**CMR ENGINEERING COLLEGE**

Accredited by NBA,Affililated to JNTU,Hyderabad

Kandlakoya(v), Medchal Road,Hyderabad-501401

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

# ACADEMIC YEAR – 2018-2019

**Subject:** Design and Analysis Algorithm **Year/Semester**: III-I, Section-C

**Innovative practice:** Flipped Class **Type of Learning:** Collaborative Learning

**Topic:** Backtracking Techniques **Date:** 18.09.2018

# Innovative Practice Description

* **Unit / Topic:** Unit V **/** Backtracking Techniques

# Course Outcome: CO5

* **Topic Learning Outcome:** TLO11
* **Activity Chosen:** Flipped Classroom

# Justification:

Backtracking is one of the important topics, repeatedly asked in university questions. This activity makes the students to get a sound knowledge in this concept. Students can prepare individually about a topic by watching the lecture video and share their ideas with their classmates that enhance their knowledge and oral communication skills.

* **Time Allotted for the Activity:** 30 Minutes

# Details of the Implementation:

* + Learning materials such as video and documents about backtracking algorithms were sent to the student’s mail and Canvas, one week before conducting this activity.
  + Instruct the students to watch the video individually at their home and take the notes according to their understanding level.
  + On the day of the activity, small teams were formed by their own interest.
  + Each student in a group discusses among the peer members in the concept of backtracking algorithms based on their self-learning and write the points for Presentation within 10 minutes.
  + Once they prepared the content for presentation with their peer members, each group present about the different problems solved using backtracking algorithms for maximum of 5 minutes.

# CO – PO / PSO mapping:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CO** | **PO1** | **PO2** | **PO3** | **PO4** | **PO10** | **PSO1** | **PSO2** | **PSO3** |
| **CO1** | 3 | 1 | 1 | 1 | 2 | 2 | 2 | 1 |

**(1 – Low 2 – Moderate 3 – High)**

# PO / PSO mapped:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Innovative practice** | **PO1** | **PO2** | **PO3** | **PO4** |
| 3 | 1 | 1 | 1 |
| **Justification for correlation** | Backtracking technique is used for analysis of algorithm. | Backtracking technique is used in analysis of engineering problems | To identify solution for complex problems using backtracking technique | To identify solution for complex problems using backtracking algorithm. |
| **PO10** | **PSO1** | **PSO2** | **PSO3** |
| 2 | 2 | 2 | 1 |
| To present solution for complex problem solutions with backtracking algorithm. | Able to use these algorithm design techniques in Information Technology | Able to use these algorithms in analyzing and developing reliable IT Solutions | Algorithm design techniques are helpful in solving real world problems in Industry and  Research. |

* **Images / Screenshot of the practice:**



# Reflective Critique:

## Feedback of practice from students and other stakeholders:

* + The students were actively participated.
  + Students were able to explored their knowledge on the backtracking algorithms to solve n-queens, subset sum and Hamiltonian circuit problems.
  + This activity makes all the students to gain well knowledge on how to apply backtracking techniques on different problems.

## Benefit of the practice:

* + Students were actively participated in this activity.
  + From this activity, the students can get more clarity in the particular topic by discussing and sharing their views with the other students in the class.
  + Learning at home automatically becomes student-centric rather than teacher- centric.

## Challenges faced in implementation:

* + Due to time constraints, only few group of students were made to share the concepts by a random call.
  + Some team members are hesitated and lack to share the concepts that they learnt and discussed.

# Faculty Incharge HOD(CSE)

# CS603PC: DESIGN AND ANALYSIS OF ALGORITHMS

**III Year B.Tech. CSE II-Sem L T P C**

# 3 1 0 4

**Prerequisites:**

1. A course on “Computer Programming and Data Structures”
2. A course on “Advanced Data Structures”

# Course Objectives:

* Introduces the notations for analysis of the performance of algorithms.
* Introduces the data structure disjoint sets.
* Describes major algorithmic techniques (divide-and-conquer, backtracking, dynamic programming, greedy, branch and bound methods) and mention problems for which each technique is appropriate;
* Describes how to evaluate and compare different algorithms using worst-, average-, and best- case analysis.
* Explains the difference between tractable and intractable problems, and introduces the problems that are P, NP and NP complete.

# Course Outcomes:

* Ability to analyze the performance of algorithms
* Ability to choose appropriate data structures and algorithm design methods for a specified application
* Ability to understand how the choice of data structures and the algorithm design methods impact the performance of programs

# UNIT - I

**Introduction:** Algorithm, Performance Analysis-Space complexity, Time complexity, Asymptotic Notations- Big oh notation, Omega notation, Theta notation and Little oh notation.

**Divide and conquer**: General method, applications-Binary search, Quick sort, Merge sort, Strassen’s matrix multiplication.

# UNIT - II

**Disjoint Sets**: Disjoint set operations, union and find algorithms

**Backtracking**: General method, applications, n-queen’s problem, sum of subsets problem, graph coloring

# UNIT - III

**Dynamic Programming**: General method, applications- Optimal binary search trees, 0/1 knapsack problem, All pairs shortest path problem, Traveling sales person problem, Reliability design.

# UNIT - IV

**Greedy method:** General method, applications-Job sequencing with deadlines, knapsack problem, Minimum cost spanning trees, Single source shortest path problem.

# UNIT - V

**Branch and Bound**: General method, applications - Travelling sales person problem, 0/1 knapsack problem - LC Branch and Bound solution, FIFO Branch and Bound solution.

**NP-Hard and NP-Complete problems**: Basic concepts, non deterministic algorithms, NP - Hard and NP-Complete classes, Cook’s theorem.

# TEXT BOOK:

1. Fundamentals of Computer Algorithms, Ellis Horowitz, Satraj Sahni and Rajasekharan, University Press.

# REFERENCE BOOKS:

1. Design and Analysis of algorithms, Aho, Ullman and Hopcroft, Pearson education.
2. Introduction to Algorithms, second edition, T. H. Cormen, C.E. Leiserson, R. L. Rivest, and C. Stein, PHI Pvt. Ltd./ Pearson Education.
3. Algorithm Design: Foundations, Analysis and Internet Examples, M.T. Goodrich and R. Tamassia, John Wiley and sons.

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT-V – BACKTRACKING

Backtracking: General method, Applications- N-QUEEN Problem, Sum of Sub Sets problem, Graph Coloring, Hamiltonian Cycles.

-0-0-0-0-

# Introduction

Backtracking is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (v1, ..., vm) of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found.

When invoked, the algorithm starts with an empty vector. At each stage it extends the partial vector with a new value. Upon reaching a partial vector (v1, ..., vi) which can’t represent a partial solution, the algorithm backtracks by removing the trailing value from the vector, and then proceeds by trying to extend the vector with alternative values.

ALGORITHM try(v1,...,vi)

{

IF (v1,...,vi) is a solution THEN RETURN (v1,...,vi) FOR each v DO

IF (v1,...,vi,v) is acceptable vector THEN sol = try(v1,...,vi,v)

IF sol != () THEN RETURN sol END

END RETURN ()

}

If Si is the **domain** of vi, then S1 × ... × Sm is the **solution space** of the problem. The **validity criteria** used in checking for acceptable vectors determines what portion of that space needs to be searched, and so it also determines the resources required by the algorithm.

The traversal of the solution space can be represented by a depth-first traversal of a tree. The tree itself is rarely entirely stored by the algorithm in discourse; instead just a path toward a root is stored, to enable the backtracking.

