ALGORITHMS AND DATA STRUCTURES I

Course 12

Backtracking

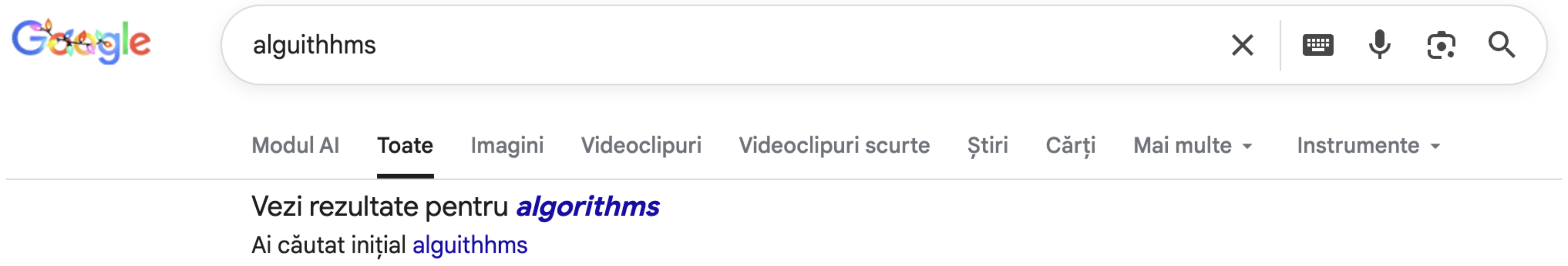
Previous Course

- What is dynamic programming?
- Main steps in applying dynamic programming
- Recurrence relationships: top-down vs. bottom-up development
- Applications
- Memoization technique

Today Course

- What is the backtracking technique ?
- Application examples:
    1. Generation of permutations
    2. The problem of placing queens on the chessboard
    3. Coloring the maps
    4. Finding routes that connect two locations
    5. The maze problem

Motivation


alguithhms

Modul AI · Toate · Imagini · Videoclipuri · Videoclipuri scurte · Știri · Cărți · Mai multe · Instrumente

Vezi rezultate pentru ***algorithms***

Ai căutat inițial alguithhms

How do you decide that I intended to write algorithms instead of alguithhms?

A measure of dissimilarity between words is evaluated

What is the backtracking technique ?

- It is a systematic search strategy in the solution space of a problem
- It is used especially for solving problems whose requirement is to determine configurations that satisfy certain restrictions (constraint satisfaction problems).
- Most of the problems that can be solved using the backtracking technique fall under following template :

  " To find a subset S of the Cartesian product A_1 x A_2 x ... x A_n (A_k – finite sets) having the property that each element (s_1, s_2 ,...,s_n) of S satisfies certain restrictions "

- Example: generating all permutations of the array {1,2,...,n} A_k = {1,2,...,n} for each k=1..n and s_i != s_j for any i != j (restriction : distinct components)

What is the backtracking technique ?

The basic idea :

