Backtracking

Backtracking is the most general technique to solve problem searching for a set of solutions or find an optimal solution by satisfying some constraints.

In Backtracking the solution is expressed as n-tuple $(x_1,x_2\ldots.x_n)$, where xi are chosen from some finite set S_i.

Some of the problems whose solutions are best done using backtracking are

1. N-queens problem

2. Sum of Subsets

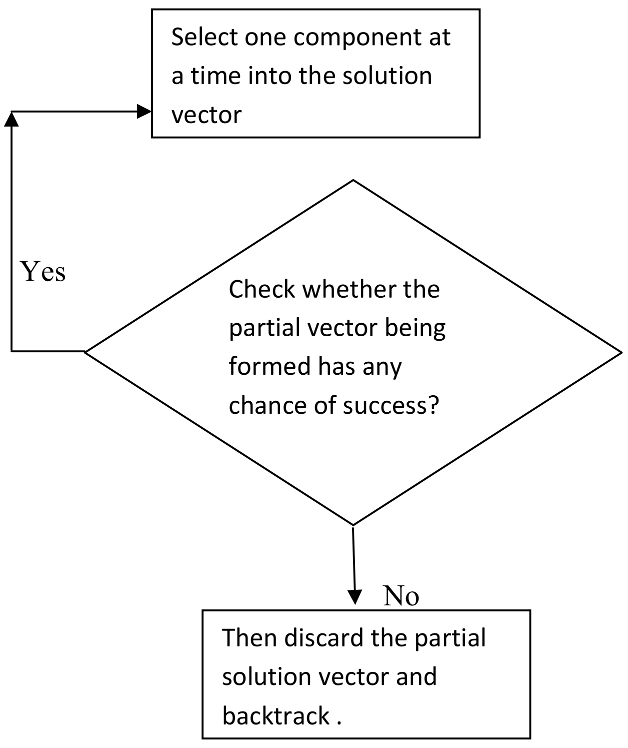
3. Graph coloring

4. Hamiltonian cycle

5. 0/1 Knapsack problem.

Principle of backtracking

Suppose mi is the size of set Si. Then there are $m=m_1m_2m_3\ldots m_n$ n-tuples that are possible for satisfying the function P. The **brute force approach** would form all these n-tuple evaluate each one with P, and save those which yield the optimum. The **backtrack algorithm** has tha ability to yield the same answer with far fewer than m trials.


Select one component at a time into the solution vector
Yes
Check whether the partial vector being formed has any chance of success?
No
Then discard the partial solution vector and backtrack .

The advantage of backtracking approach is if it realized that the partial vector $(x_1,x_2,\ldots x_i)$ cannot lead to an optimal solution then $m_{i+1}\ldots\ldots m_n$ possible test vectors can be ignored entirely.

The problem solving using backtracking approach has to satisfy a complex set of constraints. The constraints are classified into two categories: *explicit* and *implicit*

Explicit constraints are the rules that restrict the possible value of each x_i , and depend on the particular instance I of the problem. All tuples that satisfy the explicit constraints define a possible solution space for I.

Examples are

$x_i >= 0$	or $S_i = \{$ all non negative real numbers$\}$	Ex: Graph coloring
$x_i = 0$ or 1	or $S_i = \{ 0,1\}$	Ex: sum of subsets
$l_i <= x_i <= u_i$	or $S_i = \{ a: l_i <= a <= u_i \}$	Ex: 8- Queen problem

Implicit constraints are the rules that determine which of the tuples in the solution space of I satisfy the **criterion function(bounding function)**

Terminology

Backtracking algorithms determine the problem solutions by systematically searching the solution space for the given problem instance.

The tree organization of the solution space is known as **state space tree**. If the tree organizations are independent of the problem instance being solved are known as static trees. If the tree organizations are dependent of the problem instance being solved are known as dynamic trees

Problem state : Each node in the tree.

State space: All paths from the root to other nodes.

Solution states: The problem states s for which the paths from root to s define a tuple in the solution space.

Answer states: The solution states s for which the path from root to s define a tuple that is a member of the set of solutions (i.e.; it satisfies the implicit constraints) of the problem.

The problem can be solved by systematically generating **problem states**, determining which of these problem states are **solution states**, and finally determining which solution states are **answer states**.

A node which has been generated and all of whose children have not yet been generated is called a **live node**.

The live node whose children are currently being generated is called the **E-node** .

A **dead node** is a generated node which is not to be expanded further or all of whose children have been generated.

Depth first node generation with bounding functions is called **Backtracking**.

In **Branch-and-bound method** , the E-node remains E-node until it is dead .

In both methods, bounding functions are used to kill live nodes without generating all their children.