In case of greedy and dynamic programming techniques, we will use Brute force approach. It means, we will evaluate all possible solutions, among which, we select one solution as optimal solution. In backtracking technique, we will get same optimal solution with less number of steps. So we use backtracking technique. We can solve problems in an efficient way when compared to other methods like greedy method and dynamic programming. In this we will use bounding functions (criterion functions), implicit and explicit conditions. While explaining the general method of backtracking technique, there we will see implicit and explicit constraints. The major advantage of backtracking method is, if a partial solution (x1,x2,x3…..,xi) can’t lead to optimal solution then (xi+1…xn) solution may be ignored entirely.

**Explicit constraints:** These are rules which restrict each xi to take on values only from a given set.

**Example**

1. Knapsack problem, the explicit constraints are,
   1. xi=0 or 1 ii)0<=xi<=1
2. 4-queens problem : in 4 queens problem, the 4 queens can be placed in 4x4 chess board in 44 ways.

**Implicit constraints:** These are rules which determine which of the tuples in the solution space satisfy criterion function.

**Example:** In 4 queens problem, the implicit constraints are no 2 queens can be on the same row, same column and same diagonal.

**Let us see some terminology which is being used in this method.**

1. **Criterion Function:** it is a function p(x1,x2,x3,…xn) which needs to be maximized or minimized for a given problem.
2. **Solution Space** : All tuples that satisfy the explicit constraints define a possible solution space for a particular instance ‘i’ of the problem.

For example consider the following tree. ABD, ABE, AC are the tuples in solution space.

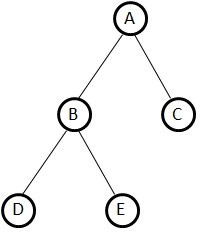


Fig) The organization of a solution space

1. **Problem state:** each node in the tree organization defines a problem state. So, A,B ,C are problem states.

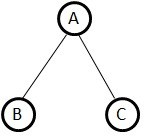


Fig) Tree (Problem State)

1. **Solution states:** These are those problem states S for which the path from the root to S define a tuple in the solution space.

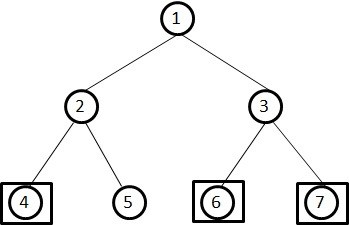
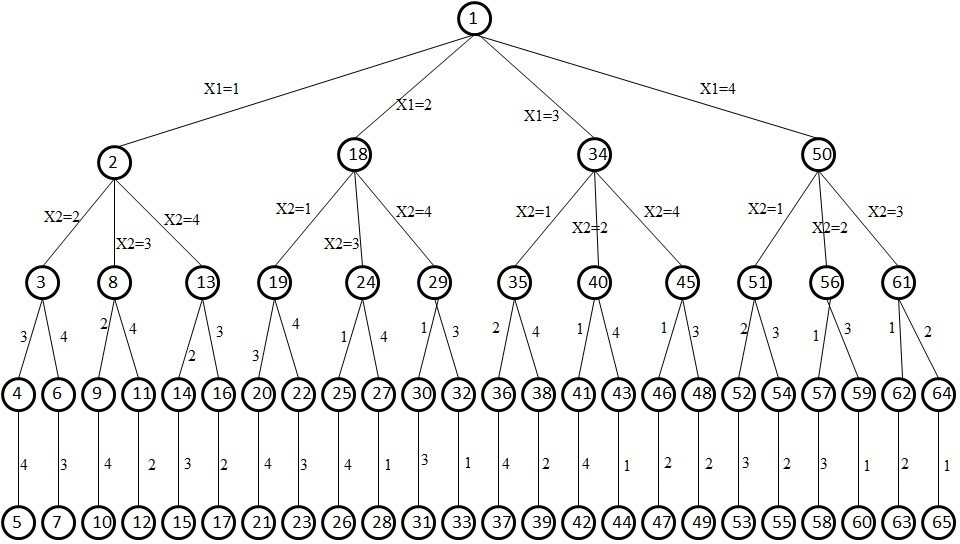


Fig) Tree (Solution state)

Here square nodes indicate solution. For the above solution space, there exists 3 solution states. These solution states represented in the form of tuples i.e. (1,2,4), (1,3,6) and (1,3,7) are the solution states.

1. **state space tree:** if we represent solution space in the form of a tree then the tree is referred as the state space tree.

For example given is the state space tree of 4-queen problem. Initially x1=1 or 2 or 3 or 4. It means we can place first queen in either of 1/2/3/4 column. If x1=1 then x2 can be paced in either 2nd,3rd , or 4th column. If x2=2 then x3 can be placed either in 3rd or 4th column. If x3=3 then x4=4. So nodes 1-2-3-4-5 is one solution in solution space. It may or may not be feasible solution. Similarly we can observe the remaining solutions in the figure.



**Fig) Tree organization of the 4 queen solution space**

1. **Answer states :** These solution states s for which the path from the root to S defines a tuple which is a member of the set of solutions.(i.e. it satisfies the implicit constraints) of the problem. Here 3,4, are answer states. (1,3) and (1,2,4) are solution states.

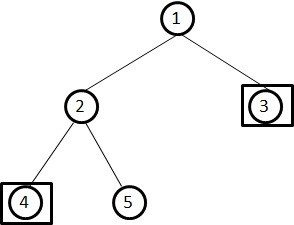
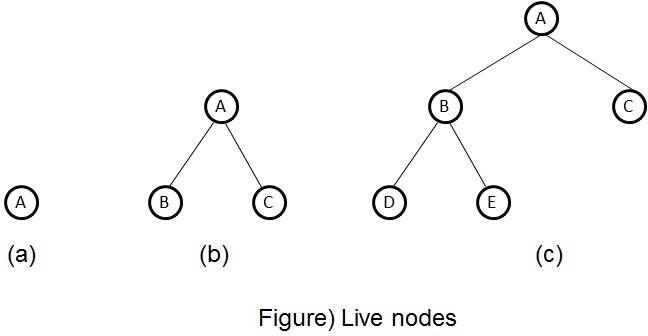


Fig) Tree (answer states)

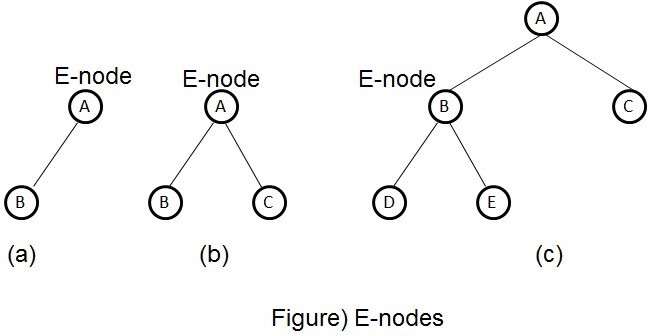
1. **Live node:** A node which has been generated and all of whose children have not yet been generated is live node. In the fig (a) node A is called live node since the children of node A have not yet been generated.



In fig (b) node A is not a live node but B,C are live nodes.

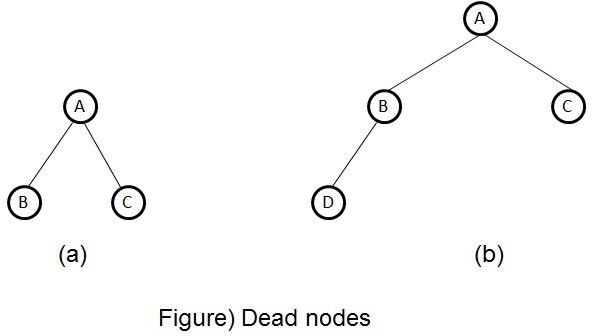
In fig(c) nodes A,B are not live and D,E C are live nodes.

