Design and Analysis of Algorithms

Mohammad GANJTABESH

mgtabesh@ut.ac.ir

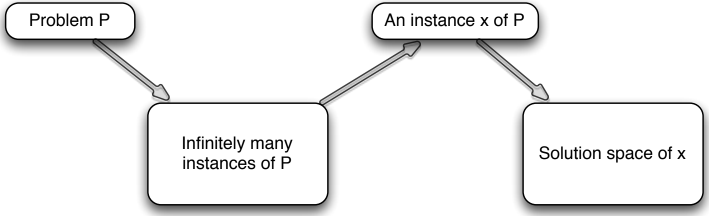
School of Mathematics, Statistics and Computer Science,
University of Tehran,
Tehran, Iran.

Techniques for the design of Algorithms

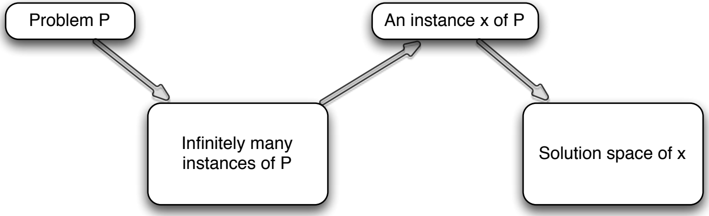
The classical techniques are as follows:

1. Divide and Conquer
2. Dynamic Programming
3. Greedy Algorithms
4. Backtracking Algorithms
5. Branch and Bound Algorithms

Backtracking Algorithms



Backtracking Algorithms



- Backtracking is a systematic way to search the configuration of solution space.
- Each possible configuration must be generated exactly once.

Backtracking Algorithms

- In general, we assume our solution is a vector $v = (a_1, a_2, \cdots, a_n)$.

Backtracking Algorithms

- In general, we assume our solution is a vector $v = (a_1, a_2, \cdots, a_n)$.
- At each step, we try to extend a partial solution $a = (a_1, a_2, \cdots, a_k)$ by adding another element at the end.

Backtracking Algorithms

- In general, we assume our solution is a vector $v = (a_1, a_2, \cdots, a_n)$.
- At each step, we try to extend a partial solution $a = (a_1, a_2, \cdots, a_k)$ by adding another element at the end.
- Then we test whether what we now have is a solution: if so, we should print it or count it.

Backtracking Algorithms

- In general, we assume our solution is a vector $v = (a_1, a_2, \cdots, a_n)$.
- At each step, we try to extend a partial solution $a = (a_1, a_2, \cdots, a_k)$ by adding another element at the end.
- Then we test whether what we now have is a solution: if so, we should print it or count it.
- If not, we check whether the partial solution is still potentially extendible to some complete solution.

Backtracking Algorithms

- In general, we assume our solution is a vector $v = (a_1, a_2, \cdots, a_n)$.
- At each step, we try to extend a partial solution $a = (a_1, a_2, \cdots, a_k)$ by adding another element at the end.
- Then we test whether what we now have is a solution: if so, we should print it or count it.
- If not, we check whether the partial solution is still potentially extendible to some complete solution.
- Backtracking algorithm is modeled by a tree of partial solutions, where each note represents a partial solution.

Backtracking Algorithms

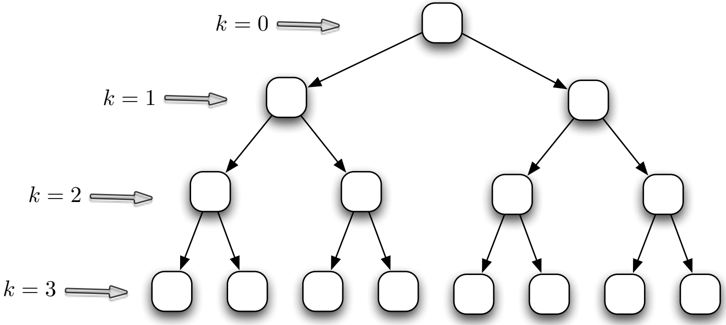
Backtrack$(A, k)\{$
    if $A = (a_1, a_2, \cdots, a_k)$ is a solution, then report it;
    else$\{$
        $k \leftarrow k + 1$;
        compute S_k;
        while$(S_k \neq \emptyset)\{$
            $a_k \leftarrow$ an element of S_k;
            $S_k \leftarrow S_k - \{a_k\}$;
            Backtrack(A, k);
        $\}$
    $\}$
$\}$