<u>General Backtrack algorithm(recursive)</u>

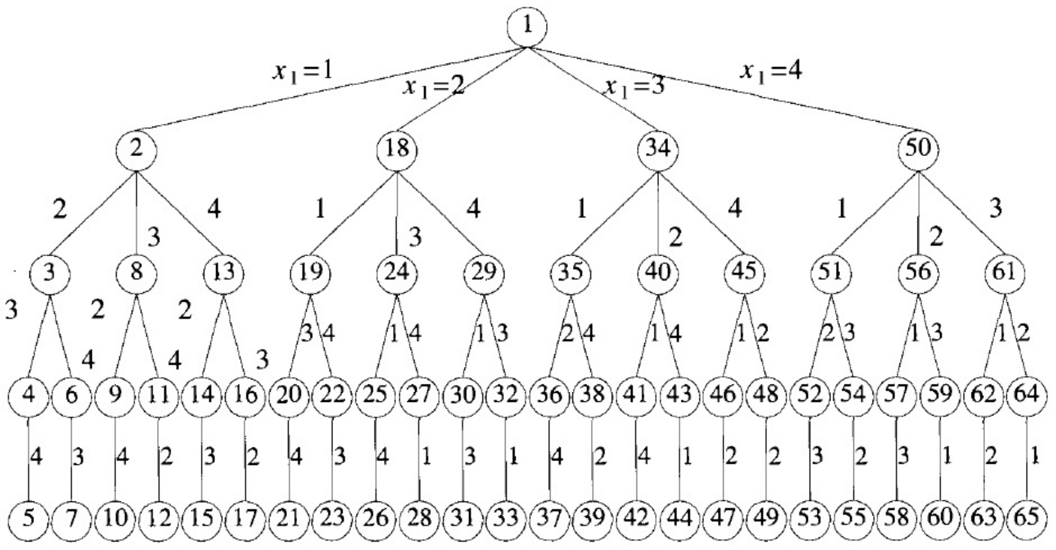
```
Algorithm Backtrack($k$)
// This schema describes the backtracking process using
// recursion. On entering, the first $k-1$ values
// $x[1], x[2], \ldots, x[k-1]$ of the solution vector
// $x[1:n]$ have been assigned. $x[\ ]$ and $n$ are global.
{
    for (each $x[k] \in T(x[1], \ldots, x[k-1])$ do
    {
        if ($B_k(x[1], x[2], \ldots, x[k]) \neq 0$) then
        {
            if ($x[1], x[2], \ldots, x[k]$ is a path to an answer node)
                then write ($x[1:k]$);
            if ($k < n$) then Backtrack($k+1$);
        }
    }
}
```

<u>General Backtrack algorithm (Iterative)</u>

```
Algorithm IBacktrack($n$)
// This schema describes the backtracking process.
// All solutions are generated in $x[1:n]$ and printed
// as soon as they are determined.
{
    $k := 1$;
    while ($k \neq 0$) do
    {
        if (there remains an untried $x[k] \in T(x[1], x[2], \ldots,$
            $x[k-1]$) and $B_k(x[1], \ldots, x[k])$ is true) then
        {
                if ($x[1], \ldots, x[k]$ is a path to an answer node)
                    then write ($x[1:k]$);
                $k := k + 1$; // Consider the next set.
        }
        else $k := k - 1$; // Backtrack to the previous set.
    }
}
```

4- Queens problem

State space tree for 4- Queen problem(Possible Solution space)



