Backtracking

Sum of Subsets – How to solve using backtracking

Sum of Subsets Problem: Given a set of positive integers, find the combination of numbers that sum to given value M.

Sum of subsets problem is analogous to the knapsack problem. The Knapsack Problem tries to fill the knapsack using a given set of items to maximize the profit. Items are selected in such a way that the total weight in the knapsack does not exceed the capacity of the knapsack. The inequality condition in the knapsack problem is replaced by equality in the sum of subsets problem. Given the set of n positive integers, W = {w1, w2, …, wn}, and given a positive integer M, the sum of the subset problem can be formulated as follows (where wi and M correspond to item weights and knapsack capacity in the knapsack problem):

$$sum_{i=1}^{n} w_i x_i = M$$

Where,

$$x_i in 0, 1$$

Numbers are sorted in ascending order, such that $w_1 < w_2 < w_3 < .... < w_n$. The solution is often represented using the solution vector X. If the ith item is included, set xi to 1 else set it to 0. In each iteration, one item is tested. If the inclusion of an item does not violet the constraint of the problem, add it. Otherwise, backtrack, remove the previously added item, and continue the same procedure for all remaining items. The solution is easily described by the state space tree. Each left edge denotes the inclusion of wi and the right edge denotes the exclusion of w_i. Any path from the root to the leaf forms a subset. A state-space tree for n = 3 is demonstrated in Fig. (a).

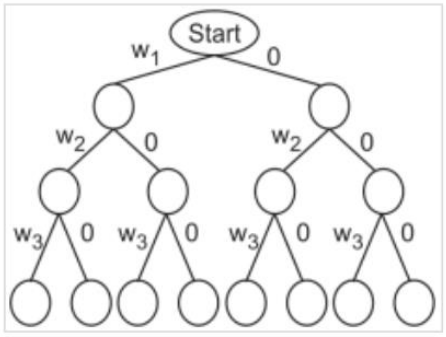


Fig. (a): State space tree for n = 3

Algorithm for Sum of subsets

The algorithm for solving the sum of subsets problem using recursion is stated below:

Algorithm

```
SUB_SET_PROBLEM(i, sum, W, remSum)
// Description : Solve sub of subset problem using backtracking
// Input :
W: Number for which subset is to be computed
i: Item index
sum : Sum of integers selected so far
remSum : Size of remaining problem i.e. (W – sum)
// Output : Solution tuple X
if
 FEASIBLE_SUB_SET(i) == 1
then
if
(sum == W)
then
   print X[1…i]
 end
else
 X[i + 1] ← 1
 SUB_SET_PROBLEM(i + 1, sum + w[i] + 1, W, remSum – w[i] + 1 )
 X[i + 1] ← 0          // Exclude the ith item
 SUB_SET_PROBLEM(i + 1, sum, W, remSum – w[i] + 1 )
end
function
FEASIBLE_SUB_SET(i)
if
(sum + remSum ≥ W) AND (sum == W) or (sum + w[i] + 1 ≤ W)
then
return0
end
return1
```

The first recursive call represents the case when the current item is selected, and hence the problem size is reduced by w[i].

The second recursive call represents the case when we do not select the current item.

Complexity Analysis

It is intuitive to derive the complexity of sum of the subset problem. In the state-space tree, at level i, the tree has 2i nodes. So, given n items, the total number of nodes in the tree would be $1 + 2 + 2^2 + 2^3 + .. 2^n$.

$T(n) = 1 + 2 + 2^2 + 2^3 + .. 2^n = 2^{n+1} – 1 = O(2^n)$

Thus, sum of sub set problem runs in exponential order.

Examples

Problem: Consider the sum-of-subset problem, n = 4, Sum = 13, and $w_1 = 3$, $w_2 = 4$, $w_3 = 5$ and $w_4 = 6$. Find a solution to the problem using backtracking. Show the state-space tree leading to the solution. Also, number the nodes in the tree in the order of recursion calls.

Solution:

The correct combination to get the sum of M = 13 for given W = {3, 4, 5, 6} is [3, 4, 6]. The solution vector for [3, 4, 6] would be X = [1, 1, 0, 1] because element 5 is not chosen, so X[3] = 0. Let's derive the solution using backtracking. The numbers in W are already sorted.

Set X = [0, 0, 0, 0]

Set Sum = 0. Sum indicates summation of selected numbers from W.

Step 1 : i = 1, Adding item w_i

$Sum = Sum + w_i = Sum + w_1 = 0 + 3 = 3$

$Sum \leq M$, so add item i to solution set.

$X[i] = X[1] = 1 \Rightarrow X = [1, 0, 0, 0]$

Step 2 : i = 2, Adding item w_2

$Sum = Sum + w_i = Sum + w_2 = 3 + 4 = 7$

$Sum \leq M$, so add item i to solution set.

$X[i] = X[2] = 1 \Rightarrow X = [1, 1, 0, 0]$

Step 3 : i = 3, Adding item w_3

$Sum = Sum + w_i = Sum + w_3 = 7 + 5 = 12$

Sum ≤ M, so add item i to solution set.

X[i] = X[3] = 1 ⇒ X =[1, 1, 1, 0]

Step 4 : i = 4, Adding item w_4

Sum = Sum + w_i = Sum + w_4 = 12 + 6 = 18

Sum > M, so backtrack and remove the previously added item from the solution set.

X[i] = X[3] = 0 ⇒ X =[1, 1, 0, 0].

Update Sum accordingly. So, Sum = Sum – w_3 = 12 – 5 = 7

And don't increment i.

Step 5 : i = 4, Adding item w_4

Sum = Sum + w_i = Sum + w_4 = 7 + 6 = 13

Sum = M, so solution is found and add item i to solution set.

X[i] = X[4] = 1 ⇒ X =[1, 1, 0, 1] A complete state space tree for given data is shown in Fig. (b)

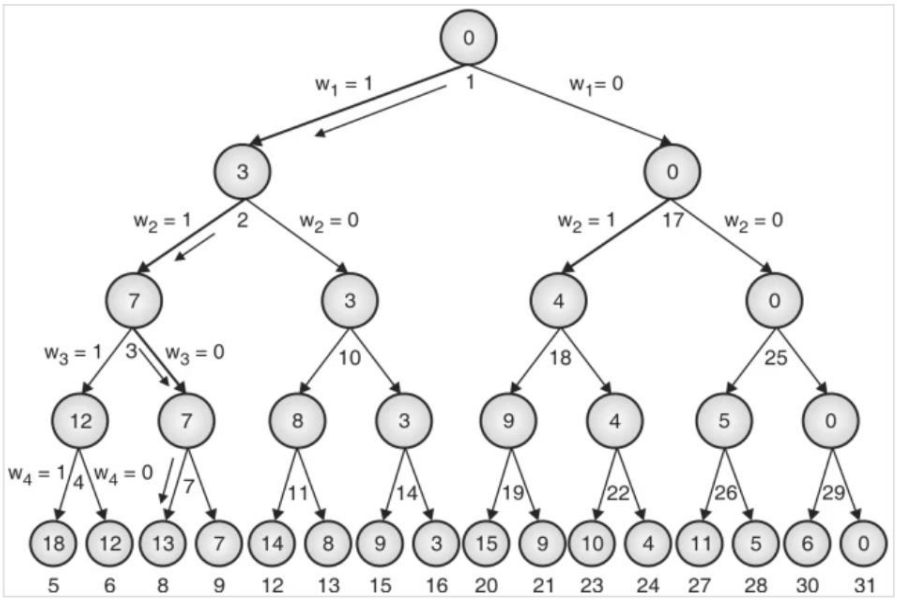


Fig. (b): State-space tree

At level i, the left branch corresponds to the inclusion of number w_i and the right branch corresponds to exclusion of number w_i. The recursive call number for the node is stated below the node. Node 8 is the solution node. The bold solid line shows the path to the output node.

