Backtracking

This technique is used to solve problem which deal with searching for a set of solutions & where optimal solution satisfying some constraints can be found.

Here,

- desired solution must be expressed as an n-tuple $(x_1, x_2 --- x_n)$ where x_i is chosen from some finite set S_i
- Solution must satisfy some criterion function $P(x_1, x_2 --- x_n)$
- If m_i is the size of S_i, no. of possible candidates are $m = m_1, m_2 --- m_n$.
- Unlike Bruteforce, Backtracking generates only fewer tuples; solution is built up one component at a time and bounding function $P(x_1 --- x_i)$ is used to test whether partial vector is suitable - else discarded.

Types of Constraints

1) Explicit Constraints

↳ rules which restrict the values of x_i

Eg:- $x_i \geq 0$ (or) $l_i \leq x_i \leq u_i$

All tuples that satisfy explicit constraints define a possible solution space for Instance I.

2) Implicit Constraints.

↳ specify how x_i must relate to each other, determine which of the tuples in the solution space of I satisfy the criterion function.

Backtracking method

* Constructs a state-space tree
  - ↳ nodes: partial solutions
  - ↳ edges: choices in extending partial soln.
* Explore the state space tree using depth-first search (DFS)
* "Prune" nonpromising nodes.
  - ↳ dfs stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node's parent to continue the search.

N- Queen Problem

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

Defn

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

Solution $x = (x_1, x_2, x_3, x_4) = (2, 4, 1, 3)$

	1	2	3	4	
1		Q_1	*		← Queen 1
2	*			Q_2	← Queen 2
3	Q_3			*	← Queen 3
4		*	Q_4		← Queen 4

Explicit Constraint

$S_i = \{1, 2, 3, 4\}$ $1 \le i \le 4$

Implicit Constraint

No two x_i's can be on the same column & no two queens can be on the same diagonal.

4- Queen's problem

For n=4, There is a solution to place 4 queens in 4x4 chessboard

Step 1 start with empty chessboard

Step 2: Place queen 1 in the first possible position of its row, which is in column 1 of row 1.

	1	2	3	4
1	Q			
2				
3				
4				

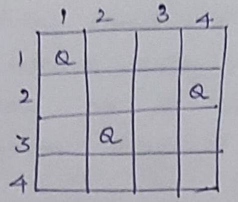
Step 3 Place queen 2, after trying unsuccessfully columns 1 and 2 in the first acceptable position for it, which is the square in row 2 & column 3

	1	2	3	4
1	Q			
2			Q	
3				
4				

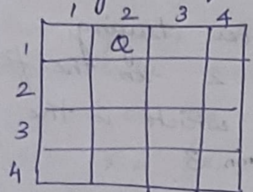
Step 4 This proves to be a dead end because there is no acceptable position for queen 3. So the algorithm backtracks and puts queen 2 in the next possible position at (2, 4)

	1	2	3	4
1	Q			
2				Q
3				
4				

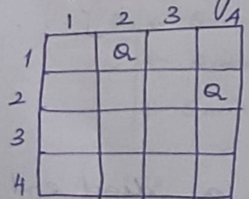
Step 5: Then queen 3 is placed at (3,2) which proves to be another dead end



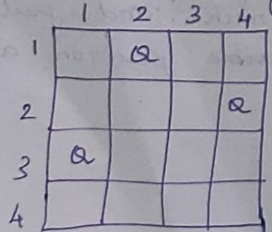
Step 6 The algorithm then backtracks all the way to queen 1 & moves to (1,2)



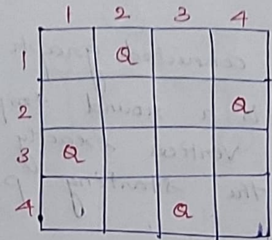
Step 7 The queen 2 goes to (2,4)



Step 8 The queen 3 goes to (3,1)

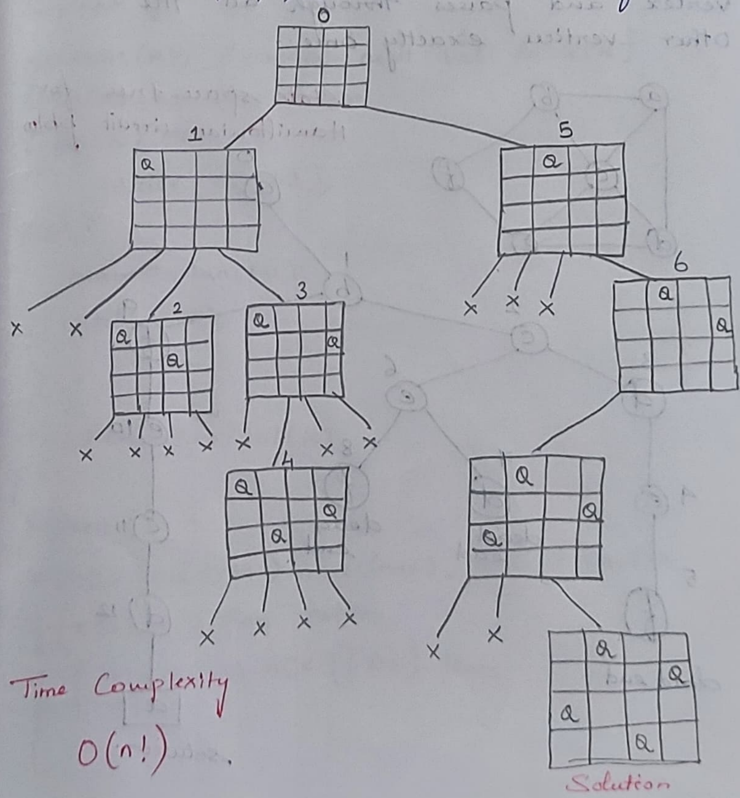


Step 9 The queen 3 goes to (4,3). This is a solution to the problem.



