Design and Analysis of Algorithms

Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem,
  - Travelling Sales Person problem
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms,
  - P, NP, NP-Complete, and NP-Hard classes

Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem,
  - Travelling Sales Person problem
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms,
  - P, NP, NP-Complete, and NP-Hard classes

Backtracking

- Some problems can be solved, by exhaustive search.
- Backtracking is a more intelligent variation of this approach.
- The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.
  - If a partially constructed solution can be developed, continue.
  - If there is no legitimate option for the next component, **backtrack** to replace the last component of the partially constructed solution with its next option.

Backtracking

- **State-space tree,** represents the processing
- Its root represents an initial state
- The nodes of the first level in the tree represent the choices made for the first component of a solution
- The nodes of the second level represent the choices for the second component, and so on.
- A node in a state-space tree is said to be promising
  - if it corresponds to a partially constructed solution that may still lead to a complete solution;
  - otherwise, it is called nonpromising.
- Leaves represent either nonpromising dead ends

Backtrackin g

- In the majority of cases, a states-pace tree for a backtracking algorithm is constructed in the manner of depth-first search.
- If the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.

Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem,
  - Travelling Sales Person problem
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms,
  - P, NP, NP-Complete, and NP-Hard classes

N-Queens Problem

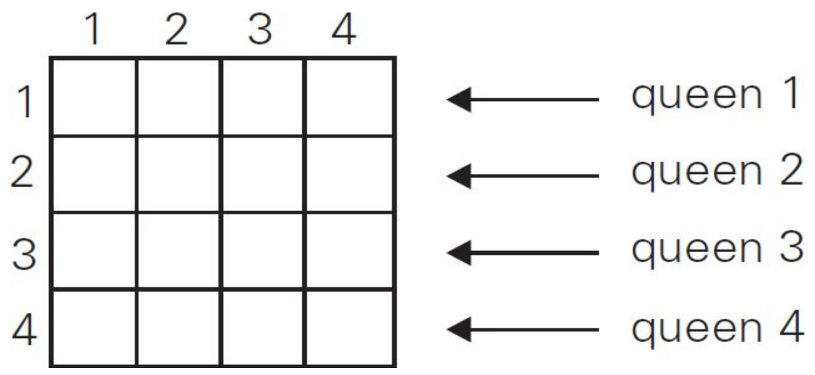
Problem Definition

- The problem is to place n queens on an **n × n** chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

- So let us consider the **4-queens problem** and solve it by the backtracking technique.

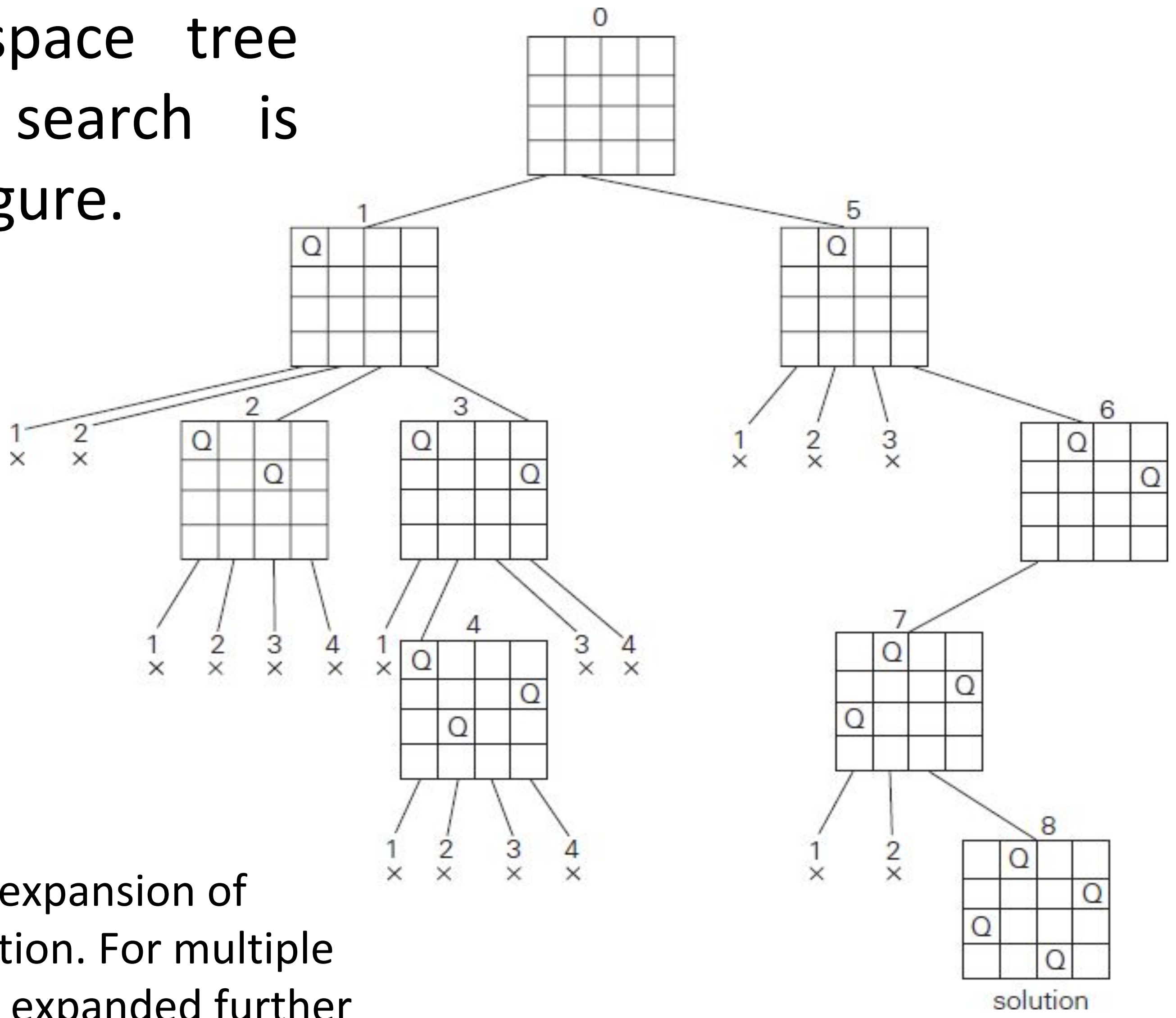
4-Queens problem

- Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board presented in figure.



4-Queens problem

- The state-space tree of this search is shown in figure.



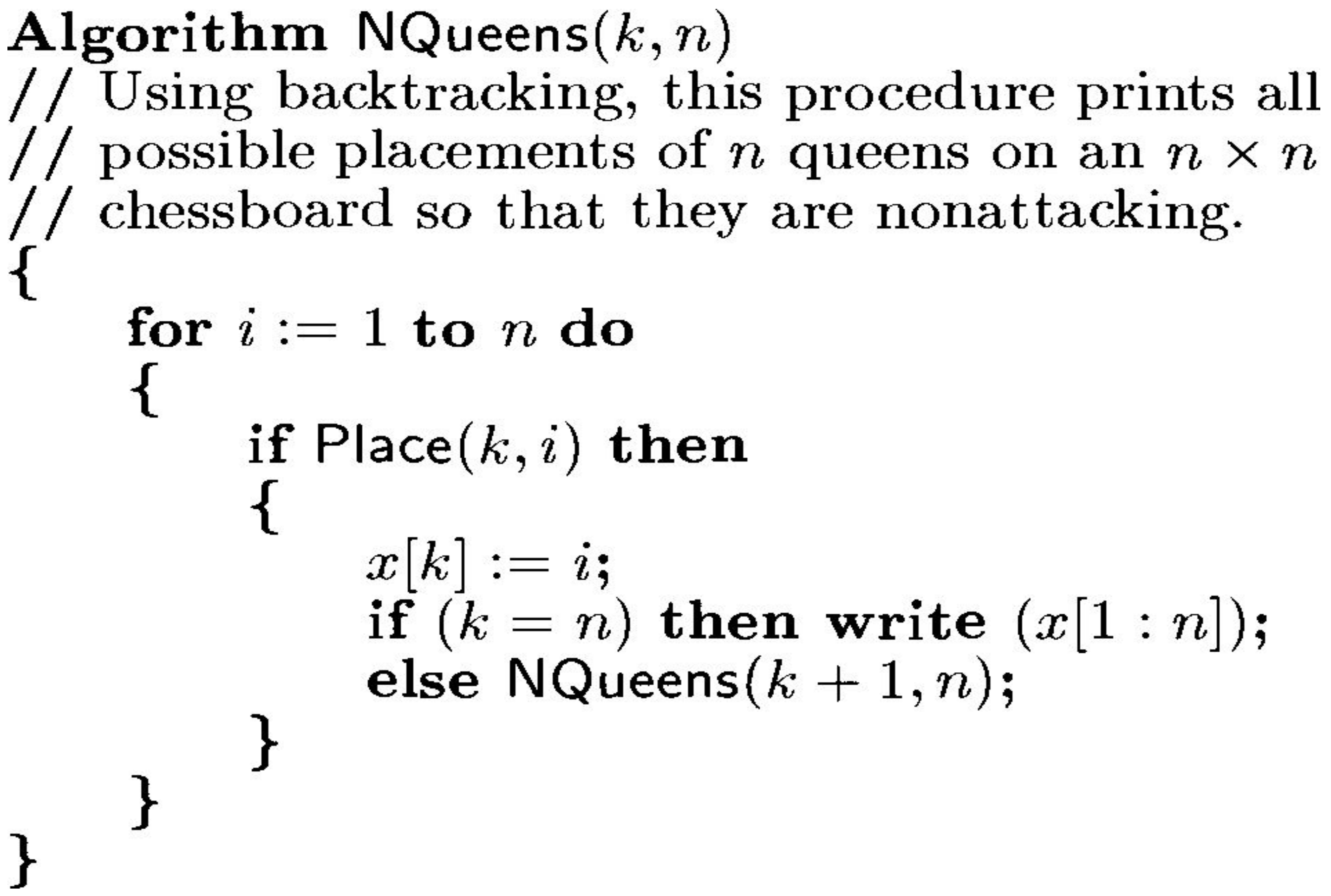
Note: This tree shows expansion of nodes till it gets a solution. For multiple solutions it need to be expanded further

4-queens problem

- If other solutions need to be found, the algorithm can simply resume its operations at the leaf at which it stopped.
- Alternatively, we can use the board's symmetry for this purpose.
- Finally, it should be pointed out that a single solution to the n-queens problem for any $n \geq 4$ can be found in **linear time**.

N-queens prblem

Algorithm to find all solutions of n-queens problem



```
Algorithm NQueens($k, n$)
// Using backtracking, this procedure prints all
// possible placements of $n$ queens on an $n \times n$
// chessboard so that they are nonattacking.
{
    for $i := 1$ to $n$ do
    {
        if Place($k, i$) then
        {
            $x[k] := i$;
            if $(k = n)$ then write $(x[1 : n])$;
            else NQueens($k + 1, n$);
        }
    }
}
```

N-queens prblem

```
Algorithm Place($k, i$)
// Returns true if a queen can be placed in $k$th row and
// $i$th column. Otherwise it returns false. $x[\,]$ is a
// global array whose first $(k-1)$ values have been set.
// Abs($r$) returns the absolute value of $r$.
{
    for $j := 1$ to $k-1$ do
        if (($x[j] = i$) // Two in the same column
            or (Abs($x[j] - i$) = Abs($j - k$)))
                // or in the same diagonal
            then return false;
    return true;
}
```

Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem,
  - Travelling Sales Person problem
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms,
  - P, NP, NP-Complete, and NP-Hard classes

Sum of Subsets Problem

Problem definition

- Find a subset of a given set $A = \{a_1, . . . , a_n\}$ of n positive integers whose sum is equal to a given positive integer d.

