Backtracking

Outline

- What is backtracking ?
- The general structure of the algorithm
- Applications: generating permutations, generating subsets, n-queens problem, map coloring, path finding, maze problem

What is backtracking?

- It is a systematic search strategy of the state-space of combinatorial problems

- It is mainly used to solve problems which ask for finding elements of a set which satisfy some constraints. Most of the problems which can be solved by backtracking have the following general form:

  " Find a subset S of A_1 x A_2 x … x A_n (A_k – finite sets) such that each element $s=(s_1,s_2,\ldots,s_n)$ satisfies some constraints"

Example: generating all permutations of {1,2,…,n}

A_k = {1,2,…,n} for all k

s_i <> s_j for all i<>j (restriction: distinct components)

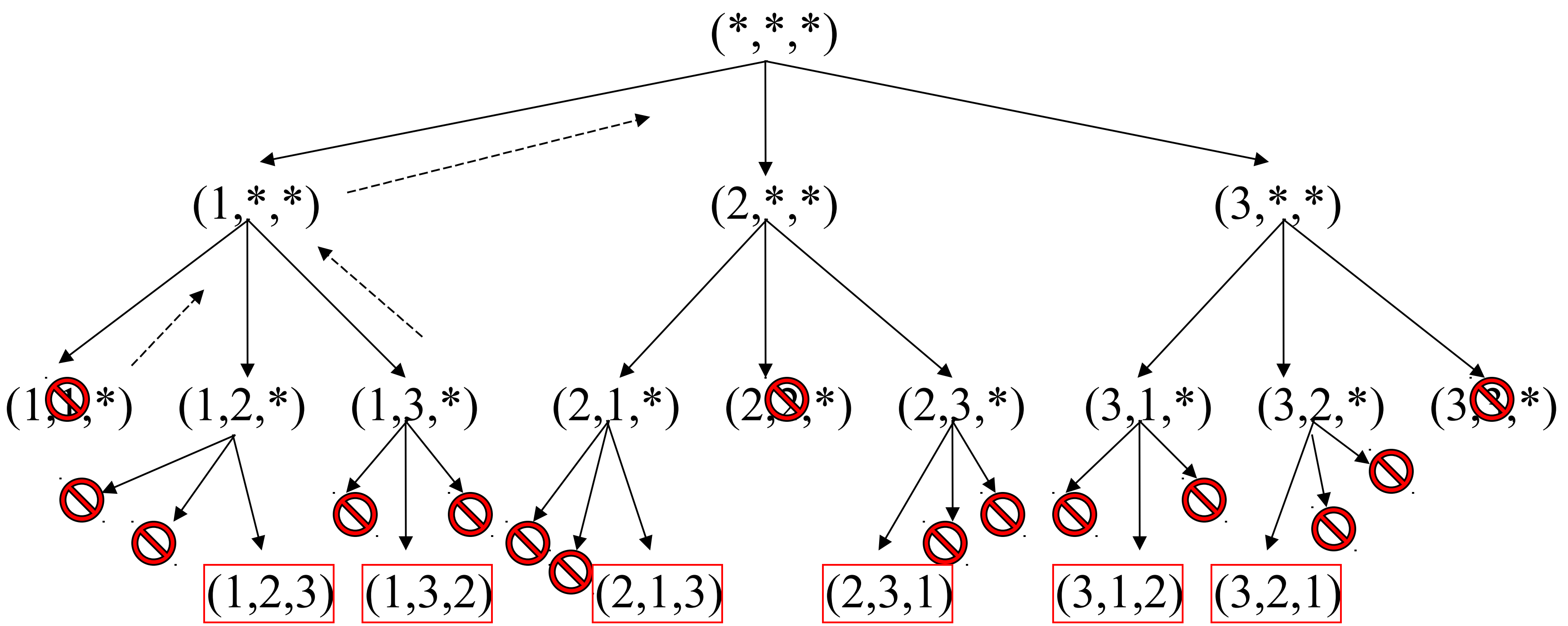
What is backtracking?

Basic ideas:

- the solutions are constructed in an incremental manner by finding the components successively
- each partial solution is evaluated in order to establish if it is promising (a promising solution could lead to a final solution while a non-promising one does not satisfy the partial constraints induced by the problem constraints)
- if all possible values for a component do not lead to a promising (valid or viable) partial solution then we come back to the previous component and try another value for it
- backtracking implicitly constructs a state space tree:
  - The root corresponds to an initial state (before the search for a solution begins)
  - An internal node corresponds to a promising partial solution
  - An external node (leaf) corresponds either to a non-promising partial solution or to a final solution

What is backtracking?

