Design & Analysis of Algorithms

(Prof. Parminder Kaur Wadhwa)

Date _____

Page _____

Backtracking :-

- General Backtracking Algorithm
- 8-Queen's problem
- Sum of subsets
- Graph coloring
- Hamiltonian cycles
- Knapsack Problem

} Solve these problems using backtracking technique.

Brute Force Approach V/S Backtracking :-

In many practical applications, the desired solution is expressed as a n-tuple.

$(x_1, x_2, \ldots, x_n)$

where x_i are chosen from some finite set S_i.

Mostly, the problems to be solved call for finding one vector that maximizes or minimizes or satisfies a CRITERION FUNCTION $P(x_1, \ldots, x_n)$.

e.g. Sorting the array of integers in $a[1:n]$ is a problem whose solution is expressible by a n-tuple; where x_i is the index in array a of the ith smallest element.

The criterion function P for this is $a[x_i] \leq a[x_{i+1}]$ for $1 \leq i \leq n$. The set S_i is finite and includes the integers

1 through n.

Suppose m_i is the size of set S_i.

Then there are $m = m_1 m_2 \ldots m_n$ n-tuples that are possible candidates for satisfying the function P.

The brute force approach would be to form all these n-tuples and evaluate each one with P, saving those which yield the optimum.

It means, in brute force approach all possible solution states are generated. But those which satisfy criterion function are then saved. So, brute force is not an efficient method as it is generating the tuples (or solution states) which will be discarded as if they don't satisfy the criterion function. It is a time consuming method.

There is a better method than brute-force approach. It is called as BACKTRACKING.

The backtrack algorithm has its virtue the ability to yield the same answer with far fewer than m trials.

The basic idea of the Backtracking algorithm is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, \ldots, x_i)$ (sometimes called as BOUNDING FUNCTIONS) to test whether the vector being formed has any chance of success.

The major advantage of this method is this:-

if it realized that the partial vector $(x_1, x_2, \ldots, x_i)$ can in no way lead to an optimal solution, then $m_{i+1} \cdots m_n$ possible test vectors can be ignored entirely.

Many of the problems that are solved using backtracking require that all the solutions satisfy a complex set of constraints For any problem, these ~~prob~~ constraints can be divided into two categories:-

explicit & implicit.

Explicit constraints :-

Explicit constraints are rules that restrict each x_i to take on values only from a given set.