1. **E-node :** The live node whose children are currently being generated is called E-node.( node being expanded).



1. **Dead node:** it is a generated node that is either not to be expanded further or one for which all of its children have been generated.

Ex)In figure(a) nodes A,B,C are dead nodes since node A’s children already generated and Nodes B,C are not expanded.



In figure (b) assumed that node B can generate one more node so nodes A,D,C are dead nodes.

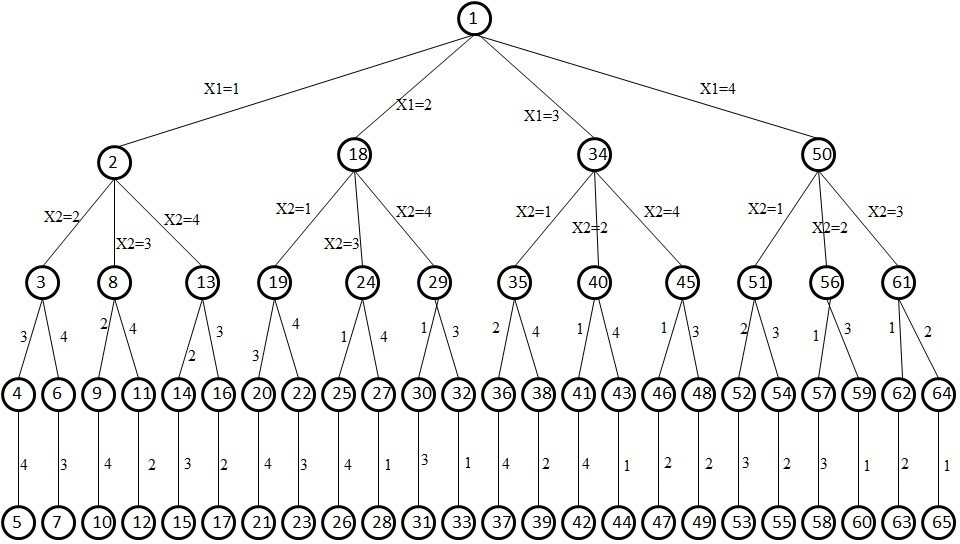
# Applications:

1. **n-Queens Problem ( 4-Queens and 8-Queens Problem)**

Consider an nxn chess board. Let there are n Queens. These n Queens are to be placed on the nxn chess board so that no two queens are on the same column, same row or same diagonal.

n-queens Problem: The n-queens problem is a generalization of the 8-queens problem. Now n-queens are to be placed on an nxn cross board so that no two attack; that is no two queens are on the same row, column, or diagonal. The solution space consists of all n! permutations of n-tuple (1,2,3,..n).

The following figure shows a possible tree organization for the case n = 4. A tree such as this is called a permutation tree. The edges are labeled by possible values of xi.



**Figure)The organization of 4-queens solution space**

Edges from level 1 to level 2 nodes specify the values for x1. Thus the left most sub-tree contains all solutions with x1=1.

Edges from level i to level i+1 are labeled with the values of xi. The solution space is defined by all paths from the root node to a leaf node. There are 4!=24 leaf nodes in the permutation tree.

If we imagine the chess board squares being numbered as the indices of the two dimensional array a[1..n,1..n] then we observe that every element on the same diagonal that runs from upper left to lower right has the same row-col value.

1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 |  |  | Q |  |  |  |  |  |  | |
| 2 |  |  |  |  | Q |  |  |  |  |  |
| 3 |  |  |  |  |  |  | Q |  |  |  |
| 4 | Q |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  | Q |  |  |  |  |
| 6 Q |  |  |  |  |  |  |  |  |  |  |
| 7 |  | Q |  |  |  |  |  |  |  |  |
| 8 |  |  |  | Q |  |  |  |  |  |  |
| Consider the | queen | at | | a[4,2]. | | | The | | squares | that | are |

diagonal to this queen (running from upper left to lower right) are a[3,1],a[5,3], a[6,4], a[7,5],a[8,6]. All these squares have a (row – column) value of 2. Also every element on the same diagonal that goes from the upper right to the lower left has the same (row + column) value.

Suppose two queens are placed at positions (i,j) and (k,l)then by the above we can say they are on the same diagonal if

i-j=k-l which is primary diagonal or

i+j=k+l which is secondary diagonal

Equation for primary diagonal

i-j = k-l

this can be written as follows

j-l = i-k

Equation for Secondary diagonal

i+j = k+l

this can be written as follows

j-l = k-i

Therefore two queens lie on the same diagonal if and only if |j-l| = |i-k|.

The algorithm place(k,i) returns a Boolean value that is true if kth queen can be placed in column ‘i’. it tests both whether ‘i’ is distinct from all previous values x[i]..x[k-1] and whether there is no other queen on the same diagonal.

Its computing time is O(k-1).

The array x[1..n] is a global array. Let (x1,x2,x3,…xn) be the solution vector where xi is the column number on which the ith queen is placed. (i may be row number).

Using the algorithm place() a queen is placed in kth row , ith column and return true otherwise false.

**Algorithm place(k,i)**

{

for j:=1 to k-1 do

if ((x[j]=i) or (abs(x[j]-i) = abs(j-k)) then return false;

return true;

}

This algorithm is invoked by nqueens(1,n).

The algorithm for obtaining solution to n-queens problem is given below.

**Algorithm nqueens(k,n)**

{

for i:=1 to n do

{

if (place(k,i) then

{

X[k]:=i;

if (k=n) then write(x[i:n);

else

nqueens(k+1,n);

}

}

}

For an 8x8 chess board there are 64C8 possible ways to place 8 Queens using brute force approach. However by allowing only placements of queens on distinct rows and columns, we require the examination of at most 8! Tuples.

For a 4x4 chess board there are 16C4 possible ways to place

4 Queens using brute force approach. However by allowing only placements of queens on distinct rows and columns, we require the examination of at most 4! Tuples.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Place first queen in the | first | row |  | 1 | 2 3 | | 4 |
| in the first column. |  |  | 1 | 1 |  |  |  |
| As it is the first queen | it is |  | 2 |  |  |  |  |
| not under attack. |  |  | 3 |  |  |  |  |
| X[1]=1 (column value is assigned) | | | 4 |  |  |  |  |

X[1]=1

To place second queen in second row

Start with first column. 1

It is under attack 2

Second column also Under attack 3

Third column not under attack by other queens. 4

So we place queen in 3rd column. X[2]=3

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
|  |  | 2 |  |
| - | - | - | - |
|  |  |  |  |

1 2 3 4

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
| - | - | 2 |  |
|  |  |  |  |
|  |  |  |  |

X[2]=3

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| To place third queen in third row  first col under attack |  |  | 1 |  | 2 3 4 |  |
| second column under attack | 1 |  |  |  |  |  |
| third column under attack | 2 |  |  |  |  |  |
| fourth column under attack | 3 |  |  |  |  |  |
| not possible to place queen in third row | 4 |  |  |  |  |  |
| because placement of previous queens is not correct.So backtrack to previous row  and move the queen to another possible place and continue. |  |  |  |  |  |  |
|  |  |  | 1 |  | 2 3 4 |  |
| Go to second row | 1 |  |  |  |  |  |
| Move the queen to another col. | 2 |  |  |  |  |  |
| Another possibility is column 4. | 3 |  |  |  |  |  |
| Move to col 4. | 4 |  |  |  |  |  |
| Now X[2]=4 |  |  |  |  | X[2]=4 |  |

Go to third row to place 3rd queen

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
| - | - | - | 2 |
|  |  |  |  |
|  |  |  |  |

First col under attack 1

Second column not under attack by other 2

queens 3

So place the queen in 2nd col. 4

X[3]=2

Now to place 4th queen in 4th row First col under attack by other queen

Second col under attack by other queen Third col under attack by other queen

Fourth col under attack by other queen 1

Not possible to place the queen in 4th row 2

as there is a problem in the placement of 3

previous queens 4

Back track to previous placements

Goto 3rd row and try to move the queen to another place.

