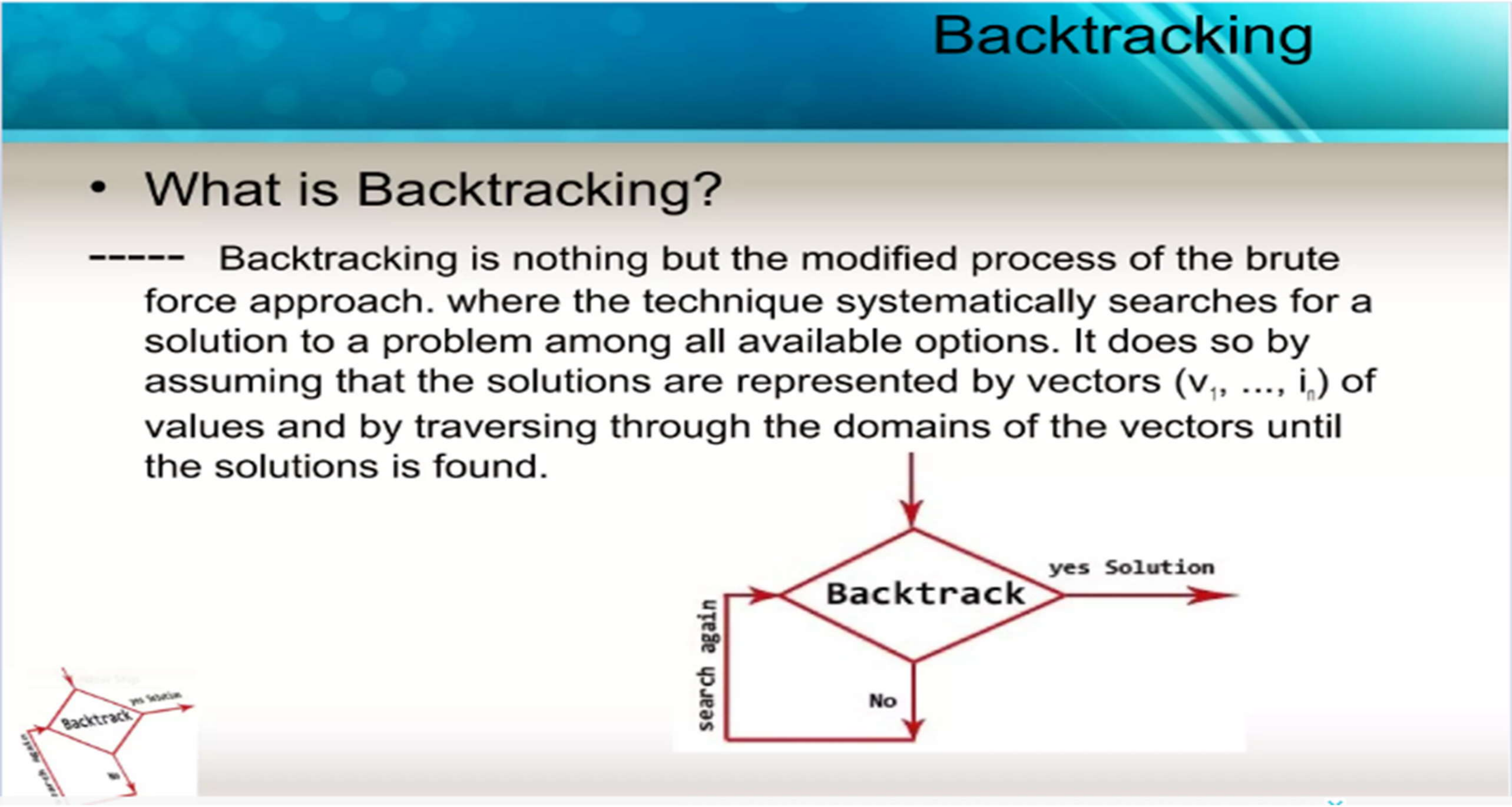
Backtracking

- Backtracking algorithms determine problem solutions by systematically Searching the solution space for the given problem instance.
- This search is facilitated by using a tree organization for the solution space [STATE SPACE TREE]



Backtracking

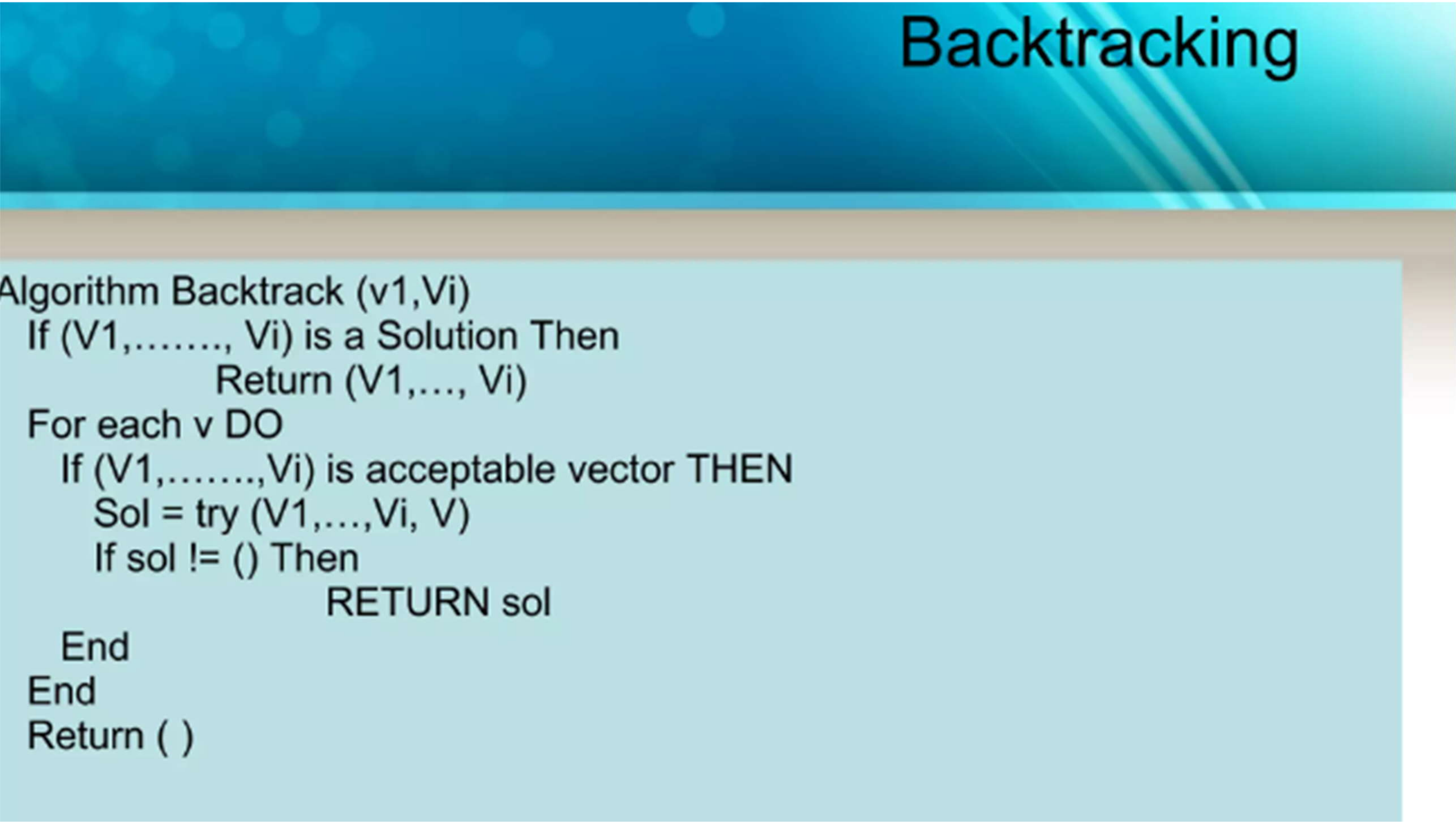
- What is Backtracking?

----- Backtracking is nothing but the modified process of the brute force approach. where the technique systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (v_1, ..., i_n) of values and by traversing through the domains of the vectors until the solutions is found.

Backtracking

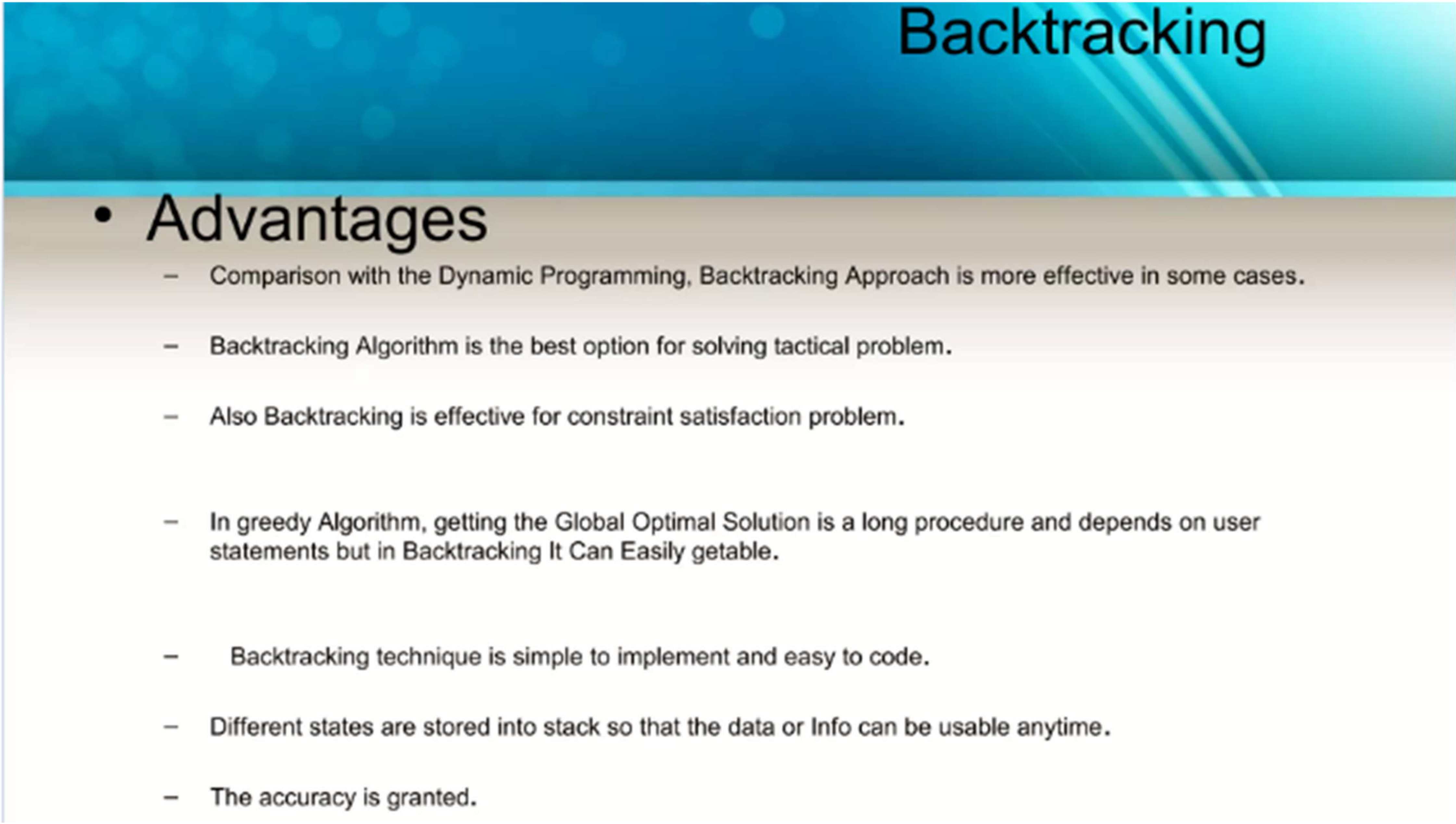
- The Algorithmic Approach
  - Backtracking systematically try and search possibilities to find the solution. Also it is an important process for solving constraint satisfaction problem like crossword, Sudoku and many other puzzles. It can be more continent technique for parsing other combinatorial optimization problem.

  - Basically the process is used when the problem has a number of option and just one solution have to be selected. After having a new option set means recursion, the procedure is repeated over and over until final stage.