Example: state space tree for permutations generation



Outline

- What is backtracking ?
- The general structure of the algorithm
- Applications: generating permutations, generating subsets, n-queens problem, map coloring, path finding, maze problem

The general structure of the algorithm

Basic steps:

1. Choose the representation of solutions

1. Establish the sets $A_1,\ldots,A_n$ and the order in which their elements are processed

1. Derive from the problem restrictions the conditions which a partial solution should satisfy in order to be promising (valid). These conditions are sometimes called continuation conditions.

1. Choose a criterion to decide when a partial solution is a final one

The general structure of the algorithm

Example: generating permutations

1. Solution representation: each permutation is a vector $s=(s_1,s_2, \ldots s_n)$ satisfying: $s_i<>s_j$ for all $i<>j$

1. Sets $A_1,\ldots,A_n$: $\{1,2,\ldots,n\}$. Each set will be processed in the natural order of the elements

1. Continuation conditions: a partial solution $(s_1,s_2,\ldots,s_k)$ should satisfy $s_k<>s_i$ for all $i<k$

1. Criterion to decide when a partial solution is a final one: $k=n$

The general structure of the algorithm

Some notation:

$(s_1,s_2,\ldots,s_k)$ partial solution

k – index for constructing s

$A_k = \{a^k_1,\ldots,a^k_{mk}\}$

$m_k=card\{A_k\}$

i_k - index for scanning A_k

The general structure of the algorithm

Search for a value of k-th component which leads to a promising partial solution

If such a value exists check if a final solution was obtained

If it is a solution then process it and go to try the next possible value

If it is not a final solution go to the next component

If it doesn't exist then go back to the previous component

Backtracking(A_1, A_2, …, A_n)

  $k:=1$; $i_k:=0$

  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 "$(s_1,\dots,s_k)$ is a final solution"

      THEN "process the final solution"

      ELSE $k:=k+1$; $i_k:=0$ ENDIF

  ELSE $k:=k-1$ ENDIF

ENDWHILE

The general structure of the algorithm

The recursive variant:

- Suppose that $A_1,\ldots,A_n$ and s are global variables

- Let k be the component to be filled in

The algorithm will be called with BT_rec(1)

```
BT_rec(k)
IF "(s_1,…,s_{k-1}) is a solution"
    THEN  "process it"
    ELSE
     FOR j:=1,m_k  DO
        s_k:=a^k_j
        IF "(s_1,…s_k) is valid"
        THEN BT_rec(k+1) ENDIF
     ENDFOR
ENDIF
```

Try each possible value

Go to fill in the next component

Outline

- What is backtracking ?
- The general structure of the algorithm
- Applications: generating permutations, generating subsets, n-queens problem, map coloring, path finding, maze problem

Application: generating permutations

```
Backtracking(A_1, A_2, …, A_n)
  k:=1;  i_k:=0
  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 the 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 valid(s[1..k])
           THEN v:=True
           ELSE s[k]:=s[k]+1
     ENDWHILE
   IF v=True THEN
    IF k=n
       THEN WRITE s[1..n]
        ELSE k:=k+1; s[k]:=0
   ELSE k:=k-1
  ENDIF ENDIF ENDWHILE
```

Application: generating permutations

Function to check if a partial solution is a valid one

```
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
```

Outline

- What is backtracking ?
- The general structure of the algorithm
- Applications: generating permutations, generating subsets, n-queens problem, map coloring, path finding, maze problem

Application: generating subsets

Let $A=\{a_1,\ldots,a_n\}$ be a finite set. Generate all subsets of A having m elements.

Example: A={1,2,3}, m=2, S={{1,2},{1,3},{2,3}}

- Solution representation: each subset is represented by its characteristic vector ($s_i=1$ if a_i belongs to the subset and $s_i=0$ otherwise)
- Sets $A_1,\ldots,A_n$: {0,1}. Each set will be processed in the natural order of the elements (first 0 then 1)
- Continuation conditions: a partial solution $(s_1,s_2,\ldots,s_k)$ should satisfy $s_1+s_2+\ldots+s_k \le m$ (the partial subset contains at most m elements)
- Criterion to decide when a partial solution is a final one: $s_1+s_2+\ldots+s_k = m$ (m elements were already selected)

Application: generating subsets

Iterative algorithm

```
subsets(n,m)
  k:=1
  s[k]:=-1
  WHILE k>0 DO
     s[k]:=s[k]+1;
     IF s[k]<=1 AND
         sum(s[1..k])<=m
     THEN
        IF sum(s[1..k])=m
         THEN s[k+1..n]=0
                 WRITE s[1..n]
        ELSE k:=k+1; s[k]:=-1
      ENDIF
   ELSE k:=k-1
   ENDIF  ENDWHILE
```