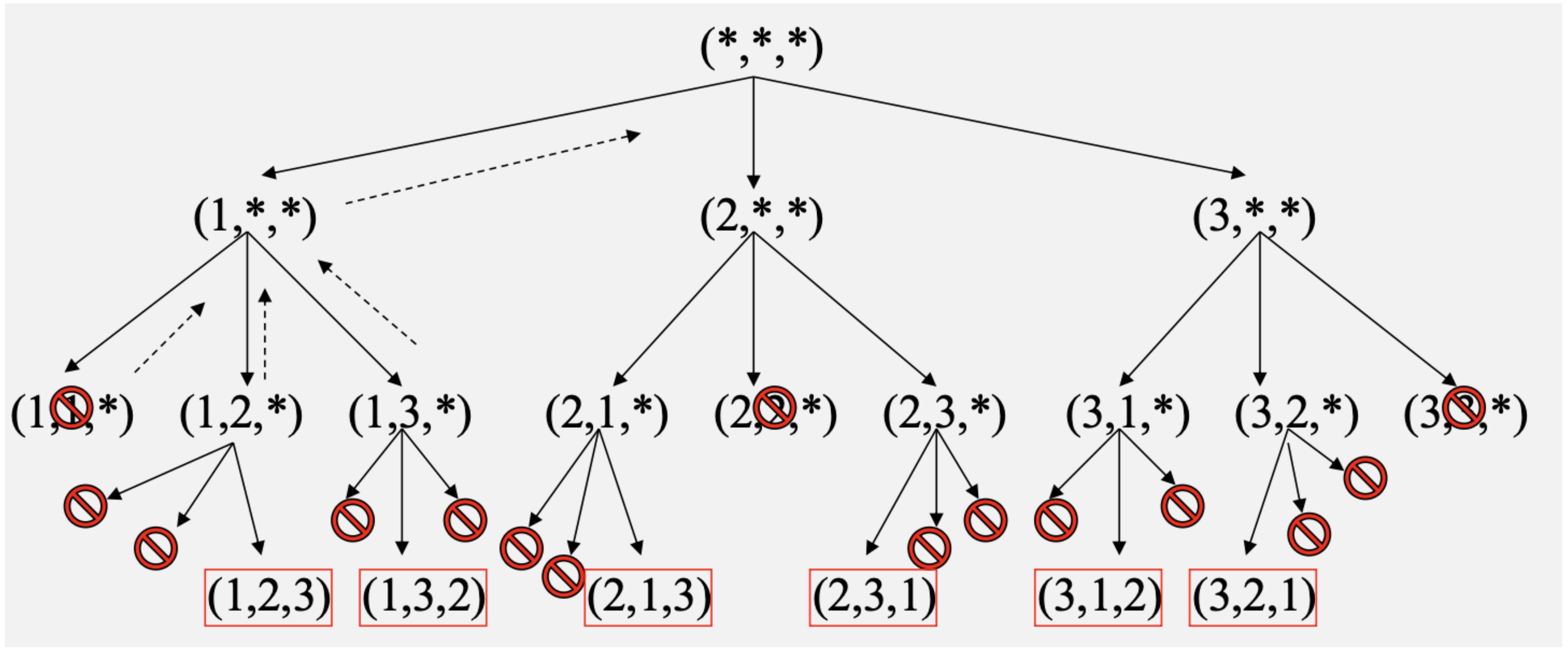
- The solutions are built incrementally by successively finding suitable values for the components of a potential solution (at each step one component is completed; a solution in which only part of the components are completed is called partial solution)

- Each partial solution is evaluated to determine if it is valid (promising). If a partial solution violates partial constraints, it is deemed invalid, and will never lead to a solution that satisfies all constraints of the problem

- If none of the values corresponding to a component leads to a valid partial solution, then it returns to the previous component and another value is tried for it .

What is the return search technique ?

- Example: the traversal tree of the solution space in the case of the permutation generation problem (n=3)



- Remark : visiting nodes is similar to tree-depth traversal

General structure of the algorithm

Steps in designing the algorithm:

1. Representation of the solutions is chosen
2. The search space is identified and the sets A_1 ,...,A_n are determined, plus the order in which they are completed
3. The conditions that the partial solutions must satisfy in order to be valid are deduced, as restrictions. These conditions are called continuation conditions
4. The criterion is established on the basis of which it is decided whether a partial solution is considered a final solution

General structure of the algorithm

Steps in designing the algorithm:

1. Representation of the solutions is chosen: each permutation is a vector $s=(s_1,s_2,\dots s_n)$ that respects s_i != s_j for any i != j

2. The search space is identified and the sets A_1 ,…,A_n : {1, …, n}, Each set is iterated in the natural order of the elements (from 1 to n)

3. Continuation conditions : a partial solution$(s_1,s_2,\dots,s_k)$ satisfy s_k != s_i for any $i<k$

4. Criterion for deciding whether a partial solution is final : $k=n$ (all components were set)

General structure of the algorithm

Notations:

- $(s_1,s_2,\ldots,s_k)$ partial solution
- k – index in s
- $A_k = \{a^k{}_1,\ldots,a^k{}_{mk}\}$
- m_k=card(A_k)
- i_k- index in A_k

General structure of the algorithm

The recursive variant :

- We assume that $A_1, \dots, A_n$ and s are global variables
- Let k be the component to be completed

The algorithm is called by BT_rec(1)

```
BT_rec(k)
 IF "($s_1$ ,..., $s_{k-1}$ ) is a final solution "
 THEN " process final solution "
 ELSE
   FOR j $\leftarrow$ 1,$m_k$ DO
      $s_k \leftarrow a^k_j$
      IF "($s_1$, ..., $s_k$) is valid "
      THEN BT_rec(k+1) ENDIF
   ENDFOR
 ENDIF
```

Try all possible values

Complete next component

Application 1 : generating permutations

Function that checks if a partial solution is valid

```
valid(s[1..k])
 FOR i ← 1,k-1 DO
  IF s[k]=s[i]
  THEN RETURN FALSE
  ENDIF
 ENDFOR
 RETURN TRUE
```