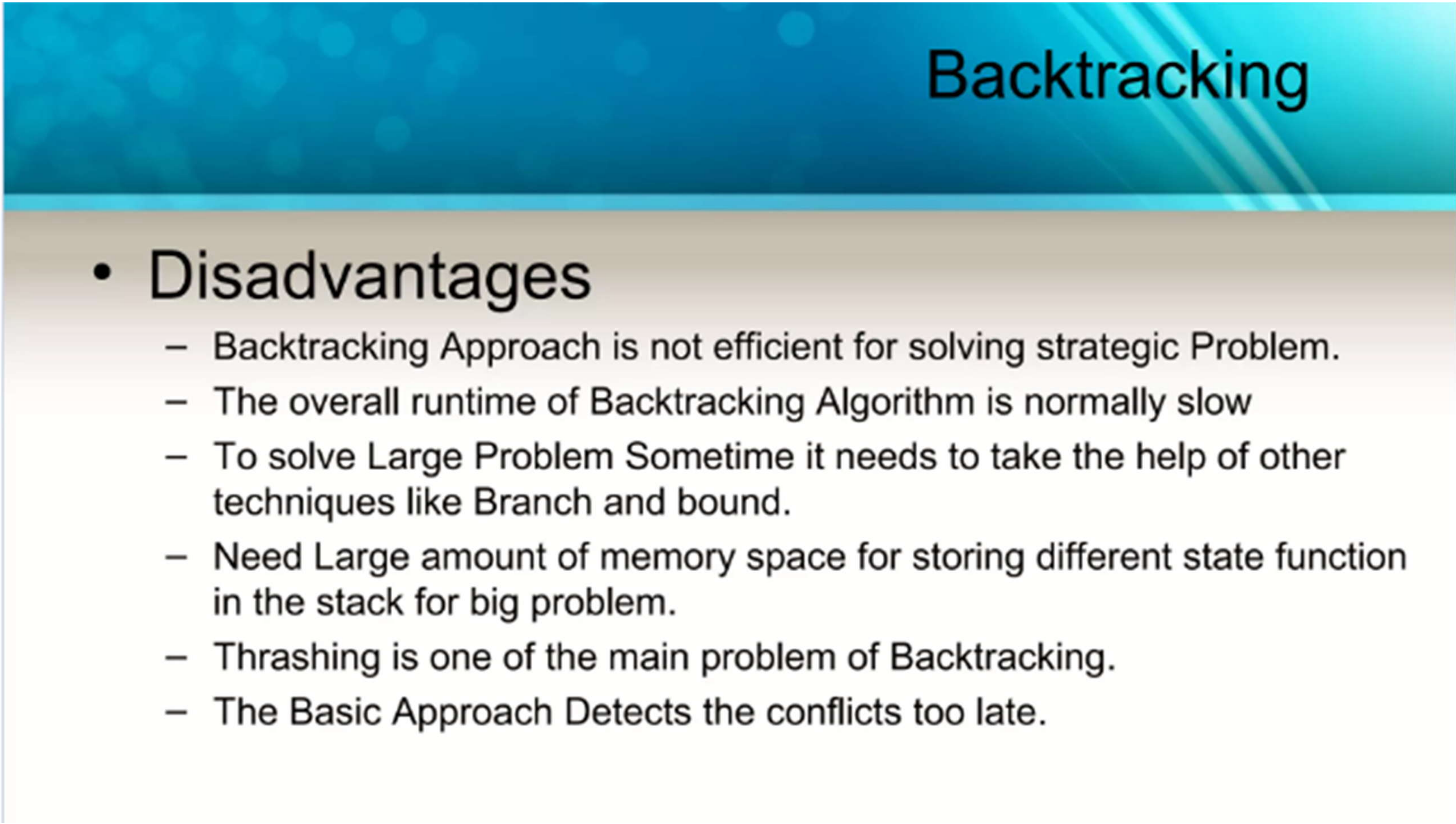
Backtracking

```
lgorithm Backtrack (v1,Vi)
 If (V1,……., Vi) is a Solution Then
               Return (V1,…, Vi)
 For each v DO
   If (V1,…….,Vi) is acceptable vector THEN
     Sol = try (V1,…,Vi, V)
     If sol != () Then
                       RETURN sol
   End
 End
 Return ( )
```



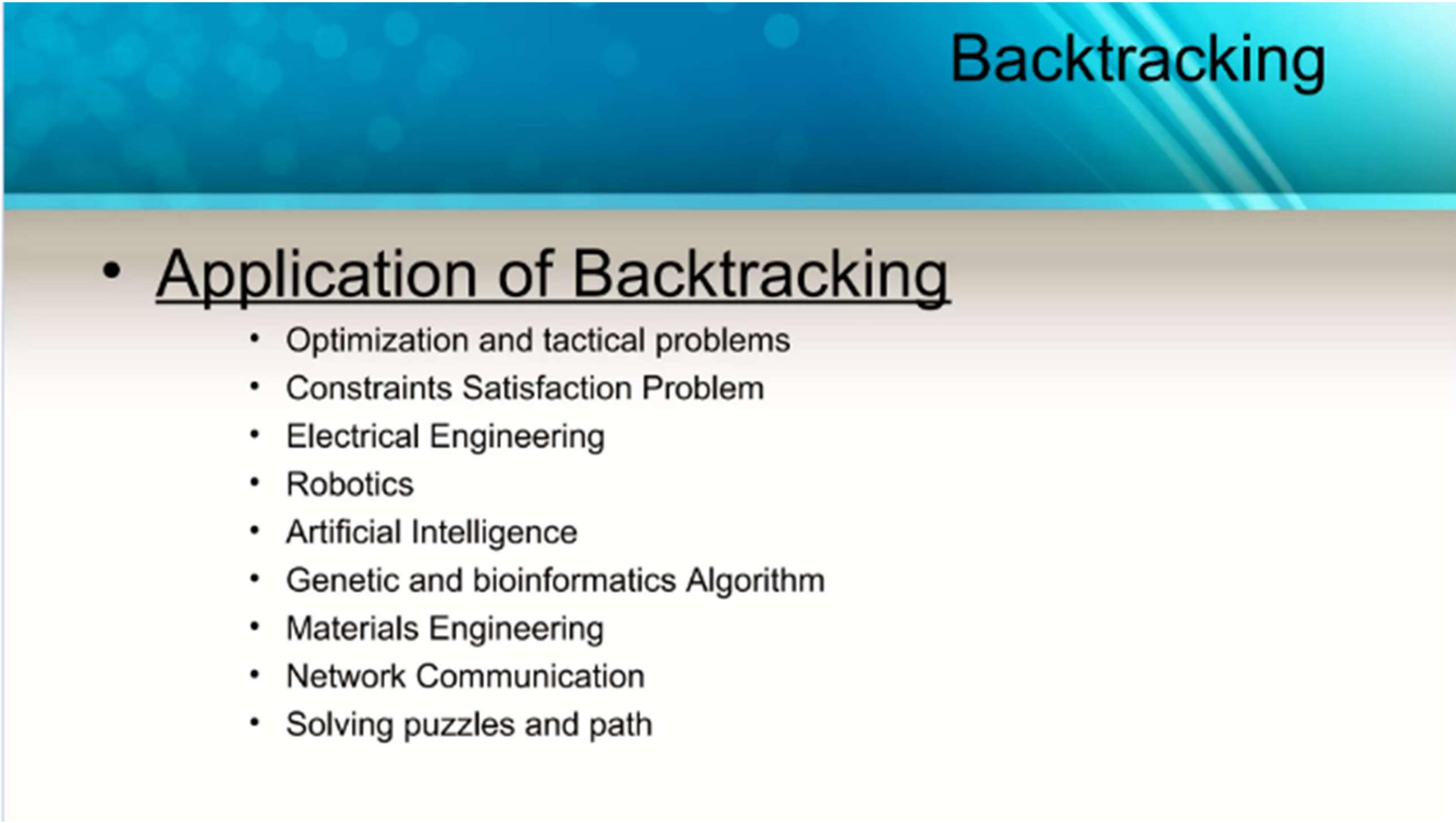
Backtracking

- Advantages
  - Comparison with the Dynamic Programming, Backtracking Approach is more effective in some cases.
  - Backtracking Algorithm is the best option for solving tactical problem.
  - Also Backtracking is effective for constraint satisfaction problem.
  - In greedy Algorithm, getting the Global Optimal Solution is a long procedure and depends on user statements but in Backtracking It Can Easily getable.
  - Backtracking technique is simple to implement and easy to code.
  - Different states are stored into stack so that the data or Info can be usable anytime.
  - The accuracy is granted.



Backtracking

- Disadvantages
  - Backtracking Approach is not efficient for solving strategic Problem.
  - The overall runtime of Backtracking Algorithm is normally slow
  - To solve Large Problem Sometime it needs to take the help of other techniques like Branch and bound.
  - Need Large amount of memory space for storing different state function in the stack for big problem.
  - Thrashing is one of the main problem of Backtracking.
  - The Basic Approach Detects the conflicts too late.



Backtracking

- <u>Application of Backtracking</u>
  - Optimization and tactical problems
  - Constraints Satisfaction Problem
  - Electrical Engineering
  - Robotics
  - Artificial Intelligence
  - Genetic and bioinformatics Algorithm
  - Materials Engineering
  - Network Communication
  - Solving puzzles and path

Some Problem Solved with Backtracking Technique

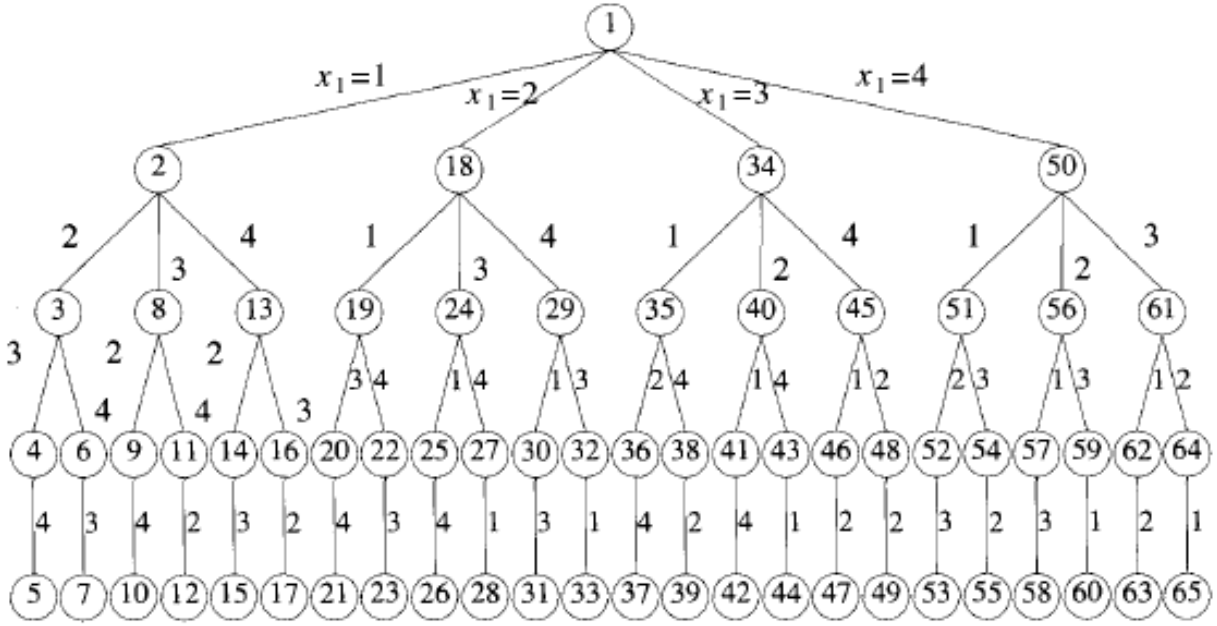
1. N Queens Problem
2. 0/1 Knapsack Problem
3. Travelling Salesperson Problem(TSP)
4. Graph Coloring
5. Hamiltonian Circuit

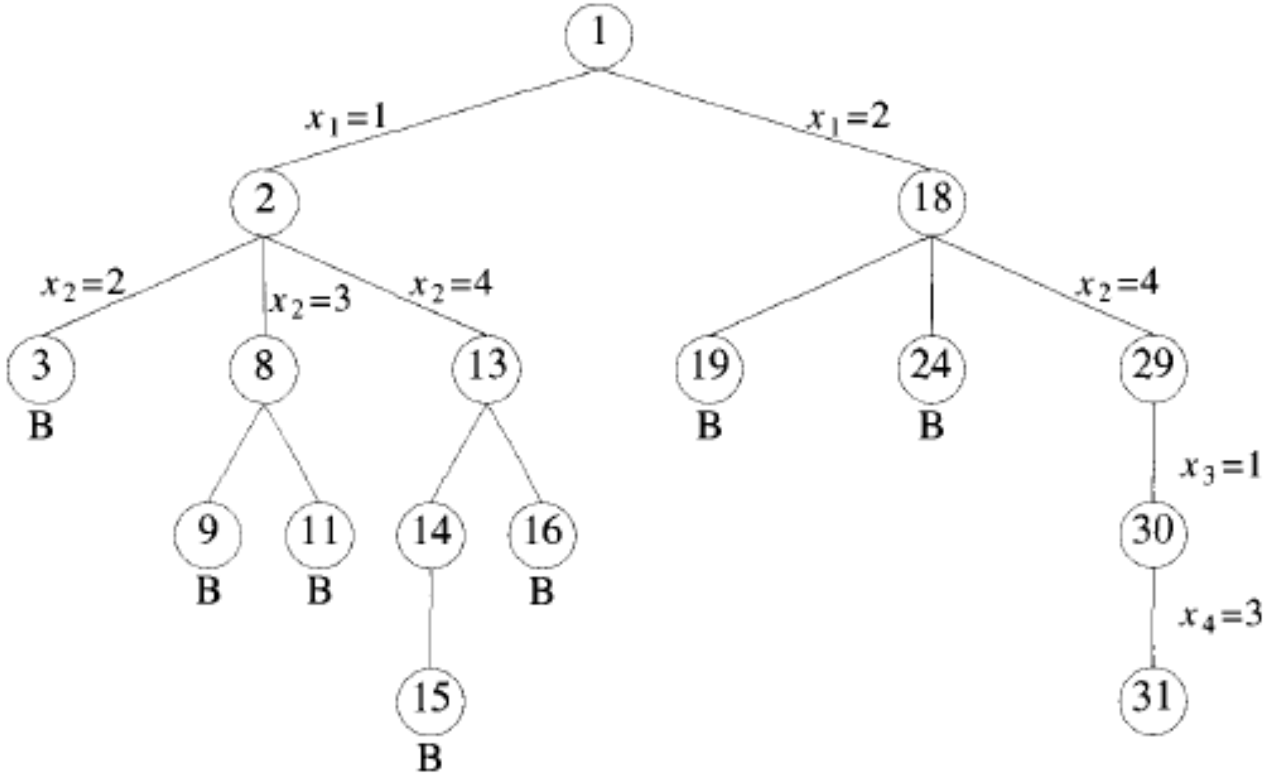
N – Queens Problem

- Consider n*n chess board and try to find all ways to place n non-attacking queens