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

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

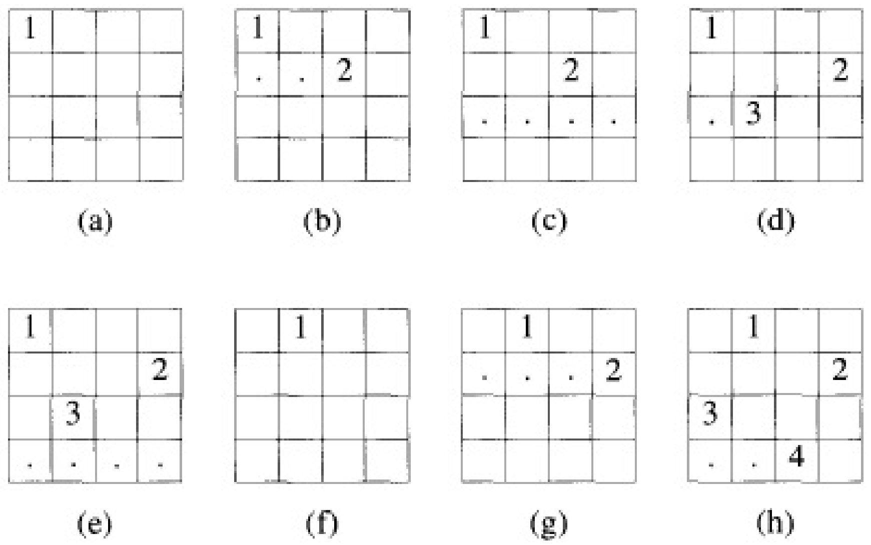
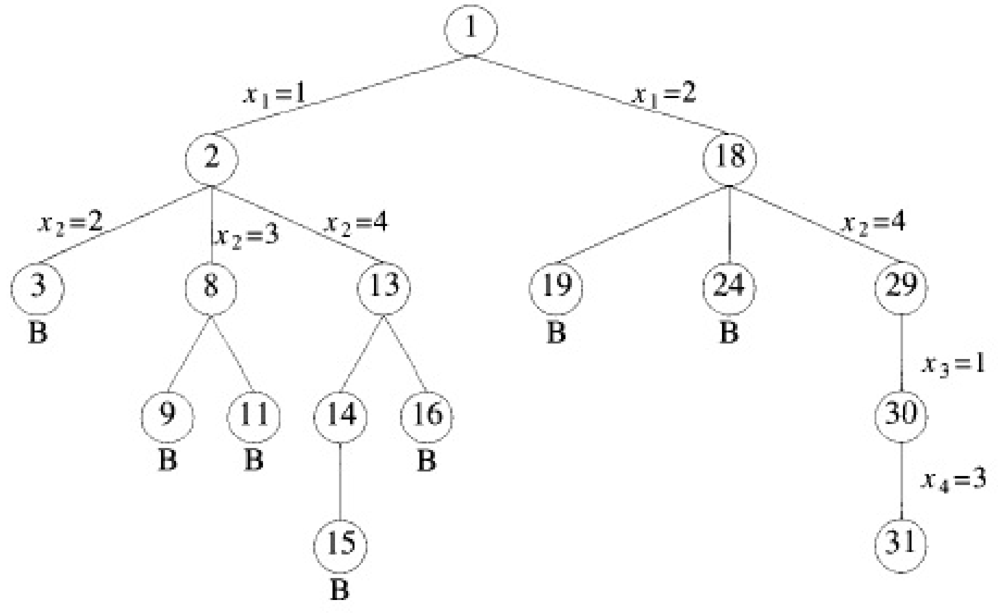


Figure 7.5 Example of a backtrack solution to the 4-queens problem



After calling Nqueens(1,4) algorithm ,the above partial state space tree is generated having only one solution. Refer the class notes for Complete state space tree having all solutions.

Sum of Subsets problem

Given n distinct positive numbers and find all combinations of these numbers whose sum is equal to m.

We can formulate the sum of subsets problem using either fixed or variable-sized tuples. If an element x_i is included in the solution vector then $x_i=1$, otherwise $x_i=0$.

The bounding function for sum of subsets problem

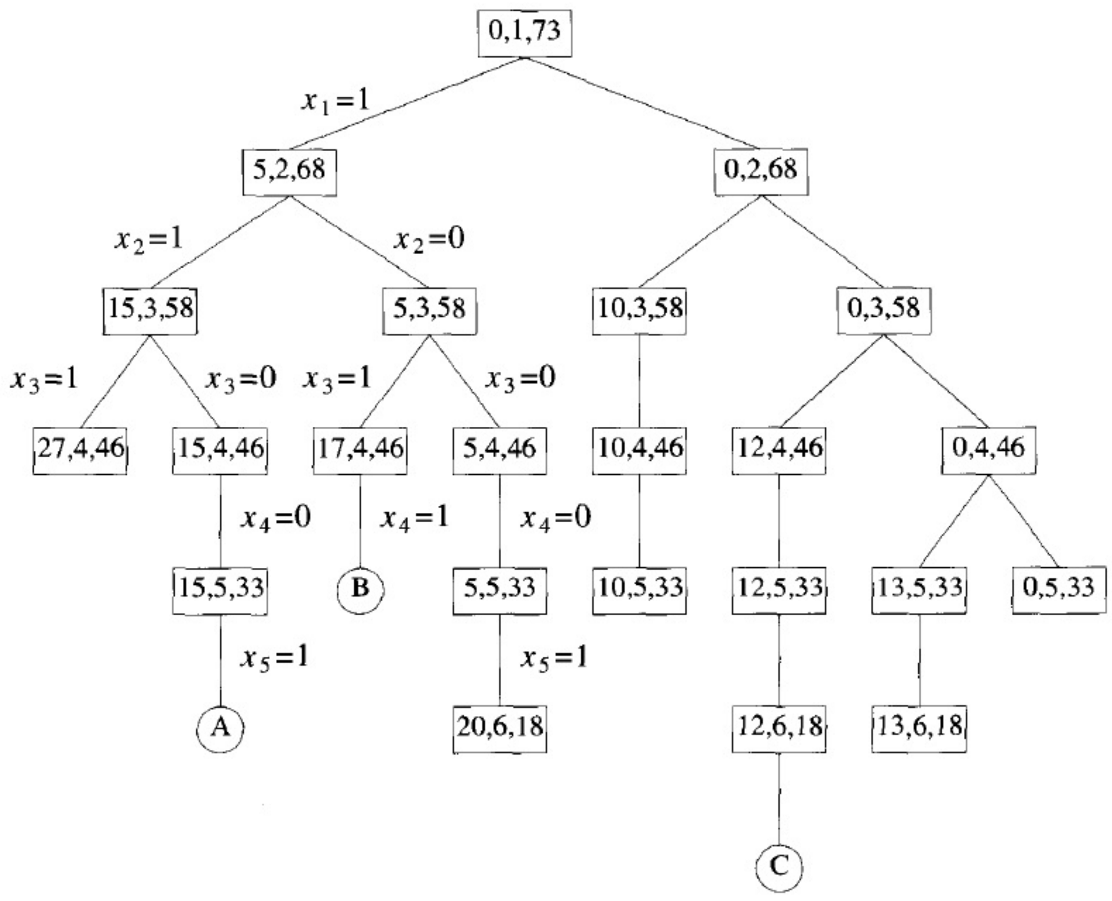
$$B_k(x_1, \ldots, x_k) = true \text{ iff } \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