Example:

Analyze sum of subsets algorithm on data :

M = 35 and

i) w = {5, 7, 10, 12, 15, 18, 20}

ii) w = {20, 18, 15, 12, 10, 7, 5}

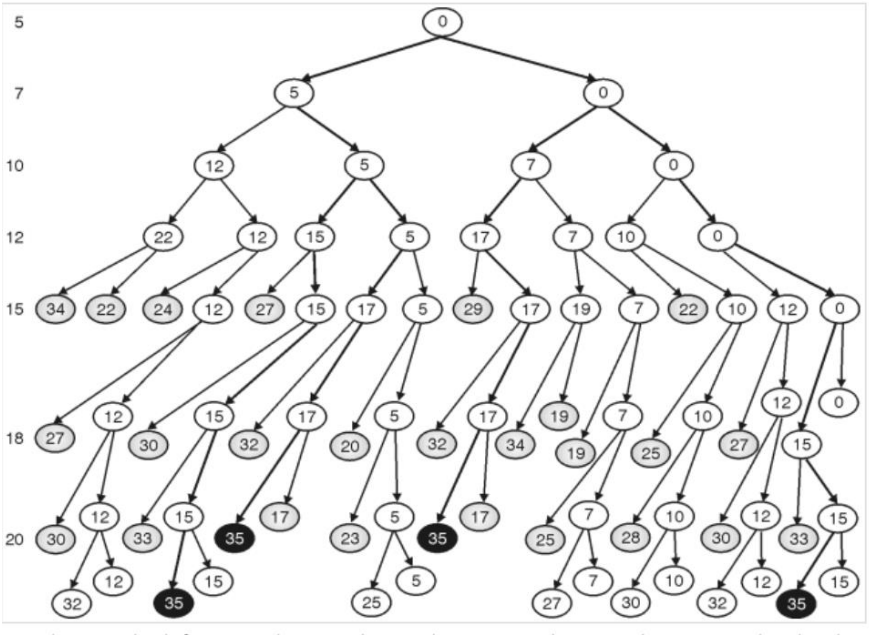
iii) w = {15, 7, 20, 5, 18, 10, 12} Are there any discernible differences in the computing time ?

Solution:

Let us run the algorithm on first instance w = {5, 7, 10, 12, 15, 18, 20}.

Items in sub set	Condition	Comment
{ }	0	Initial condition
{ 5 }	5 < 35	Select 5 and Add next element
{ 5, 7 }	12 < 35	Select 7 and Add next element
{ 5, 7, 10 }	22 < 35	Select 20 and Add next element
{ 5, 7, 10, 12 }	34 < 35	Select 12 and Add next element
{ 5, 7, 10, 12, 15 }	49 > 35	Sum exceeds M, so backtrack and remove 12
{ 5, 7, 10, 15 }	37 > 35	Sum exceeds M, so backtrack and remove 15
{ 5, 7, 12 }	24 < 35	Add next element
{ 5, 7, 12, 15 }	39 > 35	Sub set sum exceeds, so backtrack
{ 5, 10 }	15 < 35	Add next element
{ 5, 10, 12 }	27 < 35	Sub set sum exceeds, so backtrack
{ 5, 10, 12, 15 }	42 > 35	Sub set sum exceeds, so backtrack
{ 5, 10, 15 }	30 < 35	Add next element
{ 5, 10, 15, 18 }	48 > 35	Sub set sum exceeds, so backtrack
{ 5, 10, 18 }	33 < 35	Add next element
{ 5, 10, 18, 20 }	53 > 35	Sub set sum exceeds, so backtrack
{ 5, 10, 20 }	35	Solution Found

There may be multiple solutions. A state-space tree for the above sequence is shown here: The number in the leftmost column shows the element under consideration. The left and right branches in the tree indicate inclusion and exclusion of the corresponding element at that level, respectively.



Numbers in the leftmost column indicate elements under consideration at that level. A gray circle indicates the node that cannot accommodate any of the next values, so we will cut sort them from further expansion. White leaves do not lead to a solution. An intermediate white node may or may not lead to a solution. The black circle is the solution state.

We get the four solutions:

- {5, 10, 20}
- {5, 12, 18}
- {7, 10, 18}
- {15, 20}

For efficient execution of the subset sum problems, input elements should be sorted in non-decreasing order. If elements are not in non-decreasing order, the algorithm does more backtracking. So second and third sequences would take more time for execution and may not find as many solutions as we get in the first sequence.

Example:

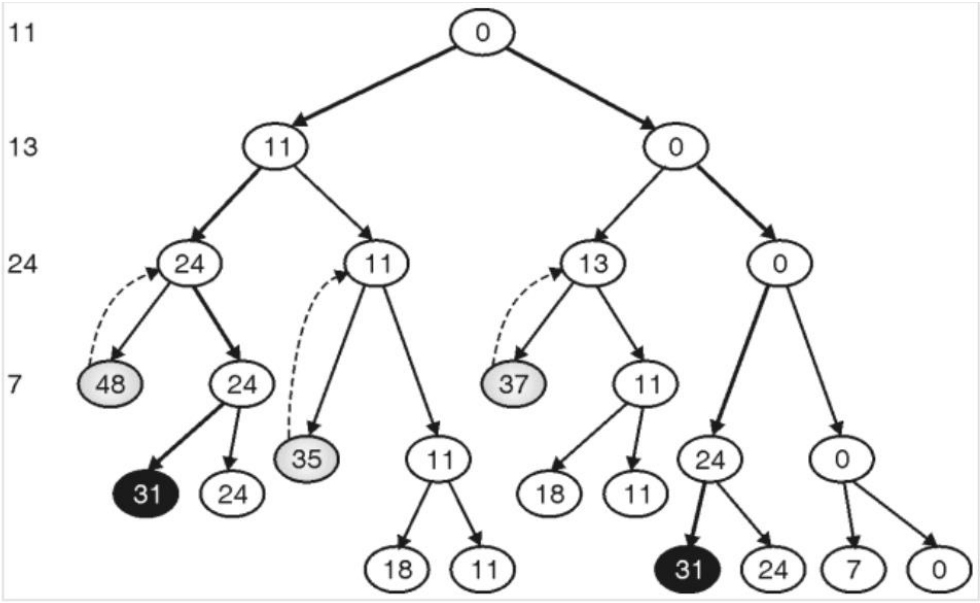
Solve the sum of subset problems using backtracking algorithmic strategy for the following data: n = 4 W =(w_1, w_2, w_3, w_4) = (11, 13, 24, 7) and M = 31.

Solution:

Items in sub set	Condition	Comment
{ }	0	Initial condition

{ 11 }	11 < 31	Add next element
{ 11, 13 }	24 < 31	Add next element
{ 11, 13, 24 }	48 < 31	Sub set sum exceeds, so backtrack
{ 11, 13, 7 }	31	Solution Found

State-space tree for a given problem is shown here:



In the above graph, the black circle shows the correct result. The gray node shows where the algorithm backtracks. Numbers in the leftmost column indicate elements under consideration at that level. The left and right branches represent the inclusion and exclusion of that element, respectively.

We get two solutions:

- {11, 13, 7}
- {24, 7}

Graph Coloring Problem

Introduction to Graph Coloring Problem

Graph coloring refers to the problem of coloring vertices of a graph in such a way that no two adjacent vertices have the same color. This is also called the **vertex coloring** problem.

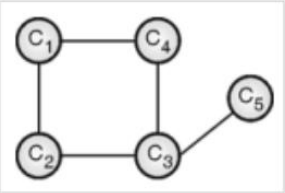
- If coloring is done using at most k colors, it is called **k-coloring**.

- The smallest number of colors required for coloring graph is called its **chromatic number**.
- The chromatic number is denoted by X(G). Finding the chromatic number for the graph is NP-complete problem.
- Graph coloring problem is both, decision problem as well as an optimization problem. A decision problem is stated as, “With given M colors and graph G, whether such color scheme is possible or not?”.
- The optimization problem is stated as, “Given M colors and graph G, find the minimum number of colors required for graph coloring.”
- Graph coloring problem is a very interesting problem of graph theory and it has many diverse applications. Few of them are listed below.

Applications of Graph Coloring Problem

- Design a timetable.
- Sudoku
- Register allocation in the compiler
- Map coloring
- Mobile radio frequency assignment:

The input to the graph is an adjacency matrix representation of the graph. Value M(i, j) = 1 in the matrix represents there exists an edge between vertex i and j. A graph and its adjacency matrix representation are shown in Figure (a)

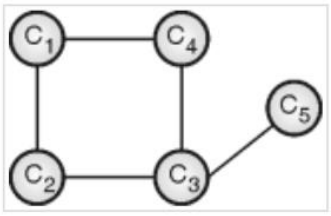


(a). Graph G

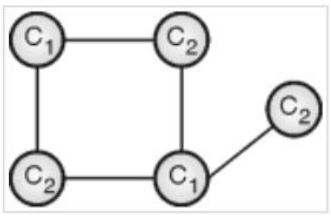
	C_1	C_2	C_3	C_4	C_5
C_1	0	1	0	1	0
C_2	1	0	1	0	0
C_3	0	1	0	1	1
C_4	1	0	1	0	1
C_5	0	0	1	0	0

Adjacency matrix for graph G

The problem can be solved simply by assigning a unique color to each vertex, but this solution is not optimal. It may be possible to color the graph with colors less than |V|. Figure (b) and figure (c) demonstrate both such instances. Let Ci denote the i^{th} color.