Recursive algorithm

```
subsets_rec(k)
  IF sum(s[1..k-1])=m
  THEN
         s[k..n]=0
         WRITE s[1..n]
  ELSE
    s[k]:=0; subsets_rec(k+1);
    s[k]:=1; subsets_rec(k+1);
  ENDIF
```

Rmk: sum(s[1..k]) computes the sum of the first k components of s[1..n]

Outline

- What is backtracking ?
- The general structure of the algorithm
- Applications: generating permutations, generating subsets, n-queens problem, map coloring, path finding, maze problem

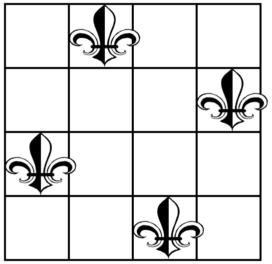
Application: n-queens problem

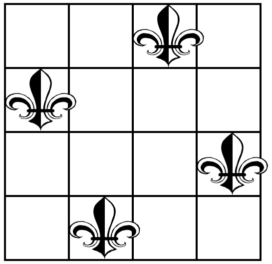
Find all possibilities of placing n queens on a n-by-n chessboard such that they does not attack each other:

- each line contains only one queen
- each column contains only one queen
- each diagonal contains only one queen

This is a classical problem proposed by Max Bezzel (1850) an studied by several mathematicians of the time (Gauss, Cantor)

Examples: if n<=3 there is no solution; if n=4 there are two solutions





As n becomes larger the number of solutions becomes also larger (for n=8 there are 92 solutions)

Application: n-queens problem

1. Solution representation: we shall consider that queen k will be placed on row k. Thus for each queen it suffices to explicitly specify only the column to which it belongs: The solution will be represented as an array $(s_1,\ldots,s_n)$ with s_k = the column on which the queen k is placed
2. Sets $A_1,\ldots,A_n$: $\{1,2,\ldots,n\}$. Each set will be processed in the natural order of the elements (starting from 1 to n)
3. Continuation conditions: a partial solution $(s_1,s_2,\ldots,s_k)$ should satisfy the problems restrictions (no more than one queen on a line, column or diagonal)
4. Criterion to decide when a partial solution is a final one: $k = n$ (all n queens have been placed)

Application: n-queens problem

Continuation conditions: Let $(s_1,s_2,\ldots,s_k)$ be a partial solution. It is a valid partial solution if it satisfies:

- All queens are on different rows - implicitly satisfied by the solution representation (each queen is placed on its own row)
- All queens are on different columns:

  $s_i <> s_j$ for all $i <> j$

  (it is enough to check that $s_k <> s_i$ for all $i <= k-1$)
- All queens are on different diagonals:

  $|i-j| <> |s_i - s_j|$ for all $i <> j$

  (it is enough to check that $|k-i| <> | s_k - s_i|$ for all $1 <= i <= k-1$)

Indeed ….

Application: n-queens problem

Remark:

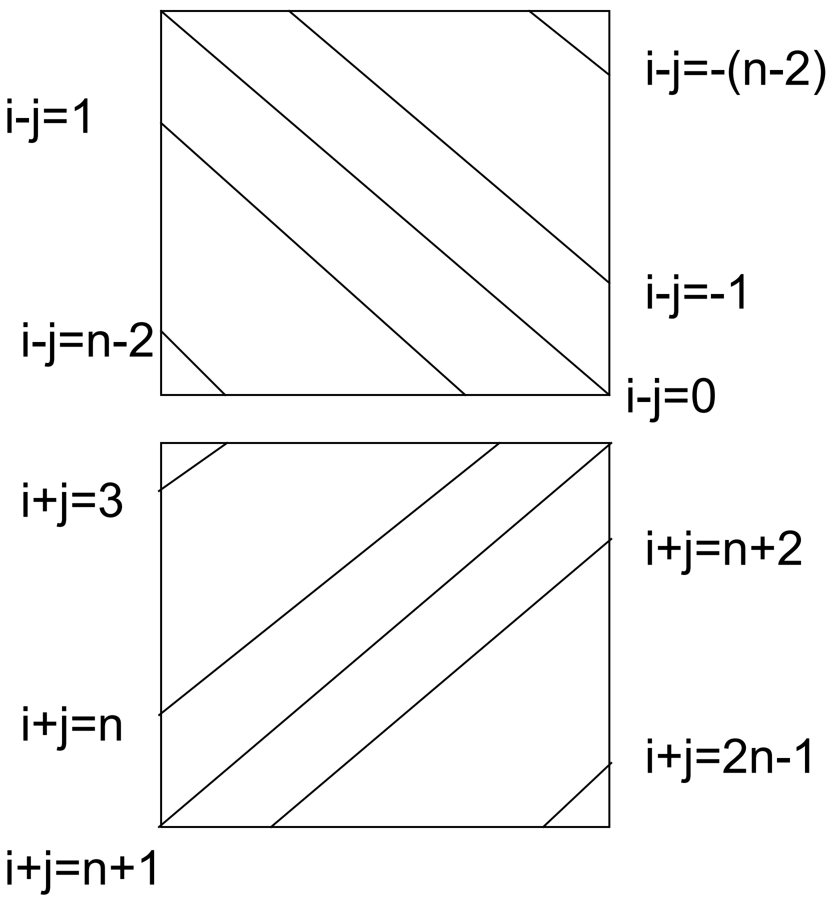
two queens i and j are on the same diagonal if either