Example

- For A = {1, 2, 5, 6, 8} and d = 9, there are two solutions: {1, 2, 6} and {1, 8}.
- Of course, some instances of this problem may have no solutions.
- State space tree is shown in next slide

Sum of Subsets Problem

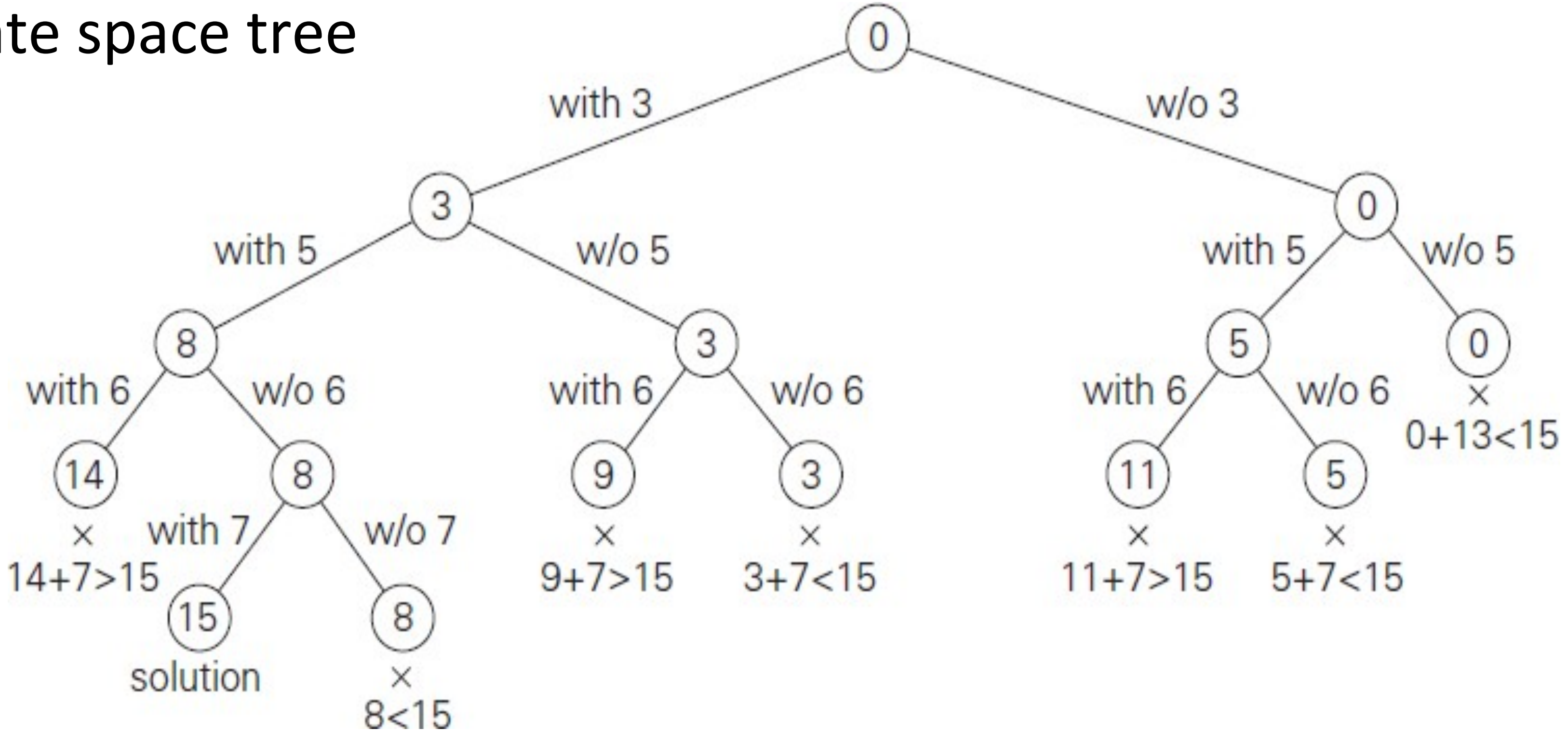
- It is convenient to sort the set's elements in increasing order.
- In the state space tree at a node, If sum is not equal to d, we can terminate the node as non-promising if either of the following two inequalities holds:

$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large),}$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small).}$$

Sum of Subsets Problem

- State space tree



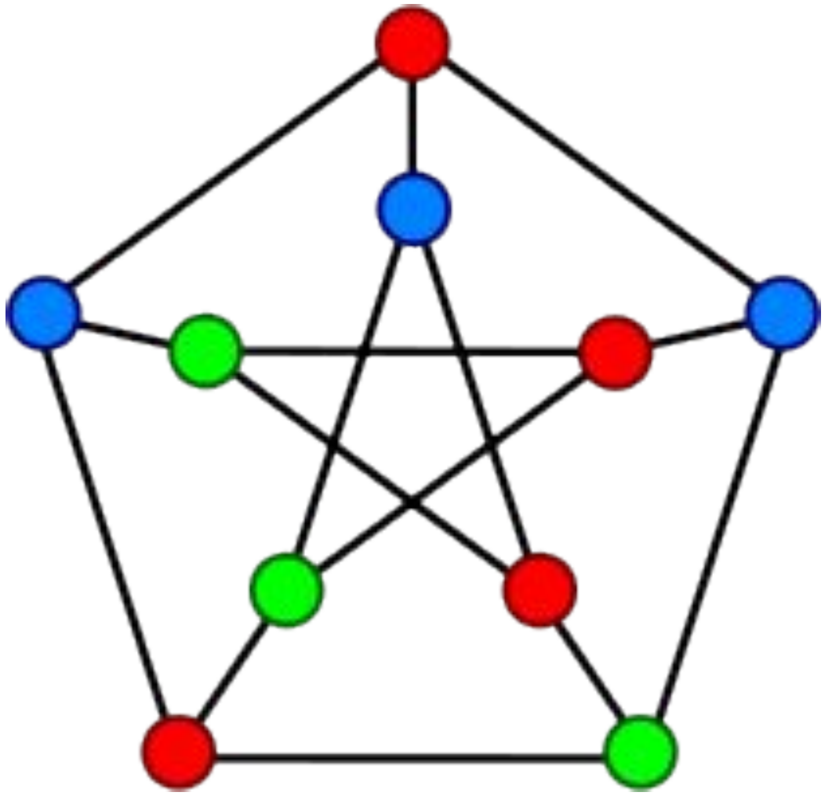
$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large),}$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small).}$$

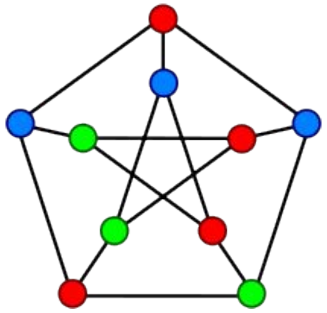
Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem,
  - Travelling Sales Person problem
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms,
  - P, NP, NP-Complete, and NP-Hard classes

Graph Coloring



Graph Coloring



- **Problem Statement**
  - Given an undirected graph and a number m,
  - determine if the graph can be colored with at most m colors
  - such that no two adjacent vertices of the graph are colored with same color.
- Problem is also called as *m-colorability decision problem*
- Here coloring of a graph means assignment of colors to all vertices.

Graph Coloring

Applications

To make exam schedule for a university

- This problem can be represented as a graph where every vertex is a subject and an edge between two vertices mean there is a common student.
- So this is a graph coloring problem where minimum number of time slots is equal to the chromatic number of the graph.

Graph Coloring

Applications

Mobile Radio Frequency Assignment

- When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different.
- How to assign frequencies with this constraint? What is the minimum number of frequencies needed?
- This problem is also an instance of graph coloring problem where every tower represents a vertex and an edge between two towers represents that they are in range of each other.

Graph Coloring

Applications

Map Coloring

- Geographical maps of countries or states where no two adjacent cities cannot be assigned same color.
- Four colors are sufficient to color any map (really?)

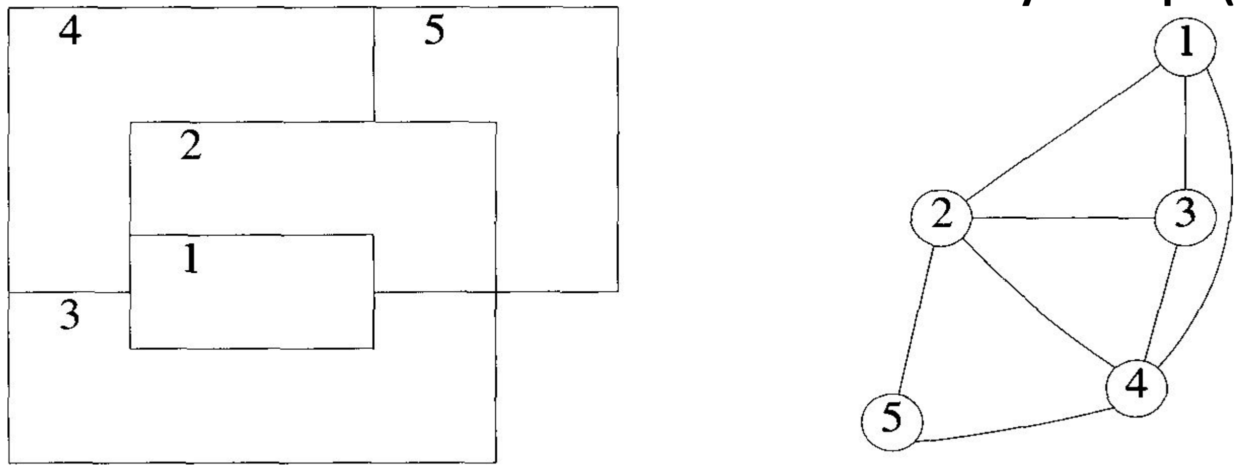
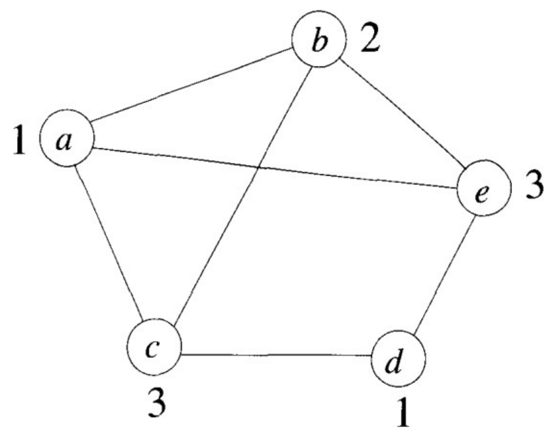


Figure 7.12 A map and its planar graph representation

Graph Coloring

- If the degree of the graph is d, it can be colored with d+1 colors
- *m-colorability optimization* problem finds the smallest integer m, for which the graph G can be colored.
- This integer is referred as the chromatic number of the graph.
- Example: 3 colors sufficient for the graph given below



Graph coloring

- Represent the graph by adjacency matrix G[1:n,1:n]
- Colors represented by 1,2,3 … m
- Solutions are given by **n-tuple (x_1,x_2,….. X_n),** x_i is the color of node i

k - Index of the vertex to color

Algorithm

X[1:m] is initialized to zeros

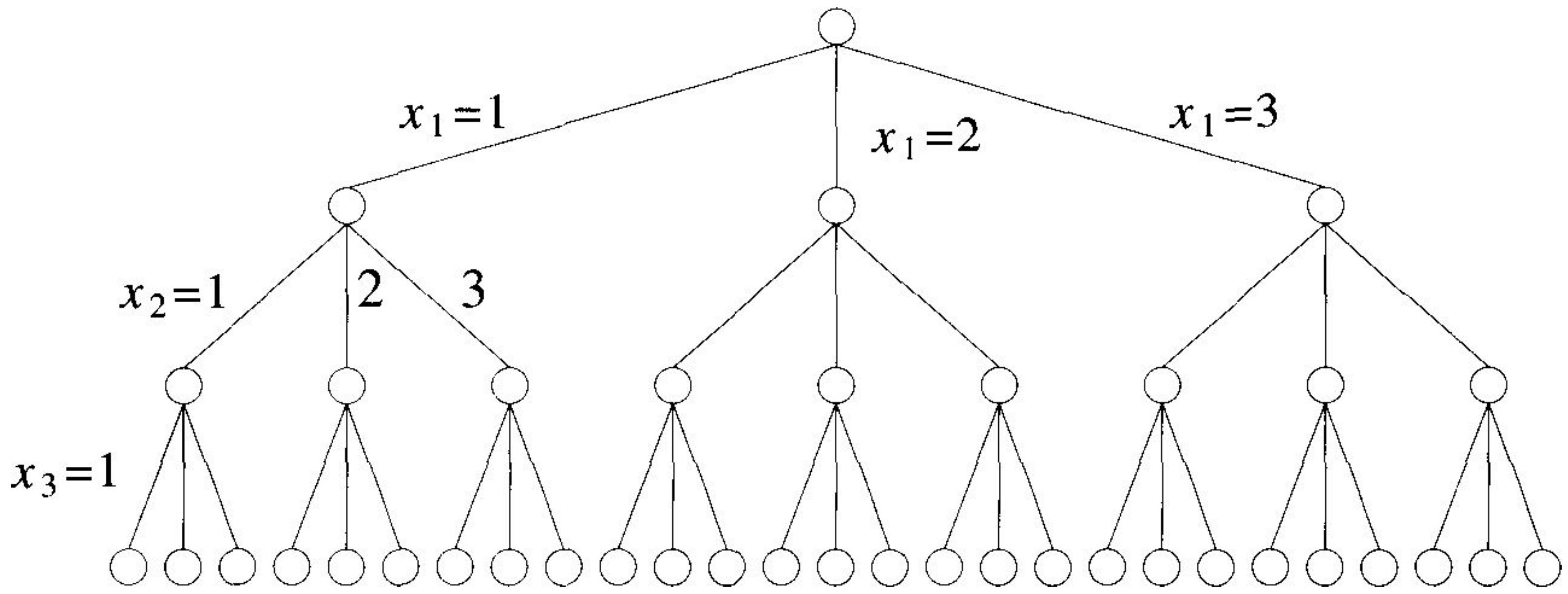
```
Algorithm mColoring(k)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. k is the index
// of the next vertex to color.
{
    repeat
    {// Generate all legal assignments for x[k].
        NextValue(k); // Assign to x[k] a legal color.
        if (x[k] = 0) then return; // No new color possible
        if (k = n) then        // At most m colors have been
                               // used to color the n vertices.
            write (x[1 : n]);
        else mColoring(k + 1);
    } until (false);
}
```