The other places are under attack go to 2nd row Already we checked all possibilities in 2nd row we backtrack to first row.

1 2 3 4

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
|  |  |  | 2 |
| - | 3 |  |  |
|  |  |  |  |

X[3]=2

1 2 3 4

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
|  |  |  | 2 |
|  | 3 |  |  |
| - | - | - | - |

First queen is moved to 2nd column X[1]=2

Second queen in second row First col under attack Second column under attack Third col under attack

4th col not under attck So place queen in 4th col X[2]=4

To place third queen in third row First col not under attack

So place the queen in first col X[3]=1

To place 4th queen in 4th row first col under attack second col under attack third col not under attack

so place the 4th queen in 3rd col x[4]=3

1 2 3 4

1

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

2

3

4

X[1]=2

1 2 3 4

1

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 |  |  |
| - | - | - | 2 |
|  |  |  |  |
|  |  |  |  |

2

3

4

X[2]=4

1 2 3 4

1

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 |  |  |
|  |  |  | 2 |
| 3 |  |  |  |
|  |  |  |  |

2

3

4

X[3]=1

1 2 3 4

1

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 |  |  |
|  |  |  | 2 |
| 3 |  |  |  |
| - | - | 4 |  |

2

3

4

X[4]=3

All four queens are placed in the

4x4 chess board without attacking each other.

In the same way it is possible to place all 8 queens in an 8x8 chess board without attacking each other.

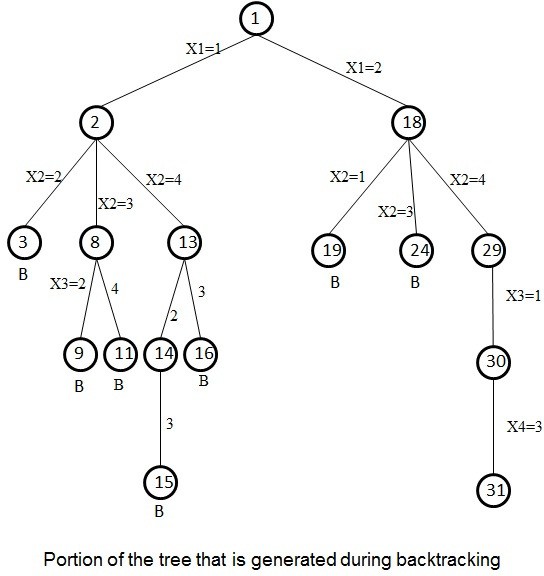


Figure shows the part of the solution space tree that is generated. The tree generated as per the above processing. Nodes are numbered in the order in which they are generated. A node that gets killed as a result of backtracking has a B under it.

Tracing of the algorithm to place 4 queens on a 4x4 cross board such that no two queens attack each other.

nqueens(k,n) place(k,i)

nqueens(1,4) place(1,1) returns True so x[1]=1 nqueens(2,4) place(2,1) returns False

place(2,2) returns False place(2,3) returns True so X[2]=3

nqueens(3,4) place(3,1) returns False

place(3,1) returns False place(3,1) returns False place(3,1) returns False

Backtracking nqueens(2,4) place(2,4) returns True so X[2]=4

nqueens(3,4) place(3,1) returns False

place(3,2) returns True so X[3]=2 nqueens(4,4) place(4,1) returns False

place(4,2) returns False place(4,3) returns False place(4,4) returns False

Backtracking nqueens(1,4) place(1,2) returns True so x[1]=2

nqueens(2,4) place(2,1) returns False

place(2,2) returns False place(2,3) returns False place(2,4) returns True so X[2]=4

nqueens(3,4) place(3,1) returns True so X[3]=1 nqueens(4,4) place(4,1) returns False

place(2,2) returns False

place(2,3) returns True so X[4]=3

The solution vector for a 4x4 cross board to place 4 non attacking queens is

x[1]=2

x[2]=4

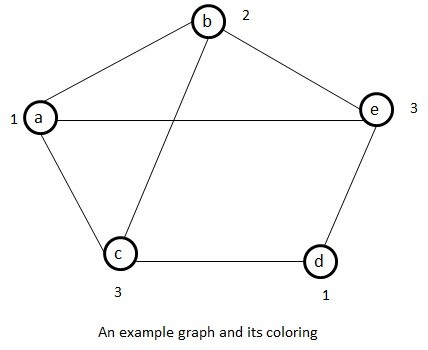
x[3]=1

x[4]=3

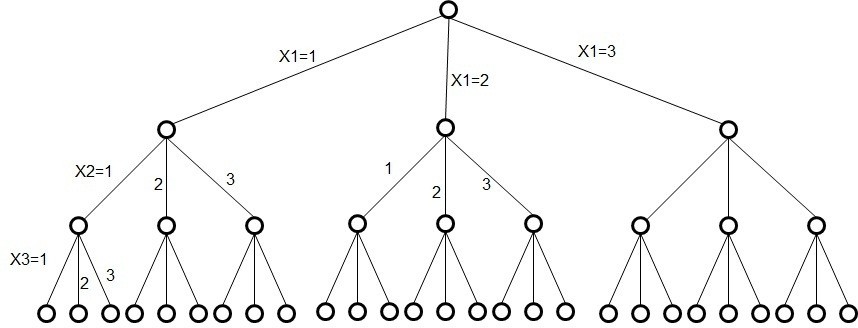
1. GRAPH COLORING

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

This is termed the m-colorability decision problem. Note that if d is the degree of the given graph, then it can be colored with d+1 colors. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored. The integer is referred to as the chromatic number of the graph.

For example the following graph can be colored with three colors 1, 2 and 3. The color of each node is indicated next to it. It can also be seen that thee colors are needed to color this graph and hence this graph’s chromatic number is 3.

**State space tree for coloring a graph containing 3 nodes using 3 colors**



**Fig)State space tree for mColoring when n=3 and m=3**

The algorithm mcoloring 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.

**Algorithm mcoloring(k)**

**{**

**repeat**

**{**

**nextvalue(k);**

**if (x[k] = 0) then return; if (k=n) then**

**write(x[1:n]); else**

**mcoloring(k+1);**

**}until(false);**

**}**

No of vertices= n No of colors= m

Solution vector = X[1], X[2], X[3] X[n]

The values of solution vector may belongs to {0,1,2,3..m} The following Algorithm is used to generate next color.

Assume that X[1],..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, the x[k]=0.