e.g. Explicit constraints :-

a) S_i = { all non negative real numbers }

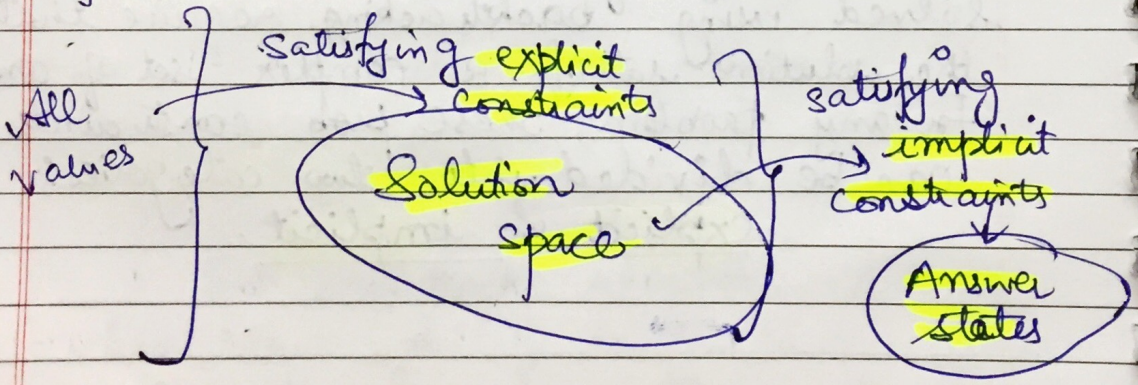
or S_i is $x_i \geq 0$

b) $S_i = \{0, 1\}$ i.e. $x_i = 0$ or 1

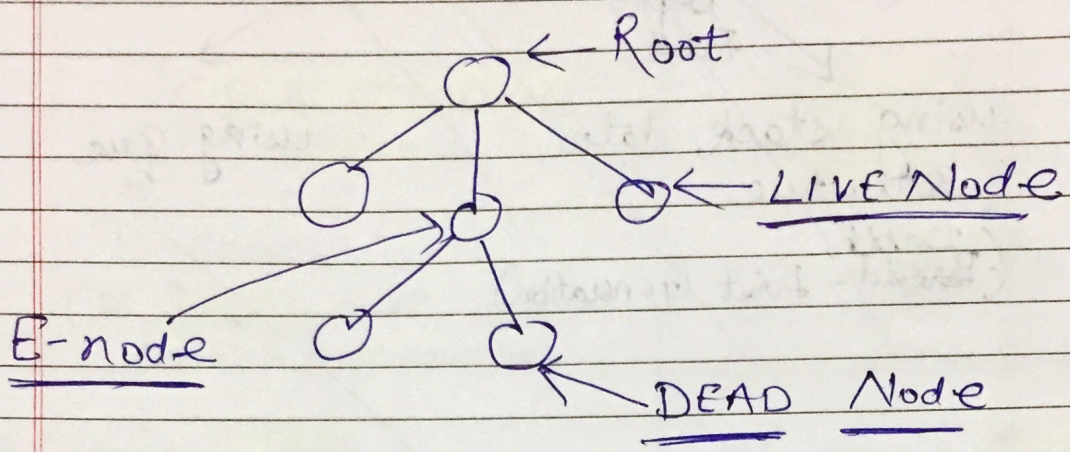
All tuples that satisfy the explicit constraints define a possible SOLUTION SPACE

Implicit Constraints :-

The implicit constraints are rules that determine which of the tuples in the solution space satisfy the criterion function.



Tree Organization of Solution Space



Terms :-

Root node is the initial state.

Any node is a problem-state. (i.e. current state of the problem)

Live node :- A node which has been generated and all whose children have not yet been generated is called as a live node.

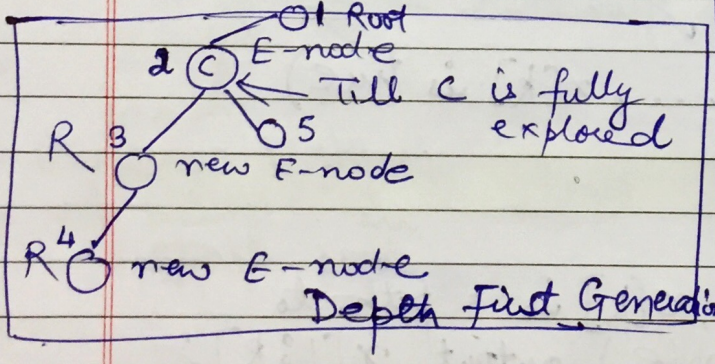
E-node :- The live node whose children are currently being generated is called as E-node (node being expanded).

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

Tree Generation Methods

(BACKTRACKING) METHOD

- D-search (or Depth First search) generation)
- (using STACK data structure)
- In this method, as soon as a new child C of the current E-node R is generated, this child will become the new E-node.



Then R will become the E-node again when the subtree C has been fully explored

(BRANCH & BOUND) METHOD

- Breadth First Generation
- (using Queue as data structure)
- In this method, E-node remains as the E-node until it is dead.

Each node is placed into a queue when all the children of the current E-node have been generated, the next node at the front of the queue becomes the new E-node.

Algorithm Representing the General Backtracking method

```
Void IterativeBacktrack (int n)
// In this algorithm, all solutions are
// generated in x[1:n] and printed as
// soon as they are determined.
{
    int k=1;                    //initialized
    while (k) {
        if (there remains an untried x[k]
        such that x[k] is in
        T(x[1], x[2], ..., x[k-1]) and
        Bk(x[1], ..., x[k] is true)
        {
            if (x[1], .., x[k] is a path to
            an answer node) output x[1:k];
            k++;    // consider the next set.
        }
        else k--; // Backtrack to the previous set.
    }
}
```

A solution state with specific values of components of solution vector

a component of solution vector

Bounding function is satisfied

means implicit constraints are fulfilled for this state

display as output if gives the answer-state.

Summarizing :- General Backtracking Algo.

```
void IterativeBacktrack (int n)
{
    int k = 1;
    while (k) {
    if (there remains an untried x[k] such
    that x[k] is in T(x[1], x[2], ..., x[k-1])
    and Bk (x[1], ..., x[k]) is true)
    { if (x[1], ..., x[k] is a path to
    an answer node) output x[1:k];
        k++; // consider the next set.
    }
    else k--; // Backtrack to the
                previous set.
    }
}
```

Explanation of above algorithm :-

T( ) will yield the set of all possible values that can be placed as the first component x_1 of the solution vector. The component x_1 will take on those value for which the Bounding function $B_1(x_1)$ is true.

The elements are generated in a depth first manner (using stack data structure).