X[ ] is initialized to zeros

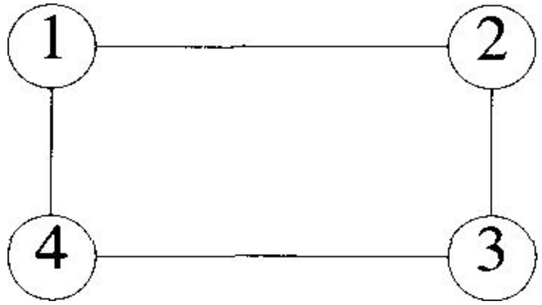
```
Algorithm NextValue($k$)
// $x[1],\ldots,x[k-1]$ have been assigned integer values in
// the range $[1,m]$ such that adjacent vertices have distinct
// integers. A value for $x[k]$ is determined in the range
// $[0,m]$. $x[k]$ is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex $k$. If no such color exists, then $x[k]$ is 0.
{
    repeat
    {
        $x[k] := (x[k]+1)$ mod $(m+1)$; // Next highest color.
        if $(x[k] = 0)$ then return; // All colors have been used.
        for $j := 1$ to $n$ do
        {   // Check if this color is
            // distinct from adjacent colors.
            if $((G[k,j] \neq 0)$ and $(x[k] = x[j]))$
            // If $(k,j)$ is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if $(j = n+1)$ then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

Logic: For a vertex, start coloring from 1, if the adjacent nodes have same color keep incrementing till m.

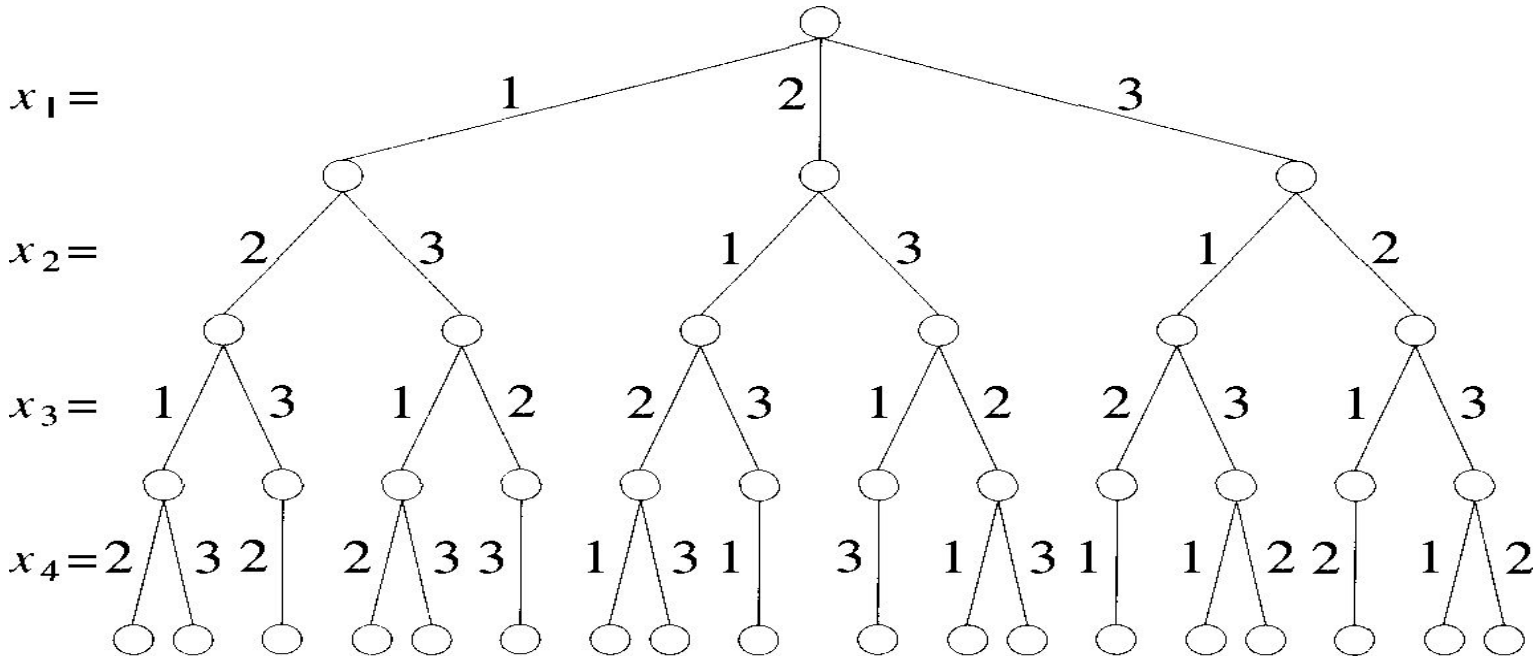
The color of vertex k & j are same



State space tree for mColoring when n =3 and m = 3



This state space tree shows all the the successful coloring (promising nodes leading to the solution)



A 4-node graph and all possible 3-colorings

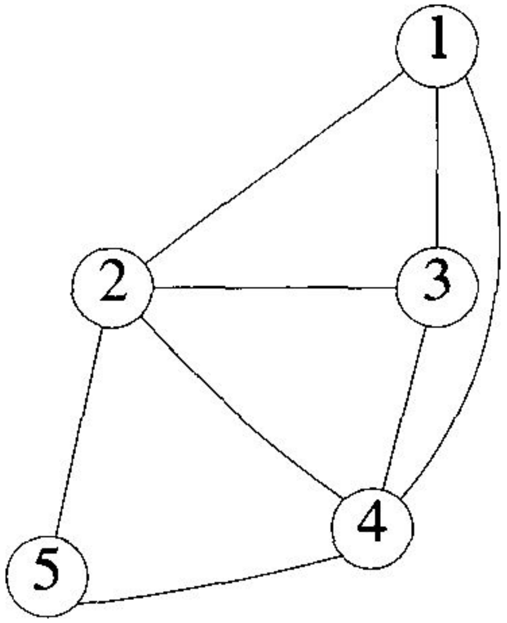
Analysis

An upper bound on the computing time of mColoring can be arrived at by noticing that the number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$. At each internal node, $O(mn)$ time is spent by NextValue to determine the children corresponding to legal colorings. Hence the total time is bounded by $\sum_{i=0}^{n-1} m^{i+1}n = \sum_{i=1}^{n} m^i n = n(m^{n+1} - 2)/(m - 1) = \boxed{O(nm^n).}$

m – degree of the graph

Problem

- Apply graph coloring algorithm for the graph given below. Assume m=3



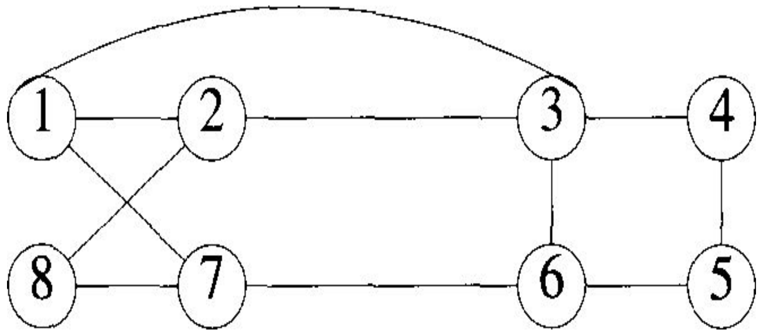
- Solve the same problem with m=4

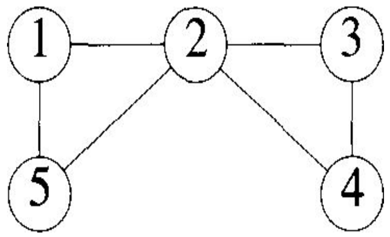
Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem,
  - Travelling Sales Person problem
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms,
  - P, NP, NP-Complete, and NP-Hard classes

Hamiltonian cycle

Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle (suggested by Sir William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other words if a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices



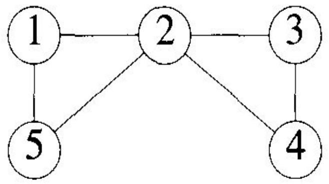


Hamiltonian Cycle

- TSP tour is a Hamiltonian cycle.
- Here we discuss, Backtracking algorithm that finds all distinct cycles
- Solution will be in the form $(x_1, x_2, \ldots\ldots, x_n)$

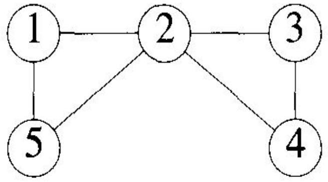
Hamiltonian Cycle

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1 : n, 1 : n]. All cycles begin at node 1.
{
    repeat
    { // Generate values for x[k].
        NextValue(k); // Assign a legal next value to x[k].
        if (x[k] = 0) then return;
        if (k = n) then write (x[1 : n]);
        else Hamiltonian(k + 1);
    } until (false);
}
```



X[1:n] is initialized to zeros

```
**Algorithm** NextValue($k$)
// $x[1:k-1]$ is a path of $k-1$ distinct vertices. If $x[k]=0$, then
// no vertex has as yet been assigned to $x[k]$. After execution,
// $x[k]$ is assigned to the next highest numbered vertex which
// does not already appear in $x[1:k-1]$ and is connected by
// an edge to $x[k-1]$. Otherwise $x[k]=0$. If $k=n$, then
// in addition $x[k]$ is connected to $x[1]$.
{
    repeat
    {
        $x[k] := (x[k]+1)$ mod $(n+1)$; // Next vertex.
        if $(x[k]=0)$ then return;
        if $(G[x[k-1], x[k]] \neq 0)$ then
        { // Is there an edge?
            for $j := 1$ to $k-1$ do if $(x[j] = x[k])$ then break;
                            // Check for distinctness.
            if $(j = k)$ then // If true, then the vertex is distinct.
                if $((k < n)$ or $((k = n)$ and $G[x[n], x[1]] \neq 0))$
                    then return;
        }
    } until (false);
}
```



Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem
  - Travelling Sales Person problem
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms
  - P, NP, NP-Complete & NP-Hard classes

Branch and Bound

- Backtracking - cut off a branch of the problem's state-space tree as soon as we can deduce that it cannot lead to a solution
- This idea can be strengthened further if we deal with an optimization problem.
- An optimization problem seeks to minimize or maximize some objective function like
  - a tour length – in TSP,
  - the value of items selected – in knapsack

  subject to some constraints.
- An optimal solution is a feasible solution with the best value of the objective function

Branch and Bound

Compared to backtracking, branch-and-bound requires two additional items:

1. a way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution,

   that can be obtained by adding further components to the partially constructed solution represented by the node
2. the value of the best solution seen so far

Branch and Bound

In general, we terminate a search path for any one of the following three reasons:

1. The value of the node's bound is not better than the value of the best solution seen so far.
2. The node represents no feasible solutions because the constraints of the problem are already violated.
3. The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)

Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem
  - Travelling Sales Person problem
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms
  - P, NP, NP-Complete & NP-Hard classes

Assignment Problem

Problem Definition

- Assigning **n people** to **n jobs** so that the total cost of the assignment is as small as possible.

- An instance of the assignment problem is specified by an n × n cost matrix **C**
- We can state the problem as follows:

  "select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible"

Example

$$C = \begin{array}{c} \\ \\ \\ \\ \end{array}\begin{array}{cccc|c} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} & \\ 9 & 2 & 7 & 8 & \text{person } a \\ 6 & 4 & 3 & 7 & \text{person } b \\ 5 & 8 & 1 & 8 & \text{person } c \\ 7 & 6 & 9 & 4 & \text{person } d \end{array}$$

- How can we find a lower bound on the cost of an optimal selection without actually solving the problem?