Backtracking Algorithms

```
Backtrack(A, k){
    if A = (a_1, a_2, ..., a_k) is a solution, then report it;
    else{
        k ← k + 1;
        compute S_k;
        while(S_k ≠ ∅){
            a_k ← an element of S_k;
            S_k ← S_k − {a_k};
            Backtrack(A, k);
        }
    }
}
```

- Backtracking ensures correctness by checking all possibilities.
- It ensures efficiency by never visiting a configuration more than once.

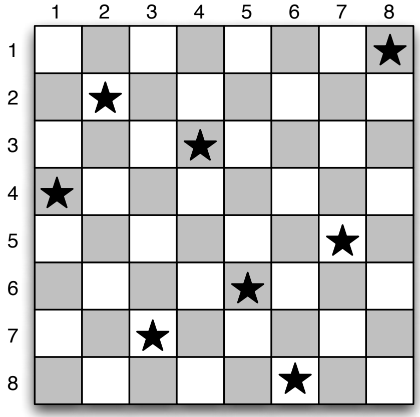
Backtracking Algorithms



n—Queens Problem

Definition

The problem is to locate n queens on an $n \times n$ chess board.



$n-$Queens Problem

We can use different approaches:

- Search all the solution space of size $\binom{n^2}{n}$.

$n-$Queens Problem

We can use different approaches:

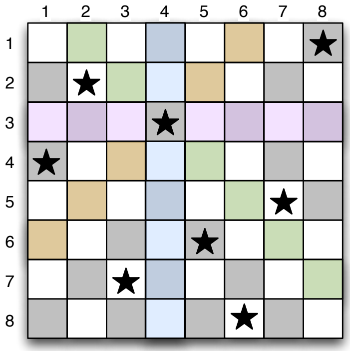
- Search all the solution space of size $\binom{n^2}{n}$.
- Using eight loops, each is inside the other, which implies the size of n^n for the solution space.

$n-$Queens Problem

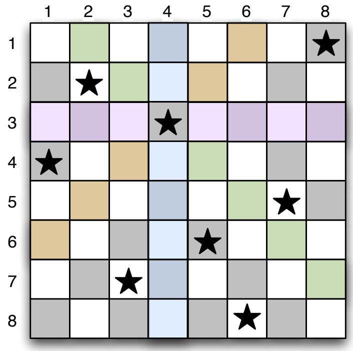
We can use different approaches:

- Search all the solution space of size $\binom{n^2}{n}$.
- Using eight loops, each is inside the other, which implies the size of n^n for the solution space.
- Using 1-dimensional array in order to remove more conflicts and reducing the search space.

$n-$Queens Problem: Row and Column conflicts



$n-$Queens Problem: Row and Column conflicts



- **Row:** Queen $Q[i,k]$ conflicts with Queen $Q[j,l] \iff i = j$.
- **Column:** Queen $Q[i,k]$ conflicts with Queen $Q[j,l] \iff k = l$.

$n-$Queens Problem: Diagonal conflicts

Diagonal

	1	2	3	4	5	6	7	8
1	0	-1	-2	-3	-4	-5	-6	-7
2	1	0	-1	-2	-3	-4	-5	-6
3	2	1	0	-1	-2	-3	-4	-5
4	3	2	1	0	-1	-2	-3	-4
5	4	3	2	1	0	-1	-2	-3
6	5	4	3	2	1	0	-1	-2
7	6	5	4	3	2	1	0	-1
8	7	6	5	4	3	2	1	0

$Q[i,k]$ conflicts with $Q[j,l]$

$$\Updownarrow$$

$$i-k=j-l$$

Back-Diagonal

	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10
3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12
5	6	7	8	9	10	11	12	13
6	7	8	9	10	11	12	13	14
7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16