**Algorithm nextvalue(k)**

**{**

**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**

**{**

**if ((G[k,j]!=0) and (x[k]=x[j])) then break;**

**//g[k,j] an edge and**

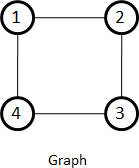
**//vertices k and j have same color**

**}**

**if (j=n+1) then return;**

**}until (false);**

**}**

Adjacency Matrix G 1 2 3 4

1

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |

2

3

4

Assume that n=4 and m=3 X[1]=0,x[2]=0,x[3]=0,x[4]=0

If we call the algorithm mcoloring(k)

mcoloring(1) i.e. k=1 nextvalue(1)

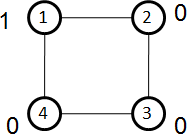
k=1

x[1]=(x[1]+1) mod (m+1)

x[1]= 0+1 mod 4 x[1]=1

G[k,j]!=0 and x[k]=x[j]

j=1 G[1,1] false and true = false j=2 G[1,2] true and false = false j=3 G[1,3] false and false = false j=4 G[1,4] true and false = false

x[1]=1 ,x[2]=0,x[3]=0,x[4]=0

mcoloring(2) i.e. k=2 nextvalue(2)

k=2

x[2]=(x[2]+1) mod (m+1)

x[2]= 0+1 mod 4 x[2]=1

G[k,j]!=0 and x[k]=x[j]

j=1 G[2,1] True and True = True break G[2,1] is an edge and

adjacent vertices have same color

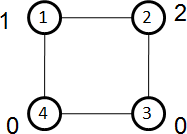
x[2]=(x[2]+1) mod (m+1)

x[2]=(1+1) mod 4=2 mod 4 x[2]=2

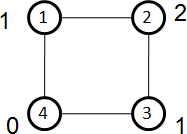
G[k,j]!=0 and x[k]=x[j]

j=1 G[2,1] True and False = False j=2 G[2,2] False and True = False j=3 G[2,3] True and False = False j=4 G[2,4] False and False = False

x[1]=1 ,x[2]=2,x[3]=0,x[4]=0

assume that the number mentioned outside the node belongs to color

|  |  |  |  |
| --- | --- | --- | --- |
| mcoloring(3) nextvalue(3) k=3  x[3]=(x[3]+1) | i.e. k=3  mod (m+1) |  | |
| x[3]= 0+1 mod x[3]=1 | 4 |
| j=1 G[3,1] | G[k,j]!=0 and False and | x[k]=x[j] True = | False |
| j=2 G[3,2] | True and | False = | False |
| j=3 G[3,3] | False and | True = | False |
| j=4 G[3,4] | True and | False = | False |

x[1]=1 ,x[2]=2,x[3]=1,x[4]=0.

mcoloring(4) i.e. k=4 nextvalue(4)

k=4

x[4]=(x[4]+1) mod (m+1)

x[4]= 0+1 mod 4 x[4]=1

G[k,j]!=0 and x[k]=x[j]

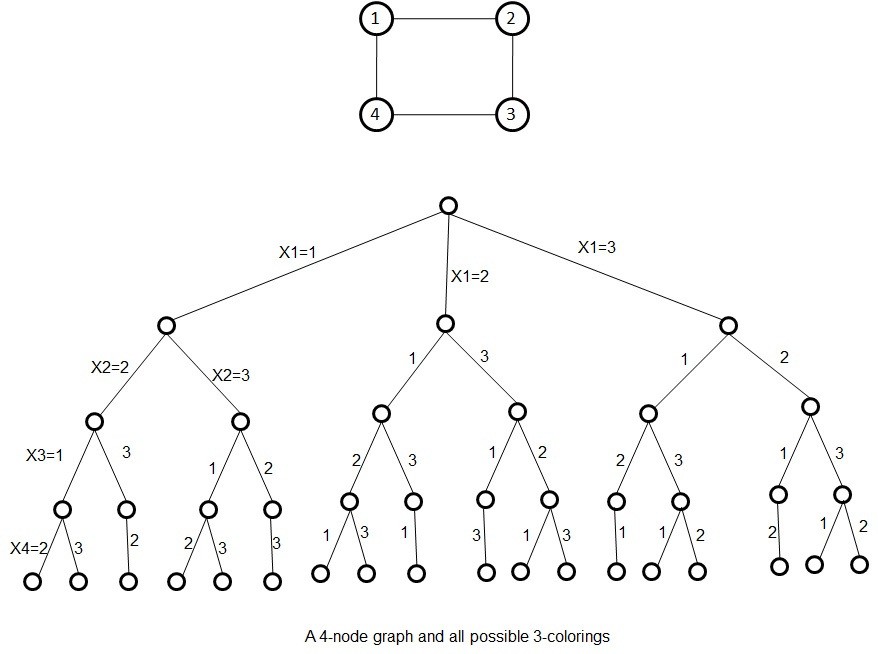
j=1 G[4,1] true and true = True so break

adjacent vertices have same color

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x[4]=(x[4]+1) | mod (m+1) |  |  |  | |
| x[4]= 1+1 mod | 4 |  |  |
| x[4]=2 |  |  |  |
|  | G[k,j]!=0 | and | x[k]=x[j] |
| j=1 G[4,1] | True | and | False | = | False |
| j=2 G[4,2] | False | and | True | = | False |
| j=3 G[4,3] | True | and | False | = | False |
| j=4 G[4,4] | False | and | True | = | False |

x[1]=1 ,x[2]=2,x[3]=1,x[4]=2



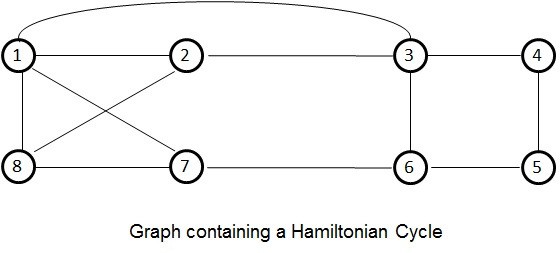


# HAMILTONIAN CYCLES

Let G=(V,E) be a connected graph with n vertices. A Hamiltonian cycle 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 v1  G and the vertices of G are visited in

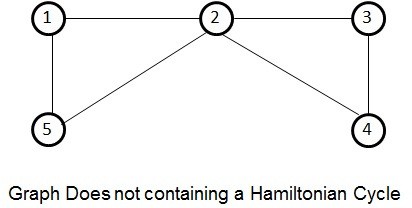
the order v1, v2,…vn+1 then the edges (vi,vi+1) are in E, 1<=i<=n, and the vi are distinct except for v1 and vn+1, which are equal.



The above graph contains the Hamiltonian cycles 1,2,3,4,5,6,7,8,1

1,3,4,5,6,7,8,2,1

1,2,8,7,6,5,4,3,1



The graph contains no Hamiltonian cycle.

To check whether there is a Hamiltonian cycle or not we may use backtracking method. The graph may be directed or undirected. Only distinct cycles are output.

The backtracking solution vector (X1,X2,X3,…Xn) is defined so that xi represents the ith visited vertex of the proposed cycle.

Now all we need to do is determine how to compute the set of possible vertices for xk if x1,..xk-1 have already been chosen. If k=1 then x1 can be any of the n vertices.

The algorithm nextvalue(k) which determines a possible next vertex for the proposed cycle.

Using nextvalue we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix G[1:n,1:n], then setting x[2:n] to 0 and x[1] to 1 and then executing Hamiltonian(2).

// x[1:k-1] is a path of k-1 distinct vertices

// if x[k]=0 then no vertex has 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]. Otherwise x[k]=0.

// if k=n then in addition x[k] is connected to x[1].

**Algorithm nextvalue(k)**

**{**

**Repeat**

**{**

**X[k]:=(x[k]+1)mod(n+1); if (x[k]=0) then**

**return;**

**if (G[x[k-1],x[k]]]!=0) then**

**{**

**For j:=1 to k-1 do**

**if (x[j]=x[k]) then break;**

**if (j=k) then**

**if((k<n) or ((k=n)and G[x[n],x[1]]!=0))then return;**

**}**

**}until (false);**

**}**

**Algorithm to generate next vertex.**

The algorithm Hamiltonian() 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.