(b). Nonoptimal solution (uses 5 colors)



(c). Optimal solution (uses 2 colors)

- This problem can be solved using backtracking algorithms as follows:
  - List down all the vertices and colors in two lists
  - Assign color 1 to vertex 1
  - If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2.
  - Repeat the process until all vertices are colored.
- Algorithm backtracks whenever color i is not possible to assign to any vertex k and it selects next color i + 1 and test is repeated. Consider the graph shown in
  Figure (d)

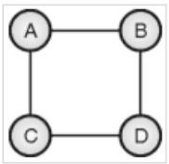


Figure (d)

If we assign color 1 to vertex A, the same color cannot be assigned to vertex B or C. In the next step, B is assigned some different colors 2. Vertex A is already colored and vertex D is a neighbor of B, so D cannot be assigned color 2. The process goes on. State-space tree is shown in Figure (e)

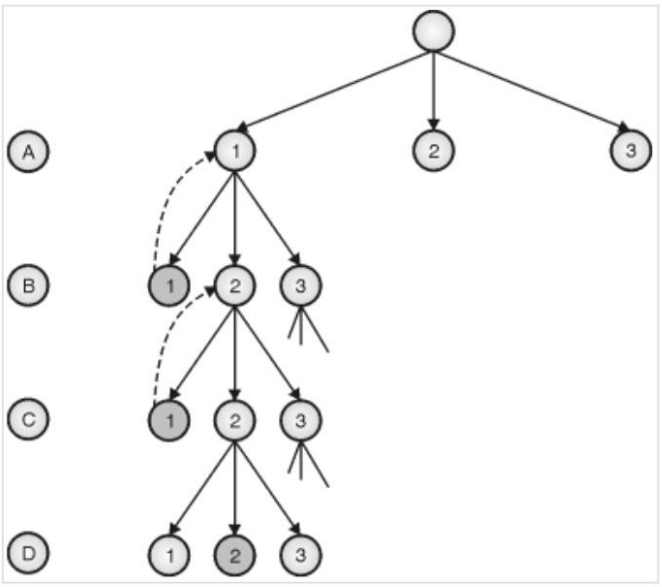


Figure (e): State-space tree of the graph of Figure (d)

Thus, vertices A and C will be colored with color 1, and vertices B and D will be colored with color 2.

Algorithm

Algorithm for graph coloring is described here:

Algorithm

```
GRAPH_COLORING(G, COLOR, i)
// Description :  Solve the graph coloring problem using backtracking
// Input : Graph G with n vertices, list of colors, initial
vertex i
// COLOR[1…n] is the array of n different colors
// Output : Colored graph with minimum color
if
CHECK_VERTEX(i) == 1
then

if
```

```
i == N
then

print
COLOR[1…n]
else
   j ← 1

while
(j ≤ M)
do
    COLOR(i + 1) ← j
     j ← j + 1
end

end


end
Function
CHECK_VERTEX(i)
for
j ← 1 to i – 1
do

if
Adjacent(i, j)
then
if
COLOR(i) == COLOR(j)
then

return
0
end

end
end
return
1
```

Complexity Analysis

The number of anode increases exponentially at every level in state space tree. With M colors and n vertices, total number of nodes in state space tree would be $1 + M + M^2 + M^3 + ... + M^n$

Hence, $T(n) = 1 + M + M^2 + M^3 + ... + M^n$

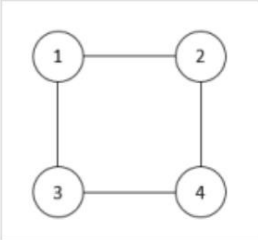
$=fracM^{n+1}-1M-1$

So, $T(n) = O(M^n)$.

Thus, the graph coloring algorithm runs in exponential time.

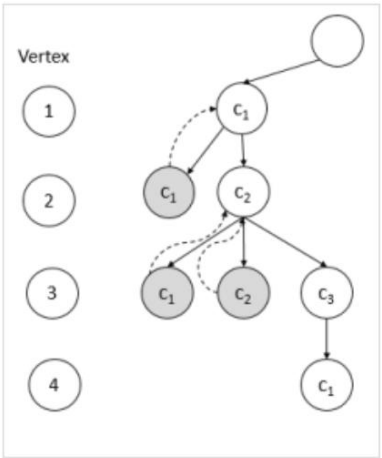
Examples on Graph Coloring Problem

Example: Apply backtrack on the following instance of graph coloring problem of 4 nodes and 3 colors



Solution:

This problem can be solved using backtracking algorithms. The formal idea is to list down all the vertices and colors in two lists. Assign color 1 to vertex 1. If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2. The process is repeated until all vertices are colored. The algorithm backtracks whenever color i is not possible to assign to any vertex k and it selects the next color i + 1 and the test is repeated. This graph can be colored with 3 colors. The solution tree is shown in Fig. (i):



Flg. (i): Solution tree

Vertex	Assigned Color
1	c_1
2	c_2
3	c_3
4	c_1

Hamiltonian Cycle using Backtracking

Hamiltonian Cycle: What it is?

*The **Hamiltonian cycle** is the cycle in the graph which visits all the vertices in graph exactly once and terminates at the starting node. It may not include all the edges*

- **The** Hamiltonian cycle problem is the problem of finding a Hamiltonian cycle in a graph if there exists any such cycle.
- The input to the problem is an undirected, connected graph. For the graph shown in Figure (a), a path A – B – E – D – C – A forms a Hamiltonian cycle. It visits all the vertices exactly once, but does not visit the edges <B, D>.

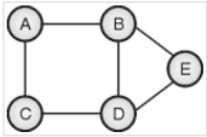


Figure (a)

- The Hamiltonian cycle problem is also both, decision problem and an optimization problem. A decision problem is stated as, “Given a path, is it a Hamiltonian cycle of the graph?”.
- The optimization problem is stated as, “Given graph G, find the Hamiltonian cycle for the graph.”
- We can define the constraint for the Hamiltonian cycle problem as follows:
  - In any path, vertex i and (i + 1) must be adjacent.
  - 1st and (n – 1)th vertex must be adjacent (nth of cycle is the initial vertex itself).
  - Vertex i must not appear in the first (i – 1) vertices of any path.
- With the adjacency matrix representation of the graph, the adjacency of two vertices can be verified in constant time.

Algorithm for Hamiltonian Cycle Problem using Backtracking

Algorithm

```
HAMILTONIAN (i)
// Description : Solve Hamiltonian cycle problem using backtracking.
// Input : Undirected, connected graph G = <V, E> and initial vertex i
// Output : Hamiltonian cycle
if
FEASIBLE(i)
then
if
(i == n - 1)
then
  Print V[0… n – 1]
else
  j ← 2
while
(j ≤ n)
do
    V[i] ← j
    HAMILTONIAN(i + 1)
    j ← j + 1
  end
end
end
function
FEASIBLE(i)
flag ← 1
for
j ← 1 to i – 1
do
if
 Adjacent(Vi, Vj)
```

then

flag ← 0

end

end

if

Adjacent (Vi, Vi-1)

then

flag ← 1

else

flag ← 0

end

return

flag

Complexity Analysis

Looking at the state space graph, in worst case, total number of nodes in tree would be,

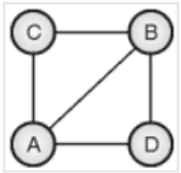
$T(n) = 1 + (n - 1) + (n - 1)^2 + (n - 1)^3 + \ldots + (n - 1)^{n-1}$

=frac(n−1)n−1n−2

$T(n) = O(n^n)$. Thus, the Hamiltonian cycle algorithm runs in exponential time.

Examples

Example: Show that given graph has a Hamiltonian cycle



Solution:

We can start with any random vertex. Let us start with vertex A. Neighbors of A are {B, C, D}. The inclusion of B does not lead to a complete solution. So explore it as shown in Figure (c).