- We can do this by several methods.
  - For example, optimal solution cannot be smaller than the sum of the smallest elements in each of the matrix's rows.
  - For the instance here, this sum is 2 + 3+ 1+ 4 = 10.

$$C = \begin{array}{cccccc} & \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} & \\ & 9 & 2 & 7 & 8 & \text{person } a \\ & 6 & 4 & 3 & 7 & \text{person } b \\ & 5 & 8 & 1 & 8 & \text{person } c \\ & 7 & 6 & 9 & 4 & \text{person } d \end{array}$$

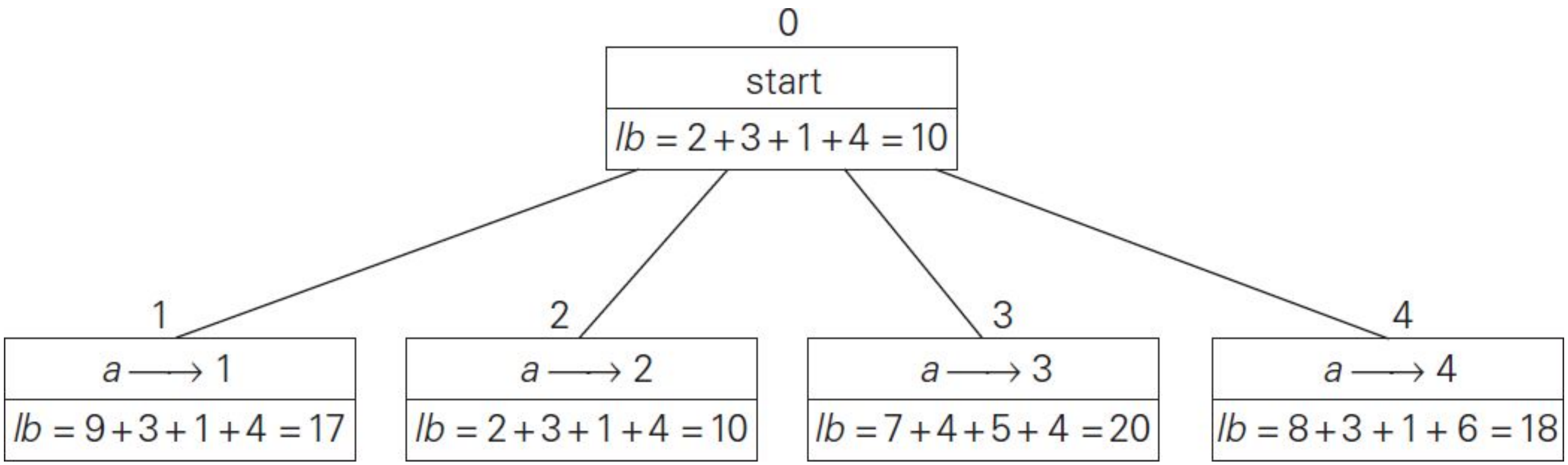
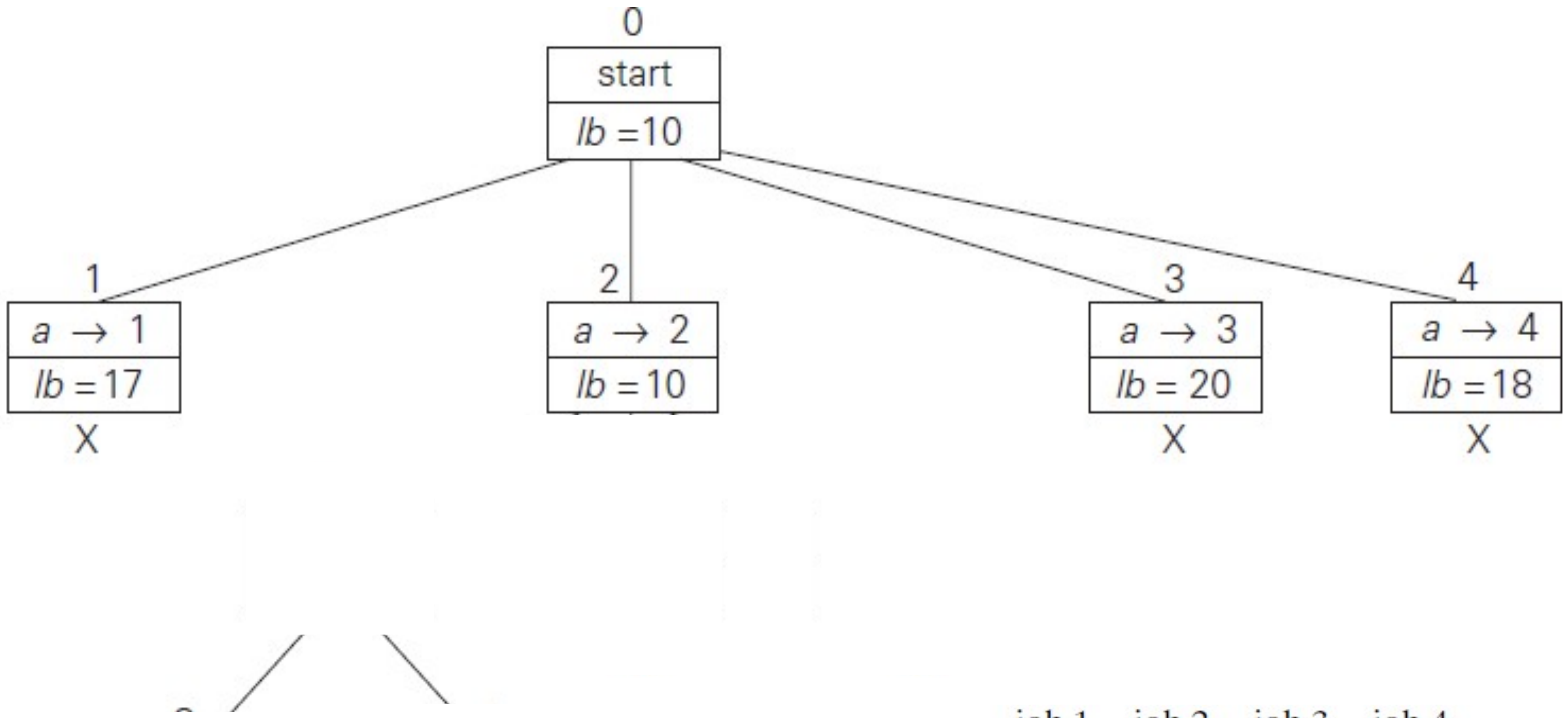


Figure: Lower bound computation

Note: Avoid the assigned jobs column for computation of lb, as it is not feasible

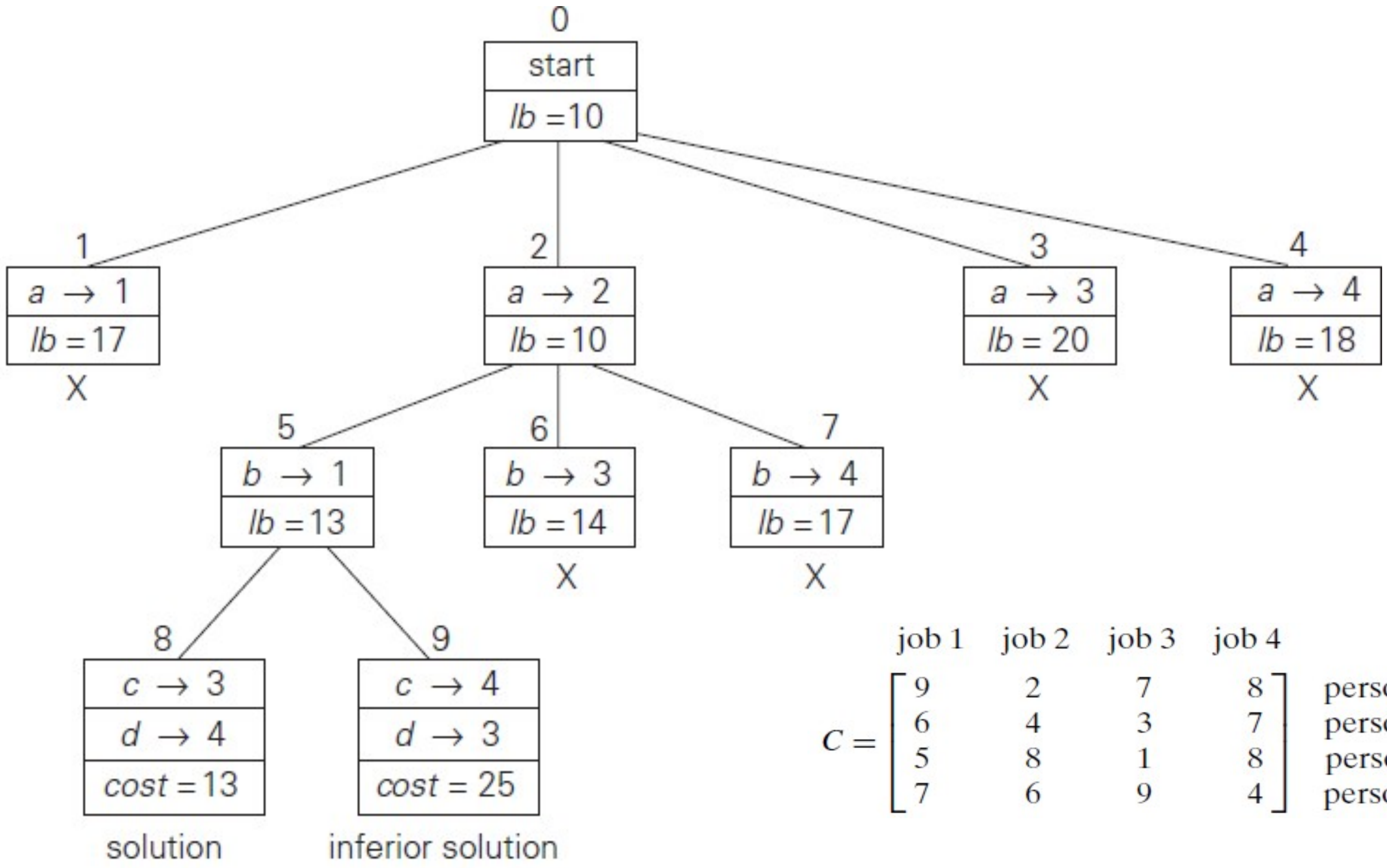
- Rather than generating a single child of the last promising node as we did in backtracking, we will generate **all the children of the most promising node** among non-terminated leaves in the current tree.
- How can we tell which of the nodes is most promising?
  - We can do this by comparing the lower bounds of the live nodes.
  - It is sensible to consider a node with the best bound as most promising (as of now)
- This variation of the strategy is called the **best-first branch-and-bound**.



$$C = \begin{array}{c|cccc|c} & \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} & \\ & 9 & 2 & 7 & 8 & \text{person } a \\ & 6 & 4 & 3 & 7 & \text{person } b \\ & 5 & 8 & 1 & 8 & \text{person } c \\ & 7 & 6 & 9 & 4 & \text{person } d \end{array}$$

FIGURE 12.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

Note: After Computation of 9th Node, We have to explore nodes that have better ***lb*** than 13. In this example we do not have such nodes. So we stop at 9.



$$C = \begin{array}{c|cccc|l} & \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} & \\ & 9 & 2 & 7 & 8 & \text{person } a \\ & 6 & 4 & 3 & 7 & \text{person } b \\ & 5 & 8 & 1 & 8 & \text{person } c \\ & 7 & 6 & 9 & 4 & \text{person } d \end{array}$$

FIGURE 12.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

Note: After Computation of 9th Node, We have to explore nodes that have better ***lb*** than 13. In this example we do not have such nodes. So we stop at 9.

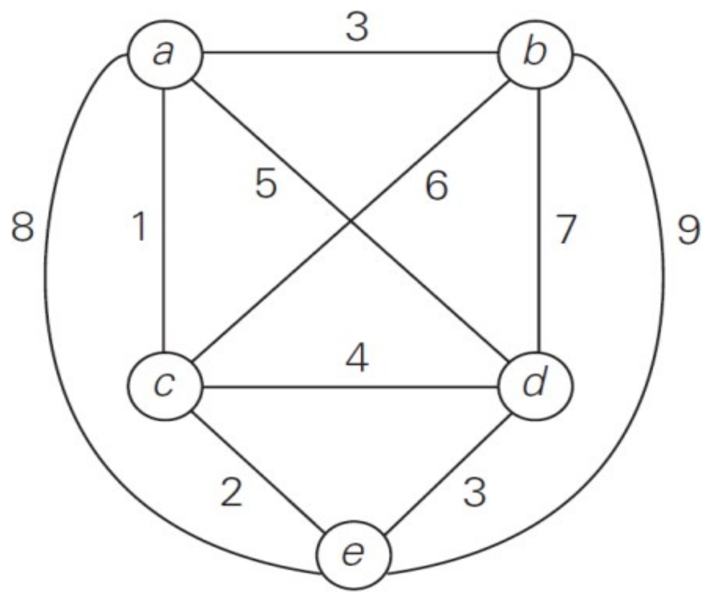
Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem
  - Travelling Sales Person problem
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms
  - P, NP, NP-Complete & NP-Hard classes

Branch & Bound - TSP

- We will be able to apply the branch-and-bound technique to instances of the TSP if we come up with a reasonable lower bound on tour lengths.
- One very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix D and multiplying it by the number of cities n.
- But there is a less obvious and more informative lower bound for instances with symmetric matrix D
- For each city i, 1≤ i ≤ n,
  - Find s_i – sum of the distances from city i to the 2 nearest cities;
  - compute **s**, the sum of $\mathbf{s_i}$ i=1..n, divide the result by 2

Branch & Bound – TSP - Example

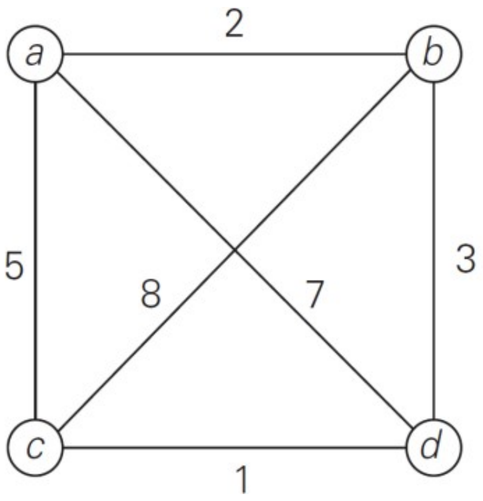


$$lb = \lceil[(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2\rceil = 14.$$

Branch & Bound - TSP

To reduce the amount of potential work, **we take advantage of two observations**.