**Algorithm Hamiltonian(k)**

**{**

**Repeat**

**{**

**nextvalue(k);**

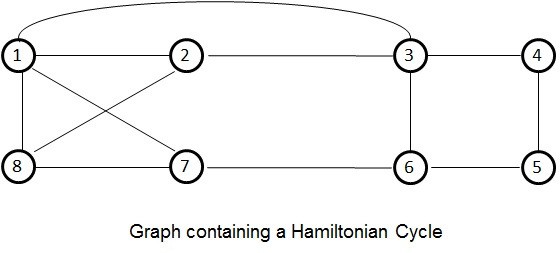
**if (x[k]=0) then return;**

**if (k=n) then write (x[1:n]); else**

**Hamiltonian(k+1);**

**}until(false);**

**}**

**Algorithm to find all Hamiltonian cycles. Example)**

**No of vertices n=8**

**Adjacency matrix G Solution vertex 1 2 3 4 5 6 7 8**

**1 X[1]**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **1** | **0** | **0** | **0** | **1** | **1** |
| **1** | **0** | **1** | **0** | **0** | **0** | **0** | **1** |
| **1** | **1** | **0** | **1** | **0** | **1** | **0** | **0** |
| **0** | **0** | **1** | **0** | **1** | **0** | **0** | **0** |
| **0** | **0** | **0** | **1** | **0** | **1** | **0** | **0** |
| **0** | **0** | **1** | **0** | **1** | **0** | **1** | **0** |
| **1** | **0** | **0** | **0** | **0** | **1** | **0** | **1** |
| **1** | **1** | **0** | **0** | **0** | **0** | **1** | **0** |

|  |
| --- |
| **1** |
| **0** |
| **0** |
| **0** |
| **0** |
| **0** |
| **0** |
| **0** |

**2 X[2]**

**3 X[3]**

**4 X[4]**

**5 X[5]**

**6 X[6]**

**7 X[7]**

**8 X[8]**

**Algorithm starts with vertex 1 as initial vertex. Solution vertex must contain a series of vertices in the cycle.**

**X[1:n] i.e. x[1:8]**

**X[1]=1**

**and x[2:8]=0**

**we will add one by one vertices to the solution vector.**

**Hamiltonian(2) k=2 Nextvalue(2) k=2**

**X[2]=(x[2]+1) mod (8+1)=(0+1)mod 9 = 1**

**Is there an edge between k and k-1 G[x[k-1],x[k]]!=0**

**G[1,1] no edge False X[2]=(x[2]+1) mod (8+1)**

**= (1+1) mod 9**

**= 2**

**If (G[x[k-1],x[k]]!=0)**

**G[1,2] edge True**

**Are there duplicate vertices in the path J=1 is x[j]=x[k]**

**is x[1]=x[2] no false**

**Solution vector**

**X[1]**

|  |
| --- |
| **1** |
| **2** |
| **0** |
| **0** |
| **0** |
| **0** |
| **0** |
| **0** |

**X[2]**

**X[3]**

**X[4]**

**X[5]**

**X[6]**

**X[7]**

**X[8]**

**k < n return still we need to add vertices**

**Hamiltonian(3) k=3 Nextvalue(3) k=3**

**X[3]=(x[3]+1) mod (8+1)=(0+1)mod 9 = 1 X[3]=1**

**Is there an edge between k and k-1**

**If (G[x[k-1],x[k]]!=0)**

**G[2,1] edge True**

**Are there duplicate vertices in the path J=1 is x[j]=x[k]**

**is x[1]=x[3]**

**1=1 True break**

**X[3]=(x[3]+1) mod (8+1)**

**X[3]=(1+1)mod 9**

**= 2**

**If (G[x[k-1],x[k]]!=0)**

**G[2,2] edge False**

**X[3]=(x[3]+1) mod (8+1)**

**X[3]=(2+1)mod 9**

**= 3**

**If (G[x[k-1],x[k]]!=0)**

**G[2,3] edge True**

**J=1 is x[j]=x[k]**

**Is 1=2 no false J=2 is 2=3 no false**

**as k < n return still we need to add vertices**

**Hamiltonian(4) k=4 Nextvalue(4) k=4**

**X[4]=(x[4]+1) mod (8+1)**

**= (0+1)mod 9 = 1 X[4]=1**

**Is there an edge between k and k-1**

**If (G[x[k-1],x[k]]!=0)**

**G[3,1] edge True**

**Are there duplicate vertices in the path J=1 is x[j]=x[k]**

**is x[1]=x[4]**

**1=1 True break**

**Solution vector**

**X[1]**

|  |
| --- |
| **1** |
| **2** |
| **3** |
| **0** |
| **0** |
| **0** |
| **0** |
| **0** |

**X[2]**

**X[3]**

**X[4]**

**X[5]**

**X[6]**

**X[7]**

**X[8]**

**X[4]=(x[4]+1) mod (8+1)**

**X[4]= 2**

**If (G[x[k-1],x[k]]!=0)**

**G[3,2] edge True**

**Are there duplicate vertices in the path J=1 is x[j]=x[k]**

**is x[1]=x[4]**

**1=2 False**

**J=2 is x[j]=x[k] is x[2]=x[4]**

**2=2 True Break**

**X[4]=(x[4]+1) mod (8+1) X[4]=3**

**If (G[x[k-1],x[k]]!=0)**

**G[3,3] edge False**

**X[4]=(x[4]+1) mod (8+1) X[4]=4**

**If (G[x[k-1],x[k]]!=0)**

**G[3,4] edge True**

**J=1 is x[j]=x[k]**

**Is 1=4 no false J=2 is 2=4 no false J=3 is 3=4 no false**

**Solution vector**

**X[1]**

|  |
| --- |
| **1** |
| **2** |
| **3** |
| **4** |
| **0** |
| **0** |
| **0** |
| **0** |

**X[2]**

**X[3]**

**X[4]**

**X[5]**

**X[6]**

**X[7]**

**X[8]**

**as k < n return still we need to add vertices**

**Hamiltonian(5) k=5 Nextvalue(5) k=5**

**X[5]=(x[5]+1) mod (8+1)**

**= (0+1)mod 9 = 1 X[5]=1**

**Is there an edge between k and k-1**

**If (G[x[k-1],x[k]]!=0)**

**G[4,1] no edge False**

**X[5]=(x[5]+1) mod (8+1)**

**= (1+1)mod 9 = 2**

**If (G[x[k-1],x[k]]!=0)**

**G[4,2] no edge False X[5]=(x[5]+1) mod (8+1)**

**= (3+1)mod 9 = 3 If (G[x[k-1],x[k]]!=0)**

**G[4,3] edge True**

**Are there duplicate vertices in the path J=1 is x[j]=x[k]**

**is x[1]=x[5]**

**1=3 False J=2 is x[j]=x[k]**

**is x[2]=x[5]**

**2=3 False J=3 is x[j]=x[k]**

**is x[3]=x[5]**

**3=3 True duplicate found break**

**X[5]=(x[5]+1) mod (8+1)**

**X[5]= 5**

**If (G[x[k-1],x[k]]!=0)**

**G[4,5] edge True**

**Are there duplicate vertices in the path J=1 is x[j]=x[k]**

**is x[1]=x[5]**

**1=5 False**

**J=2 is x[j]=x[k] is x[2]=x[5]**

**2=5 False**

**J=3 is x[j]=x[k] is x[3]=x[5]**

**3=5 False**

**J=4 is x[j]=x[k] is x[4]=x[5]**

**4=5 False**

**Solution vector**

**X[1]**

|  |
| --- |
| **1** |
| **2** |
| **3** |
| **4** |
| **5** |
| **0** |
| **0** |
| **0** |

**X[2]**

**X[3]**

**X[4]**

**X[5]**

**X[6]**

**X[7]**

**X[8]**

**as k < n return still we need to add vertices**

**Hamiltonian(6) k=6 Nextvalue(6) k=6**

**The solution vector for hamiltonian cycles 1,2,3,4,5,6,7,8,1**

**1,8,2,3,4,5,6,7,1**

**1,3,4,5,6,7,8,2,1**

**SUM OF SUBSETS**

Suppose we are given n distinct positive numbers( usually called weights) and we desire to find all combinations of these numbers whose sum are m.