Recursive variant:

```
perm_rec(k)
 IF k=n+1 THEN WRITE s[1..n]
 ELSE
  FOR i ← 1,n DO
   s[k] ← i
   IF valid(s[1..k])=True
   THEN perm_rec(k+1)
   ENDIF
  ENDFOR
 ENDIF
```

General structure of the algorithm

The iterative variant

Looking for a value for the component k which leads to a valid partial solution

If such a value exists then it is checked if we reached a final solution

If a solution is found, then it can be processed, and the next value of the same component is computed next

If it is not a final solution, move on to the next component

If there are no more valid values involving the current component, return to the previous component

```
Backtracking($A_1$ , $A_2$ , …, $A_n$ )
k ← 1; $i_k$ ← 0 // traversal index $A_k$
WHILE k>0 DO
 $i_k$ ← $i_k$ +1
 v ← False
 WHILE v=False AND $i_k$ <=$m_k$ DO
   $s_k$ ← $a^k_{ik}$
   IF "($s_1$, .., $s_k$ ) is valid" THEN v ← True
   ELSE $i_k$ ← $i_k$ +1 ENDIF ENDWHILE
   IF v=True THEN
     IF "($s_1$ , …, $s_k$ ) is a final solution "
     THEN “ process final solution ”
     ELSE k ← k+1; i k ← 0 ENDIF
   ELSE k ← k-1 ENDIF
ENDWHILE
```

Application 1 : generating permutations

```
Backtracking($A_1$ , $A_2$ , …, $A_n$ )
k ← 1; $i_k$ ← 0 // traversal index $A_k$
WHILE k>0 DO
 $i_k$ ← $i_k$ +1
 v ← False
 WHILE v=False AND $i_k$ <=$m_k$ DO
   $s_k$ ← $a^k_{ik}$
   IF "($s_1$, .., $s_k$ ) is valid" THEN v ← True
   ELSE $i_k$ ← $i_k$ +1 ENDIF ENDWHILE
   IF v=True THEN
     IF "($s_1$ , …, $s_k$ ) is a final solution "
     THEN “ process final solution ”
     ELSE k ← k+1; i k ← 0 ENDIF
   ELSE k ← k-1 ENDIF
ENDWHILE
```

```
Permutations(n)
 k ← 1; s[k] ← 0
 WHILE k>0 DO
  s[k] ← s[k]+1
   v ← False
   WHILE v=False AND s[k]<=n DO
     IF (s[1..k])
     THEN v ← True
     ELSE s[k] ← s[k]+1 ENDIF ENDWHILE
   IF v=True THEN
     IF k=n
     THEN WRITE s[1..n]
     ELSE k ← k+1; s[k] ← 0 ENDIF
   ELSE k ← k-1 ENDIF
 ENDWHILE
```

Application 2 : N-Queens Problem

Problem: Determine all the possibilities of placing n queens on a chessboard of size nxn such that they do not attack each other:

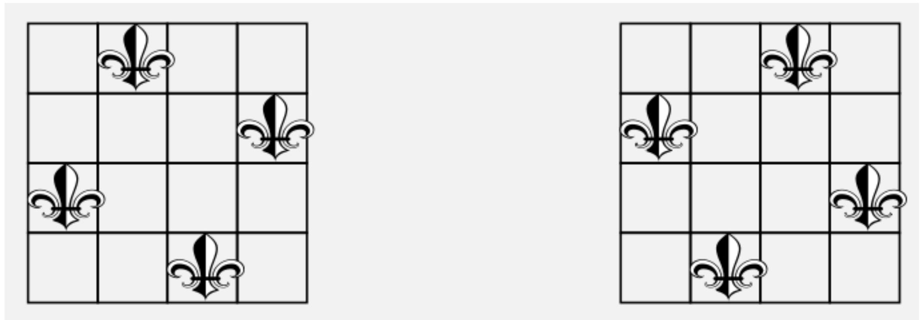
- each line contains exactly one queen

- each column contains exactly one queen

- each diagonal contains at most one queen

Remark. It is a classical problem proposed by Max Bezzel (1850) and studied by several mathematicians of the time (Gauss, Cantor)

Example: if n <=3 has no solution ; if n =4 there are 2 solutions



As it increases n, the number of solutions increases (for n = 8 are 92 solutions)