The variable k is continually incremented and a solution vector is grown until either a solution is found or no untried value of x_k remains.

When k is decremented, the algorithm must resume the generation of possible elements for the kth position that have not yet been tried.

Therefore, one must develop a procedure that generates these values in some order.

If only one solution is desired, replacing output x[1:k]; with :-

suffices as this

```
output x[1:k];
return;
```

Recursive Backtracking Algorithm

The following recursive version is initially invoked by:-

```
Backtrack(1);
```

initial value of k

```
Void Backtrack(int k)
{
    for (each x[k] such that
         x[k] ∈ T(x[1], ..., x[k-1]))
    {
        if (B_k(x[1], x[2], ..., x[k]))
        {
            if (x[1], x[2], ..., x[k] is
                a path to an answer node)
                output x[1:k];

            if (k < n) Backtrack(k+1);
        }
    }
}
```

Include in answer if Bounding function is satisfied

if all values of k are still not checked then make a recursive call

for next value of k

Explanation of the above Recursive Backtracking Algorithm

The solution vector $(x_1, \ldots, x_n)$ is treated as a global array $x[1:n]$.

All the possible elements for the kth position of the tuple that satisfy B_k are generated, one by one, and adjoined to the current vector $(x_1, x_2, \ldots, x_{k-1})$.

Each time x_k is attached, a check is made to determine whether a solution has been found.

Then the algorithm is recursively invoked.

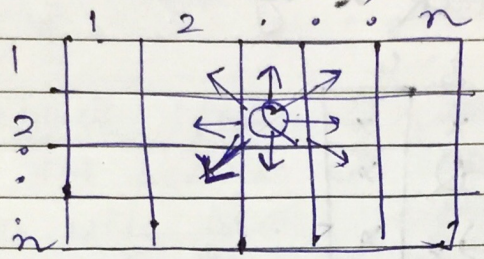
When the outer for loop is exited, no more values for x_k exist and the current copy of "Backtrack" ends, and the last unresolved call now resumes.

---

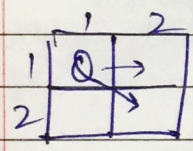
n – Queen's Problem

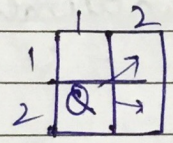
The problem is to place n queens on a n x n chessboard, so that no two "attack";

i.e. so that no two of them are on the same row, column, or diagonal

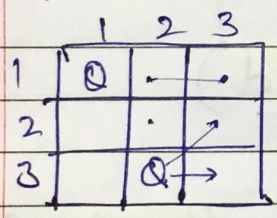


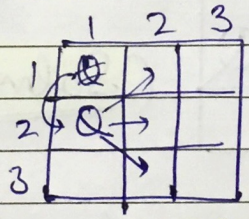
1-queen problem is SOLVABLE

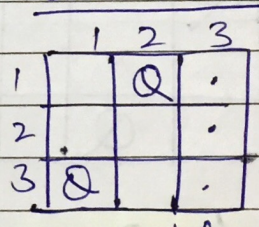




2-queen problem is unsolvable





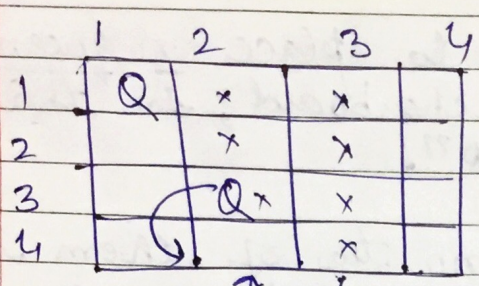


3-Queen problem is unsolvable

Implicit constraints:–

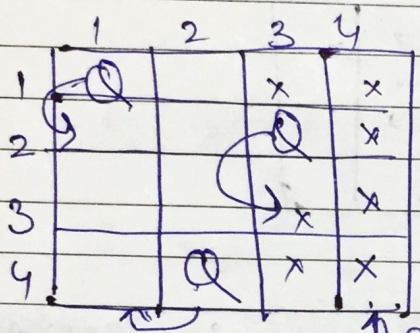
no two queens can be in the same row or column or diagonal.