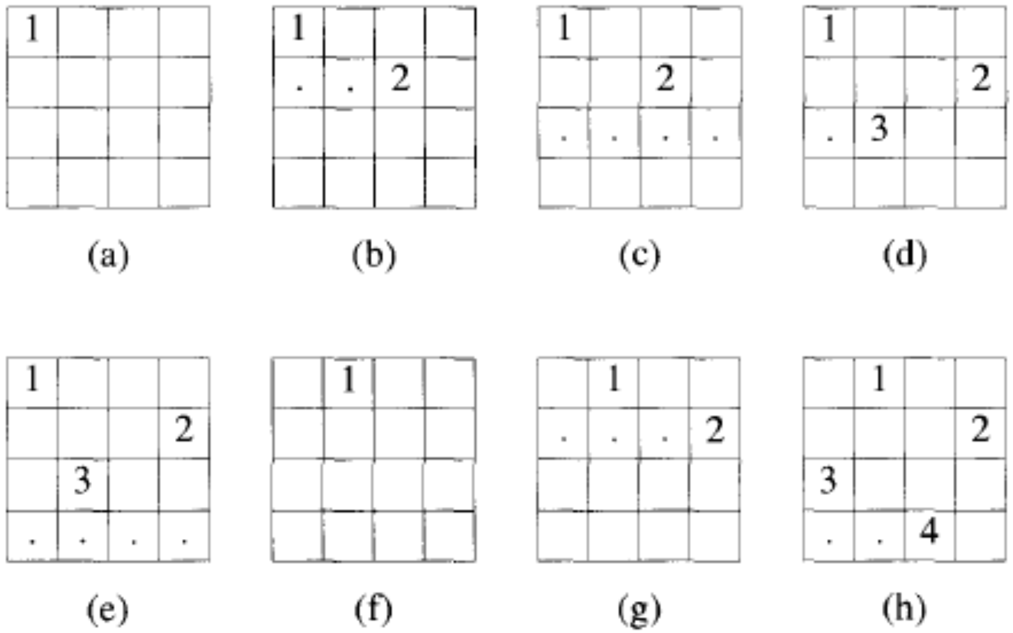
- N=4

Solution Space- tree organization




1
$x_1=1$
$x_1=2$
2
18
$x_2=2$
$x_2=3$
$x_2=4$
$x_2=4$
3
B
8
13
19
B
24
B
29
9
B
11
B
14
16
B
$x_3=1$
30
$x_4=3$
15
B
31

N-Queens Problem



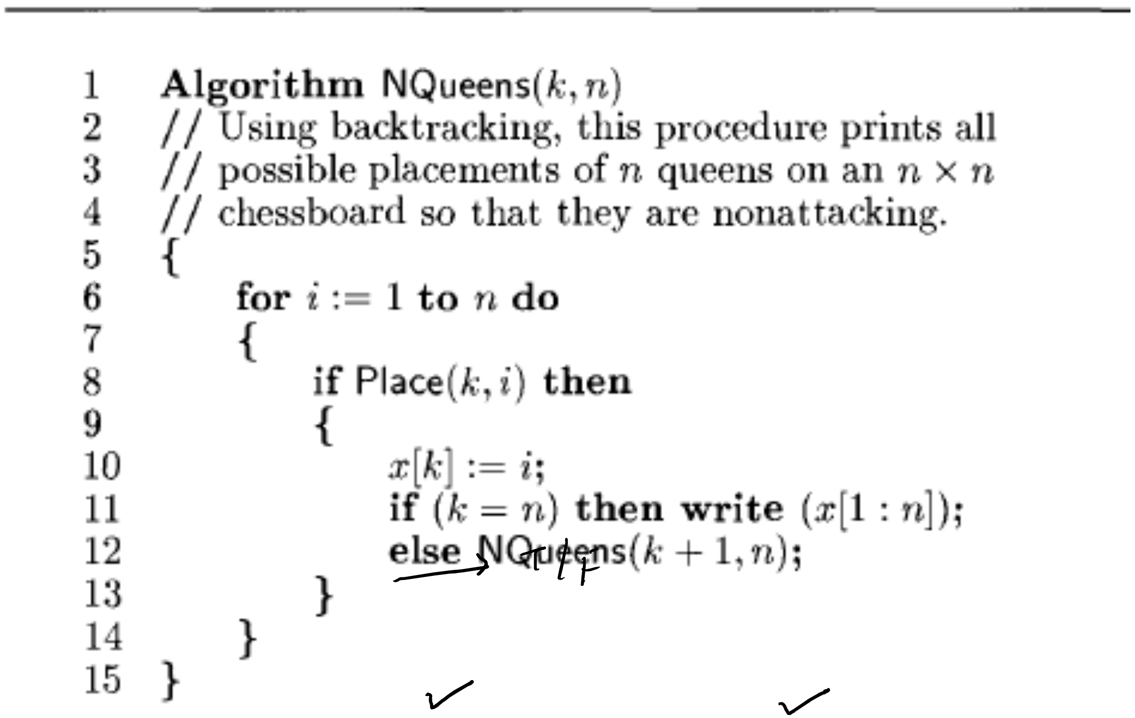
(a) (b) (c) (d)

(e) (f) (g) (h)

N-Queens Problem

- Every element on the same diagonal that runs from upper left to lower right has same row-column value

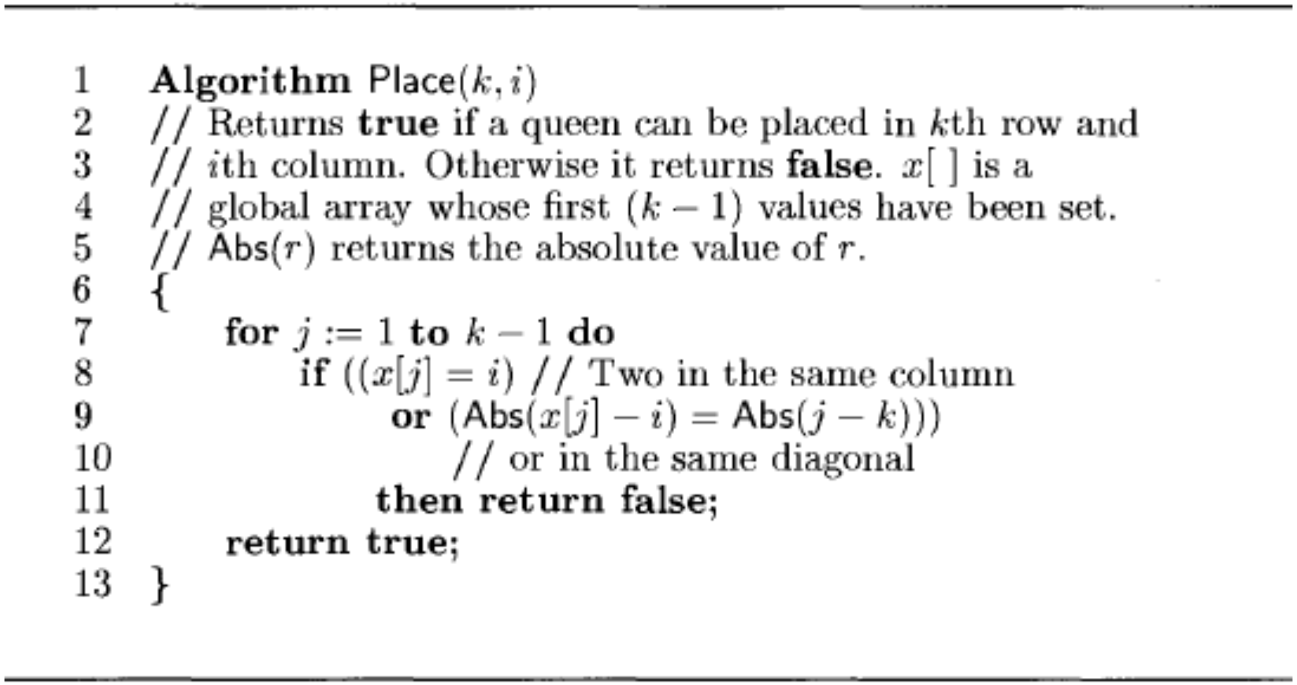
- Every element on the same diagonal that runs from upper right to lower left has same row + column value.



```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × 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);
        }
    }
}
```

✓ ✓

✓



```
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;
}
```

Sum of Subsets

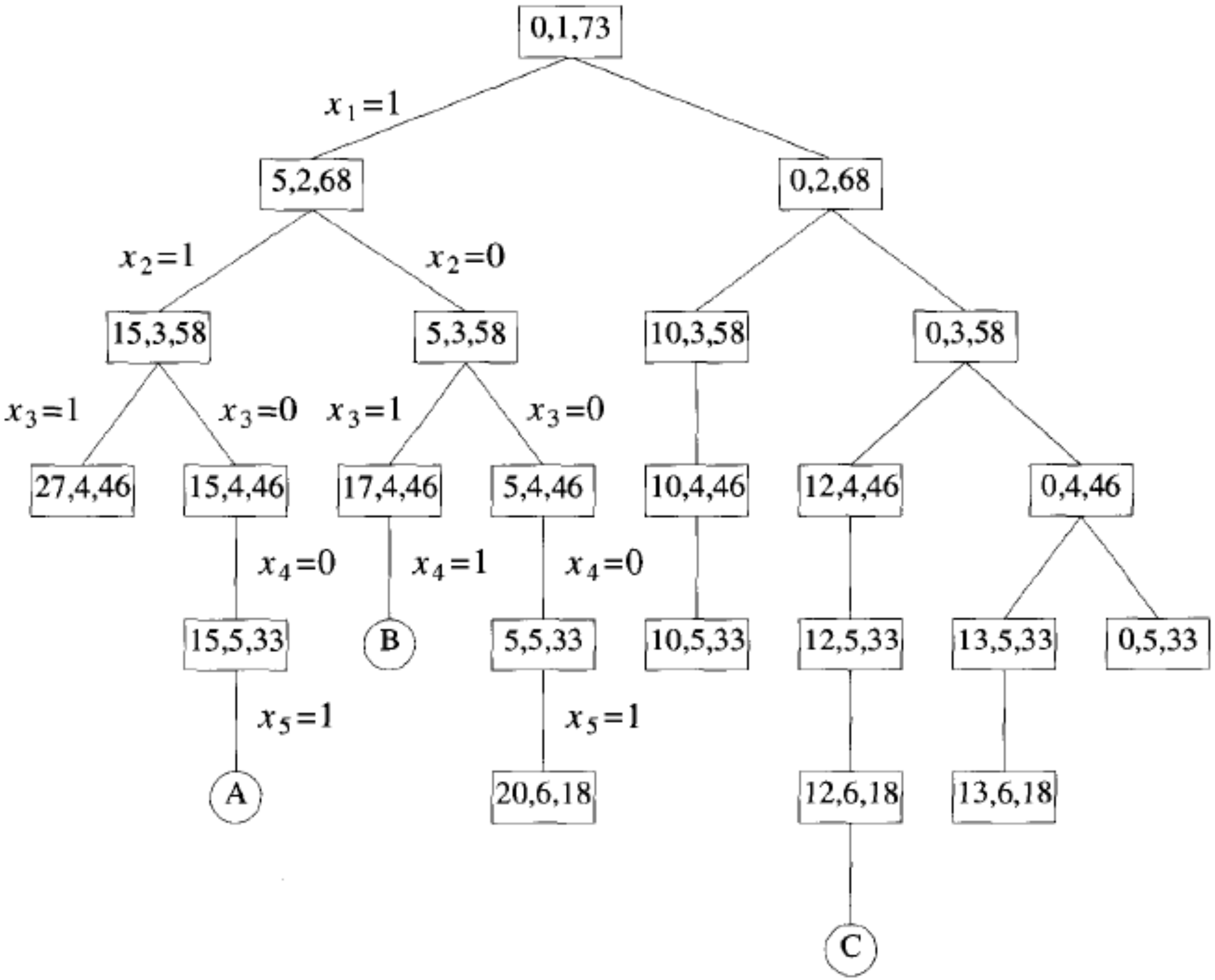
- Given n distinct positive integers, Desire to find all combinations of these numbers whose sum = m