Application 2 : N-Queens Problem

1. Representation of the solution :
   - lets consider that queen k is always placed on line k (the queens are identical) . Thus for each queen it is sufficient to identify the column on which it will be placed. The solution will thus be represented by a table (s_1, ..., s_n) where s_k = index of the column on which queen k is placed
2. The sets A_1, ..., A_n : {1,2,...,n}. Every set will be processed in the natural order of elements (from 1 to n)
3. Continuation conditions : a partial solution (s_1, s_2, ..., s_k) must satisfy the problem restrictions
   - no more than one queen on each line, column or diagonally
4. Criterion for deciding whether a partial solution is final : k = n (all queens have been placed)

Application 2 : N-Queens Problem

3. Continuation conditions : Let (s_1 , s_2, ... , s_k) be a partial solution . It is valid if :

- The queens are on different lines – the condition is implicitly satisfied due to the way of representation used (queen i is always placed on line i)
- The queens are on different columns:

  s_i != s_j for any i != j

(it is enough to verify that s_k != s_i for any 1<=i<=k-1)

- Queens are on different diagonals:

  |i-j| != |s_i – s_j | for any i != j

(it is enough to check | k-i | != | s_k - s_i | for any 1<=i<=k-1)

Application 2 : N-Queens Problem

Remark:

Two queens i and j are on the same diagonal if:

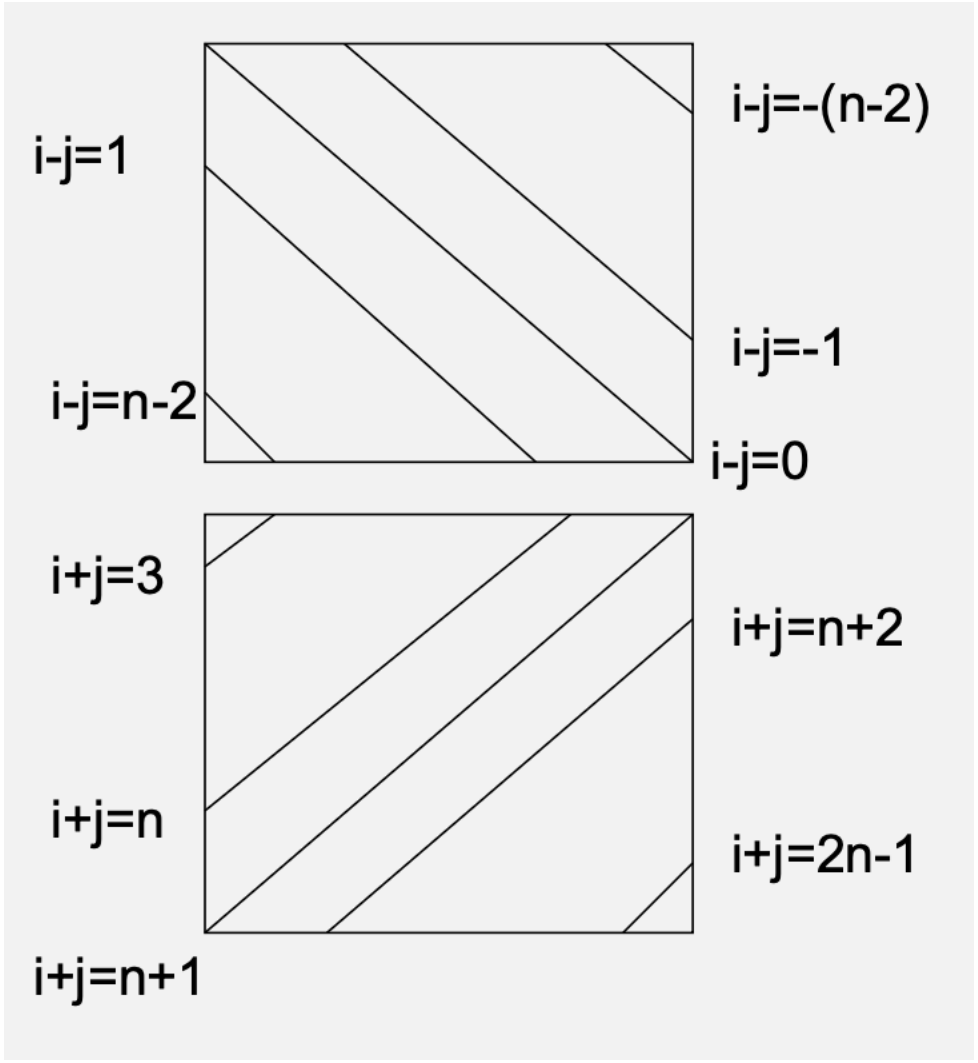
$i-s_i = j-s_j$ equivalent with $i-j = s_i - s_j$