Solution for 4 Queens Problem

Solutions : (2,4,1,3) and (3,1,4,2) reflection



Time Complexity

O(n!)

Hamiltonian Circuit Problem

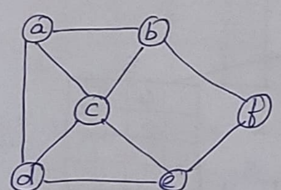
Defn

Let $G = \{V, E\}$ be a connected graph

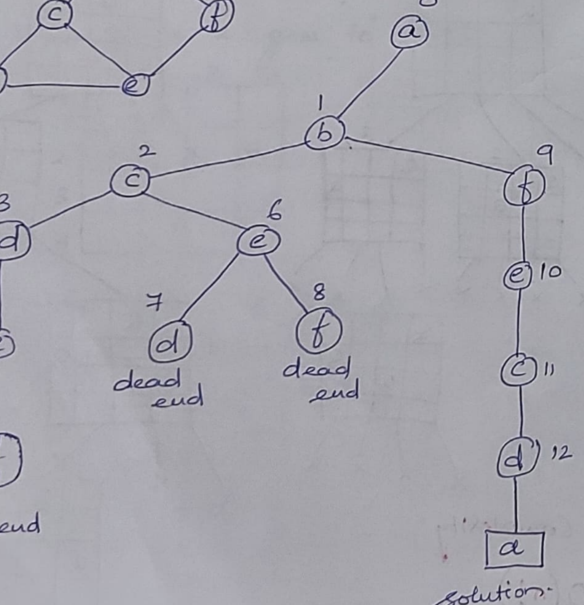
A Hamiltonian cycle is a round trip path that visits all vertices exactly once and comes back to the starting point.

(or)

Hamiltonian circuit of a graph is a path that starts and ends at the same vertex and passes through all the other vertices exactly once.

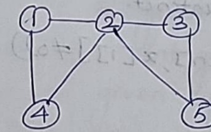


State space tree for Hamiltonian circuit pblm



To form a Hamiltonian circuit / cycle the graph should follow a closed loop.

Eg:



→ No cycle Hamiltonian circuit not possible

Algorithm

```
Algorithm Hamiltonian (k)
{
  Repeat {
    Nextval (k); // assign next val to x[k].
    if (x[k] = 0) then return.
    if (k = n) then
        write (x[1:n]);
    else
        Hamiltonian (k+1);
  } until (false);
}.
```

```
Algorithm Nextval (k)
{
  Repeat {
    x[k] = (x[k]+1) mod (n+1); // Next vertex.
    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 vertex is distinct.
if ((K<n) or (K=n) + G[x[n], x[1]] ≠ 0))
then return;
}}
until (false);}
```

Subset sum problem

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

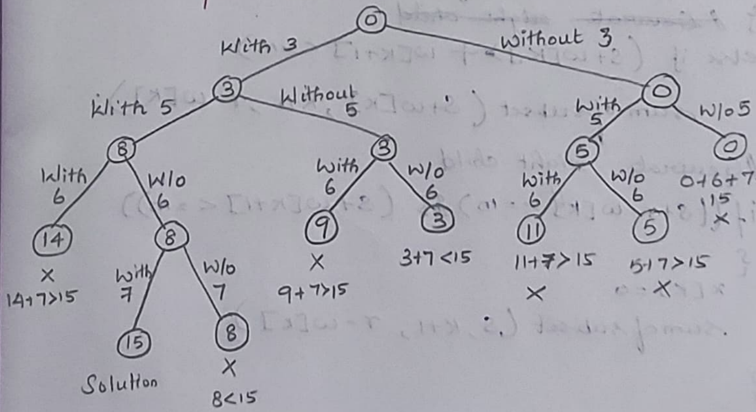
Example $A = \{1,2,5,6,8\}$ and $d=9$

solutions: $\{1,2,6\}$ and $\{1,8\}$

- We record the value of s the sum of the first 'i' numbers in the node.
- if s is equal to d, we have a solution to the problem.
- We can either report this result + stop or if all the solutions need to be found, continue by backtracking to the node's parent.

Example $A = \{3,5,6,7\}$ $d=15$ soln = $\{3,5,7\}$

State space tree (DFS)



In the above state space tree, for a node at level i, the left child corresponds to $x_i = 1$ & the right to $x_i = 0$ such that,

$$\sum_{i=1}^{K} w_i x_i + w_{K+1} \le m \qquad [m = \text{total sum}]$$

Algorithm

```
void sumofsubset(float s, int K, float r)
{
 // Generate left child.
 x[K] = 1
 if (s+w[K] == m)
 {
   // subset found
   for (j=1 ; j<=K ; j++)
        cout << x[j]
 }
 else if (s+w[K] + w[K+1] <= m)
        sumofsubset(s+w[K], K+1, r-w[K]);
 // Generate right child.
 if ((s+r-w[K] >= m) && (s+w[K+1] <= m))
 {
   x[K] = 0
   sumofsubset(s, K+1, r-w[K]);

 }
}
```