$Q[i,k]$ conflicts with $Q[j,l]$

$$\Updownarrow$$

$$i+k=j+l$$

$n-$Queens Problem: Diagonal conflicts

Diagonal

	1	2	3	4	5	6	7	8
1	0	-1	-2	-3	-4	-5	-6	-7
2	1	0	-1	-2	-3	-4	-5	-6
3	2	1	0	-1	-2	-3	-4	-5
4	3	2	1	0	-1	-2	-3	-4
5	4	3	2	1	0	-1	-2	-3
6	5	4	3	2	1	0	-1	-2
7	6	5	4	3	2	1	0	-1
8	7	6	5	4	3	2	1	0

$Q[i,k]$ conflicts with $Q[j,l]$

$$\Updownarrow$$

$$i-k=j-l$$

Back-Diagonal

	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10
3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12
5	6	7	8	9	10	11	12	13
6	7	8	9	10	11	12	13	14
7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16

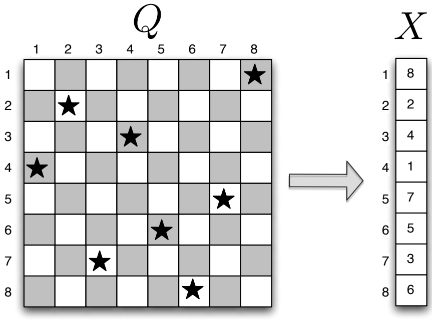
$Q[i,k]$ conflicts with $Q[j,l]$

$$\Updownarrow$$

$$i+k=j+l$$

Queen $Q[i,k]$ conflicts with Queen $Q[j,l]$ $\iff$ $|i-j|=|k-l|$

n—Queens Problem: One-Dimensional representation



Suppose that $X[i] = k$ and $X[j] = l$. Queen i conflicts with Queen j if and only if:

- $X[i] = X[j]$, or
- $|i - j| = |X[i] - X[j]|$.

$n-$Queens Problem: Algorithm

The following procedure decides whether the j-th Queen is correctly placed with respect to the previous queens.

CanPlace$(X, j)\{$

  for $i \leftarrow 1$ to $j-1$ do$\{$

    if$(X[i] = X[j]$ or $|X[i] - X[j]| = |i - j|)$ then

      return false;

  $\}$

  return true;

$\}$

$n-$Queens Problem: Recursive Backtrack Algorithm

```
n − Queen1(X, i){
    if( CanPlace(X, i)){
        if (i = n) then report(X);
        else{
            for k ← 1 to n do{
                X[i+1] ← k;
                n − Queen(X, i+1);
            }
        }
    }
}
```

$n-$Queens Problem: Iterative Backtrack Algorithm

```
n − Queen2(X, n){
    X[1] ← 0;
    k ← 1;
    while (k > 0){
        X[k] ← X[k] + 1;
        while (X[k] ≤ n and CanPlace(X, k) = false )
            X[k] ← X[k] + 1;
        if(X[k] ≤ n){
            if(k = n) then report(X);
            else{
                k ← k + 1;
                X[k] ← 0;
            }
        }
        else{
            k ← k − 1;
        }
    }
}
```

Exercises

1. Suppose that $S = \{1, 2, \cdots, n\}$. Write a backtracking algorithm to generate all permutations of S.
2. Suppose that $S = \{1, 2, \cdots, n\}$. Write a backtracking algorithm to generate all k-subsets of S.
3. (Set Cover Problem) Suppose that $S = \{1, 2, \cdots, n\}$ and $C \subseteq Powerset(S)$ is a collection of subsets of S. Write a backtracking algorithm to find a $C' \subseteq C$ such that:

$$\bigcup_{c \in C'} c = S$$

, where $|C'|$ is minimum.

4. Devise a backtracking algorithm to solve the SUDOKU puzzle from an initial state.
5. A derangement is a permutation p of $\{1, 2, \cdots, n\}$ such that no item is in its proper position, i.e. $p_i \neq i$ for all $1 \leq i \leq n$. Write a backtracking program that constructs all the derangements of n items.
6. For a given number n, write a backtracking algorithm to generate all it partitions, i.e.

$$\begin{aligned}
4 &= 1+1+1+1, \\
4 &= 2+1+1, \\
4 &= 2+2, \\
4 &= 3+1, \\
4 &= 4.
\end{aligned}$$