or

$i + s_i = j + s_j$ equivalent with $i-j = s_j - s_i$

This means that

$|i-j| = |s_i - s_j|$



Application 2 : N-Queens Problem

Recursive algorithm

```
queen(s)
 IF k=n+1 THEN WRITE s[1..n]
 ELSE
   FOR i ← 1,n DO
     s[k] ← i
     IF Validate(s[1..k])=True
     THEN queen(k+1)
    ENDIF
  ENDFOR
 ENDIF
```

Validation algorithm

```
Validate(s[1..k])
 FOR i ← 1,k-1 DO
   IF s[k]=s[i] OR |i-k|=|s[i]-s[k]|
     THEN RETURN False
   ENDIF
 ENDFOR
RETURN True
```

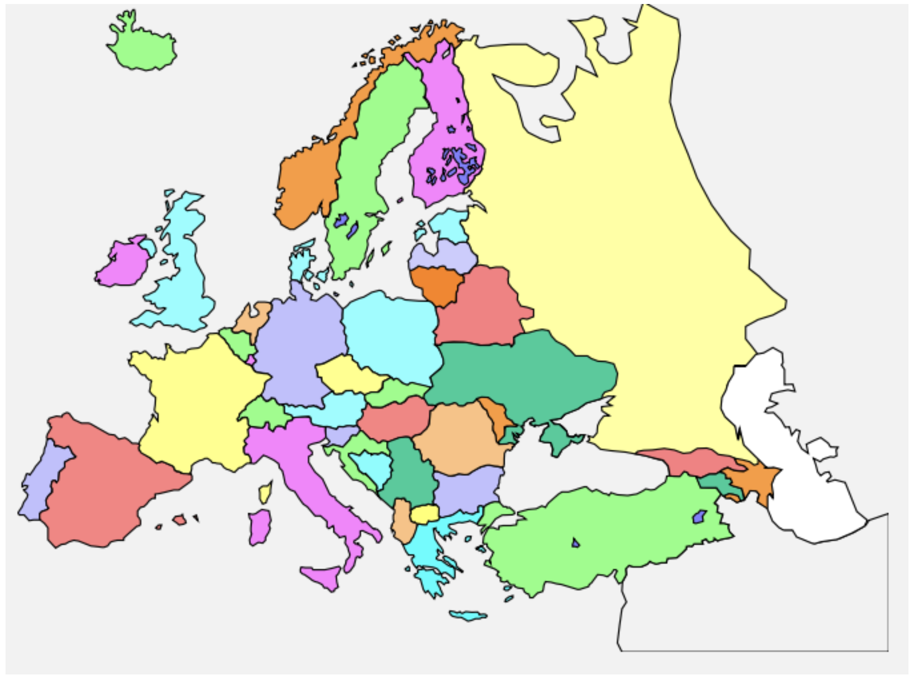
Call: Queen(1)

Application 3 : map coloring problem

Problem : Consider a geographic map that contains n countries.

Given m colors ($4 <= m < n$), assign a color to each country, so that any two neighbouring countries are colored differently.

Mathematical inspiration : any map can be colored using no more than 4 colors (result proved in 1976 by Appel and Haken – one of the first results obtained using the computer-aided proof technique)



Application 3 : map coloring problem

- Problem : Consider a geographic map that contains Given 4<=m<n colors, assign a color to each country so that any two neighboring countries are colored differently.
- Formalization of the problem: Consider how the "neighborhood" relationship is specified by a matrix N having the elements :

$$\mathrm{N}(i,j) = \begin{cases} 0 & \text{if i and j are not neighboring countries} \\ 1 & \text{if i and j are neighboring countries} \end{cases}$$

- Goal: Determine an array $S=(s_1,\dots,s_n)$ with s_k in $\{1,\dots,m\}$, specifying the associated color for country k and such that for all pairs (i,j) with N(i,j)=1, the elements s_i and s_j are different ($s_i \mathrel{!=} s_j$)

Application 3 : map coloring problem

1. Representation of the solution
   - $S=(s_1 ,\ldots,s_n)$ where s_k is the associated color for country k

2. The sets $A_1, \ldots, A_n : \{1,2,\ldots,m\}$.
   - Every set will be processed in the natural order of the elements (indexing starts from 1 to m)