Adjacent vertices to B are {C, D}. The inclusion of C does not lead to a complete solution. All adjacent vertices of C are already members of the path formed so far. So it leads to a dead end.

Backtrack and go to B and explore its next neighbor i.e. D.

The inclusion of D does not lead to a complete solution, and all adjacent vertices of D are already a member of the path formed so far. So it leads to a dead end.

Figur e: (b) initial tree

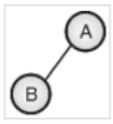


Figure (c): Added node B

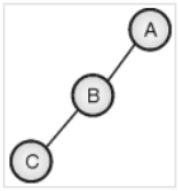


Figure (d): Node C added, dead-end

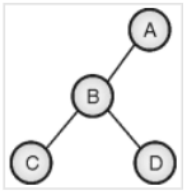


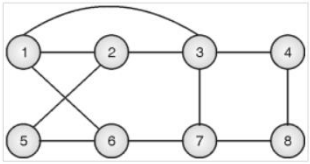
Figure (e): Node D added, dead-end

Backtrack and go to B. Now B does not have any more neighbors, so backtrack and go to A. Explore the next neighbor of A i.e. C. By repeating the same procedure, we get the state space trees as shown below. And path A – C – B – D – A is detected as the Hamiltonian cycle of the input graph.

Another Hamiltonian cycle with A as a start vertex is A – D – B – C – A.

Example

Example: Explain how to find Hamiltonian Cycle by using Backtracking in a given graph



Solution:

The backtracking approach uses a state-space tree to check if there exists a Hamiltonian cycle in the graph. Figure (f) shows the simulation of the Hamiltonian cycle algorithm. For simplicity, we have not explored all possible paths, the concept is self-explanatory.

Step 1: Tour is started from vertex 1. There is no path from 5 to 1. So it's the dead-end state.

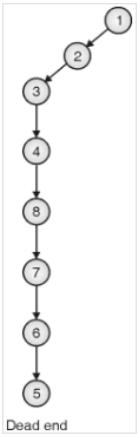
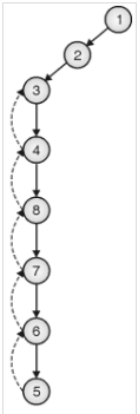
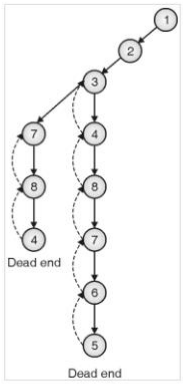


Figure (f)

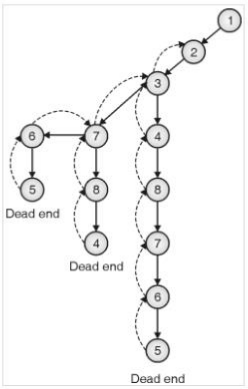
Step 2: Backtrack to the node from where the new path can be explored, that is 3 here



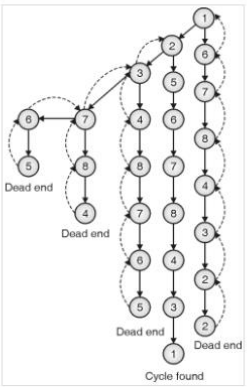
Step 3: New path also leads to a dead end so backtrack and explore all possible paths



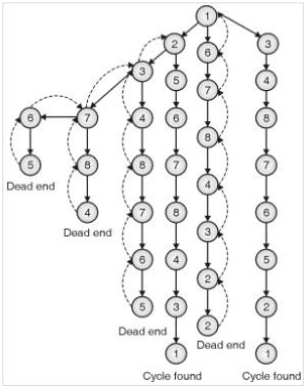
Step 4: Next path is also leading to a dead-end so keep backtracking until we get some node that can generate a new path, i.e .vertex 2 here



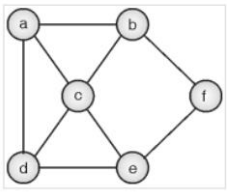
Step 5: One path leads to Hamiltonian cycle, next leads to a dead end so backtrack and explore all possible paths at each vertex



Step 6: Total two Hamiltonian cycles are detected in a given graph



Example: Find the Hamiltonian cycle by using the backtracking approach for a given graph.



Solution:

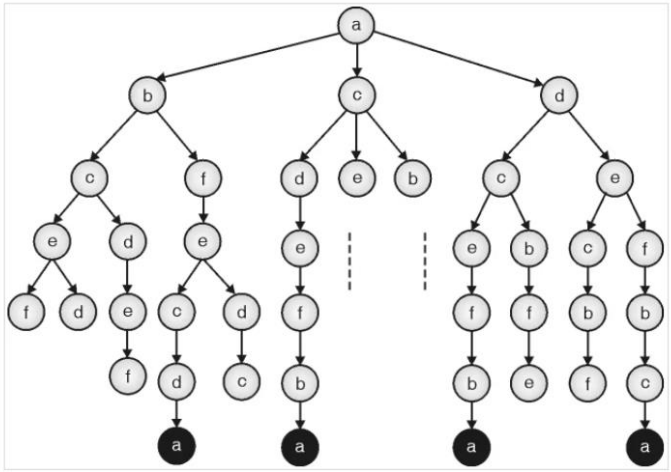


Figure (g)

The backtracking approach uses a state-space tree to check if there exists a Hamiltonian cycle in the graph. Figure (g) shows the simulation of the Hamiltonian cycle algorithm. For simplicity, we have not explored all possible paths, the concept is self-explanatory. It is not possible to include all the paths in the graph, so few of the successful and unsuccessful paths are traced in the graph. Black nodes indicate the Hamiltonian cycle.

N Queen Problem

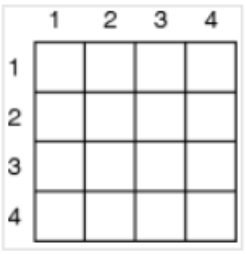
What is N Queen Problem?

N Queen problem is the classical Example of backtracking. N-Queen problem is defined as, "given N x N chess board, arrange N queens in such a way that no two queens attack each other by being in same row, column or diagonal".

- For N = 1, this is trivial case. For N = 2 and N = 3, solution is not possible. So we start with N = 4 and we will generalize it for N queens.

4-Queen Problem

Problem : Given 4 x 4 chessboard, arrange four queens in a way, such that no two queens attack each other. That is, no two queens are placed in the same row, column, or diagonal.



4 x 4 Chessboard

- We have to arrange four queens, Q1, Q2, Q3 and Q4 in 4 x 4 chess board. We will put ith queen in ith row. Let us start with position (1, 1). Q1 is the only queen, so there is no issue. partial solution is <1>
- We cannot place Q2 at positions (2, 1) or (2, 2). Position (2, 3) is acceptable. partial solution is <1, 3>.
- Next, Q3 cannot be placed in position (3, 1) as Q1 attacks her. And it cannot be placed at (3, 2), (3, 3) or (3, 4) as Q2 attacks her. There is no way to put Q3 in third row. Hence, the algorithm backtracks and goes back to the previous solution and readjusts the position of queen Q2. Q2 is moved from positions(2,3)to (2, 4). Partial solution is <1, 4>
- Now, Q3 can be placed at position (3, 2). Partial solution is <1, 4, 3>.
- Queen Q4 cannot be placed anywhere in row four. So again, backtrack to the previous solution and readjust the position of Q3. Q3 cannot be placed on (3, 3) or(3, 4). So the algorithm backtracks even further.
- All possible choices for Q2 are already explored, hence the algorithm goes back to partial solution <1> and moves the queen Q1 from (1, 1) to (1, 2). And this process continues until a solution is found.
- All possible solutions for 4-queen are shown in fig (a) & fig. (b)

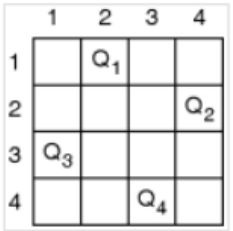


Fig. (a): Solution – 1

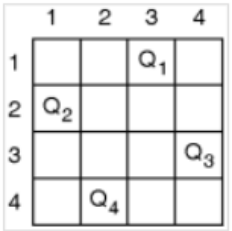


Fig. (b): Solution – 2

We can see that backtracking is a simple brute force approach, but it applies some intelligence to cut out unnecessary computation. The solution tree for the 4-queen problem is shown in Fig. (c).

