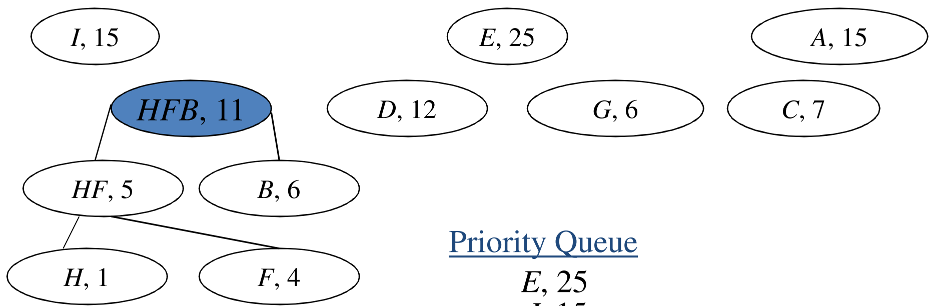
Huffman Tree



Priority Queue

E, 25
I, 15
A, 15
D, 12
C, 7
G, 6
B, 6
HF, 5

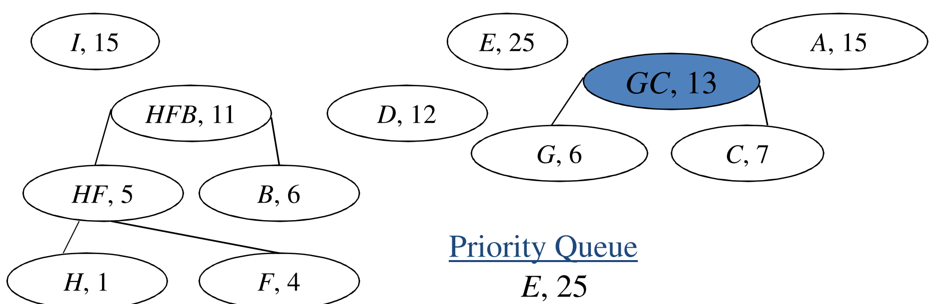
Huffman Tree



Priority Queue

E, 25
I, 15
A, 15
D, 12
HFB, 11
C, 7
G, 6

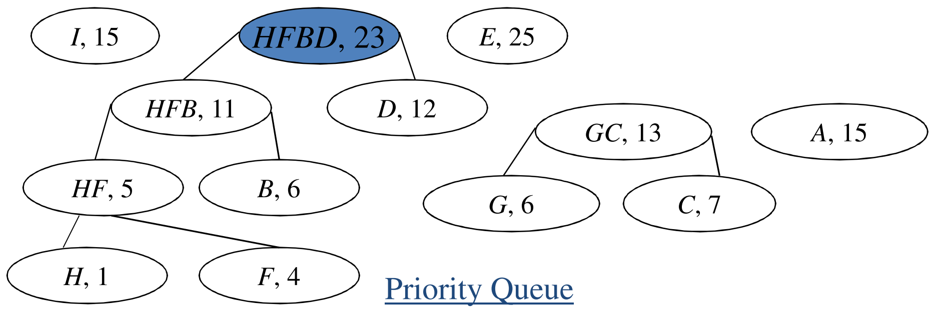
Huffman Tree



Priority Queue

E, 25
I, 15
A, 15
GC, 13
D, 12
HFB, 11

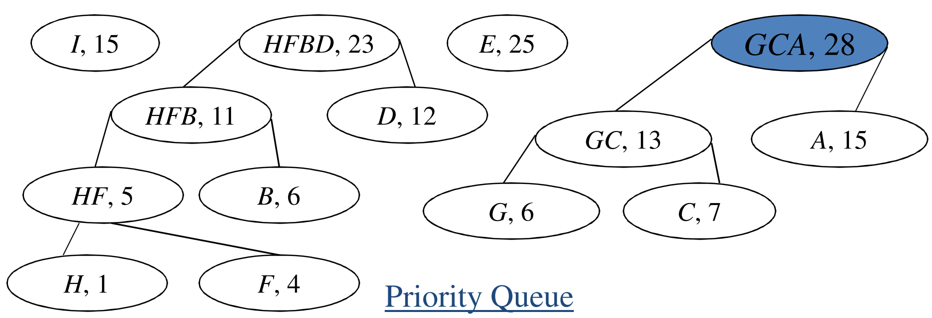
Huffman Tree



Priority Queue

E, 25
HFBD, 23
I, 15