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

- N=6
- M=30
- W[1 : 6]={5,10,12,13,15,18}

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > m$$

STATE SPACE TREE



$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > m$$

$$s = \sum_{j=1}^{k-1} w_j x_j$$

$$r = \sum_{j=k}^{n} w_j$$

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \le m.$$

$$\sum_{i=1}^{k-1} w_i . x_i + w_k x_k + w_{k+1} \le m.$$

$x_k = 1$ $x_k = 0$

$$\boxed{s + w_k + w_{k+1} \le m.} \qquad \boxed{s + w_{k+1} \le m.}$$

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > m.$$

$$\sum_{i=1}^{k-1} w_i x_i + w_k x_k + \sum_{i=k+1}^{n} w_i \ge m.$$

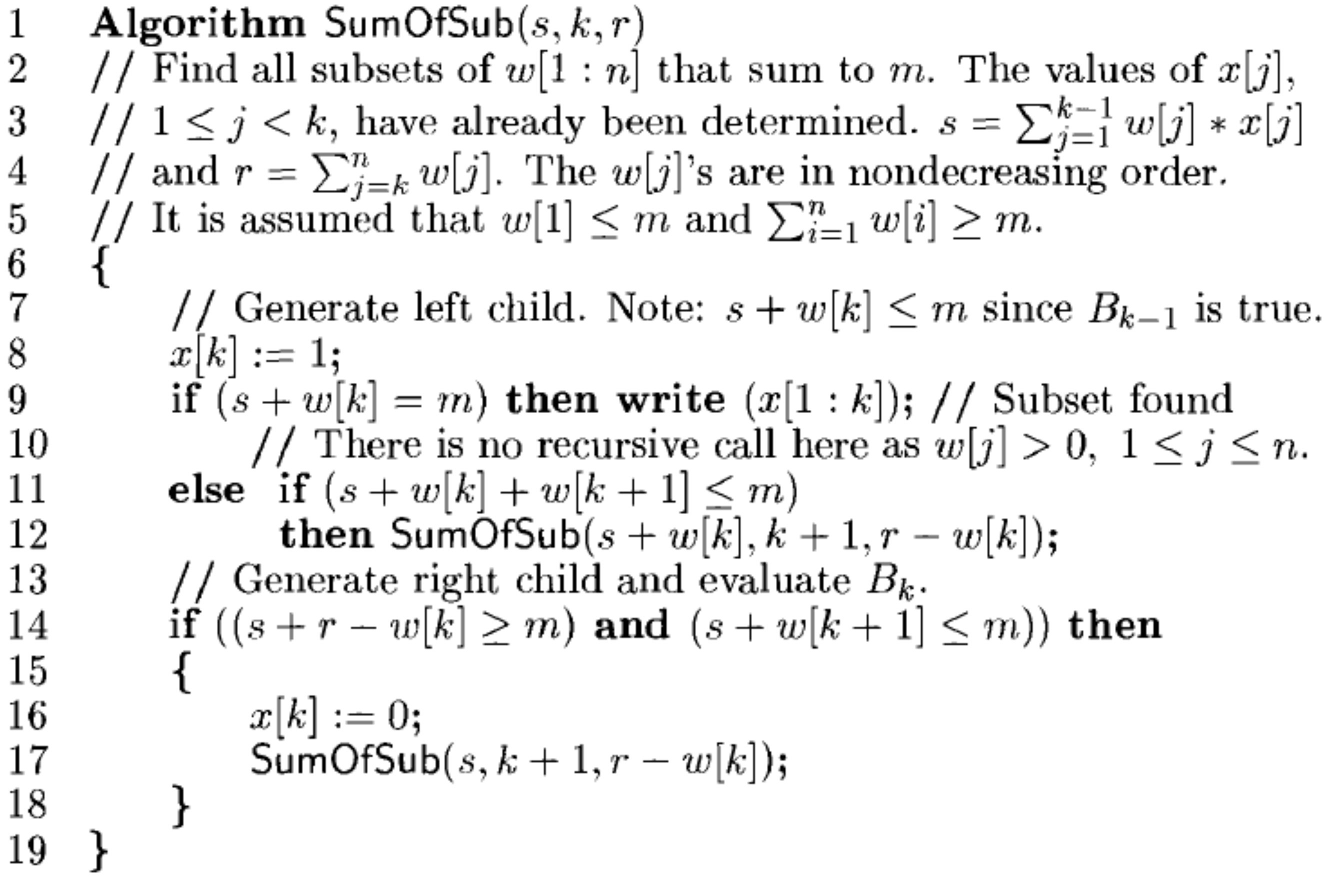
$x_k = 1$ $x_k = 0$

$$s + r \ge m \qquad s + r - w_k \ge m.$$

$x_k = 1$ $s + w_k + w_{k+1} \le m$ && $s + r \ge m.$

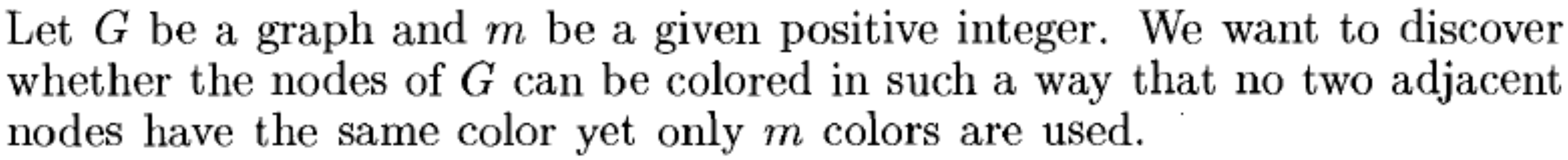
$x_k = 0$ $s + w_{k+1} \le m$ && $s + r - w_k \ge m.$

Algorithm

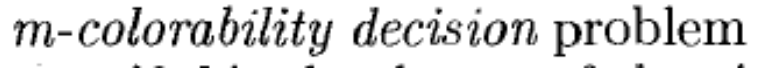


```
Algorithm SumOfSub($s, k, r$)
// Find all subsets of $w[1:n]$ that sum to $m$. The values of $x[j]$,
// $1 \leq j < k$, have already been determined. $s = \sum_{j=1}^{k-1} w[j] * x[j]$
// and $r = \sum_{j=k}^{n} w[j]$. The $w[j]$'s are in nondecreasing order.
// It is assumed that $w[1] \leq m$ and $\sum_{i=1}^{n} w[i] \geq m$.
{
    // Generate left child. Note: $s + w[k] \leq m$ since $B_{k-1}$ is true.
    $x[k] := 1$;
    if ($s + w[k] = m$) then write ($x[1:k]$); // Subset found
        // There is no recursive call here as $w[j] > 0,\ 1 \leq j \leq n$.
    else  if ($s + w[k] + w[k+1] \leq m$)
          then SumOfSub($s + w[k], k+1, r - w[k]$);
    // Generate right child and evaluate $B_k$.
    if (($s + r - w[k] \geq m$) and ($s + w[k+1] \leq m$)) then
    {
        $x[k] := 0$;
        SumOfSub($s, k+1, r - w[k]$);
    }
}
```

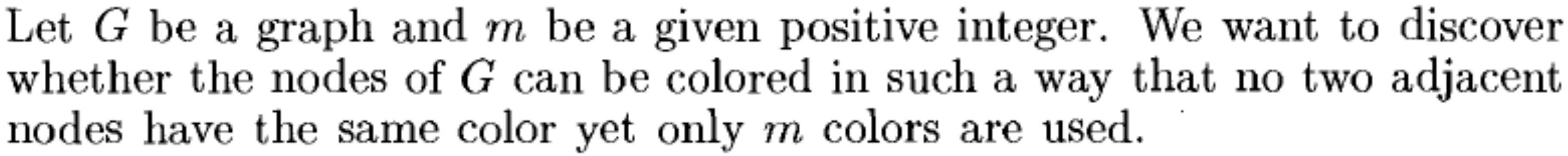
Graph Coloring



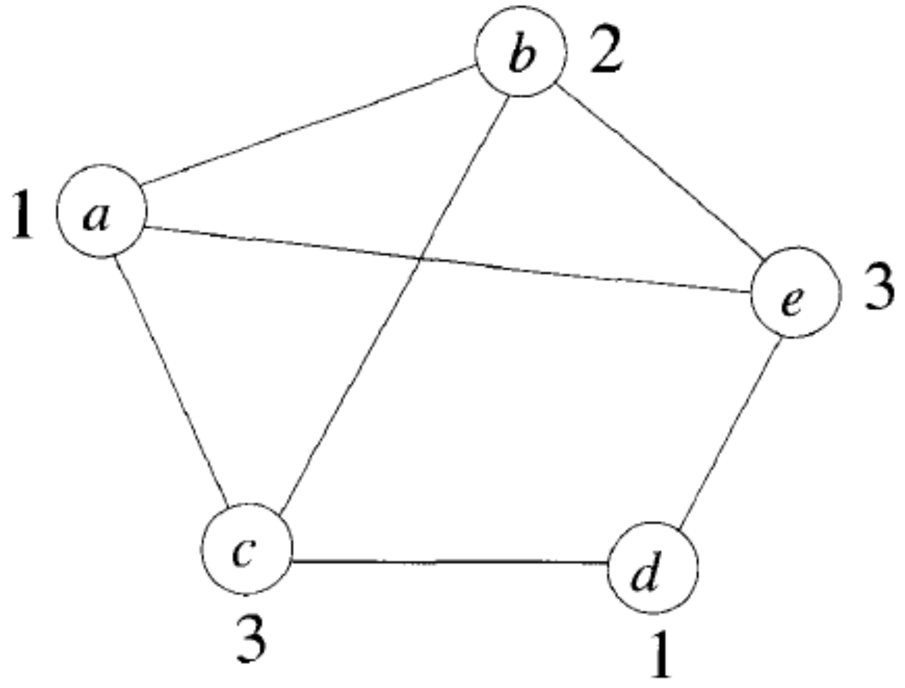
Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used.



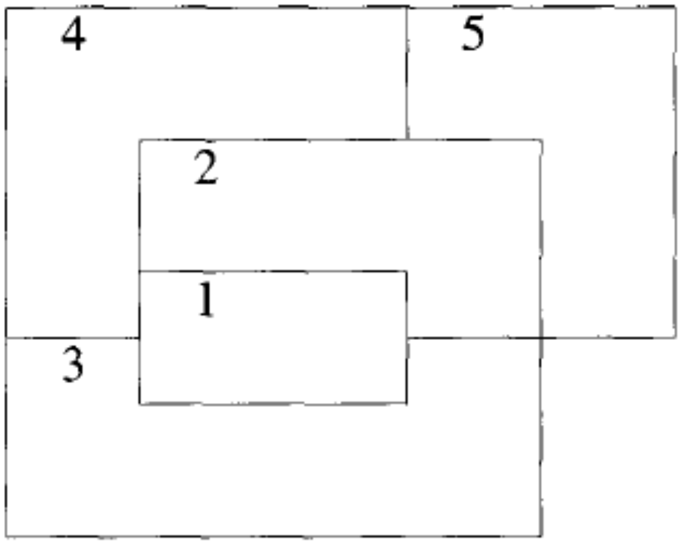
m-colorability decision problem

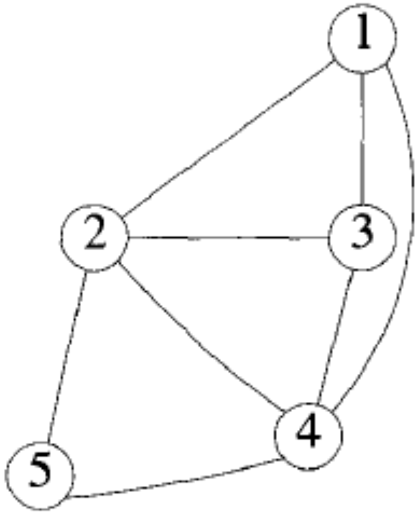


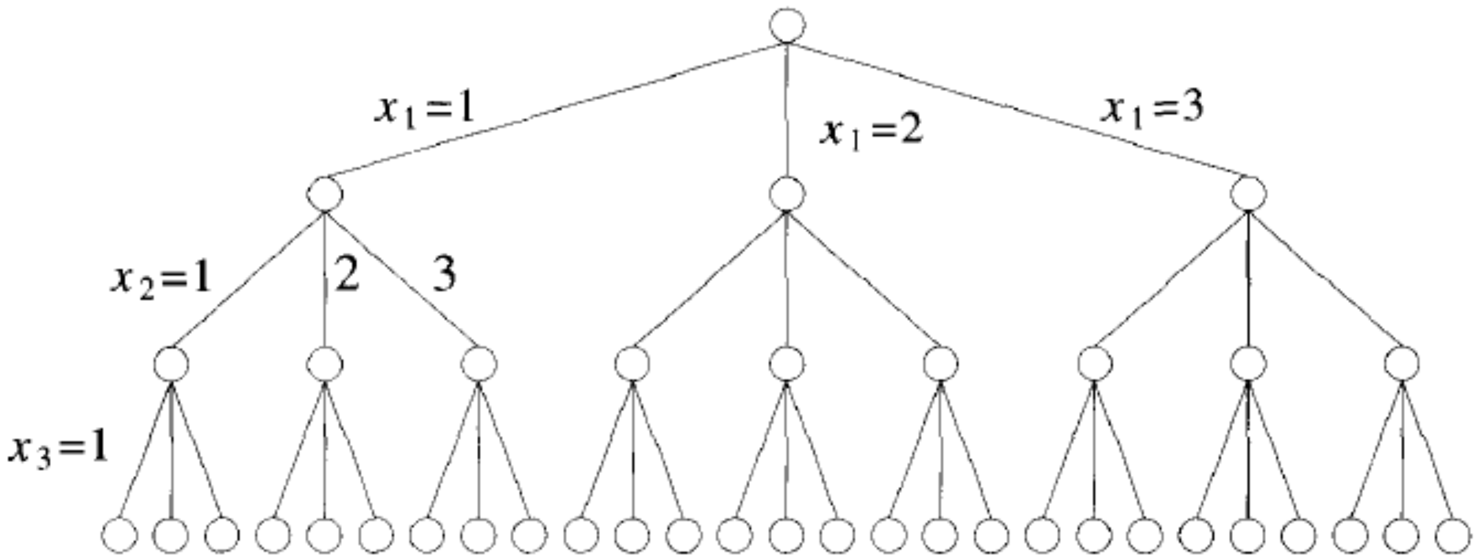
Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used.


b
2
1
a
e
3
c
d
3
1

A graph is said to be *planar* iff it can be drawn in a plane in such a way that no two edges cross each other.





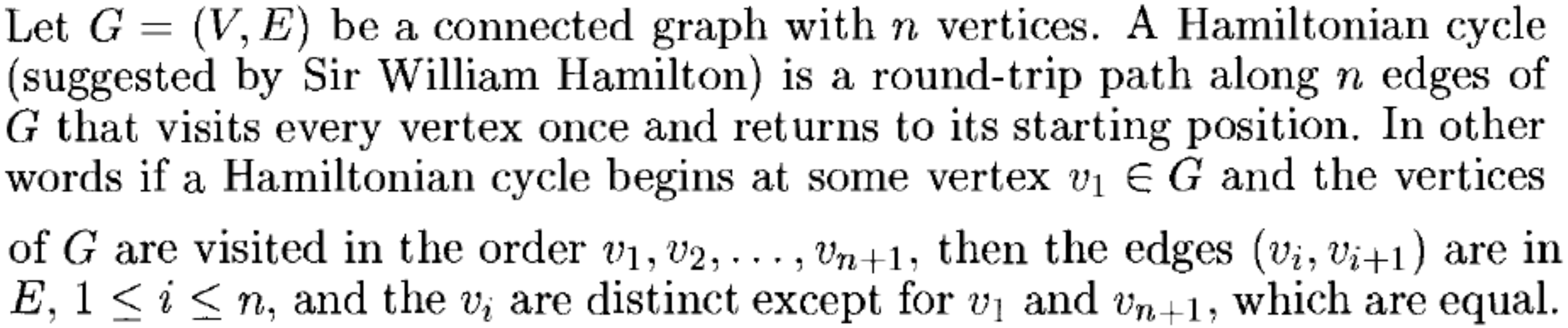


State space tree for mColoring when n =3 and m = 3

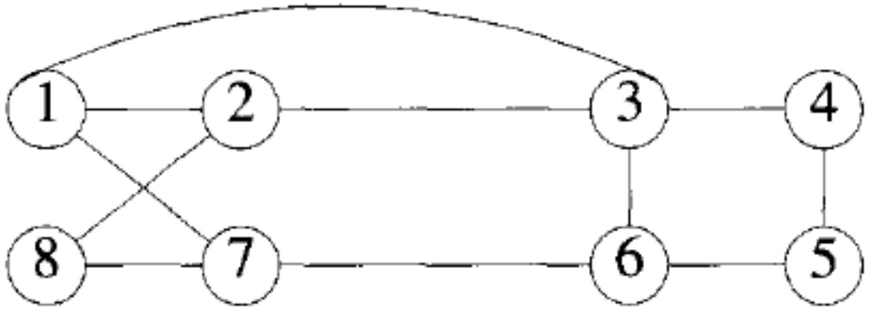
```
Algorithm mColoring($k$)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix $G[1:n, 1:n]$. All assignments of $1, 2, \ldots, m$ to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers 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);
}
```

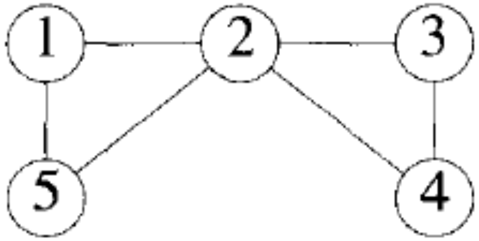
```
Algorithm NextValue($k$)
// $x[1], \ldots, x[k-1]$ have been assigned integer values in
// the range $[1, m]$ such that adjacent vertices have distinct
// integers. A value for $x[k]$ is determined in the range
// $[0, m]$. $x[k]$ is assigned the next highest numbered color
// while maintaining distinctness 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.
}
```

Hamiltonian Cycles



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 of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct except for v_1 and v_{n+1}, which are equal.