3. Continuation conditions: a partial solution ($s_1, s_2, \ldots , s_k$) must satisfy
   - s_i != s_j for all pairs (i,j) for which N(i,j)=1. For each k, it is enough to verify that s_k != s_j for all values i in {1,2,…,k-1} satisfy N (i,k)=1

4. Criterion to decide whether a partial solution is a final solution :
   - k = n (all countries have been colored)

Application 3 : map coloring problem

Recursive algorithm

```
Coloring(k)
 IF k=n+1 THEN WRITE s[1..n]
 ELSE
   FOR i ← 1, m DO
     s[k] ← i
     IF valid(s[1..k])=True THEN Coloring(k+1)
     ENDIF
   ENDFOR
 ENDIF
```

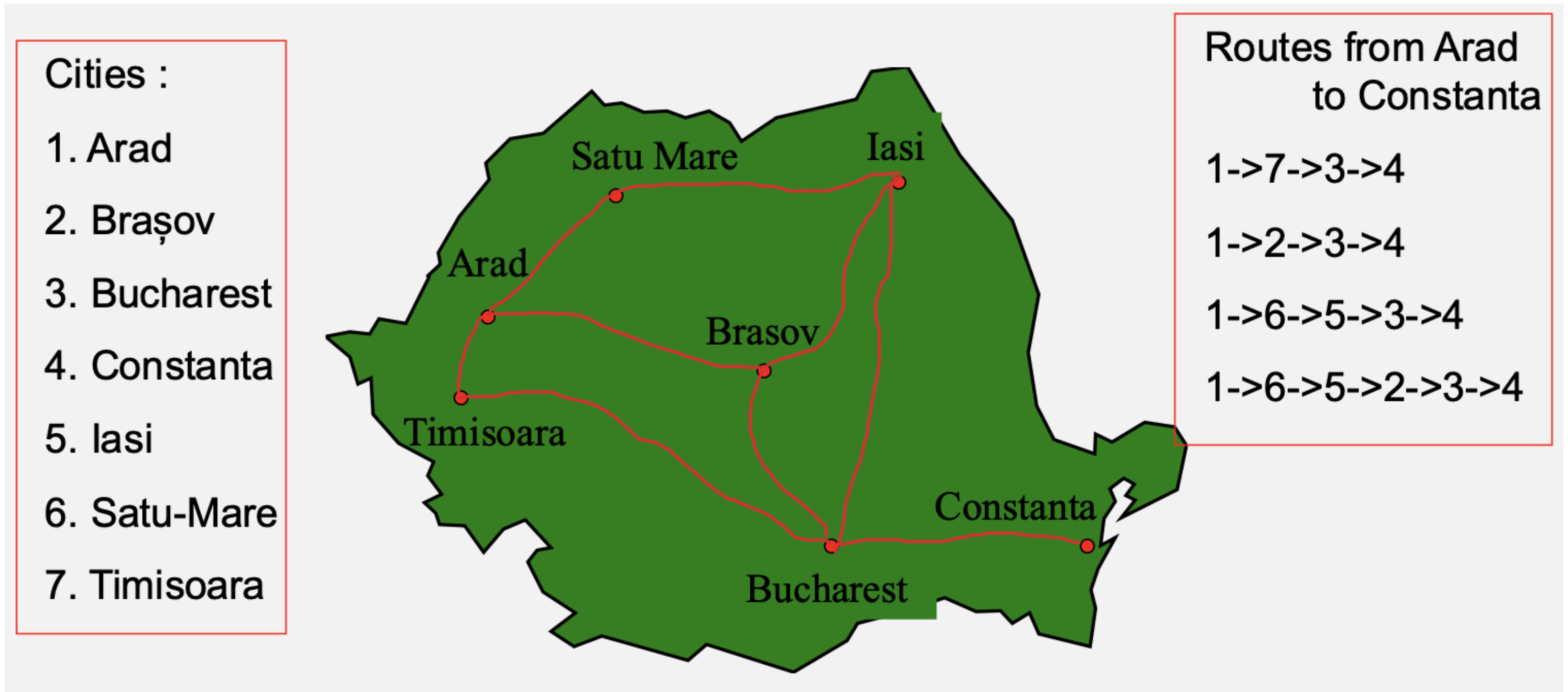
Validation algorithm

```
valid(s[1..k])
  FOR i ← 1,k-1 DO
   IF N[i,k]=1 AND s[i]=s[k]
   THEN RETURN False
   ENDIF
  ENDFOR
RETURN True
```

Call: Coloring(1)

Application 4 : route finding

- Consider a set of cities and the network of roads that connect them.
- Determine all the routes that connect two given times (a route cannot pass through the same place twice).