1. First, without loss of generality, we can consider only tours that start at a.
2. Second, because our graph is undirected, we can generate only tours in which b is visited before c. (Why?)

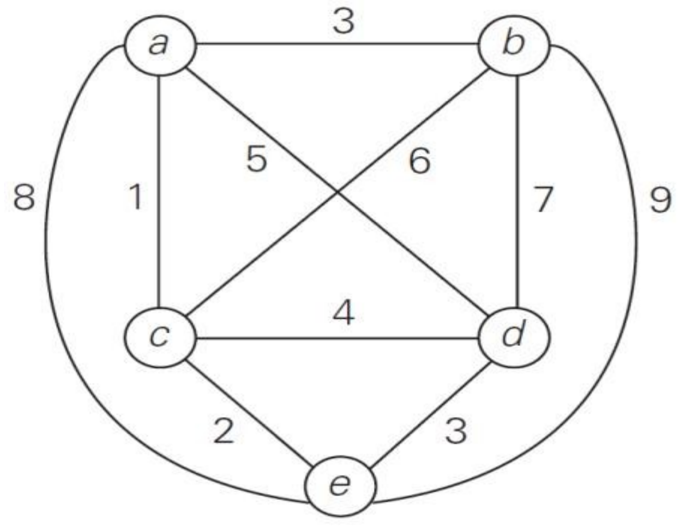


Tour	Length	
$a \dashrightarrow b \dashrightarrow c \dashrightarrow d \dashrightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \dashrightarrow b \dashrightarrow d \dashrightarrow c \dashrightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \dashrightarrow c \dashrightarrow b \dashrightarrow d \dashrightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \dashrightarrow c \dashrightarrow d \dashrightarrow b \dashrightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \dashrightarrow d \dashrightarrow b \dashrightarrow c \dashrightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \dashrightarrow d \dashrightarrow c \dashrightarrow b \dashrightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

TSP

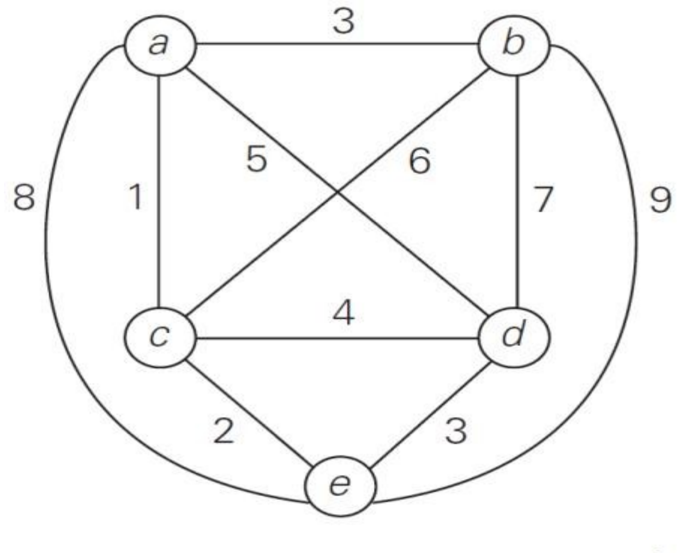
3. In addition, after visiting n−1= 4 cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one

- The state-space tree tracing the algorithm's application is given in Figure



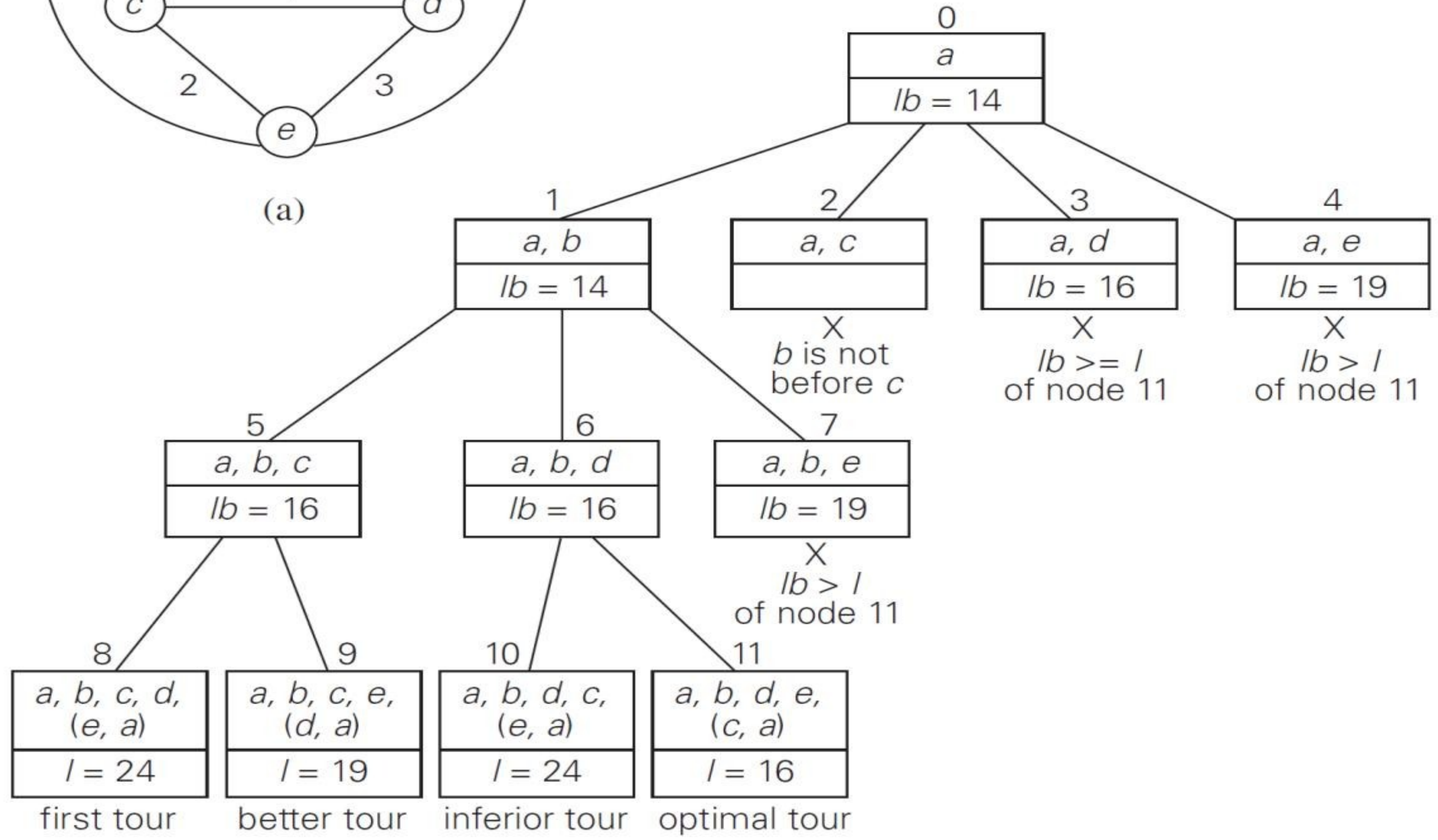
(a)

$$lb = \lceil [(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2 \rceil = 14.$$



(a)

$$lb = \lceil [(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2 \rceil = 14.$$



(b)

Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem
  - Travelling Sales Person problem
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms
  - P, NP, NP-Complete & NP-Hard classes
- `

0/1 Knapsack problem

- Branch and Solution to 0/1 Knapsack problem

Problem Definition

- Given n items of known weights w_i and values v_i , i = 1, 2, . . . , n, and a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack.

$$\sum_{1 \le i \le n} w_i x_i \le W \text{ and } \sum_{1 \le i \le n} p_i x_i \text{ is maximized, where } x_i = 0 \text{ or } 1$$

0/1 Knapsack problem

- It is convenient to order the items of a given instance in descending order by their value-to-weight ratios

$$v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$$

- Each node on the ith level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items
- We record the total ***weight w*** and the total ***value v*** of this selection in the node, along with some upper bound ***ub***

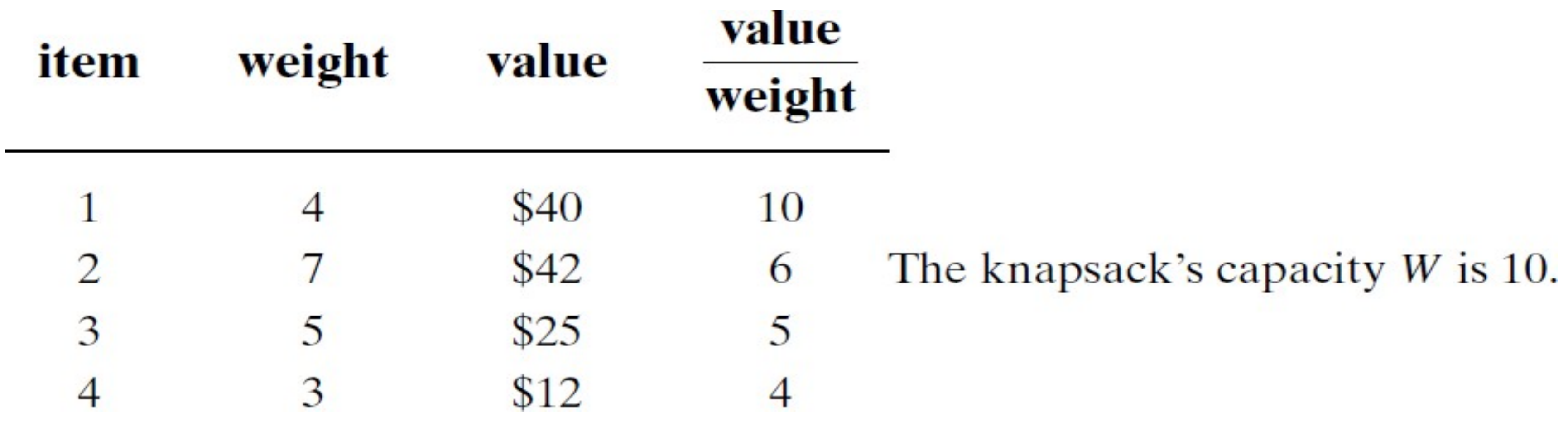
How to compute the upper bound?

- A simple way to compute the upper bound ***ub*** is to
  - add to ***v***, the total value of the items already selected,
  - the product of the remaining capacity of the knapsack ***W* – *w*** and the best per unit payoff among the remaining items, which is $\mathbf{v_{i+1}/w_{i+1}}$

$$ub = v + (W - w)(v_{i+1}/w_{i+1}).$$

Example

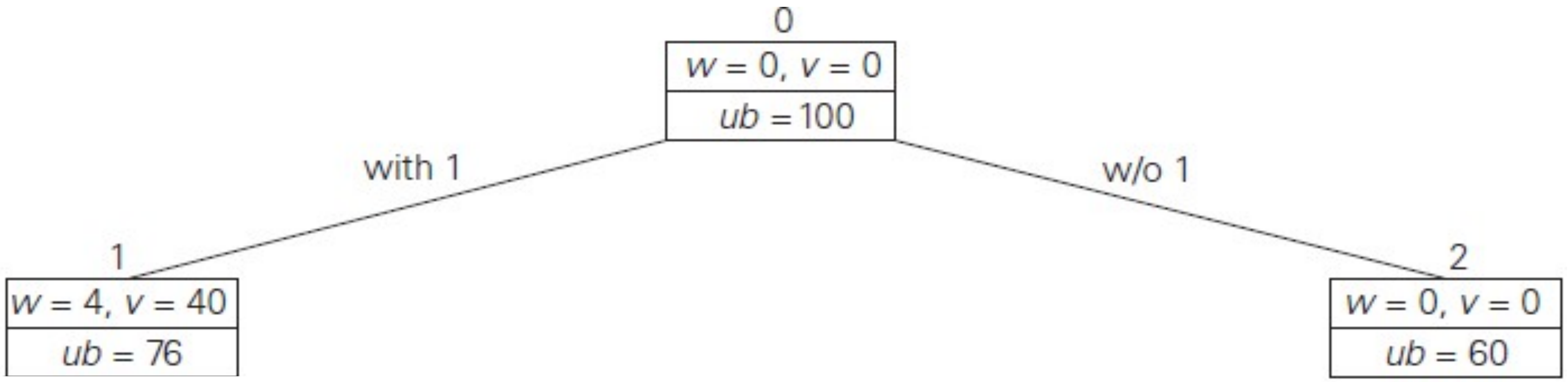
- Consider the following problem.
- The items are already ordered in descending order of their value-to-weight ratios.



item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity W is 10.