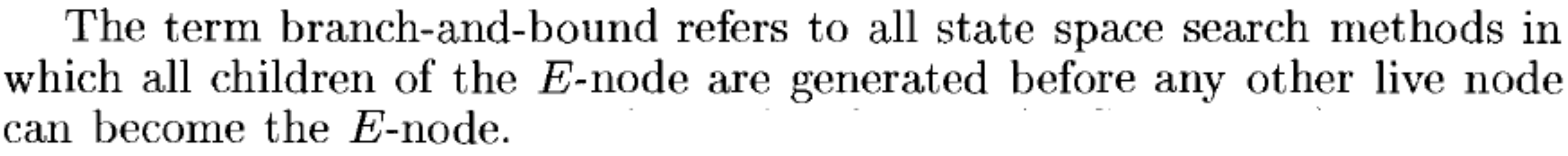

1
2
3
4
8
7
6
5


1
2
3
5
4

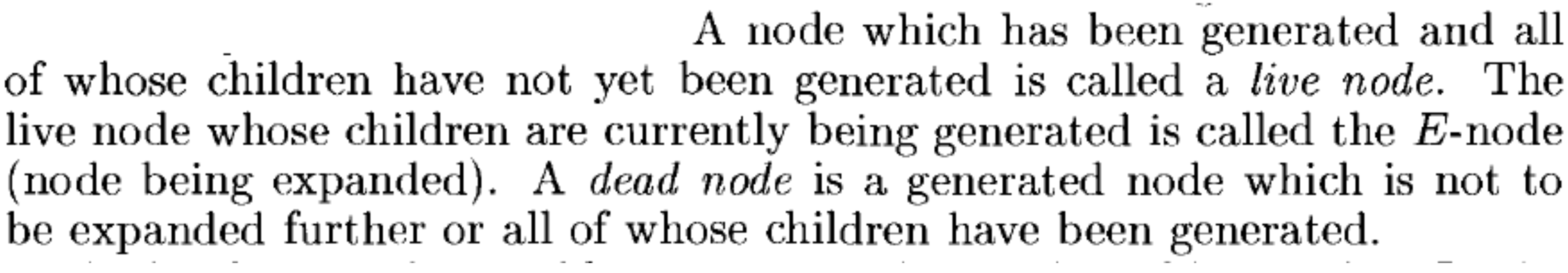
```
Algorithm NextValue($k$)
// $x[1:k-1]$ is a path of $k-1$ distinct vertices. If $x[k]=0$, then
// no vertex has as yet been assigned to $x[k]$. After execution,
// $x[k]$ is assigned to the next highest numbered vertex which
// does not already appear in $x[1:k-1]$ and is connected by
// an edge 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);
}
```

```
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);
}
```

Branch and Bound



The term branch-and-bound refers to all state space search methods in which all children of the *E*-node are generated before any other live node can become the *E*-node.



A node which has been generated and all of whose children have not yet been generated is called a *live node*. The live node whose children are currently being generated is called the *E*-node (node being expanded). A *dead node* is a generated node which is not to be expanded further or all of whose children have been generated.

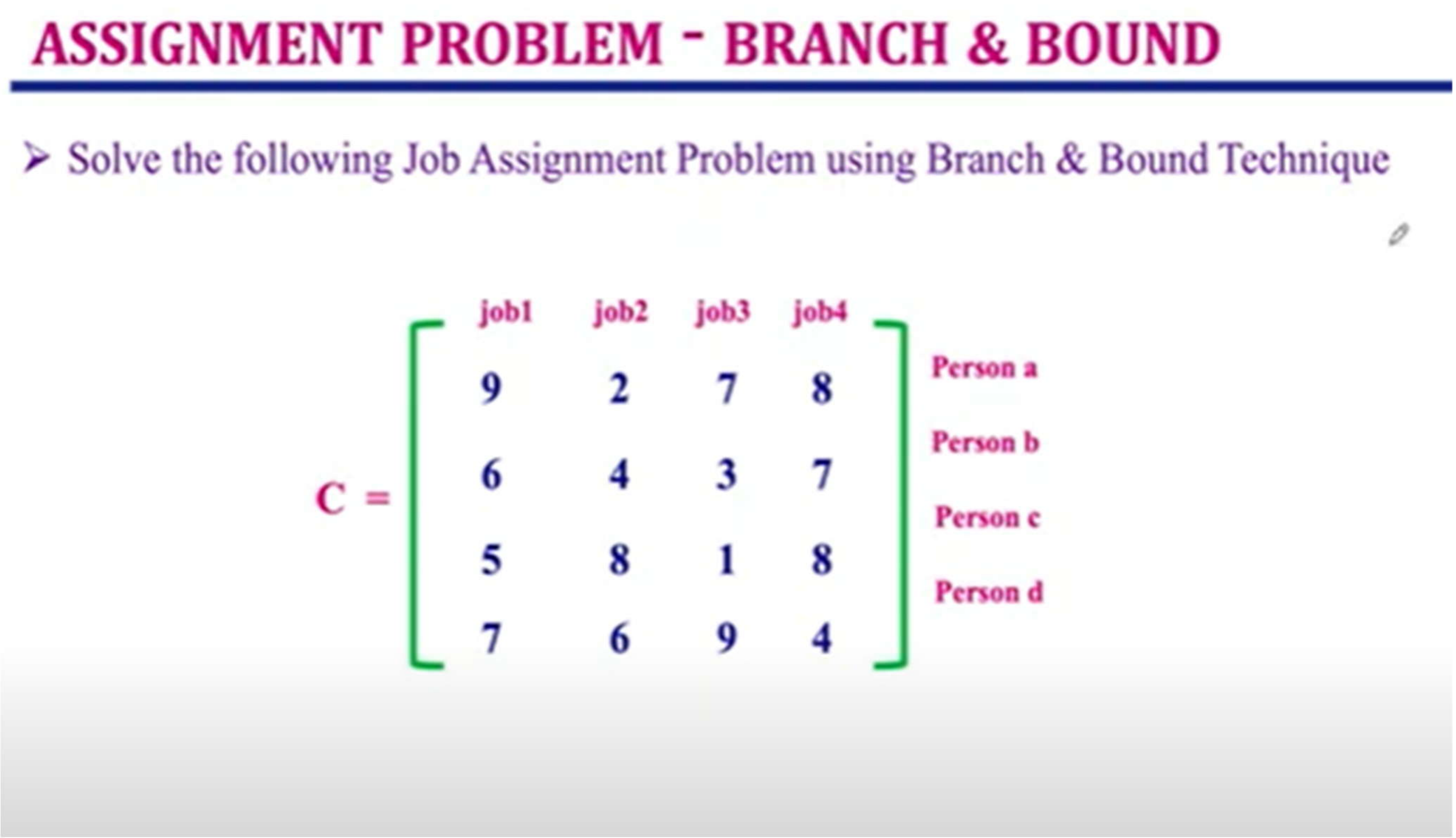
- LC Branch and Bound
- FIFO Branch and Bound
- LIFO Branch and Bound

1.Assignment Problem

2.Knapsack Problem

3.TSP

4. 15 Puzzle Problem



ASSIGNMENT PROBLEM - BRANCH & BOUND

- Solve the following Job Assignment Problem using Branch & Bound Technique

$$C = \begin{array}{cccc|l} \text{job1} & \text{job2} & \text{job3} & \text{job4} & \\ 9 & 2 & 7 & 8 & \text{Person a} \\ 6 & 4 & 3 & 7 & \text{Person b} \\ 5 & 8 & 1 & 8 & \text{Person c} \\ 7 & 6 & 9 & 4 & \text{Person d} \end{array}$$

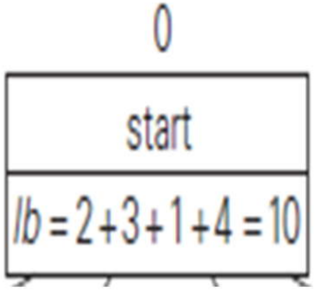
ASSIGNMENT PROBLEM - BRANCH & BOUND

Initial Lower Bound

Take minimum of each row and add them

	j1	j2	j3	j4	
a	9	2	7	8	⇒ 2
b	6	4	3	7	⇒ 3
c	5	8	1	8	⇒ 1
d	7	6	9	4	⇒ 4

Lower bound lb = 10



0

start

$lb = 2+3+1+4 = 10$

ASSIGNMENT PROBLEM - BRANCH & BOUND

Consider Person a

Assign various jobs to person a and compute lower bound

j1

	j1	j2	j3	j4	
a	9	~~2~~	~~7~~	~~8~~	⇒ 9
b	~~6~~	4	3	7	⇒ 3
c	~~5~~	8	1	8	⇒ 1
d	~~7~~	6	9	4	⇒ 4

lb = 17

j2

	j1	j2	j3	j4	
a	~~9~~	2	~~7~~	~~8~~	⇒ 2
b	6	~~4~~	3	7	⇒ 3
c	5	~~8~~	1	8	⇒ 1
d	7	~~6~~	9	4	⇒ 4

lb = 10

j3

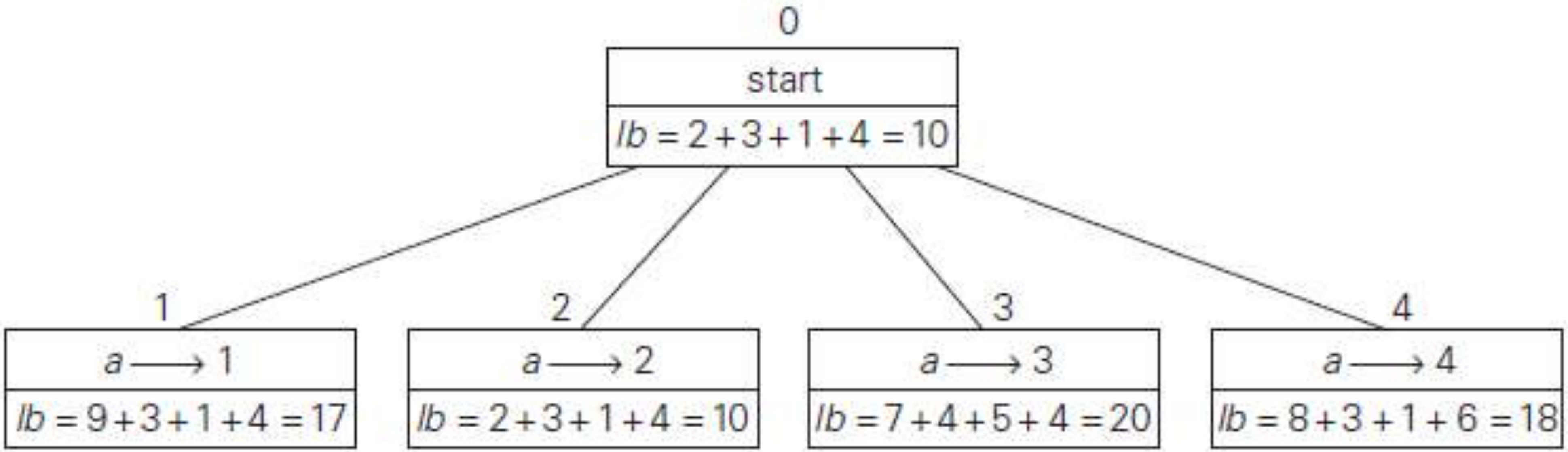
	j1	j2	j3	j4	
a	~~9~~	~~2~~	7	~~8~~	⇒ 7
b	6	4	~~3~~	7	⇒ 4
c	5	8	~~1~~	8	⇒ 5
d	7	6	~~9~~	4	⇒ 4

lb ⇒ 20

j4

	j1	j2	j3	j4	
a	~~9~~	~~2~~	~~7~~	8	⇒ 8
b	6	4	3	~~7~~	⇒ 3
c	5	8	1	~~8~~	⇒ 1
d	7	6	9	~~4~~	⇒ 6

lb ⇒ 18


0
start
lb = 2+3+1+4 = 10
1
a → 1
lb = 9+3+1+4 = 17
2
a → 2
lb = 2+3+1+4 = 10
3
a → 3
lb = 7+4+5+4 = 20
4
a → 4
lb = 8+3+1+6 = 18