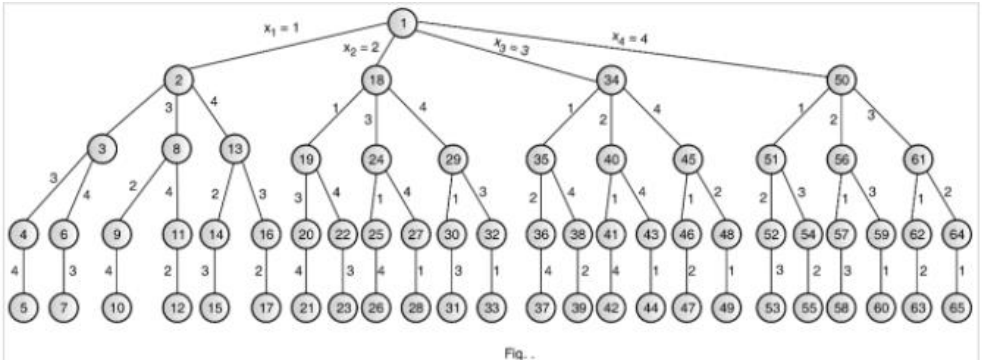


Fig. (c): State-space diagram for 4 – queen problem

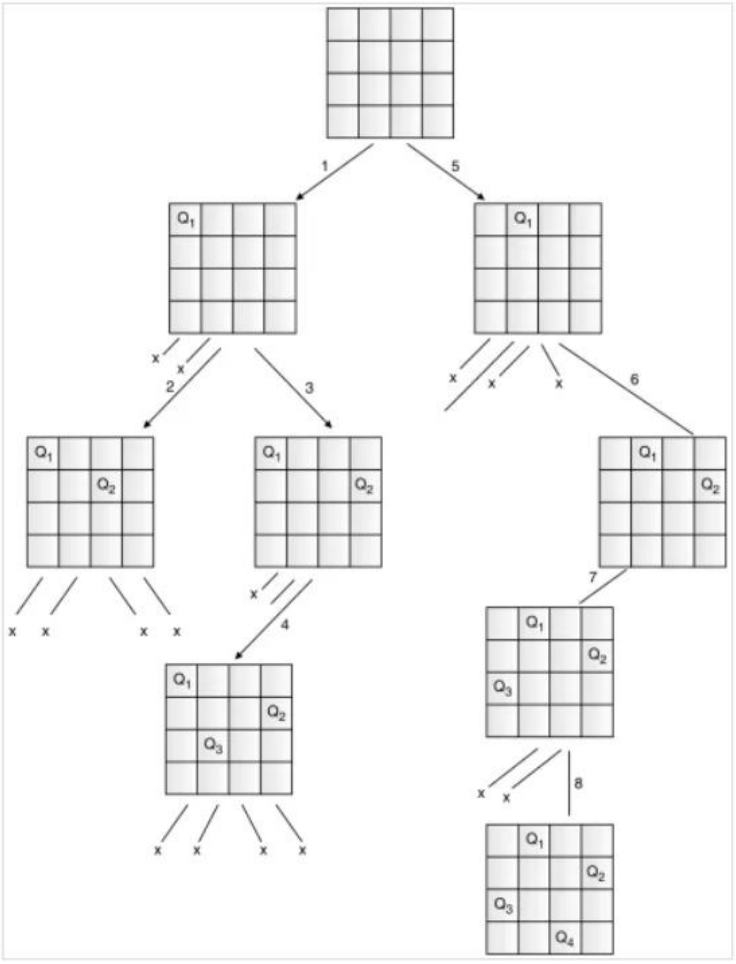


Fig. (d): Backtracking sequence for 4-queen

The solution of the 4-queen problem can be seen as four tuples (x_1, x_2, x_3, x_4), where x_i represents the column number of queen Q_i. Two possible solutions for the 4-queen problem are (2, 4, 1, 3) and (3, 1, 4, 2).

8-Queen Problem

Problem : Given an 8 x 8 chessboard, arrange 8 queens in a way such that no two queens attack each other.

Two queens are attacking each other if they are in the same row, column, or diagonal. Cells attacked by queen Q are shown in fig. (e).

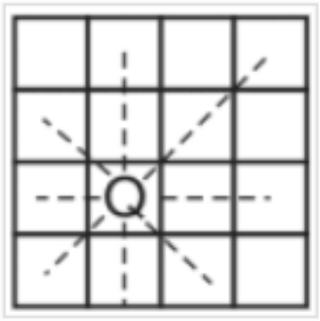


Fig. (e): Attacked cells by queen Q

- 8 queen problem has $^{64}C_8$= 4,42,61,65,368 different arrangements. Of these, only 92 arrangements are valid solutions. Out of which, only 12 are the fundamental solutions. The remaining 80 solutions can be generated using reflection and rotation.
- The 2-queen problem is not feasible. The minimum problem size for which a solution can be found is 4. Let us understand the workings of backtracking on the 4-queen problem.
- For simplicity, a partial state space tree is shown in fig. (f). Queen 1 is placed in the first column in the first row. All the positions are crossed in which Queen 1 is attacking. In the next level, queen 2 is placed in a 3rd column in row 2 and all cells that are crossed are attacked by already placed queens 1 and 2. As can be seen from fig (f), no place is left to place the next queen in row 3, so queen 2 backtracks to the next possible position and the process continues. In a similar way, if (1, 1) position is not feasible for queen 1, then the algorithm backtracks and puts the first queen in cell (1, 2), and repeats the procedure. For simplicity, only a few nodes are shown in fig. (f).

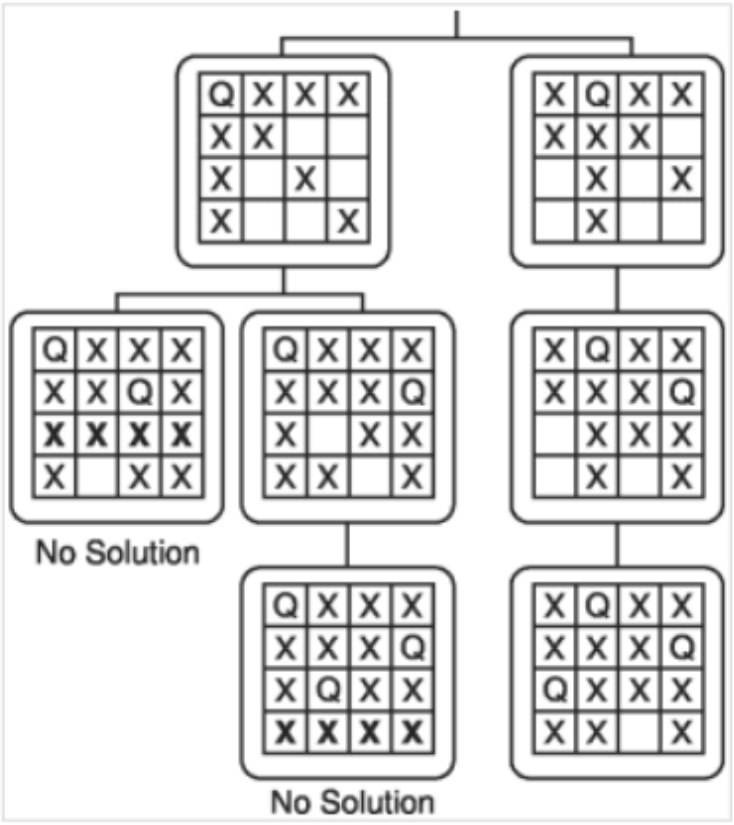


Fig. (f): Snapshot of backtracking procedure of 4-Queen problem

- A complete state space tree for the 4-queen problem is shown in fig. (g)
- The number within the circle indicates the order in which the node gets explored. The height of the e from the t indicates row and label, besides the arc indicating that the Q is placed in an ith column. Out of all the possible states, only a few are the answer states.