A, 15
GC, 13

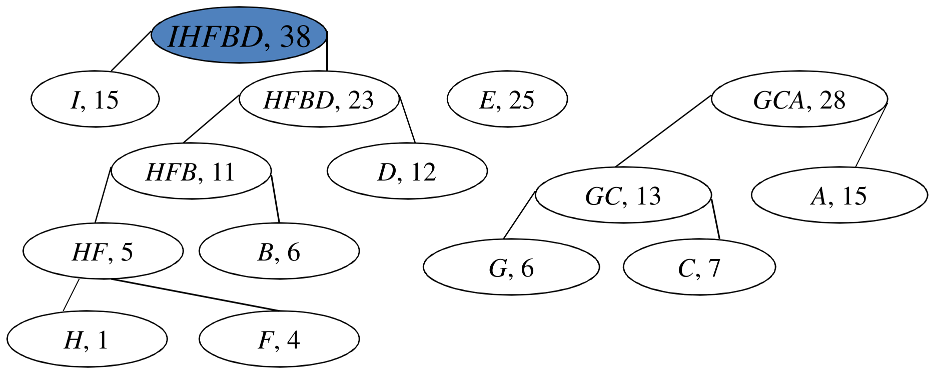
Huffman Tree



Priority Queue

GCA, 28
E, 25
HFBD, 23
I, 15

Huffman Tree



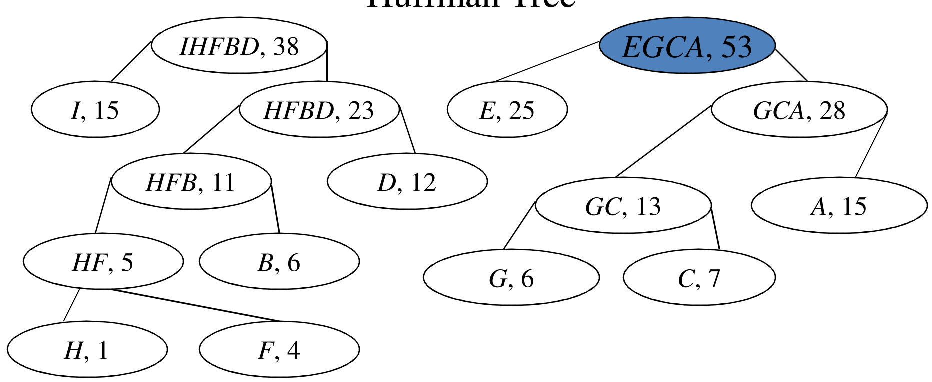
Priority Queue

IHFBD, 38

GCA, 28

E, 25

Huffman Tree

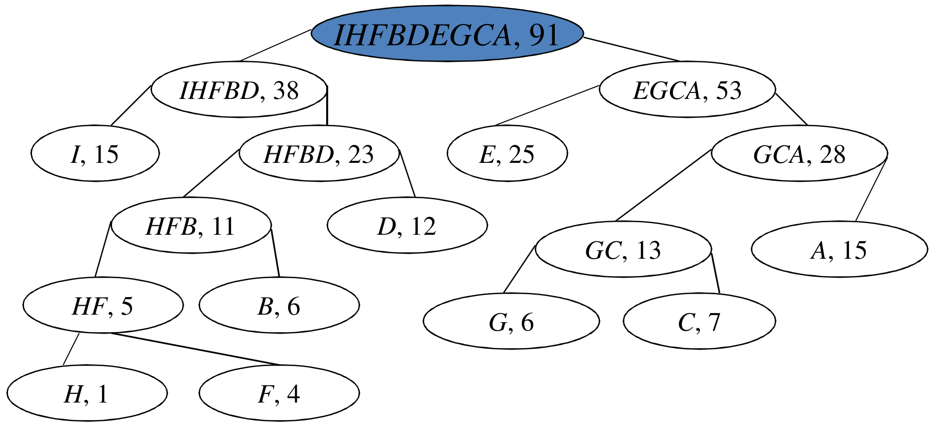


Priority Queue

EGCA, 53

IHFBD, 38

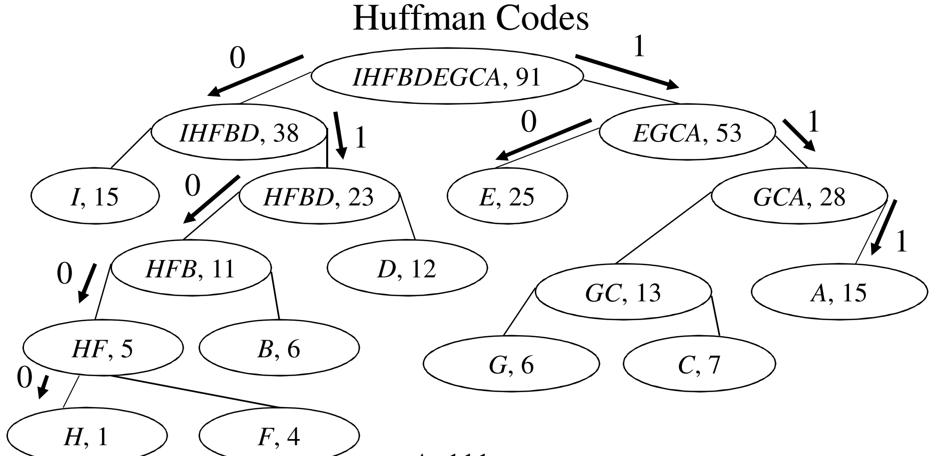
Huffman Tree



Priority Queue

IHFBDEGCA, 91

Huffman Codes



A, 111
E, 10
H, 01000