Consider Person b

a

Assign various jobs to person b by leaving job 2 which is assigned to person a and compute lower bound

b → j1

	j1	j2	j3	j4	
a		2			⇒ 2
b	6		~~3~~	~~7~~	⇒ 6
c	5		1	8	⇒ 1
d	7		9	4	⇒ 4

lb = 13

b → j3

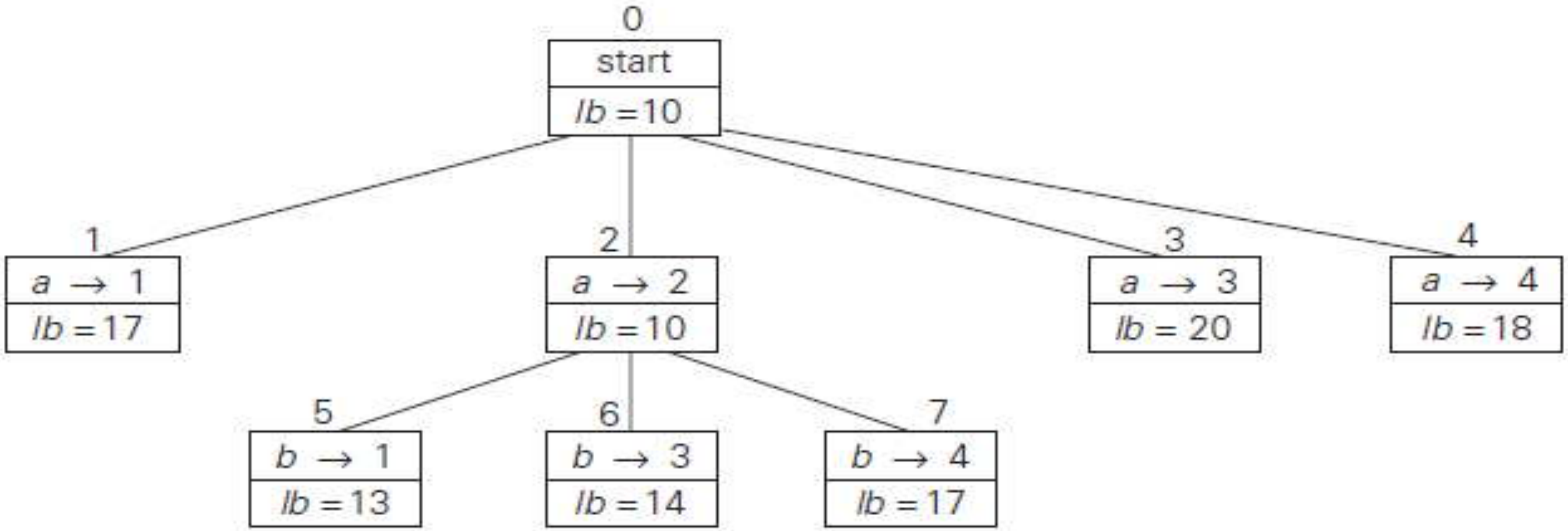
	j1	j2	j3	j3	
a		2			⇒ 2
b	~~6~~		3	~~7~~	⇒ 3
c	5		1	8	⇒ 5
d	7		9	4	⇒ 4

lb = 14

b → j4

	j1	j2	j3	j4	
a		2			⇒ 2
b	~~6~~		~~3~~	7	⇒ 7
c	5		1	8	⇒ 1
d	7		9	4	⇒ 7

lb = 17


0
start
lb = 10
1
a → 1
lb = 17
2
a → 2
lb = 10
3
a → 3
lb = 20
4
a → 4
lb = 18
5
b → 1
lb = 13
6
b → 3
lb = 14
7
b → 4
lb = 17

Consider Person C

→ a → b

Assign various jobs to person c by leaving job 2 & job 1 which is assigned to person a and person b, compute lower bound

c → j3

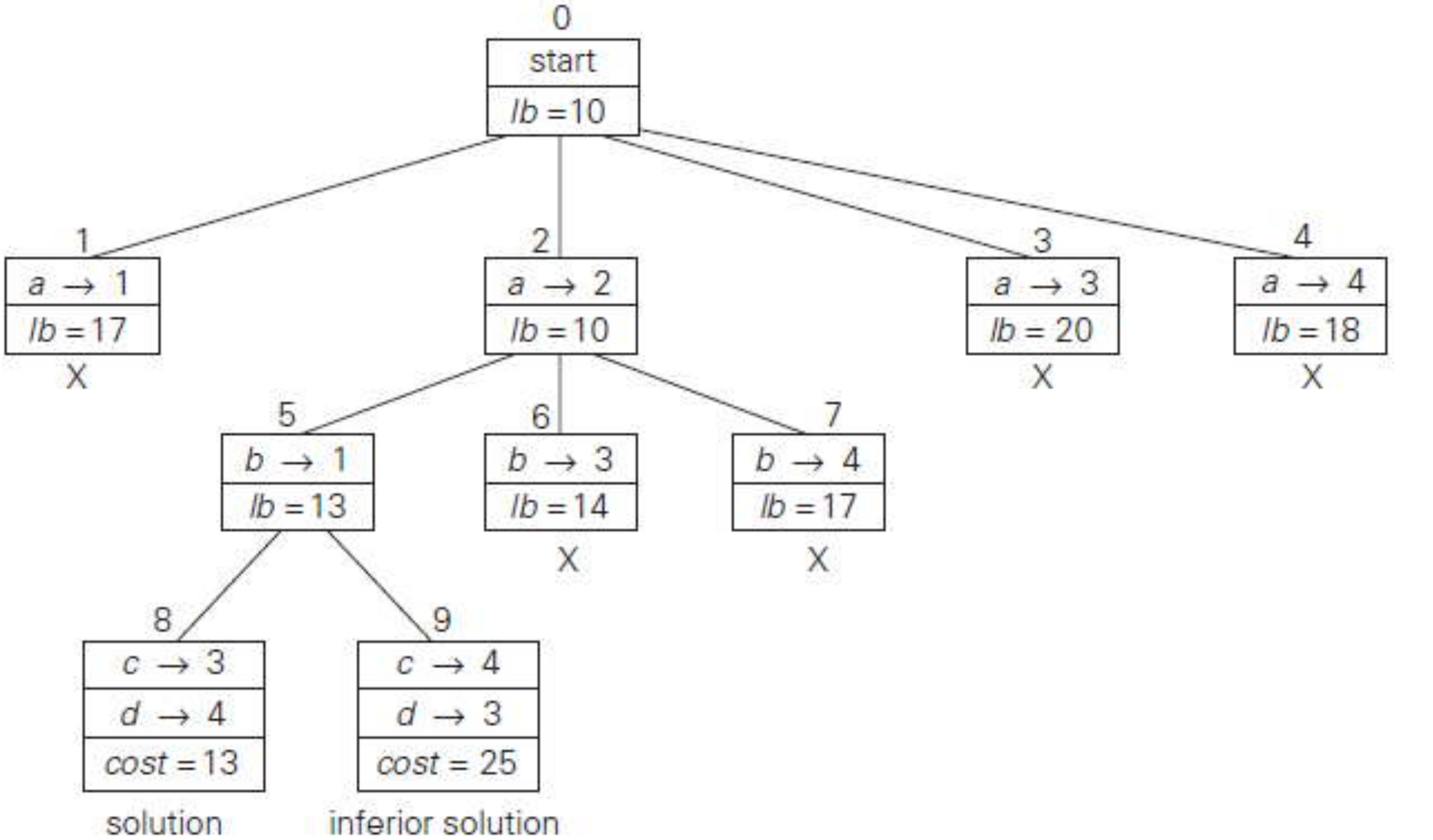
	j1	j2	j3	j4	
a		2			⇒ 2
b	6				⇒ 6
c			1	~~8~~	⇒ 1
d			9	4	⇒ 4

lb = 13

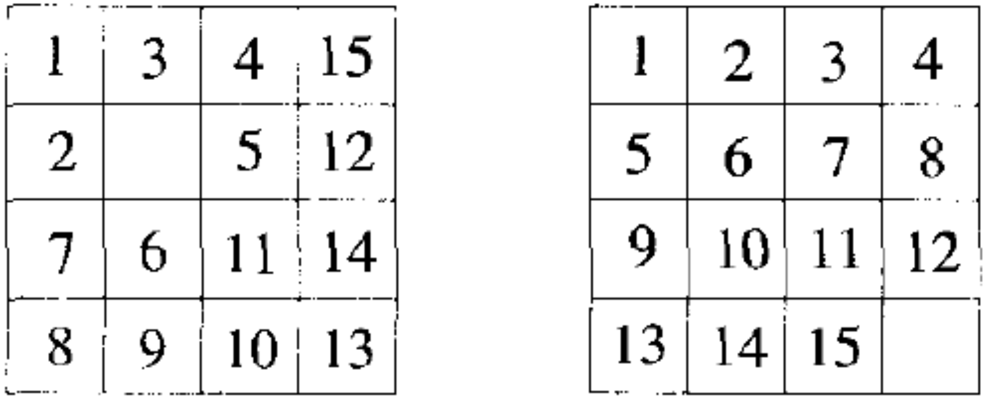
c → j4

	j1	j2	j3	j4	
a		2			⇒ 2
b	6				⇒ 6
c			~~1~~	8	⇒ 8
d			9	4	⇒ 9

lb = 25


0
start
lb =10
1
a → 1
lb =17
X
2
a → 2
lb =10
3
a → 3
lb = 20
X
4
a → 4
lb =18
X
5
b → 1
lb = 13
6
b → 3
lb =14
X
7
b → 4
lb = 17
X
8
c → 3
d → 4
cost =13
solution
9
c → 4
d → 3
cost = 25
inferior solution

15 Puzzle Problem

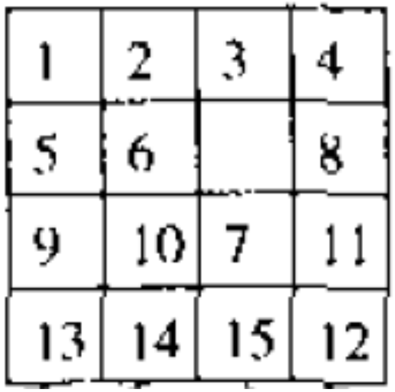


1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

(a) An arrangement

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

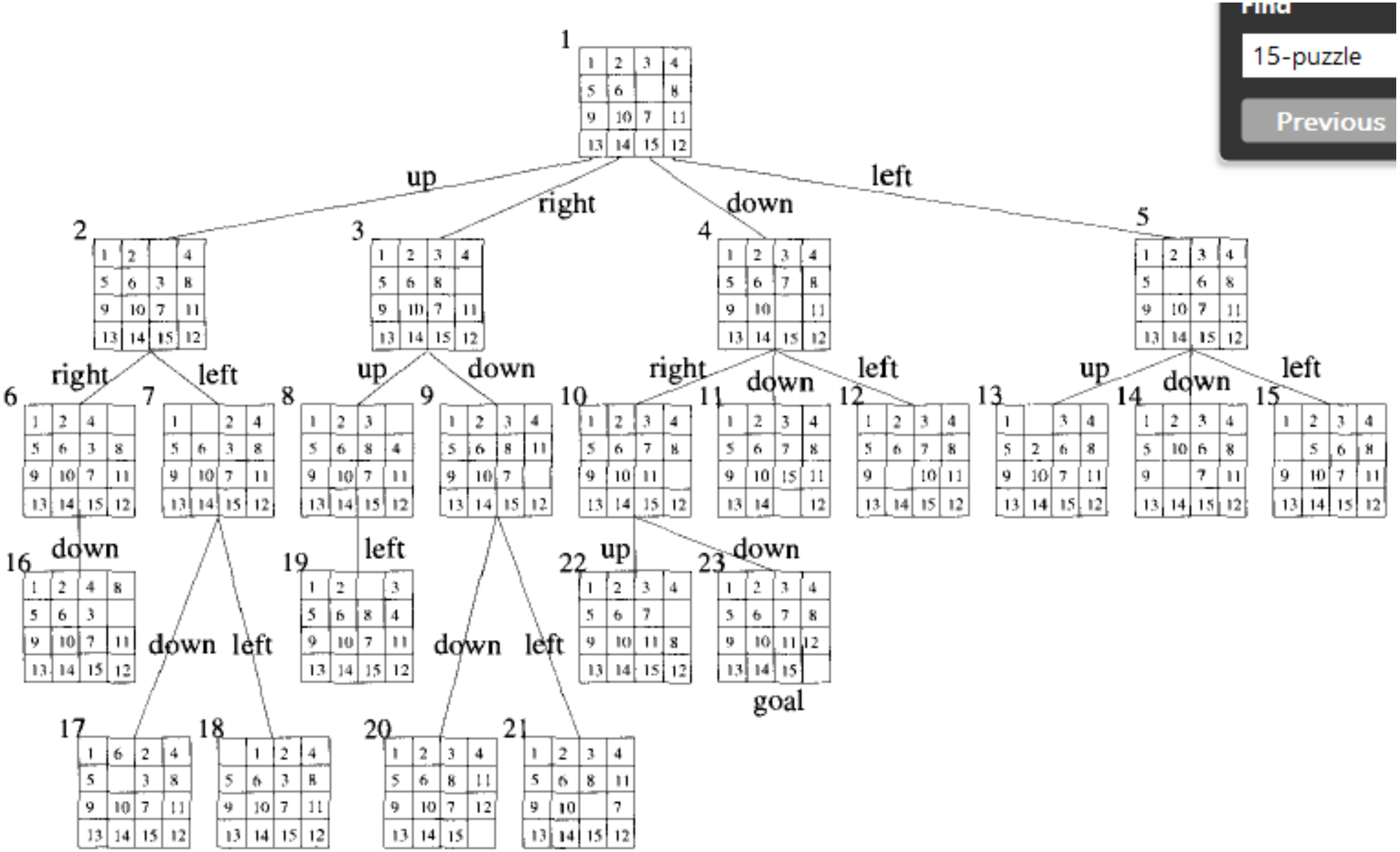
(b) Goal arrangement



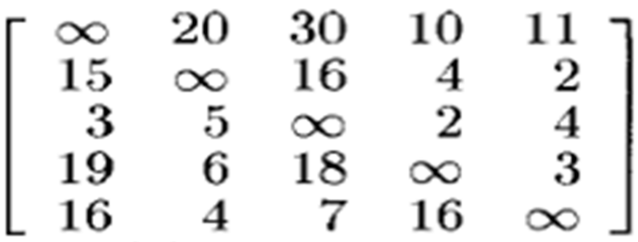
1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