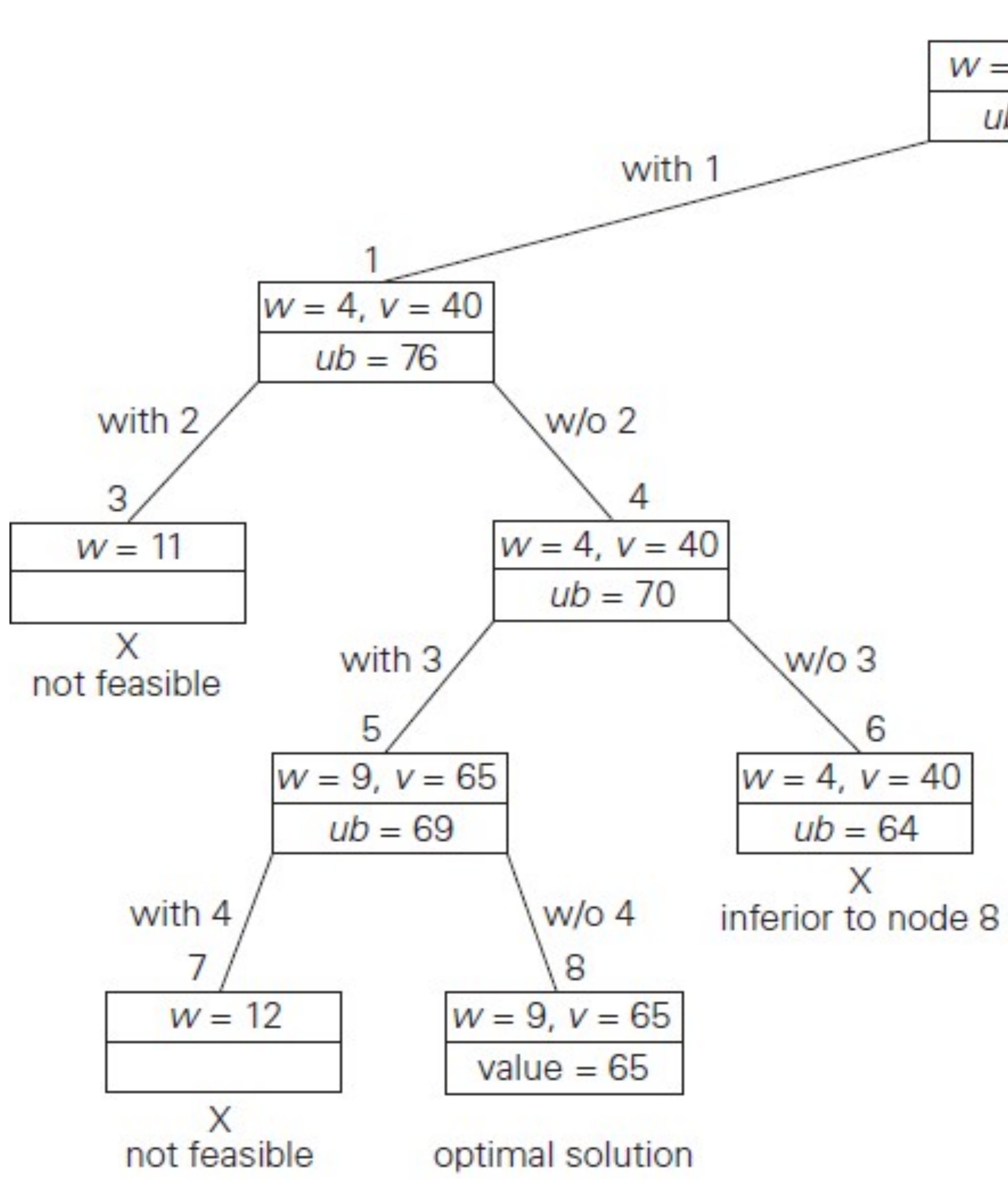
- Let us apply the branch-and-bound algorithm.



item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity W is 10.

FIGURE 12.8 State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.



item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity W is 10.

FIGURE 12.8 State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

Discussion

- Solving the knapsack problem by a branch-and-bound algorithm has a rather unusual characteristic.
- For the knapsack problem, every node of the tree represents a subset, and also a feasible solution of the items given.
- We can use this fact to update the information about the best subset seen so far after generating each new node in the tree.
- If we had done this for the instance investigated above, we could have terminated nodes 2 and 6 before node 8 was generated because they both are inferior to the subset of value 65 of node 5.

Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem
  - Travelling Sales Person problem
- **0/1 Knapsack problem**
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms
  - P, NP, NP-Complete & NP-Hard classes
- `

Some Terminologies

- ***Live node*** - a node which has been generated and all of whose children are not yet been generated.
- ***E-node*** - is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
- ***Dead node*** - a node that is either not to be expanded further, or for which all of its children have been generated
- ***Bounding Function*** - will be used to kill live nodes without generating all their children.

Some Terminologies

- ***Backtracking*** – is depth first node generation with bounding functions.
- ***Branch-And-Bound*** is a method in which E-node remains E-node until it is dead.
- ***Breadth-First-Search:*** Branch-and Bound with each new node placed in a queue. The front of the queen becomes the new E-node.
- ***Depth-Search (D-Search):*** New nodes are placed in to a stack. The last node added is the first to be explored.

Some Basic Concepts

The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{c}(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function.

Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer node from x. Node x is assigned a rank using a function $\hat{c}(\cdot)$ such that $\hat{c}(x) = f(h(x)) + \hat{g}(x)$, where $h(x)$ is the cost of reaching x from the root and $f(\cdot)$ is any nondecreasing function.

A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(\cdot)$. Hence, such a search strategy is called an LC-search (**L**east **C**ost search).

The technique discussed here is applicable for minimization problems

0/1 Knapsack - B&B based solution

- The technique discussed here is applicable for minimization problems
- So convert the knapsack problem (maximizing the profit) into minimization problem by negating the objective function

$$\text{minimize } -\sum_{i=1}^{n} p_i x_i \quad \text{subject to } \sum_{i=1}^{n} w_i x_i \leq m$$

$$x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

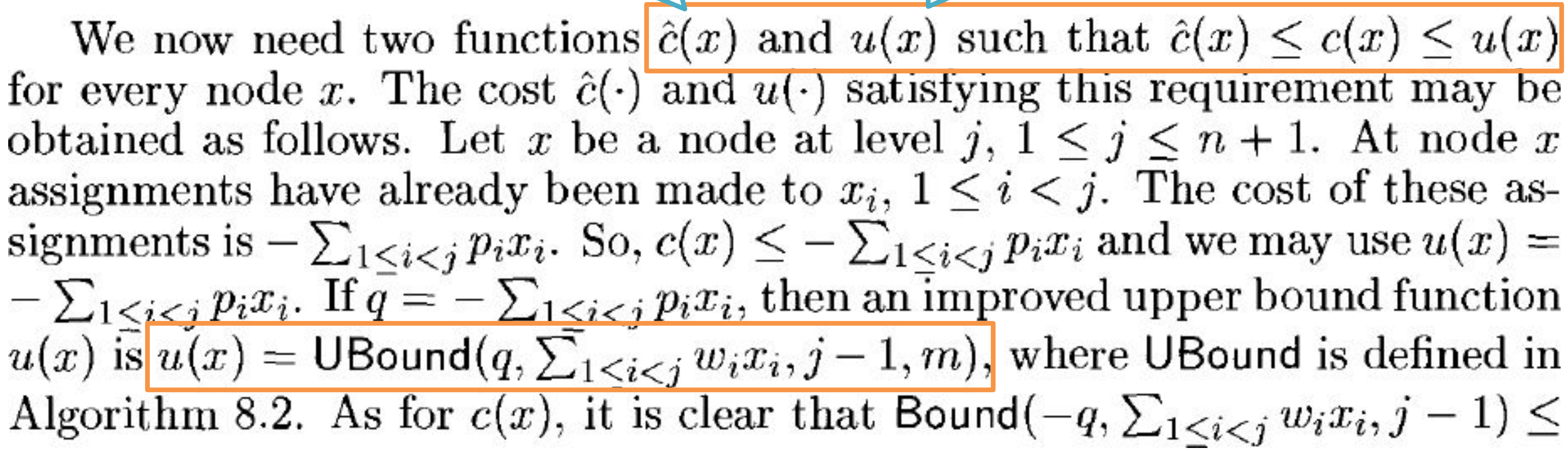
- Define cost

$$c(x) = -\sum_{1<i<n} p_i x_i$$

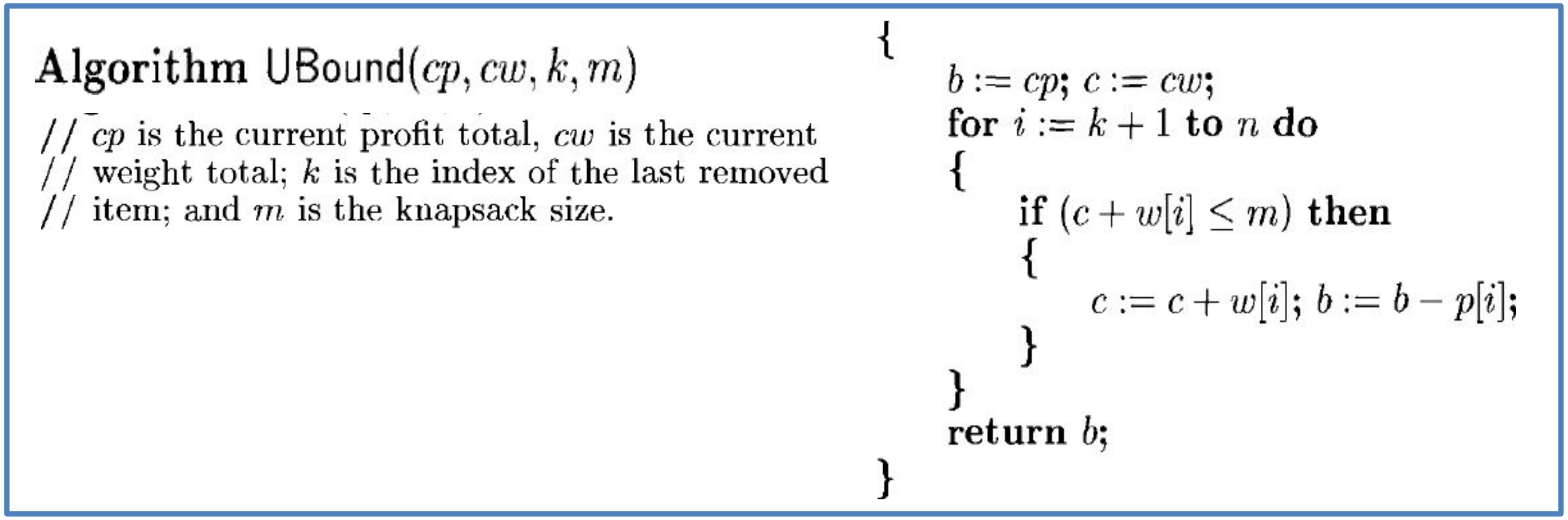
0/1 Knapsack - B&B based solution

Lower bound

Upper bound



We now need two functions $\hat{c}(x)$ and $u(x)$ such that $\hat{c}(x) \le c(x) \le u(x)$ for every node x. The cost $\hat{c}(\cdot)$ and $u(\cdot)$ satisfying this requirement may be obtained as follows. Let x be a node at level j, $1 \le j \le n+1$. At node x assignments have already been made to x_i, $1 \le i < j$. The cost of these assignments is $-\sum_{1 \le i < j} p_i x_i$. So, $c(x) \le -\sum_{1 \le i < j} p_i x_i$ and we may use $u(x) = -\sum_{1 \le i < j} p_i x_i$. If $q = -\sum_{1 \le i < j} p_i x_i$, then an improved upper bound function $u(x)$ is $u(x) = \mathsf{UBound}(q, \sum_{1 \le i < j} w_i x_i, j-1, m)$, where UBound is defined in Algorithm 8.2. As for $c(x)$, it is clear that $\mathsf{Bound}(-q, \sum_{1 \le i < j} w_i x_i, j-1) \le$