Cities :

1. Arad
2. Brașov
3. Bucharest
4. Constanta
5. Iasi
6. Satu-Mare
7. Timisoara

Routes from Arad to Constanta

1->7->3->4

1->2->3->4

1->6->5->3->4

1->6->5->2->3->4

Application 4 : route finding

Formalization of the problem: The road network is specified by matrix C:

$$C(i,j) = \begin{cases} 0 & \text{no direct road from city i to city j} \\ 1 & \text{direct road from city i to city j} \end{cases}$$

Goal: Find all the routes $S=(s_1, \ldots, s_m)$ - where s_k in {1,…,n} specifies the city visited at stage k - so that

s_1 is the source city s_m is the destination city

- $s_i \mathrel{!=} s_j$ for any $i \mathrel{!=} j$ (don't pass through the same city twice)
- $C(s_k, s_{k+1})=1$ (there is a direct road between a city and the next visited city)

Application 4 : route finding

1. Representation of the solution

   S=(s_1 ,..., s_m) with s k representing the city visited at stage k (m = number of stages; not known from the beginning)

2. The sets A_1 ,..., A_n : {1,2,...,n}.
   - Every set will be processed in the natural order of the elements (indexing starts from 1 to n)

3. Continuation conditions: a partial solution (s_1 , s_2 , ... , s_k) must satisfy :
   - s_k != s_j for any j in {1,2,...,k-1} (the times are distinct)
   - C(s_{k-1} ,s_k)=1 (you can go directly from s_{k-1} to s_k)

4. The criterion for deciding whether a partial solution is a final solution :
   - s_k = the city of destination

Application 4 : route finding

Recursive algorithm

```
routes(k)
 IF s[k-1]="destination city" THEN
    WRITE s[1..k-1]
 ELSE
   FOR j ← 1,n DO
      s[k] ← j
      IF valid(s[1..k])=True THEN
        routes(k+1)
      ENDIF
   ENDFOR
 ENDIF
```

Call: (s[1] - starting point / source city)

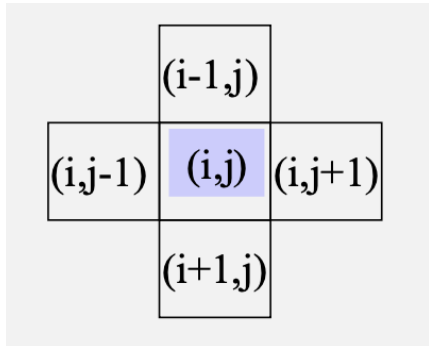
- Routes(2)

Validation algorithm

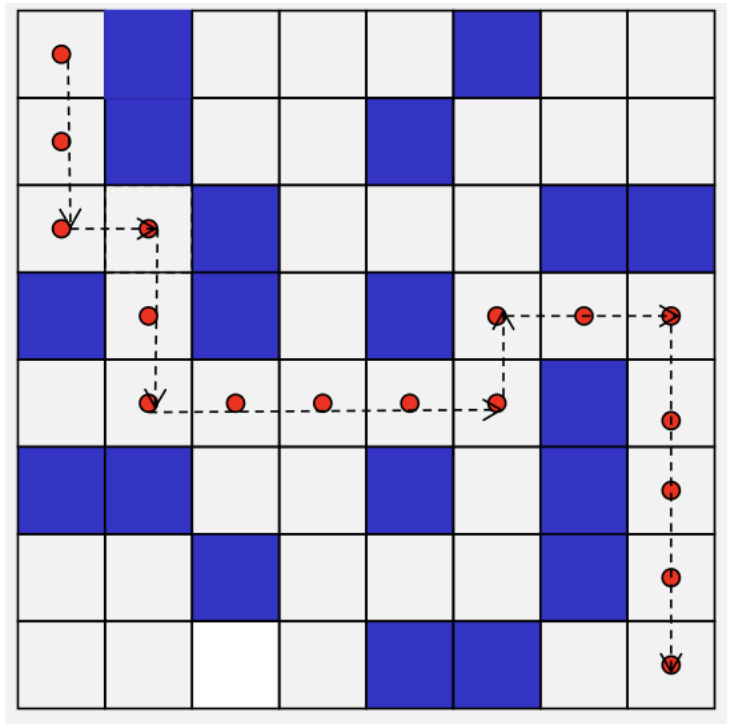
```
valid(s[1..k])
 IF C[s[k-1],s[k]]=0 THEN
   RETURN False
 ENDIF
 FOR i ← 1,k-1 DO
   IF s[i]=s[k] THEN
     RETURN False
   ENDIF
 ENDFOR
RETURN True
```