We can arrive at an easy to compute estimate $\hat{c}(x)$ of $c(x)$. We can write $\hat{c}(x) = f(x) + \hat{g}(x)$, where $f(x)$ is the length of the path from the root to node x and $\hat{g}(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x. One possible choice for $\hat{g}(x)$ is

$$\hat{g}(x) = \text{number of nonblank tiles not in their goal position}$$

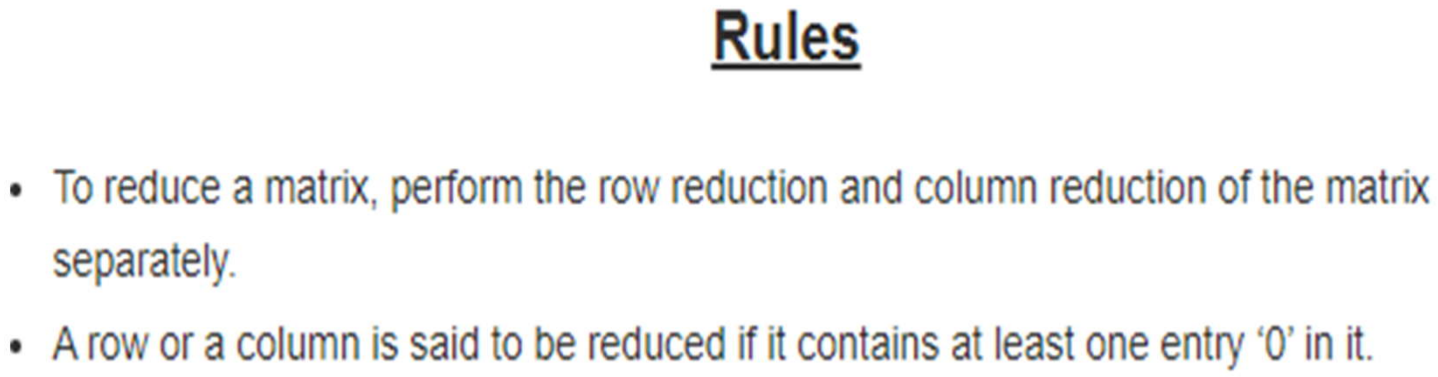

Find
15-puzzle
Previous
1
up
right
down
left
2
3
4
5
right
left
up
down
right
down
left
up
down
left
6
7
8
9
10
11
12
13
14
15
down
left
up
down
16
17
18
19
20
21
22
23
down
left
down
left
goal

TSP



$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost matrix



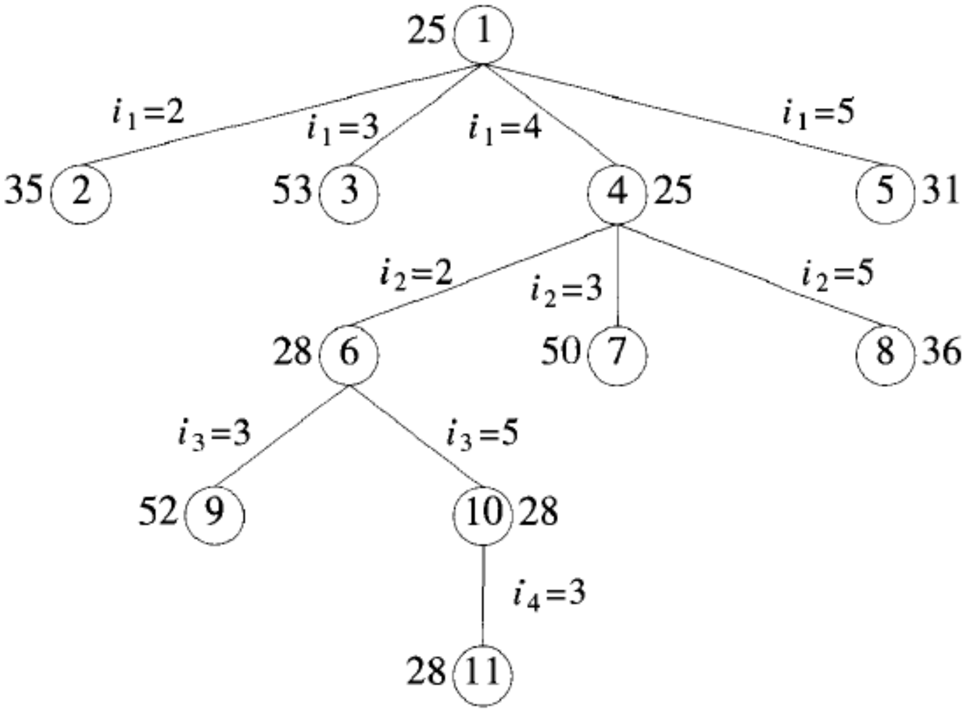
Rules

- To reduce a matrix, perform the row reduction and column reduction of the matrix separately.
- A row or a column is said to be reduced if it contains at least one entry ‘0’ in it.

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced cost matrix
L = 25

Let A be the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge $\langle i, j\rangle$ in the tour. If S is not a leaf, then the reduced cost matrix for S may be obtained as follows: (1) Change all entries in row i and column j of A to ∞. This prevents the use of any more edges leaving vertex i or entering vertex j. (2) Set $A(j, 1)$ to ∞. This prevents the use of edge $\langle j, 1\rangle$. (3) Reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞. Let the resulting matrix be B. Steps (1) and (2) are valid as no tour in the subtree s can contain edges of the type $\langle i, k\rangle$ or $\langle k, j\rangle$ or $\langle j, 1\rangle$ (except for edge $\langle i, j\rangle$). If r is the total amount subtracted in step (3) then $\hat{c}(S) = \hat{c}(R) + A(i, j) + r$.


25 1
i_1=2
i_1=3
i_1=4
i_1=5
35 2
53 3
4 25
5 31
i_2=2
i_2=3
i_2=5
28 6
50 7
8 36
i_3=3
i_3=5
52 9
10 28
i_4=3
28 11

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1,4,5; node 8

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

(h) Path 1,4,2,3; node 9

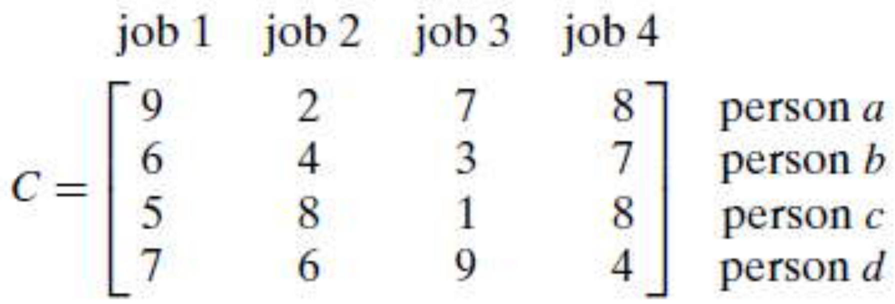
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

(i) Path 1,4,2,5; node 10

Consider the traveling salesperson instance defined by the cost matrix

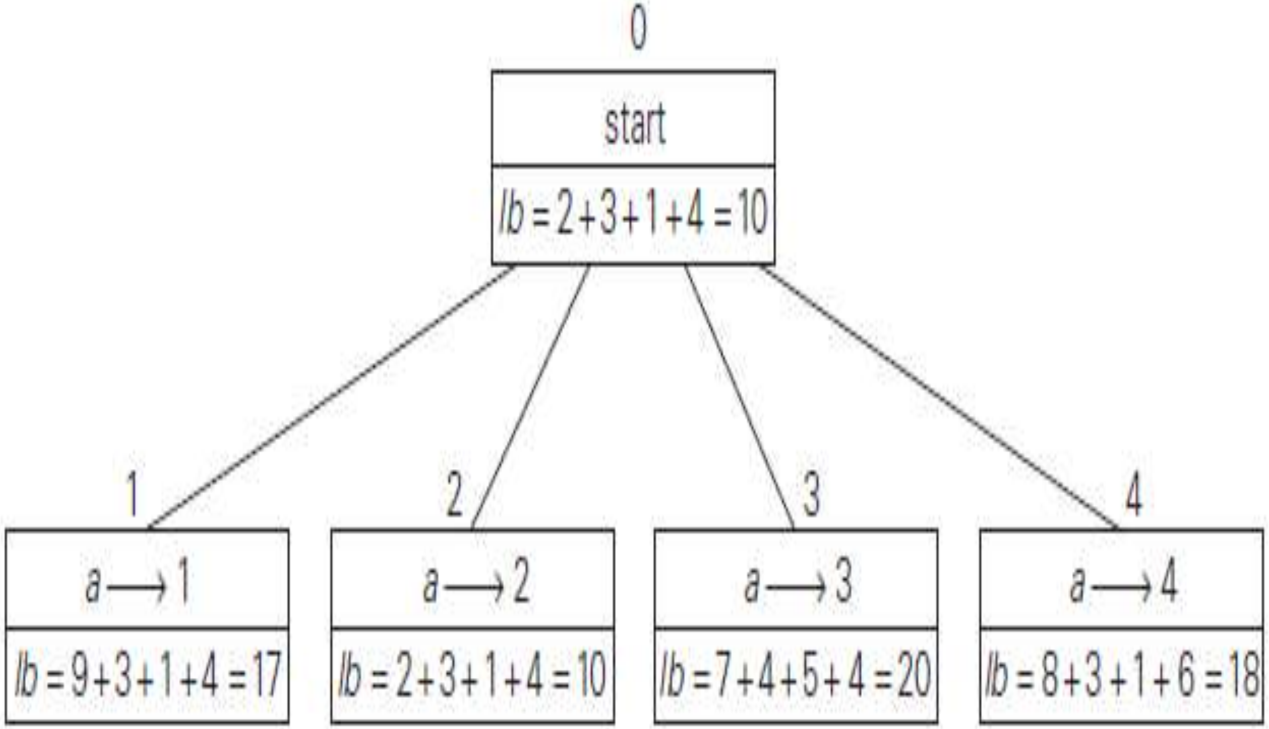
$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

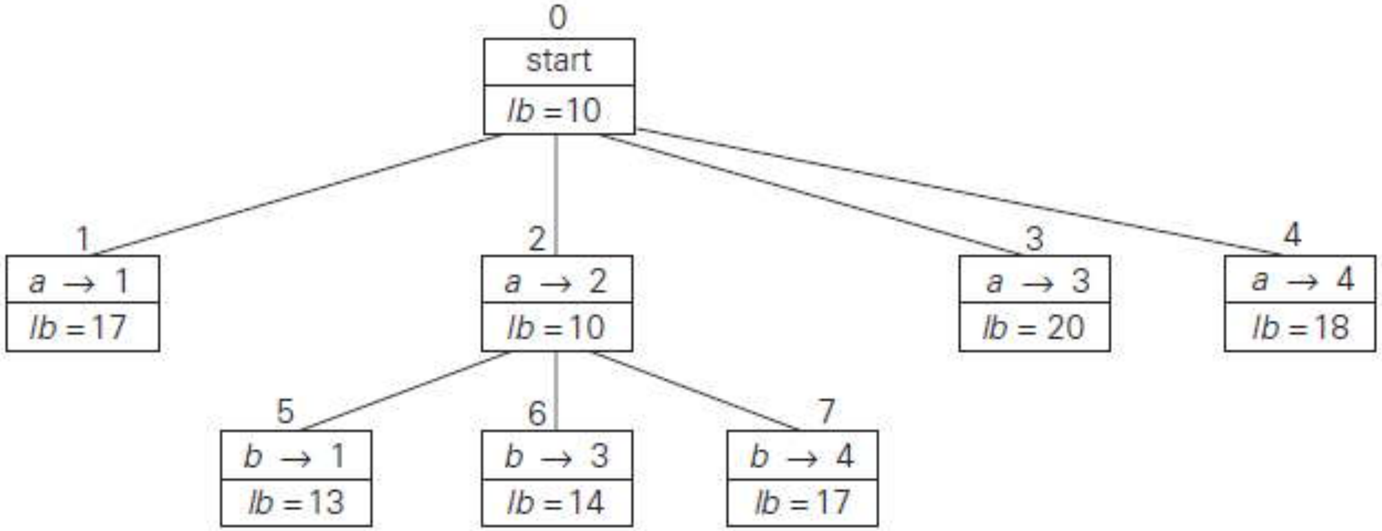
Assignment Problem

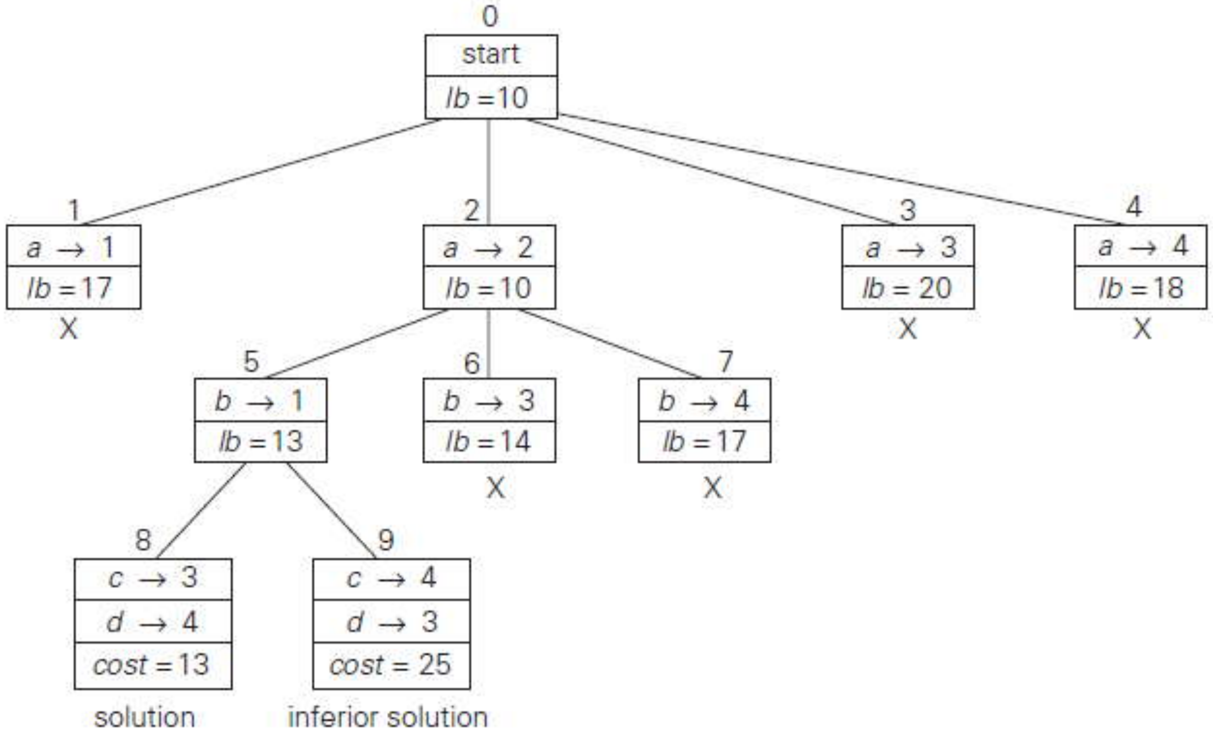


$$C = \begin{array}{c c c c c}
\text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} & \\
\left[\begin{array}{c} 9 \\ 6 \\ 5 \\ 7 \end{array}\right. & \begin{array}{c} 2 \\ 4 \\ 8 \\ 6 \end{array} & \begin{array}{c} 7 \\ 3 \\ 1 \\ 9 \end{array} & \left.\begin{array}{c} 8 \\ 7 \\ 8 \\ 4 \end{array}\right] & \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}
\end{array}$$