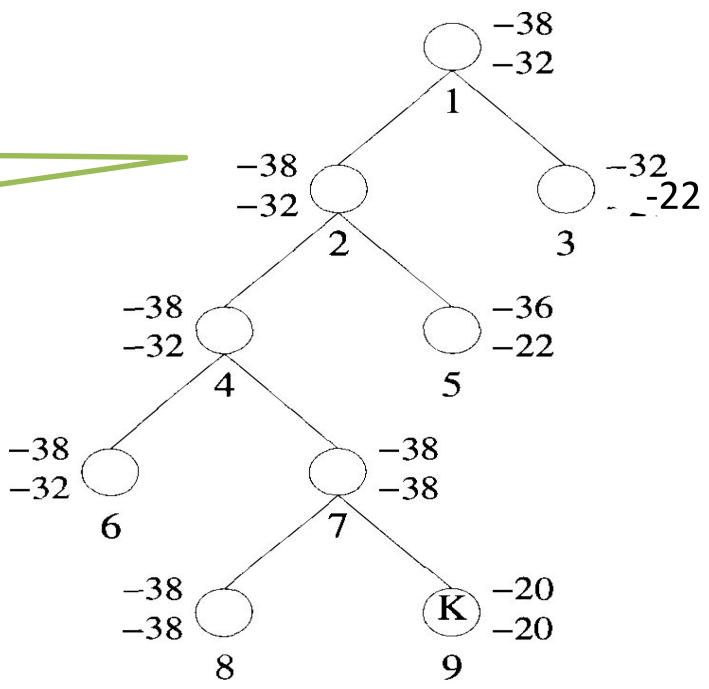
```
Algorithm UBound(cp, cw, k, m)
// cp is the current profit total, cw is the current
// weight total; k is the index of the last removed
// item; and m is the knapsack size.
{
    b := cp; c := cw;
    for i := k + 1 to n do
    {
        if (c + w[i] ≤ m) then
        {
            c := c + w[i]; b := b − p[i];
        }
    }
    return b;
}
```

LCBB - Least Cost Branch &Bound solution

Example 8.2 [LCBB] Consider the knapsack instance $n = 4$, $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$, $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$, and $m = 15$. Let us trace the working of an LC branch-and-bound search using $\hat{c}(\cdot)$ and $u(\cdot)$ as defined previously. We continue to use the fixed tuple size formulation. The search begins with the root as the E-node. For this node, node 1 of Figure 8.8, we have $\hat{c}(1) = -38$ and $u(1) = -32$.

Similar to the computation of fractional knapsack

Nodes with least $c(\cdot)$ is expanded

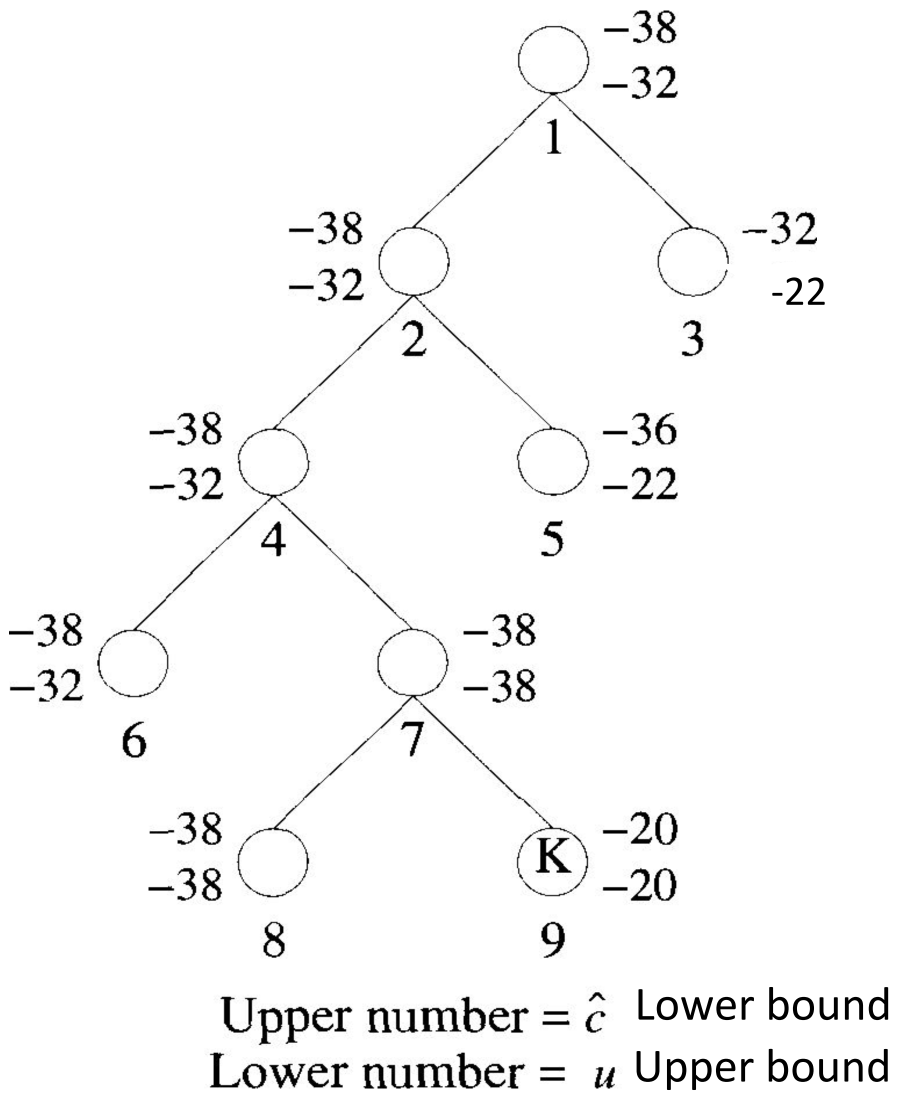


Upper number = $\hat{c}$
Lower number = u

Note: Data is assumed to be sorted in p/w order

Computation method

- Node1 – Not a solution node, becomes E-node, expanded, 2 & 3 generated & added to list (Min heap wrt ***u***) of live nodes. (Minheap: 2,3)
- Node2 - is next E-node. Expanded, 4 & 5 generated. Added to list of live nodes. (Minheap: 3,4,5)
- Next E node is 4. Expanded, 6 & 7 are generated. Added to list of live nodes. Minheap: 3,5,6,7
- Next E node: 6 or 7. Say expand 7. Nodes 8 & 9 generated. 8 is solution. Kill 9 as ***c***ⓖ჻)
  > upper(-38).
- Delete 3,5,6 as ***c***ⓖ჻)(E) > upper
- Node 8 is the solution



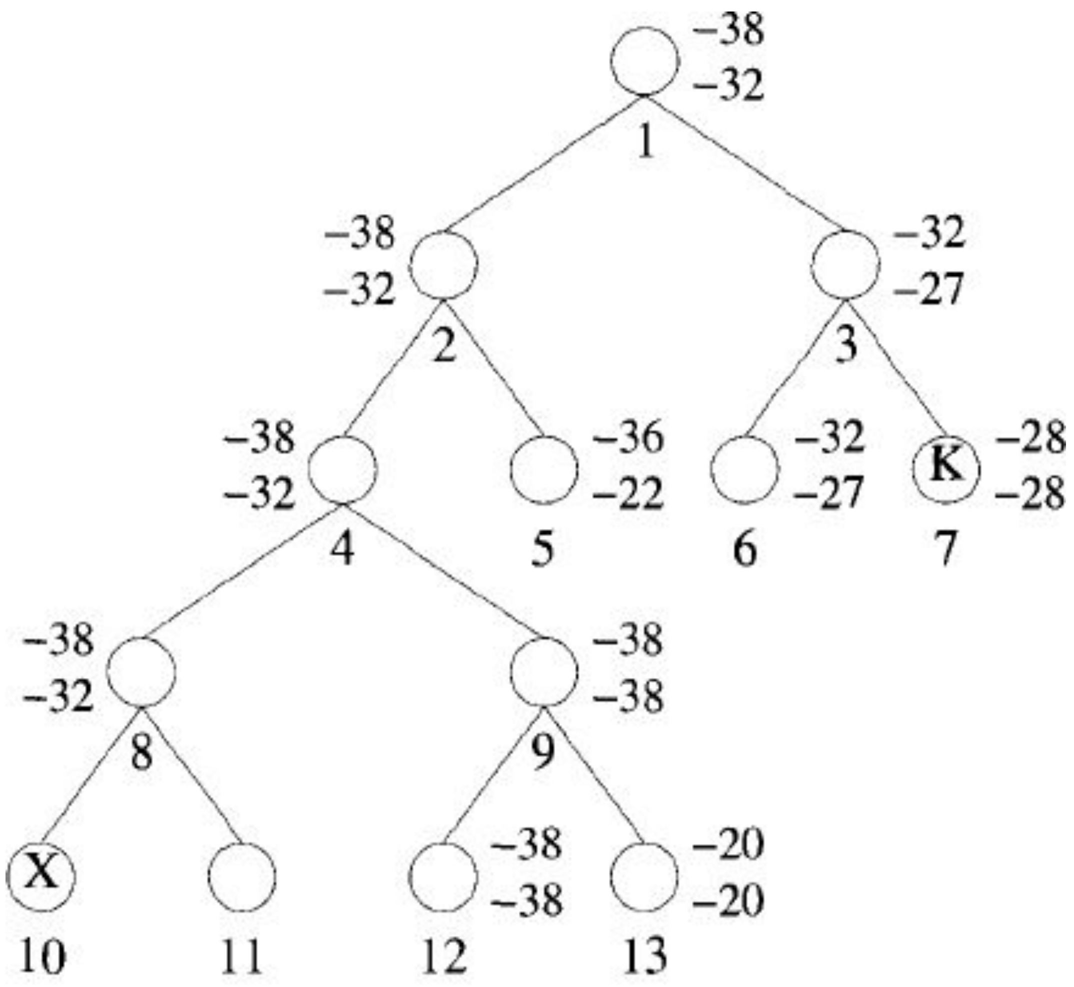
1
−38
−32
2
−38
−32
3
−32
-22
4
−38
−32
5
−36
−22
6
−38
−32
7
−38
−38
8
−38
−38
K
9
−20
−20
Upper number = $\hat{c}$ Lower bound
Lower number = u Upper bound

Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem
  - Travelling Sales Person problem
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - [illegible]
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms
  - P, NP, NP-Complete & NP-Hard classes
- `

FIFO Branch & Bound solution

- Example



upper number = $\hat{c}$ Lower bound
lower number = u Upper bound

Outline

- Backtracking
  - General method
  - N-Queens problem
  - Sum of subsets problem
  - Graph coloring
  - Hamiltonian cycles
- Branch and Bound
  - Assignment Problem
  - Travelling Sales Person prob
- 0/1 Knapsack problem
  - LC Branch and Bound solution
  - FIFO Branch and Bound solution
- NP-Complete and NP-Hard problems
  - Basic concepts,
  - non-deterministic algorithms
  - P, NP, NP-Complete & NP-Hard classes
- `

Basic Concepts

- For many of the problems we know and study, the best algorithms for their solution have computing times can be clustered into two groups;
  - Solutions are bounded by the **polynomial**- Examples include Binary search O(log n), Linear search O(n) or in general $O(n^k)$ where k is a constant.
  - Solutions are bounded by a **non-polynomial** - Examples include travelling salesman problem $O(n^2 2^n)$ & knapsack problem $O(2^{n/2})$.

    As the time increases exponentially, even moderate size problems cannot be solved.

Basic Concepts

- So far, no one has been able to device an algorithm which is bounded by the polynomial for the problems belonging to the non-polynomial.
- However impossibility of such an algorithm is not proved.

Non deterministic algorithms

- We also need the idea of two models of computer (Turing machine): deterministic and non- deterministic.
- A deterministic computer is the regular computer we always thinking of.
- When the result of every operation is uniquely defined then it is called **deterministic algorithm**.
- A non-deterministic computer is one that is just like we're used to except that it has unlimited parallelism, so that any time you come to a branch, you spawn a new "process" and examine both

Non deterministic algorithms

- When the outcome is not uniquely defined but is limited to a specific set of possibilities, we call it **non deterministic** algorithm.