This is called the sum of subsets problem.

Ex1) given positive numbers Wi, 1<=i<=n, and m, this problem calls for finding all subsets of wi whose sums are m. For example, if n=4, (w1,w2,w3,w4)=(7,11,13,24) and m=31, then the desired subsets are (7,11,13) and (7,24).

Rather than representing the solution vector by wi which sum to m, we could represent the solution vector by giving the indices of these wi.

Now the two solutions are described by the vectors (1,2,3) and (1,4).

In general all solution subset is represented by n-tuple (X1,X2,X3,…Xn) such that Xi {0,1},1<=i<=n. The Xi is 0 if wi is not chosen and xi=1 if wi is chosen. The solutions

to the above instances are (1,1,1,0) and (1,0,0,1). This formulation expresses all solutions using a fixed sized tuple.

The sum of sub set is based on fixed size tuple. Let us draw a tree structure for fixed tuple size formulation.

All paths from root to a leaf node define a solution space. The left subtree of the root defines all subsets containing W1 and the right subtree defines all subsets not containing W1 and so on.

Step 1)Start with an empty set

Step 2)Add next element in the list to the sub set

Step 3)If the subset is having sum = m then stop with that sub set as solution.

Step 4)If the sub set is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.

Step 5)if the subset is feasible then repeat step 2

Step 6)if we have visited all elements without finding a suitable subset and if no backtracking is possible, then stop with no solution.

s – sum of all selected elements

k – denotes the index of chosen element

r – initially sum of all elements. After selection of some element from the set subtract the chosen value from r each time.

W(1:n) – represents set containing n elements. X[i]-solution vector 1<=i<=k

**Algorithm sumofsubsets(s,k,r)**

**{**

**X[k]:=1;**

**if (s+w[k]=m) then write (x[1:k]); // subset found else**

**if (s+w[k]+w[k+1]<=m) then sumofsubsets(s+w[k],k+1,r-w[k]);**

**//generate right child and evaluate Bk. if ((s+r-w[k]>=m) and (s+w[k+1]<=m)) then**

**{**

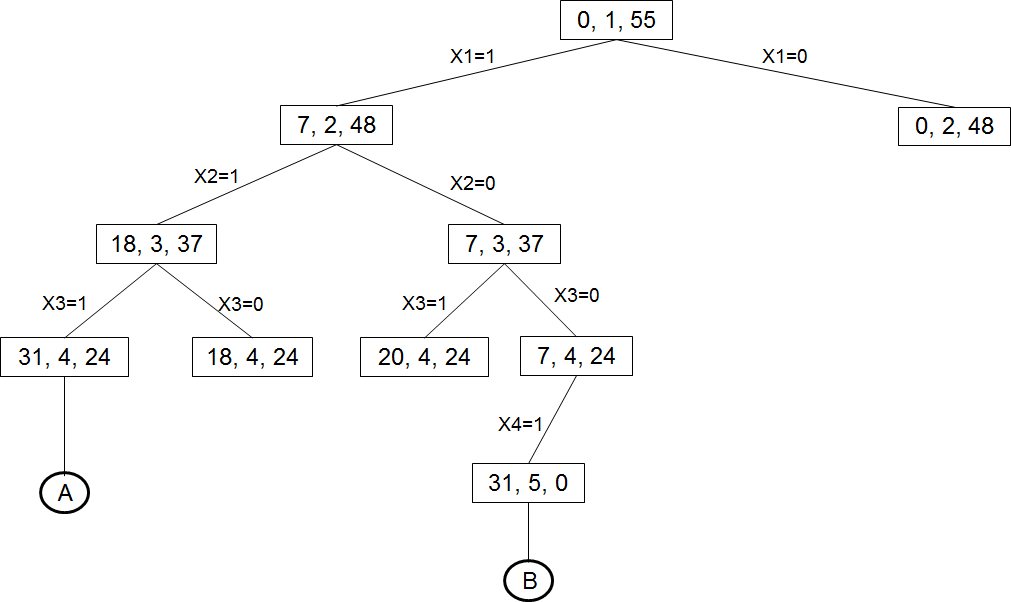
**X[k]:=0;**

**sumofsubsets(s,k+1,r-w[k]);**

**}**

**}**

Ex) n=4, (w1,w2,w3,w4)=(7,11,13,24) and m=31

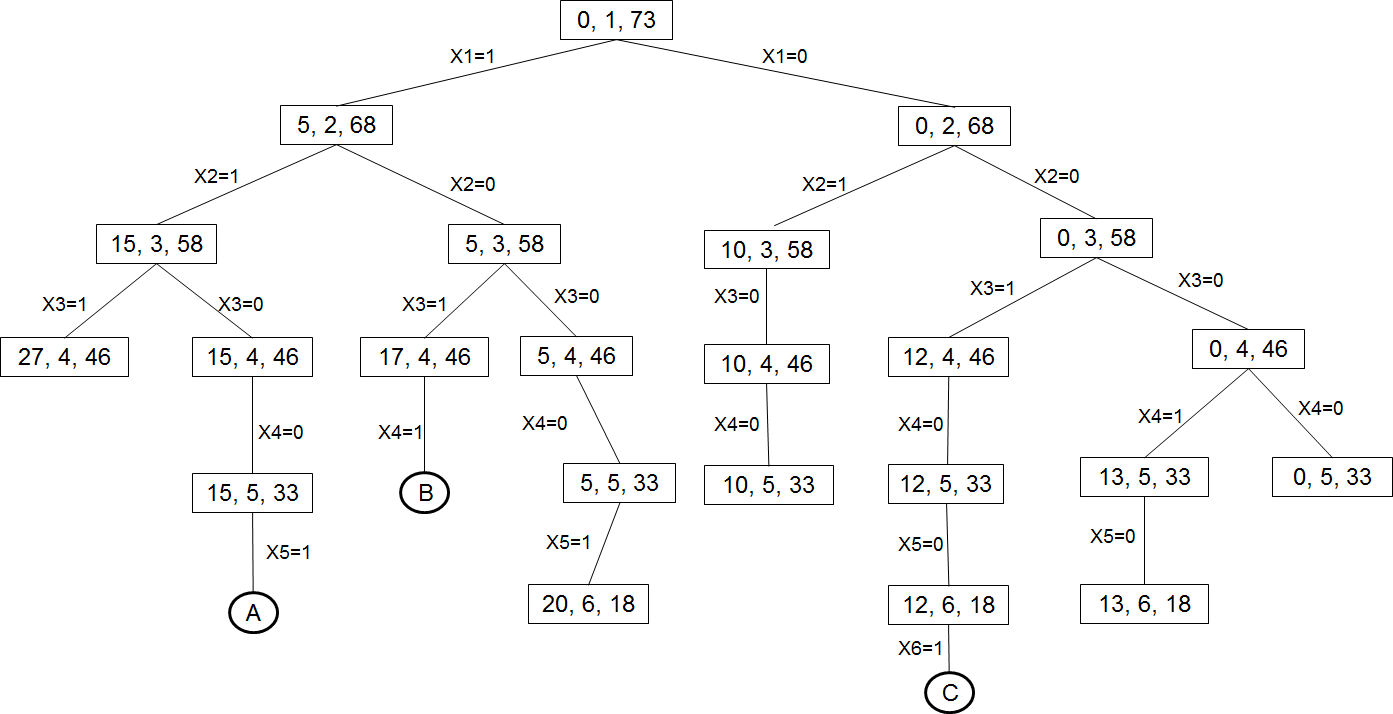
Solution Vector=(x[1],x[2],x[3],x[4])

Portion of state space Tree Solution A = {1,1,1,0}

Solution B = {1,0,0,1}

Ex2) n=6, m=30 and w[1:6]={5,10,12,13,15,18}.