4-Queen Problem

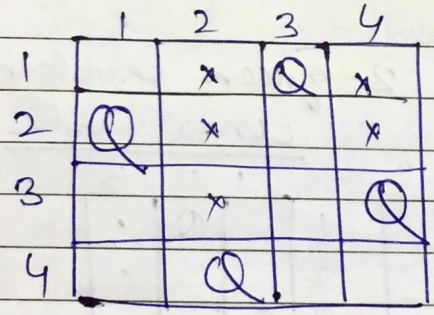


used bounding function to kill further expansion

Backtrack & try another alternative



Backtrack & try another alternative



(Solved)

4-Queens problem is SOLVABLE

When we use backtracking :— when we check that the criterion function is being violated (by using BOUNDING FUNCTION) then we kill the node & further it will not be solved. We will backtrack to the state till

where the implicit constraints were satisfied and we try another alternative to find answer state.

For a 8-queens problem :-

8 x 8 chessboard is there. And we need to place 8 queens satisfying the implicit constraints.

The explicit constraints (using the formulation that the solution to the 8-queen problem will be as 8-tuples $(x_1, \ldots, x_8)$ where x_i is the column on which queen i is placed) :-

Explicit constraints are

$$S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

$$1 \leq i \leq 8$$

i.e. 8-queens to be placed

∴ Solution Space consists of

8^8 8-tuples.

When implicit constraints are checked this reduces the solution space from 8^8 tuples to $8!$ tuples.

For a n-queen problem :-

The possible solutions are $n!$

(e.g. for a 4-queen problem, the possible solution are

$4!$ i.e. $4 \times 3 \times 2 \times 1$

$= 24$ (leaf nodes)

$\Rightarrow$ solution states)

* Solution space for a n-queen problem is $n!$

* (worst case for n-queen problem) is $O(q(n)\, n!)$

$q(n)$ is polynomial in n.

Backtracking Technique to solve the 8-Queens Problem or any n-Queens Problem

Consider an n x n chessboard, we need to find all ways to place n non-attacking queens.

If we imagine chessboard's squares being numbered as the indices of the two dimensional array a[1:n][1:n]; then :—

the two queens lie on the same diagonal if and only if :—

$$|j-l| = |i-k|$$

Suppose two queens are placed at positions (i, j) and (k, l)

(this is to check whether the queens are attacking diagonally or not. If this condition is true, it means they are attacking each other diagonally.).

The algorithm to solve N-Queens problem call another algorithm namely "Place" to check all the implicit constraints. The algorithm 'Place' will return false if the constraints are violated and the queen can not be placed at that position. The algorithm 'Place' will return true if the constraints are fulfilled and the queen can be placed at that position.

Algorithm for n-Queens Problem using Backtracking Algorithm

```
void NQueens (int k, int n)
// for a nxn chessboard
{
  for (int i=1; i<=n; i++)
  {
    if (Place (k, i))
    {
      x[k] = i;
      if (k==n)
      {
        for (int j=1; j<=n; j++)
          cout << x[j] << ' ';
        cout << endl;
      }
      else NQueens (k+1, n);
    }
  }
}
```

- k : queen to be placed
- n : total Queens
- initially k=1
- // for all queens
- // call Place(k,i)
- // if true, then place kth queen in column i
- // if all queens are placed?
- display result if all queens are placed
- Otherwise make a recursive call for next queen
- a recursive call

Place (int k, int i)

```
bool Place (int k, int i)
// returns a boolean value as "true" if the
// Kth queen can be placed in column i.
{
    for (int j=1 ; j<k ; j++)
        if ((x[j] == i) || (abs (x[j]-i)
                == abs(j-k)))
        {
            return(false);
        }
    else
        { return (true); }
}
```

check constraints for all already placed queens

queens are in same column

previous column

current column

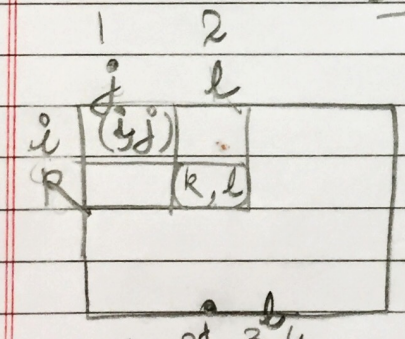
previous row

current row

queens are in the same diagonal

$|j-l| = |i-k|$

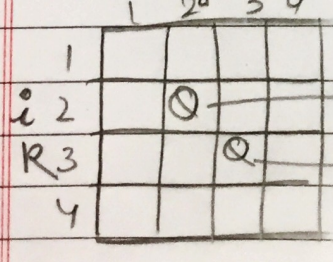
Example:-



$|1-2| = |1-2|$

$|-1| = |-1|$

$1 = 1$



i j

(2,2)

R l

(3,3)

$|j-l| = |i-k|$

$|2-3| = |2-3|$

$|-1| = |-1|$

$1 = 1$