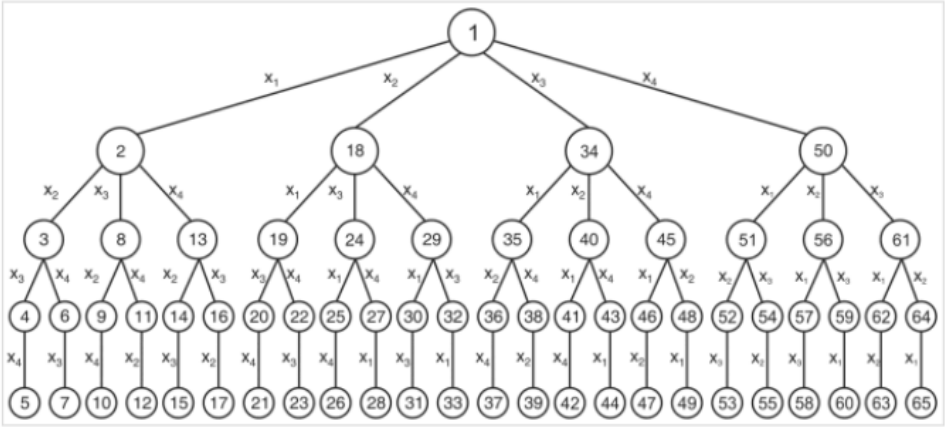


Fig. (g): Solution of 8-queen problem

-

- Solution tuple for the solution shown in fig (h) is defined as <4, 6, 8, 2, 7, 1, 3, 5>. From observations, two queens placed at (i, j) and (k, l) positions, can be in same diagonal only if,

$$(i - j) = (k - l) \text{ or}$$

$$(i + j) = (k + l)$$

From first equality, $j - l = i - k$

From second equality, $j - l = k - i$

So queens can be in diagonal only if $|j - l| = |i - k|$.

The arrangement shown in fig. (i) leads to failure. As it can be seen from fig. (i), Queen Q6 cannot be placed anywhere in the 6th row. So the position of Q5 is backtracked and it is placed in another feasible cell. This process is repeated until the solution is found.

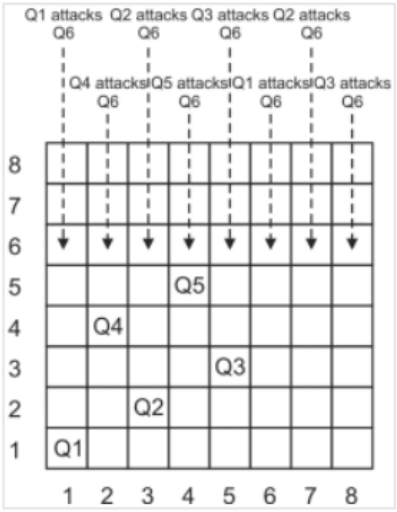


Fig. (i): Non-feasible state of the 8-queen problem

Algorithm

The following algorithm arranges n queens on n x n board using a backtracking algorithm.

Algorithm

N_QUEEN (k, n)

```
// Description : To find the solution of n x n queen problem using backtracking
// Input :
n: Number of queen
k: Number of the queen being processed currently, initially set to 1.

// Output : n x 1 Solution tuple

for
i ← 1 to n
do

if
PLACE(k , i)
then
   x[k] ← i

if
k == n
then
     print X[1…n]

else
     N_QUEEN(k + 1, n)
   end
end
end
```

- Function PLACE (k, i) returns true, if the k^{th} queen can be placed in the i^{th} column. This function enumerates all the previously kept queen's positions to check if two queens are on the same diagonal. It also checks that i is distinct from all previously arranged queens.
- Function abs(a) returns the absolute value of argument a. The array X is the solution tuple.
- Like all optimization problems, the n-queen problem also has some constraints that it must satisfy. These constraints are divided into two categories : Implicit and explicit constraints

Function

```
PLACE(k, i)
```

```
// k is the number of queen being processed
// i is the number of columns
for
 j ← 1 to k – 1
do
if
x[j] == i OR ((abs(x[j]) - i) == abs(j - k))
then
return
false
end
end
return
true
```

Explicit Constraints:

- Explicit constraints are the rules that allow/disallow selection of x_i to take value from the given set. For example, $x_i = 0$ or 1.

  x_i = 1 if $LB_i \leq x_i \leq UB_i$

  x_i = 0 otherwise
- **Solution space** is formed by the collection of all tuple which satisfies the constraint.

Implicit constraints:

- The implicit constraint is to determine which of the tuple of solution space satisfies the given criterion functions. The implicit constraint for n queen problem is that two queens must not appear in the same row, column or diagonal.
- **Complexity analysis :** In backtracking, at each level branching factor decreases by 1 and it creates a new problem of size (n – 1) . With n choices, it creates n different problems of size (n – 1) at level 1.
- PLACE function determines the position of the queen in O(n) time. This function is called n times.

- Thus, the recurrence of n-Queen problem is defined as, $T(n) = n*T(n-1) + n^2$. Solution to recurrence would be O(n!).

Travelling Salesman Problem – Solved using Branch and Bound

Travelling Salesman Problem – What it is?

- Travelling Salesman Problem (TSP) is an interesting problem. Problem is defined as "given n cities and distance between each pair of cities, find out the path which visits each city exactly once and come back to starting city, with the constraint of minimizing the travelling distance."
- TSP has many practical applications. It is used in network design, and transportation route design. The objective is to minimize the distance. We can start tour from any random city and visit other cities in any order. With n cities, n! different permutations are possible. Exploring all paths using brute force attacks may not be useful in real life applications.

LCBB using Static State Space Tree for Travelling Salseman Problem

- Branch and bound is an effective way to find better, if not best, solution in quick time by pruning some of the unnecessary branches of search tree.
- It works as follow :

Consider directed weighted graph G = (V, E, W), where node represents cities and weighted directed edges represents direction and distance between two cities.

1. Initially, graph is represented by cost matrix C, where

C_{ij} = cost of edge, if there is a direct path from city i to city j
C_{ij} = ∞, if there is no direct path from city i to city j.

2. Convert cost matrix to reduced matrix by subtracting minimum values from appropriate rows and columns, such that each row and column contains at least one zero entry.

3. Find cost of reduced matrix. Cost is given by summation of subtracted amount from the cost matrix to convert it in to reduce matrix.

4. Prepare state space tree for the reduce matrix

5. Find least cost valued node A (i.e. E-node), by computing reduced cost node matrix with every remaining node.

6. If <i, j> edge is to be included, then do following :

(a) Set all values in row i and all values in column j of A to ∞

(b) Set A[j, 1] = ∞

(c) Reduce A again, except rows and columns having all ∞ entries.

7. Compute the cost of newly created reduced matrix as,

$$\text{Cost} = L + \text{Cost}(i, j) + r$$

Where, L is cost of original reduced cost matrix and r is A[i, j].

8. If all nodes are not visited then go to step 4.

Reduction procedure is described below :

Raw Reduction:

Matrix M is called reduced matrix if each of its row and column has at least one zero entry or entire row or entire column has ∞ value. Let M represents the distance matrix of 5 cities. M can be reduced as follow:

$M_{RowRed} = \{M_{ij} - \min \{M_{ij} \mid 1 \leq j \leq n, \text{ and } M_{ij} < \infty \}\}$

Consider the following distance matrix:

Find the minimum element from each row and subtract it from each cell of matrix.

Consider the following distance matrix:

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

Find the minimum element from each row and subtract it from each cell of matrix.

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

Reduced matrix would be:

$$M_{RowRed} = \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

Row reduction cost is the summation of all the values subtracted from each rows:

Row reduction cost (M) = 10 + 2 + 2 + 3 + 4 = 21

Column reduction:

Matrix M_{RowRed} is row reduced but not the column reduced. Matrix is called column reduced if each of its column has at least one zero entry or all ∞ entries.

$M_{ColRed} = \{M_{ji} - \min \{M_{ji} \mid 1 \leq j \leq n, \text{ and } M_{ji} < \infty \}\}$

To reduced above matrix, we will find the minimum element from each column and subtract it from each cell of matrix.