Application 5 : the maze problem

- Problem. Consider a maze defined on a nxn grid . Find all paths starting from (1,1) and ending in in n (nxn)
- Remark. Only free cells can be visited. From a cell (i,j) it is possible to move to one of the neighboring cells:



- Remark : cells on the border have fewer neighbours



Application 5 : the maze problem

Formalization of the problem. The maze is stored in a nxn matrix

$$\mathrm{M}(i,j) = \begin{cases} 0 & \text{free cell} \\ 1 & \text{cell is visited} \end{cases}$$

Goal: Find S = (s_1, ..., s_m) with s_k in {1,...,n}x{1,...,n} indices corresponding to the cell visited at stage k

- s_1 is the starting cell: (1,1)
- s_m is the destination cell : (n,n)
- s_k != s_q for any k != q (a cell is visited at most once)
- M(s k)=0 (only free cells can be visited)
- s_{k-1} and s_k are neighboring cells

Application 5 : the maze problem

1. Representation of the solution

   $S=(s_1 ,..., s_n)$ with s k representing the indices of the cell visited at stage k

2. The sets $A_1 ,...,A_n$ are subsets of the set

   { 1,2 ,...,n}x{1,2,...,n}. For each cell (i,j) there is a set of 4 neighbors: (i,j-1), (i,j+1), (i-1,j), (i+1,j)

3. Continuation conditions: (s_1 , s_2 , ... , s_k) is a partial solution if:
   - s_k != s_q for any q in {1,2,...,k-1} (never visit same cell)
   - M(s k)=0 (cell is free)
   - s_{k-1} and s_k are neighboring cells

4. Condition that a partial solution (s_1, ... , s_k) is a final solution :
   - sk = (n,n) (the maze destination has been reached)

Application 5 : the maze problem

Recursive algorithm

```
maze(s)
 IF s[k-1].i=n and s[k-1].j=n THEN WRITE s[1..k]
 ELSE // all neighbors are tried
   s[k].i ← s[k-1].i-1; s[k].j ← s[k-1].j
   IF valid(s[1..k])=True THEN maze (k+1) ENDIF
   s[k].i ← s[k-1].i+1; s[k].j ← s[k-1].j
   IF valid(s[1..k])=True THEN maze (k+1) ENDIF
   s[k].i ← s[k-1].i; s[k].j ← s[k-1].j-1
   IF valid(s[1..k])=True THEN maze (k+1) ENDIF
   s[k].i ← s[k-1].i; s[k].j ← s[k-1].j+1
   IF valid(s[1..k])=True THEN maze (k+1) ENDIF
ENDIF
```

Remark:

s[k] is a structure with two fields:

- s[k].i represents the first component of the index pair
- s[k].j represents the second component of the index pair

Application 5 : the maze problem

Validation algorithm

```
valid(s[1..k])
  IF s[k].i<1 OR s[k].i>n OR s[k].j<1 OR s[k].j>n // 'cell is out of grid' check
  THEN RETURN False
  ENDIF
  IF M[s[k].i,s[k].j]=1 THEN RETURN False ENDIF // 'cell is not free' check
  FOR q ← 1,k-1 DO // 'cell is already visited' check
    IF s[k].i=s[q].i AND s[k].j=s[q].j THEN RETURN False ENDIF
   ENDFOR
RETURN True
```

Algorithm Call:

```
s[1].i:=1; s[1].j:=1
maze(2)
```

Summary

- The backtracking technique is applied to constraint satisfaction problems (when it is desired to generate all configurations that satisfy certain properties).
- The systematic traversal of the solution space guarantees correctness.
- Based on exhaustive search, the complexity of the method depends on the size of the solution space (usually n!, 2^n in the case of combinatorial problems that require the construction of solutions with n components). It can be practically applied to problems where n is at most in the order of tens.
- It can also be used to solve optimization problems with constraints (for example, to find configurations that minimize a cost), in which case partial solutions are considered viable if they do not exceed the cost of the best configuration already constructed (the variant known as branch-and-bound).

Q&A