- instance of the assignment problem is specified by an $n \times n$ cost matrix *C* so that 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.
- we start with the root that corresponds to no elements selected from the cost matrix.
- the lower-bound value for the root, denoted *lb*, is 10. The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person *a*

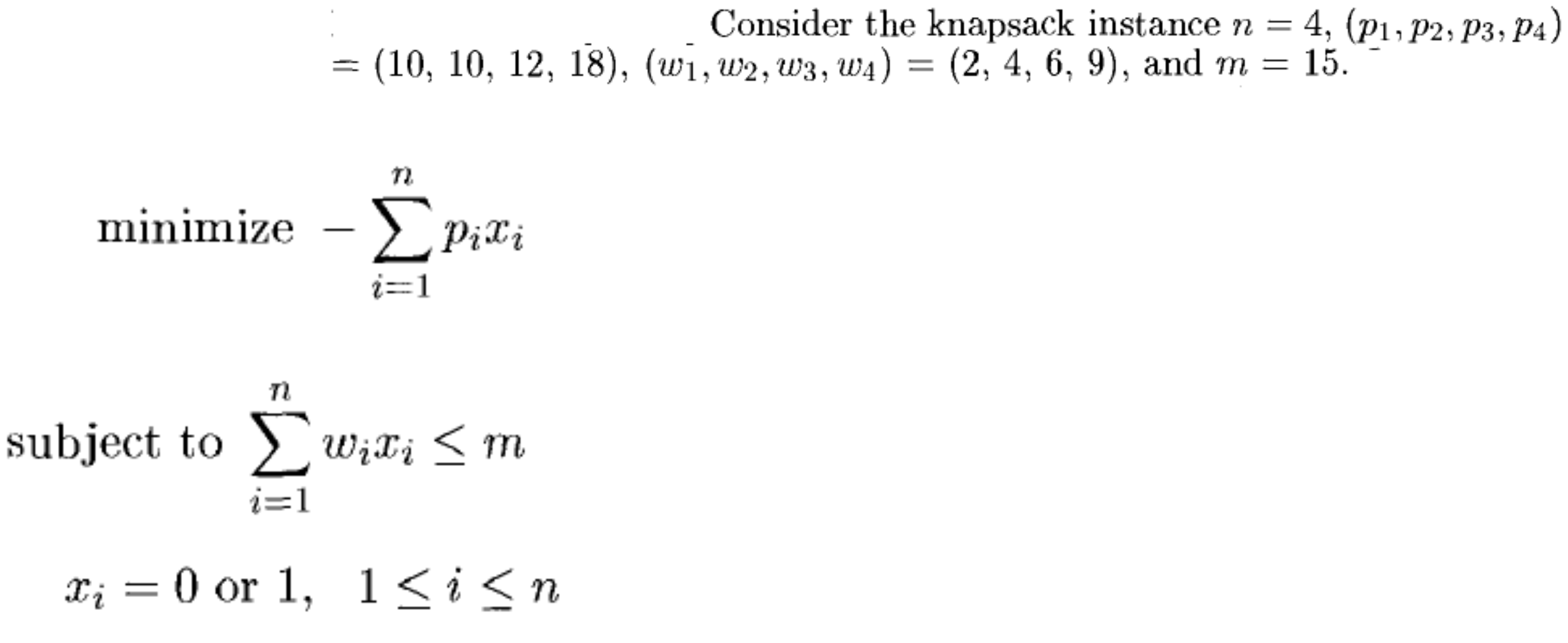

0
start
lb = 2+3+1+4 = 10
1
a ⟶ 1
lb = 9+3+1+4 = 17
2
a ⟶ 2
lb = 2+3+1+4 = 10
3
a ⟶ 3
lb = 7+4+5+4 = 20
4
a ⟶ 4
lb = 8+3+1+6 = 18


0
start
lb = 10
1
a → 1
lb = 17
2
a → 2
lb = 10
3
a → 3
lb = 20
4
a → 4
lb = 18
5
b → 1
lb = 13
6
b → 3
lb = 14
7
b → 4
lb = 17



0
start
lb = 10
1
a → 1
lb = 17
X
2
a → 2
lb = 10
3
a → 3
lb = 20
X
4
a → 4
lb = 18
X
5
b → 1
lb = 13
6
b → 3
lb = 14
X
7
b → 4
lb = 17
X
8
c → 3
d → 4
cost = 13
solution
9
c → 4
d → 3
cost = 25
inferior solution

Knapsack Problem



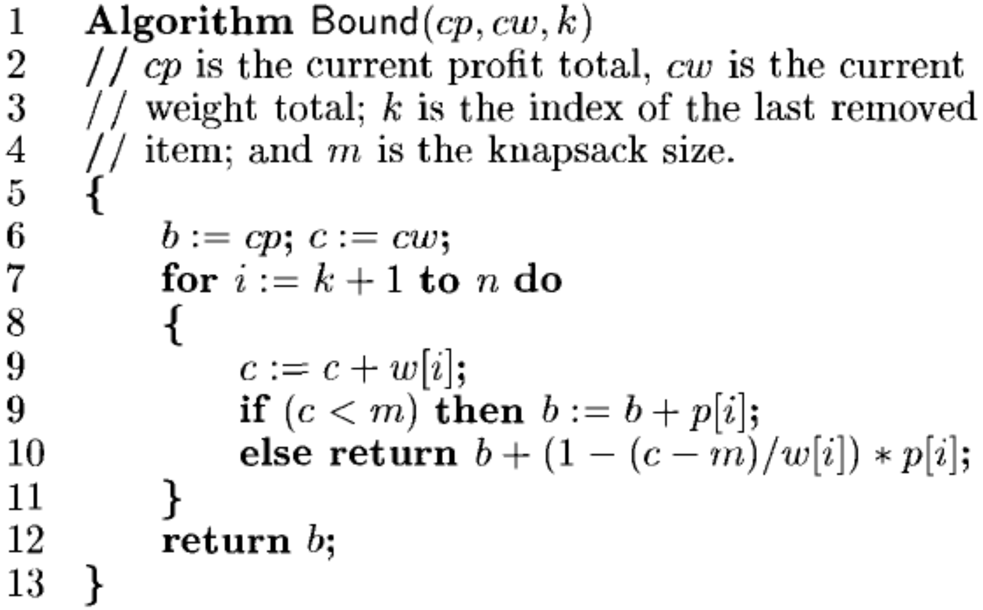
Consider the knapsack instance $n = 4$, $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$, $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$, and $m = 15$.

$$\text{minimize} \ -\sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \ \sum_{i=1}^{n} w_i x_i \le m$$

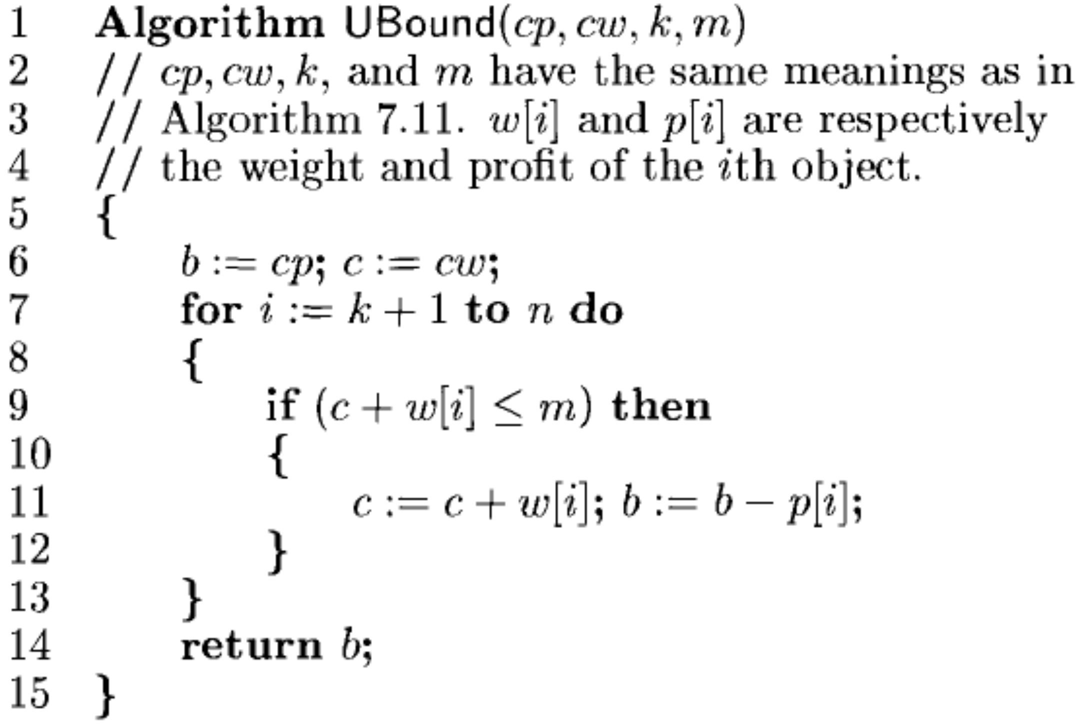
$$x_i = 0 \text{ or } 1, \quad 1 \le i \le n$$

Bound Function



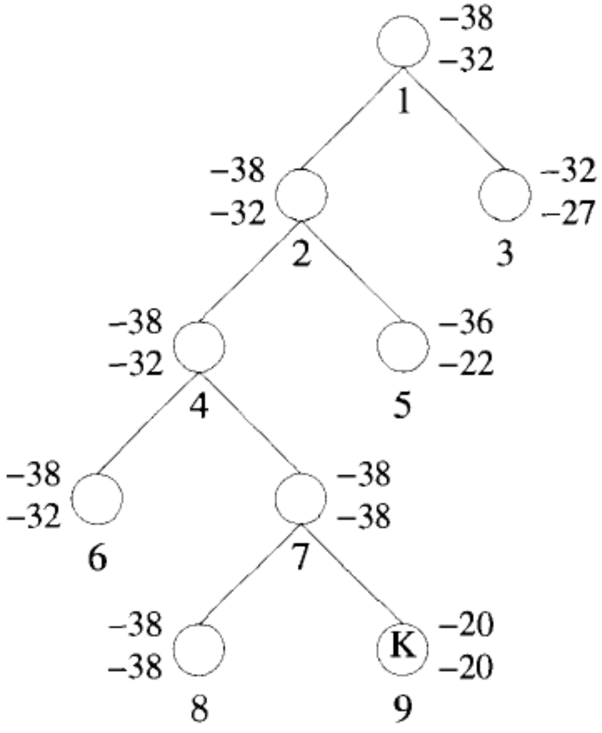
```
Algorithm Bound($cp, cw, k$)
// $cp$ is the current profit total, $cw$ is the current
// weight total; $k$ is the index of the last removed
// item; and $m$ is the knapsack size.
{
    $b := cp$; $c := cw$;
    for $i := k + 1$ to $n$ do
    {
        $c := c + w[i]$;
        if $(c < m)$ then $b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

UBound



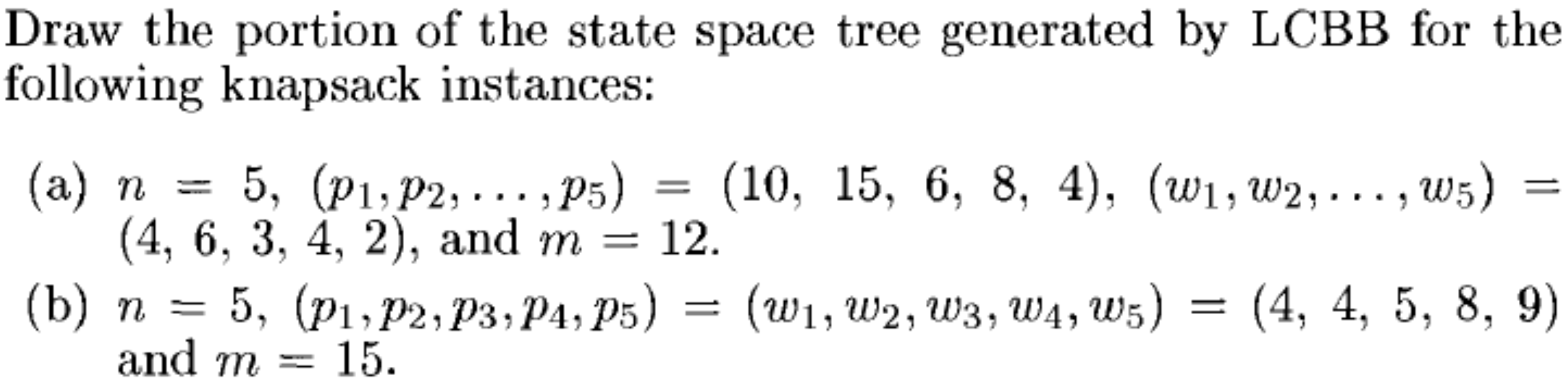
```
Algorithm UBound($cp, cw, k, m$)
// $cp, cw, k$, and $m$ have the same meanings as in
// Algorithm 7.11. $w[i]$ and $p[i]$ are respectively
// the weight and profit of the $i$th object.
{
    $b := cp$; $c := cw$;
    for $i := k + 1$ to $n$ do
    {
        if $(c + w[i] \leq m)$ then
        {
            $c := c + w[i]$; $b := b - p[i]$;
        }
    }
    return $b$;
}
```

State Space Tree



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

Examples



Draw the portion of the state space tree generated by LCBB for the following knapsack instances:

(a) $n = 5$, $(p_1, p_2, \ldots, p_5) = (10,\ 15,\ 6,\ 8,\ 4)$, $(w_1, w_2, \ldots, w_5) = (4,\ 6,\ 3,\ 4,\ 2)$, and $m = 12$.

(b) $n = 5$, $(p_1, p_2, p_3, p_4, p_5) = (w_1, w_2, w_3, w_4, w_5) = (4,\ 4,\ 5,\ 8,\ 9)$ and $m = 15$.