Decision vs Optimization algorithms

- An optimization problem tries to find an optimal solution.
- A decision problem tries to answer a yes/no question.
- Most of the problems can be specified in decision and optimization versions.
- For example, TSP can be stated as two ways
  - Optimization - find hamiltonian cycle of minimum weight
  - Decision - is there a hamiltonian cycle of weight $\leq$ k?

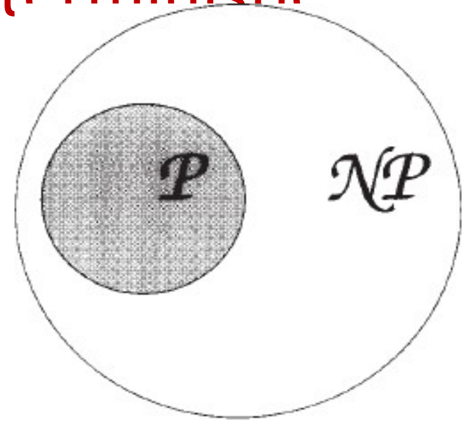
Decision vs Optimization algorithms

- For graph coloring problem,
  - Optimization – find the minimum number of colors needed to color the vertices of a graph so that no two adjacent vertices are colored the same color
  - Decision - whether there exists such a coloring of the graph's vertices with no more than m colors?

- Many optimization problems can be recast in to decision problems with the property that,
  - the decision algorithm can be solved in polynomial time if and only if the corresponding optimization problem

P, NP

- NP stands for Non-deterministic Polynomial time.
- **Definition: P** is a set of all decision problems solvable by a deterministic algorithm in polynomial time.
- **Definition: NP** is the set of all decision problems solvable by a nondeterministic algorithm in polynomial time.



- This also implies P $\subseteq$ NP
- i.e. Problems known to be in P are trivially in NP

P

- P is a complexity class that represents the set of all decision problems that can be solved in polynomial time.
- That is, given an instance of the problem, the answer yes or no can be decided in polynomial time.
- Example
  - Given a connected graph G, can its vertices be coloured using two colours so that no edge is monochromatic?
  - Algorithm: start with an arbitrary vertex, color it red and all of its neighbours blue and continue. Stop when you run out of vertices or you are forced to make an edge have both of its endpoints be the same color.

Source: Stackoverflow.com

NP

- NP is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time.
- This means that if someone gives us an instance of the problem and a certificate (sometimes called a witness) to the answer being yes, we can check that it is correct in polynomial time.

Source:
Stackoverflow.com

NP - Example

Integer factorisation is in NP.

- This is the problem that given integers n and m, is there an integer f with $1 < f < m$, such that f divides n (f is a small factor of n)?
- This is a decision problem because the answers are yes or no.
- If someone hands us an instance of the problem (so they hand us integers n and m) and an integer f with $1 < f < m$, and claim that f is a factor of n (the certificate), we can check the answer in polynomial time by performing the division n / f.

P, NP

- But there are some problems which are known to be in NP but don't know if they're in P.
- The example is the decision-problem version of the Travelling Salesman Problem (***decision-TSP***).
- It's not known whether decision-TSP is in P:
  - there's no known poly-time solution,
  - but there's no proof such a solution doesn't exist.

- The most famous unsolved problem in computer science is **"whether P=NP or P≠NP? "**

NP-Complete

Reducible Problems

- Given decision problems P and Q, if an algorithm can transform a solution for P into a solution for Q in polynomial time, it’s said that Q is ***poly-time reducible*** (or just reducible) to P.

Definition: NP-Complete

A decision problem D is said to be **NP-complete** if:

1. it belongs to class NP
2. every problem in NP is polynomially reducible to D

NP-Complet e

- NP-Complete is a complexity class which represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time.

- Intuitively this means that we can solve Y quickly if we know how to solve X quickly.
- Precisely, Y is reducible to X, if there is a polynomial time algorithm f to transform instances y of Y to instances x = f(y) of X in polynomial time, with the property that the answer to y is yes, if and only if the answer to f(y) is yes.

Source:
Stackoverflow.com

NP Complete

- Informally, these are the hardest problems in the class NP
- If any NP-complete problem can be solved by a polynomial time deterministic algorithm,
  - then every problem in NP can be solved by a polynomial time deterministic algorithm

- But Till date, no polynomial time deterministic algorithm is known to solve any of them

NP-Complet e

- Example for NP-complete is **CNF-satisfiability problem**.
- The CNF-satisfiability problem deals with boolean expressions.
- This is given by Cook in 1971.
- The CNF-satisfiability problem asks whether or not one can assign values true and false to variables of a given boolean expression in its CNF form to make the entire expression true.

Example for NP-Complete

- Traveling salesman problem
- Hamiltonian cycle problem
- Clique problem
- Subset sum problem
- Boolean satisfiability problem
- Many thousands of other important computational problems in computer science, mathematics, economics, manufacturing, communications, etc.

NP-Hard Problems

- These problems need not have any bound on their running time.
- If any NP-Complete Problem is polynomial time reducible to a problem X, that problem X belongs to **NP-Hard** class.
- All NP-Complete problems are also NP-Hard.
- In other words if a NP-Hard problem is non-deterministic polynomial time solvable, it is a NP- Complete problem.
- Example of a NP problem that is not NPC is Halting Problem.

NP-Hard Problems

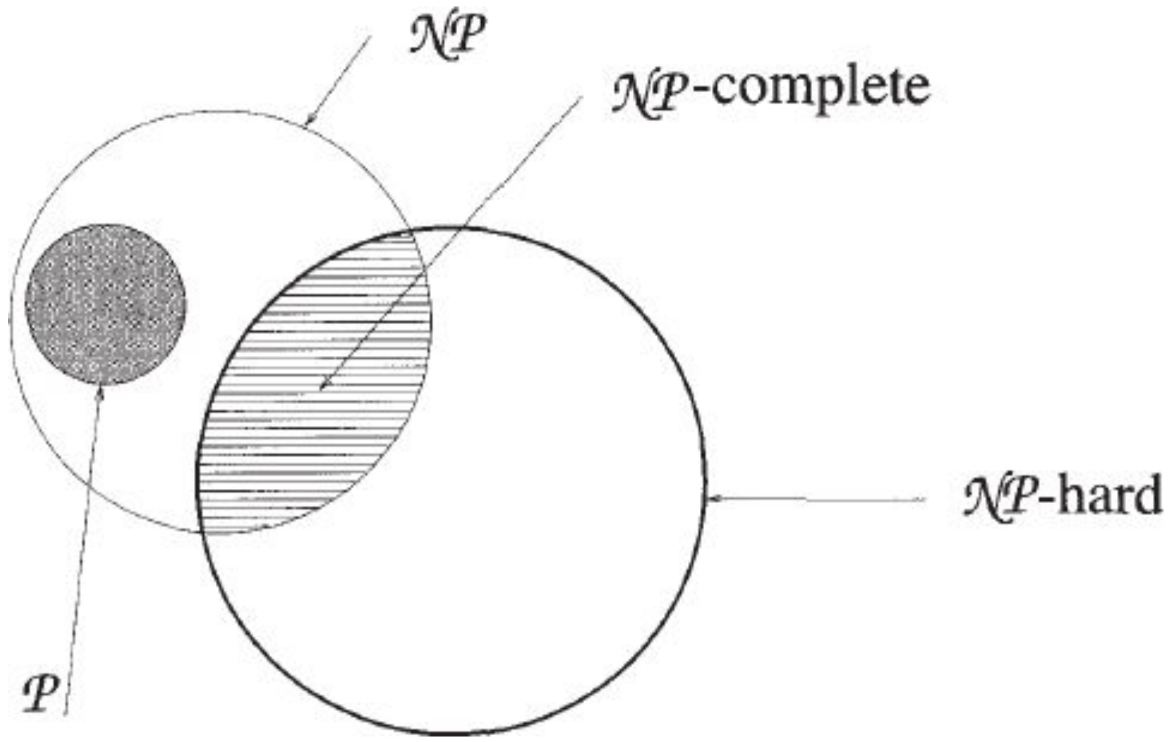


Figure: Commonly believed relationship between P, NP, NP-Complete and NP-hard problems

NP-Hard Problems

- If a NP-Hard problem can be solved in polynomial time then all NP-Complete can be solved in polynomial time.
- “All NP-Complete problems are NP-Hard but not all NP-Hard problems are not NP-Complete.”
- NP-Complete problems are subclass of NP-Hard
- The more conventional **optimization version of TSP** for finding the shortest route is NP-hard, not strictly NP-complete.

NP-Hard

- Intuitively, these are the problems that are at least as hard as the NP-complete problems.
- Note that NP-hard problems do not have to be in NP, and they

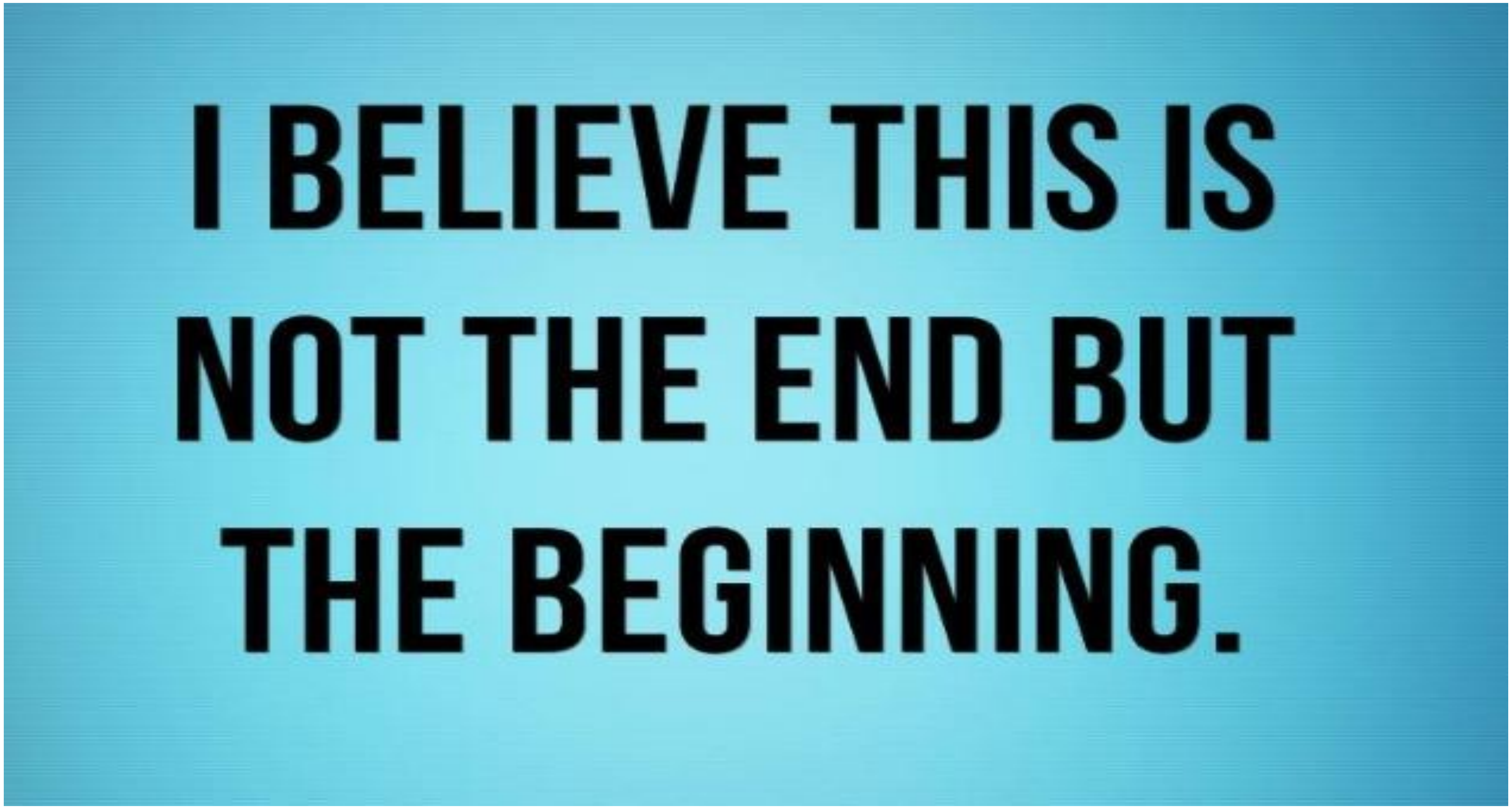
  do not have to be decision problems.
- The precise definition here is that a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time.

- But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time.
- Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems

NP-Hard

- Example: The halting problem is an NP-hard problem.
- This is the problem that given a program P and input I, will it halt? This is a decision problem but it is not in NP. It is clear that any NP-complete problem can be reduced to this one. As another example, any NP-complete problem is NP-hard.

Source:
Stackoverflow.com



I BELIEVE THIS IS NOT THE END BUT THE BEGINNING.