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

Sum of Subsets Algorithm

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

Draw the portion of state space tree for the instance n=6 ,m=30 ,and w[1:6]= {5,10,12,13,15,18}.



Graph Coloring

Problem 1: m-colorability decision problem

Given a graph and m be a postive integer. Checking whether the graph G can be colored with atmost m colors such that no two adjacent vertices in the graph should not have same color ?

Problem 2: m-colorability optimization problem

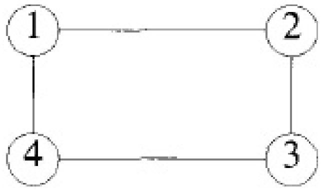
Given a graph G, find the smallest integer m for which the Graph G can be colored. The integer m is known as the chromatic number of the graph.

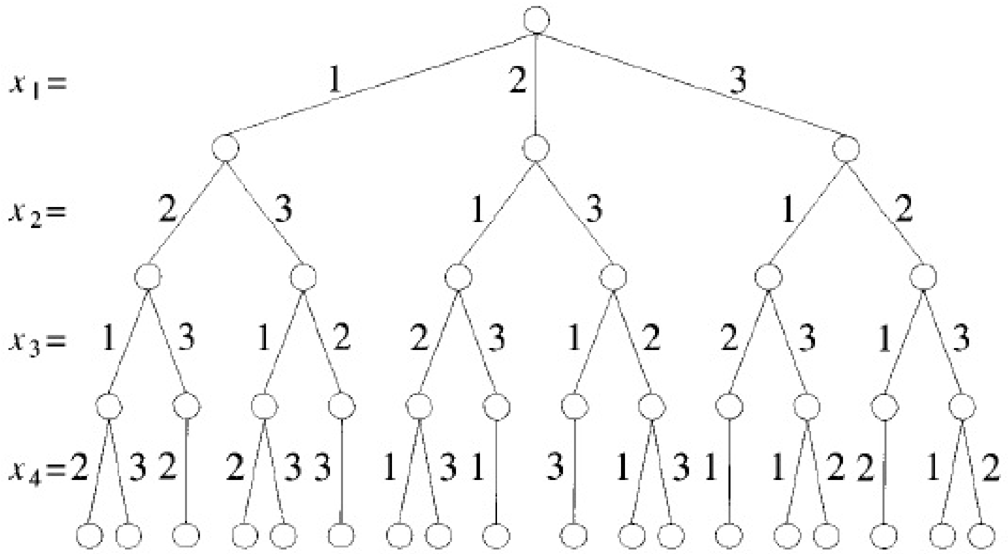
Problem 3: Find all the possible colorings for the given Graph G.

Here, we find the solution for the problem 3(Finding all possible colorings of the Graph with n vertices and m colors).

The solution is represented as n-tuple (x_1,x_2….x_n) where x_i is the color of node i .

Example: The state space tree for the following graph is





Hamiltonian cycles

Let G=(V,E) be a connected graph with n vertices . A Hamiltonian cycle is a round-trip path along with n edges of G that visits every vertex once and return to its starting position.

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

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

Refer class notes for example