Portion of the state space tree generated by sum of subsets



State space tree with solution

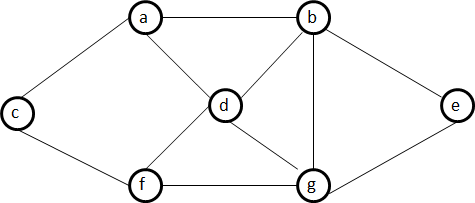
The rectangular nodes list the values of s,k and r. Circular nodes represent points at which subsets with sums m are printed out.

Solution A = (1,1,0,0,1) Solution B = (1,0,1,1) Solution C = (0,0,1,0,0,1)

Note that the tree contains only 23 rectangular nodes. The full space tree for n=6 contains 26-1=63 nodes from which calls could be made.

**Important questions**

1. Describe problem state, solution state and answer state with examples.
2. Write the control abstraction of backtracking 3)Explain the applications of backtracking 4)Describe 4-queen problem using backtracking
3. Write an algorithm of finding all m-colorings of a graph
4. Draw the state space tree for m-coloring graph using suitable graph
5. Apply backtracking to find Hamiltonian cycle in the following graph as shown in the figure.



1. Write backtracking algorithm for 8-queens problem

**CMR ENGINEERING COLLEGE**

Accredited by NBA,Affililated to JNTU,Hyderabad

Kandlakoya(v), Medchal Road,Hyderabad-501401

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**Academic Year – 2018-2019**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Roll No.** | **Name of the Student** | **Grade** |
|  | 178R1A05C1 | ANUMULA SUSHMITHA | Excellent |
|  | 178R1A05C2 | AELLA ABHISHEK REDDY | Good |
|  | 178R1A05C3 | ALISHETTY PRAVALIKA | Excellent |
|  | 178R1A05C6 | ANEGONDE VENKATA SAI PRASANTH | Satisfaction |
|  | 178R1A05C7 | ANUMALA ROHITH REDDY | Excellent |
|  | 178R1A05C9 | BATHINI LAXMAN | Excellent |
|  | 178R1A05D3 | GURRAMPALLY RAHUL YADAV | Excellent |
|  | 178R1A05D6 | GIMKALA SAMYUKTHA | Satisfaction |
|  | 178R1A05D8 | GOLIPALLY HARSHITHA REDDY | Excellent |
|  | 178R1A05E0 | IPPILI LAKSH KIRAN | Excellent |
|  | 178R1A05E1 | KOTHAPA LAKSHMI PRIYA | Excellent |
|  | 178R1A05E2 | KALAKONDA SRI VIDYA | Satisfaction |
|  | 178R1A05E3 | KASARLA CHANDANA | Good |
|  | 178R1A05E4 | KASHA SUMANTH | Good |
|  | 178R1A05E6 | KOLUGURI RUKMINI REDDY | Good |
|  | 178R1A05E8 | L K LAVANYA | Good |
|  | 178R1A05F1 | MOTATI SRUJANA | Satisfaction |
|  | 178R1A05F2 | MADIVALA RENUKA | Excellent |
|  | 178R1A05F4 | MATTAPARTHI SAI KUMAR | Excellent |
|  | 178R1A05F5 | MUSKULA MEGHANA | Good |
|  | 178R1A05F8 | NERETY NEERAJA | Good |
|  | 178R1A05F9 | NILAM SURAJ | Excellent |
|  | 178R1A05G1 | NOMULA ANANDITHA | Satisfaction |
|  | 178R1A05G2 | NUNAVATH PRASHANTH | Excellent |
|  | 178R1A05G4 | PASUPULETI SAI RAHUL | Satisfaction |
|  | 178R1A05G7 | PASUPULETI MAHIJA | Excellent |
|  | 178R1A05G9 | RAMANAGARI PRASHANTHI | Excellent |
|  | 178R1A05H0 | PODDATURI SAI ROSHAN | Satisfaction |
|  | 178R1A05H4 | TANKASALA SAI TEJA | Good |
|  | 178R1A05H9 | YEDULLA NAVEEN ARAVIND REDDY | Good |
|  | 178R1A05I0 | YERUVA RAVITEJA | Satisfaction |
|  | 178R1A05I4 | BEERAM VAMSHI | Good |
|  | 178R1A05I7 | BOLLAPELLI HARIKA | Good |
|  | 178R1A05I9 | DEVARAKONDA LAKSHMI KAMESHWARI | Satisfaction |
|  | 178R1A05J0 | DAMMU SOWJANYA |  |
|  | 178R1A05J1 | YADAV DEEPALI | Excellent |
|  | 178R1A05J3 | G SAI SHEELA | Excellent |
|  | 178R1A05J5 | GAYATHRI PAREEK | Satisfaction |
|  | 178R1A05K4 | KANAKATLA SAGARIKA | Excellent |
|  | 178R1A05K5 | KARANI NETHRA NANDINI | Excellent |
|  | 178R1A05K7 | KEVA MAHESEKAR | Excellent |
|  | 178R1A05K9 | KONIKI VENKATA SAI NEEHARIKA | Good |
|  | 178R1A05L1 | MUCHARLA PRATHYUSHA REDDY | Good |
|  | 178R1A05L2 | MACHA NITHYA SANTHOSHINI | Excellent |
|  | 178R1A05L4 | MANCHANA MANISHA | Satisfaction |
|  | 178R1A05L6 | MEDARI RAGHU VARAN | Good |
|  | 178R1A05M1 | NADINDLA ASMITHA | Good |
|  | 178R1A05M2 | NALAMKULANGARA RAHUL KRISHNA | Excellent |
|  | 178R1A05M3 | NELAPATLA SARVANI | Satisfaction |
|  | 178R1A05M4 | PANDALA SHRAVANI | Excellent |
|  | 178R1A05M7 | RAHUL SAINI | Excellent |
|  | 178R1A05M8 | REVA MAHESEKAR | Excellent |
|  | 178R1A05M9 | S NIKITHA REDDY | Satisfaction |
|  | 178R1A05N0 | SAMOD VIJAYA KUMAR REDDY | Excellent |
|  | 178R1A05N3 | SOKKAM NAGARANI | Good |
|  | 178R1A05N5 | VAIDYAM VIKRAM KUMAR | Excellent |
|  | 178R1A05N6 | YAMJALA SUPRIYA | Satisfaction |
|  | 178R1A05N7 | YESUGARI ARAVIND | Excellent |
|  | 188R5A0512 | I SAI KUMAR | Excellent |
|  | 17601A0555 | NAVAPET TEJASWINI | Good |
|  | 17601A0574 | ADURI SRILEKHA | Satisfaction |
|  | 17601A0577 | TIGULLA SNEHA | Excellent |

The following rubric used for the presentation.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Evaluation criteria** | **Excellent**  **5.0 to >3.0 pts** | | **Good**  **3.0 to >2.0 pts** | | **Satisfaction**  **2.0 to >1.0 pts** | |
| Objectives | Presented objectives. | clear | Presented that relevant concepts. | objectives somewhat to the | Presented objectives. | wrong |
| Constructive Idea | Excellent idea with evidence and relevant content to demonstrate the problem. | | Little relevant content to demonstrate the problem. | | No relevant content to demonstrate the problem. | |
| Solution | Delivered the solution and content  professionally and answered the quires | | Delivered contents answered quires. | the and  the few | Delivered the contents and not answered the quires. | |