Column reduced matrix M_{ColRed} would be:

M_{RowRed} =

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞
↓	↓	↓	↓	↓
1	0	3	0	0

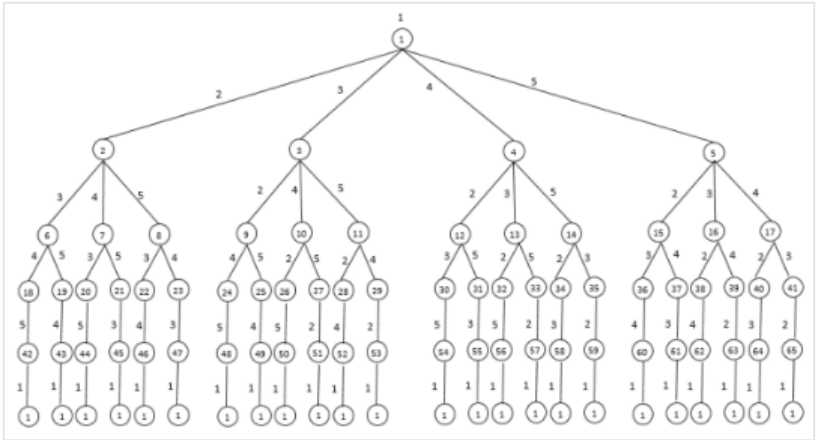
Column reduced matrix M_{ColRed} would be:

M_{ColRed} =

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

Each row and column of M_{ColRed} has at least one zero entry, so this matrix is reduced matrix.
Column reduction cost (M) = 1 + 0 + 3 + 0 + 0 = 4

State space tree for 5 city problem is depicted in Fig. 6.6.1. Number within circle indicates the order in which the node is generated, and number of edge indicates the city being visited.



Example

Example: Find the solution of following travelling salesman problem using branch and bound method.

Cost Matrix =

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Solution:

- The procedure for dynamic reduction is as follow:
- Draw state space tree with optimal reduction cost at root node.
- Derive cost of path from node i to j by setting all entries in i^{th} row and j^{th} column as ∞.

Set M[j][i] = ∞

- Cost of corresponding node N for path i to j is summation of optimal cost + reduction cost + M[j][i]
- After exploring all nodes at level i, set node with minimum cost as E node and repeat the procedure until all nodes are visited.
- Given matrix is not reduced. In order to find reduced matrix of it, we will first find the row reduced matrix followed by column reduced matrix if needed. We can find row reduced matrix by subtracting minimum element of each row from each element of corresponding row. Procedure is described below:
- Reduce above cost matrix by subtracting minimum value from each row and column.

∞	20	30	10	11	→ 10	∞	10	20	0	1	
15	∞	16	4	2	→ 2	13	∞	14	2	0	
3	5	∞	2	4	→ 2	⇒ 1	3	∞	0	2	= M_1'
19	6	18	∞	3	→ 3	16	3	15	∞	0	
16	4	7	16	∞	→ 4	12	0	3	12	∞	
						↓ 1	↓ 0	↓ 3	↓ 0	↓ 0	

M_1'

is not reduced matrix. Reduce it subtracting minimum value from corresponding column. Doing this we get,

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

$= M_1$

Cost of M_1 = C(1)

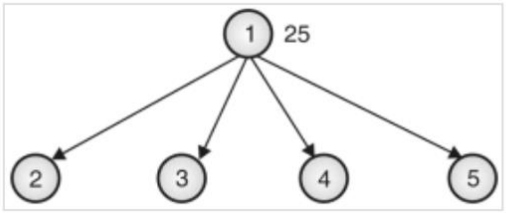
= Row reduction cost + Column reduction cost

= (10 + 2 + 2 + 3 + 4) + (1 + 3) = 25

This means all tours in graph has length at least 25. This is the optimal cost of the path.

State space tree

State space tree



Let us find cost of edge from node 1 to 2, 3, 4, 5.

Let us find cost of edge from node 1 to 2, 3, 4, 5.

Select edge 1-2:

Set M_1 [1] [ ] = M_1 [ ] [2] = ∞

Set M_1 [2] [1] = ∞

Reduce the resultant matrix if required.

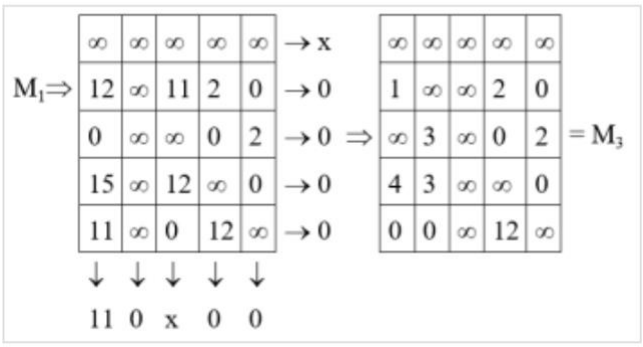
∞	∞	∞	∞	∞	→ x
∞	∞	11	2	0	→ 0
0	∞	∞	0	2	→ 0
15	∞	12	∞	0	→ 0
11	∞	0	12	∞	→ 0
↓	↓	↓	↓	↓	
0	x	0	0	0	

$= M_2$

M_2 is already reduced.
Cost of node 2 :

C(2) = C(1) + Reduction cost + M_1 [1] [2]
= 25 + 0 + 10 = 35

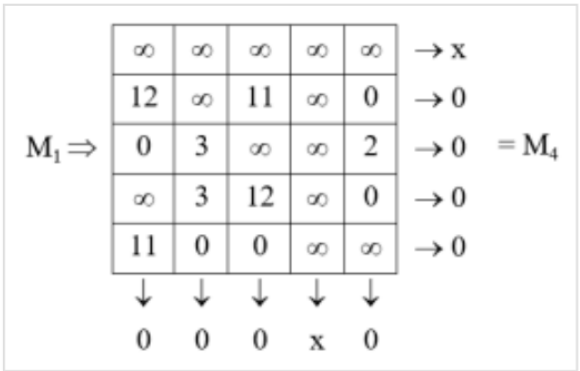
Select edge 1-3
Set M_1 [1][ ] = M_1 [ ] [3] = ∞
Set M_1 [3][1] = ∞
Reduce the resultant matrix if required.



Cost of node 3:

C(3) = C(1) + Reduction cost + M_1[1] [3]
= 25 + 11 + 17 = 53

Select edge 1-4:
Set M_1 [1][ ] = M_1[ ][4] = ∞
Set M_1 [4][1] = ∞
Reduce resultant matrix if required.



Matrix M_4 is already reduced.

Cost of node 4:

C(4) = C(1) + Reduction cost + M_1 [1] [4]
= 25 + 0 + 0 = 25

Select edge 1-5:
Set M_1 [1] [ ] = M_1 [ ] [5] = ∞
Set M_1 [5] [1] = ∞
Reduce the resultant matrix if required.

$M_1 \Rightarrow$

∞	∞	∞	∞	∞	→ x
12	∞	11	2	∞	→ 2
0	3	∞	0	∞	→ 0
15	3	12	∞	∞	→ 3
∞	0	0	12	∞	→ 0

$\Rightarrow$

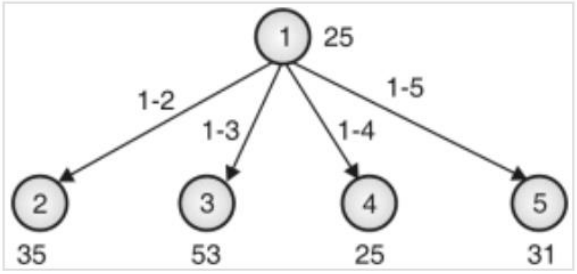
∞	∞	∞	∞	∞	
10	∞	9	0	∞	
0	3	∞	0	∞	$= M_5$
12	0	9	∞	∞	
∞	0	0	12	∞	
↓	↓	↓	↓	↓	
0	0	0	0	x	

Cost of node 5:

C(5) = C(1) + reduction cost + M_1 [1] [5]
= 25 + 5 + 1 = 31

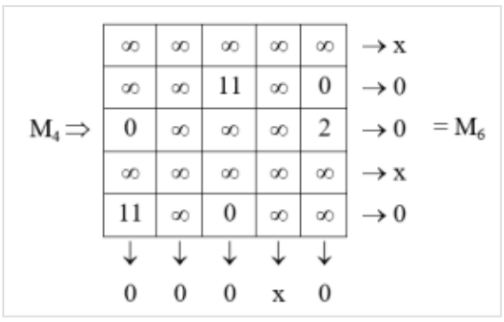
State space diagram:

State space diagram:



Node 4 has minimum cost for path 1-4. We can go to vertex 2, 3 or 5. Let's explore all three nodes.

Select path 1-4-2 : (Add edge 4-2)
Set M_4 [1] [] = M_4 [4] [] = M_4 [] [2] = ∞
Set M_4 [2] [1] = ∞
Reduce resultant matrix if required.

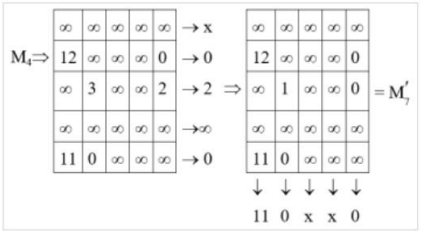


Matrix M_6 is already reduced.
Cost of node 6:

C(6) = C(4) + Reduction cost + M_4 [4] [2]
= 25 + 0 + 3 = 28

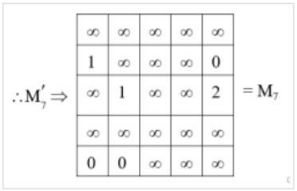
Select edge 4-3 (Path 1-4-3):
Set M_4 [1] [ ] = M_4 [4] [ ] = M_4 [ ] [3] = ∞
Set M [3][1] = ∞

Reduce the resultant matrix if required.



M_7'

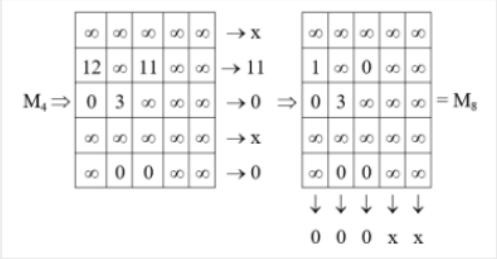
is not reduced. Reduce it by subtracting 11 from column 1.



Cost of node 7:

C(7) = C(4) + Reduction cost + M_4 [4] [3]
= 25 + 2 + 11 + 12 = 50

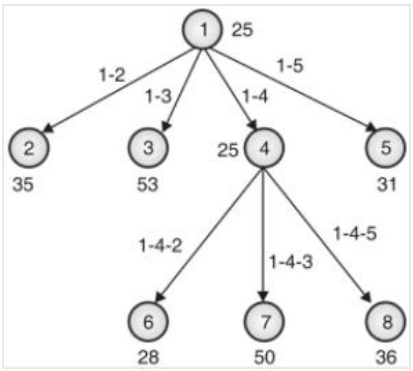
Select edge 4-5 (Path 1-4-5):



Matrix M_8 is reduced.
Cost of node 8:

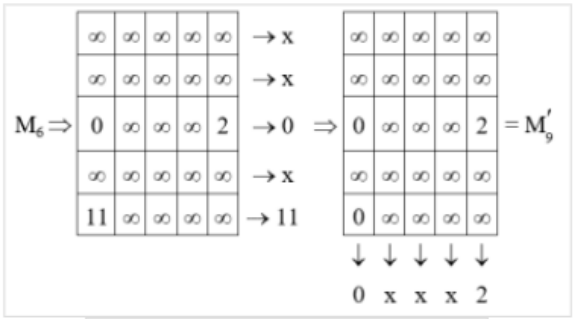
C(8) = C(4) + Reduction cost + M_4 [4][5]
= 25 + 11 + 0 = 36

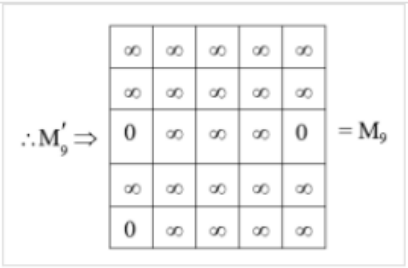
State space tree
Path 1-4-2 leads to minimum cost. Let's find the cost for two possible paths.



Add edge 2-3 (Path 1-4-2-3):
Set M_6 [1][ ] = M_6 [4][ ] = M_6 [2][ ]
= M_6 [][3] = ∞
Set M_6 [3][1] = ∞
Reduce resultant matrix if required.

Reduce resultant matrix if required.

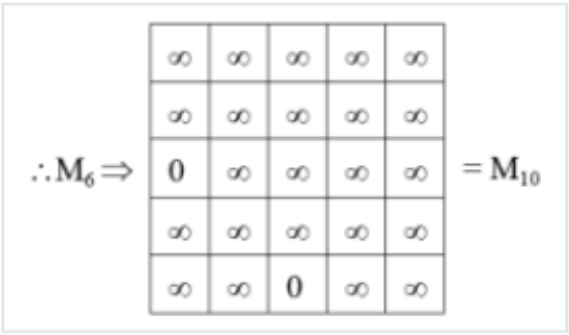




Cost of node 9:

C(9) = C(6) + Reduction cost + M_6 [2][3]
= 28 + 11 + 2 + 11 = 52

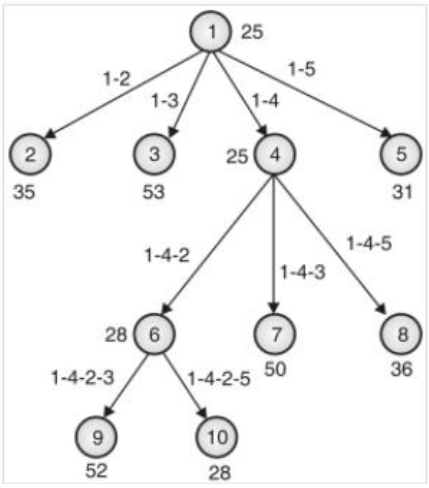
Add edge 2-5 (Path 1-4-2-5):
Set M_6 [1][ ] = M_6 [4][ ] = M_6 [2][ ] = M_6 [ ][5] = ∞
Set M_6 [5][1] = ∞
Reduce resultant matrix if required.



Cost of node 10:

C(10) = C(6) + Reduction cost + M_6 [2][5]
= 28 + 0 + 0 = 28

State space tree



Add edge 5-3 (Path 1-4-2-5-3):

$\therefore\ M_{10} \Rightarrow$

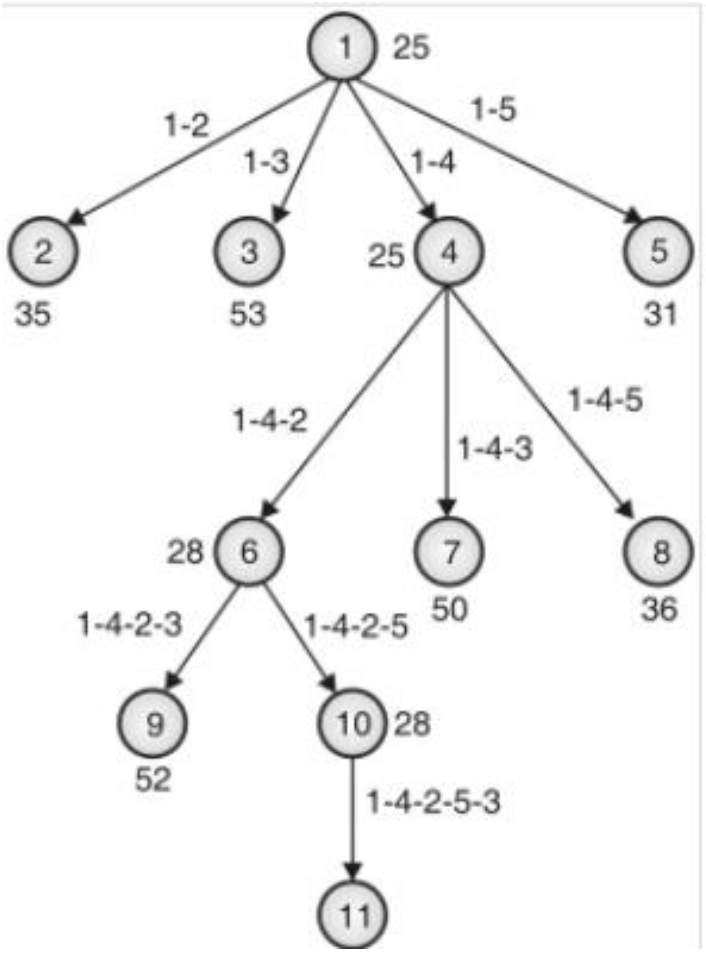
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

$= M_{11}$

Cost of node 11:

C(11) = C(10) + Reduction cost + M_{10} [5][3]
= 28 + 0 + 0 = 28

State space tree:


1
25
1-2
1-3
1-4
1-5
2
35
3
53
25
4
5
31
1-4-2
1-4-3
1-4-5
28
6
7
50
8
36
1-4-2-3
1-4-2-5
9
52
10
28
1-4-2-5-3
11