$$i-s_i=j-s_j \Leftrightarrow i-j = s_i-s_j$$

or

$$i+s_i=j+s_j \Leftrightarrow i-j= s_j-s_i$$

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



Application: n-queens problem

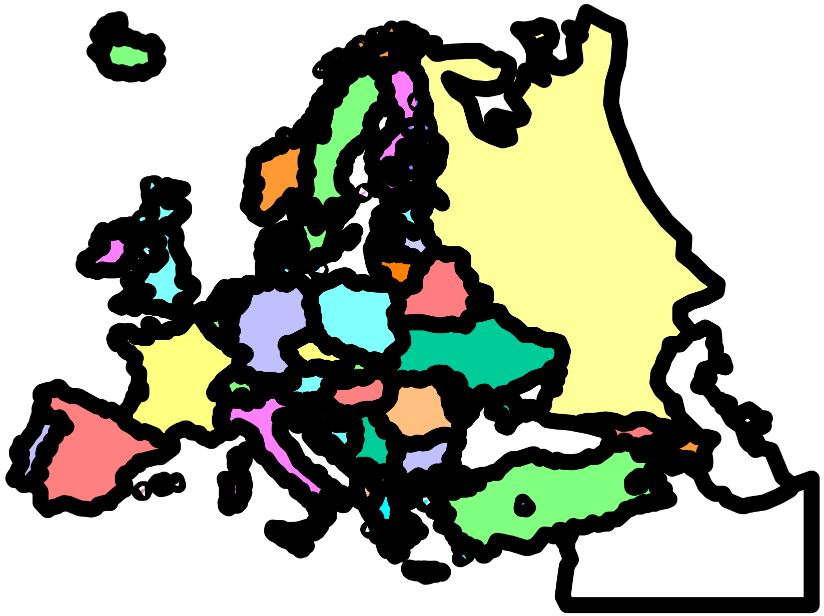
```
Validation(s[1..k])
FOR i:=1,k-1 DO
  IF s[k]=s[i] OR |i-k|=|s[i]-s[k]|
        THEN RETURN False
  ENDIF
ENDFOR
RETURN True
```

Algorithm:

```
Queens(k)
IF k=n+1 THEN WRITE s[1..n]
ELSE
   FOR i:=1,n DO
      s[k]:=i
      IF Validation(s[1..k])=True
          THEN Queens(k+1)
      ENDIF
   ENDFOR
ENDIF
```

Application: map coloring

Problem: Let us consider a geographical map containing n countries. Propose a coloring of the map by using 4<=m<n colors such that any two neighboring countries have different colors



Mathematical related problem: any map can be colored by using at most 4 colors (proved in 1976 by Appel and Haken) – one of the first results of computer assisted theorem proving

Application: map coloring

Problem: Let us consider a geographical map containing n countries. Propose a coloring of the map by using 4<=m<n colors such that any two neighboring countries have different colors

Problem formalization: Let us consider that the neighborhood relation between countries is represented as a matrix N as follows:

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

Find a map coloring $S=(s_1,\ldots,s_n)$ with s_k in $\{1,\ldots,m\}$ such that for all pairs (i,j) with N(i,j)=1 the elements s_i and s_j are different ($s_i <> s_j$)

Application: map coloring

1. Solution representation

   $S=(s_1,\dots,s_n)$ with s_k representing the color associated to country k

2. Sets $A_1,\dots,A_n$: {1,2,…,m}. Each set will be processed in the natural order of the elements (starting from 1 to m)

1. Continuation conditions: a partial solution $(s_1,s_2,\dots,s_k)$ should satisfy s_i<>s_j for all pairs (i,j) with N(i,j)=1

   For each k it suffices to check that s_k<>s_j for all pairs i in {1,2,…,k-1} with N(i,k)=1

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

Application: map coloring

Recursive algorithm

```
Coloring(k)
IF k=n+1 THEN WRITE s[1..n]
ELSE
   FOR j:=1,m DO
      s[k]:=j
      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: path finding

Let us consider a set of n towns. There is a network of routes between these towns. Generate all routes which connect two given towns such that the route doesn't reach twice the same town

Towns:

1.Arad

2.Brasov

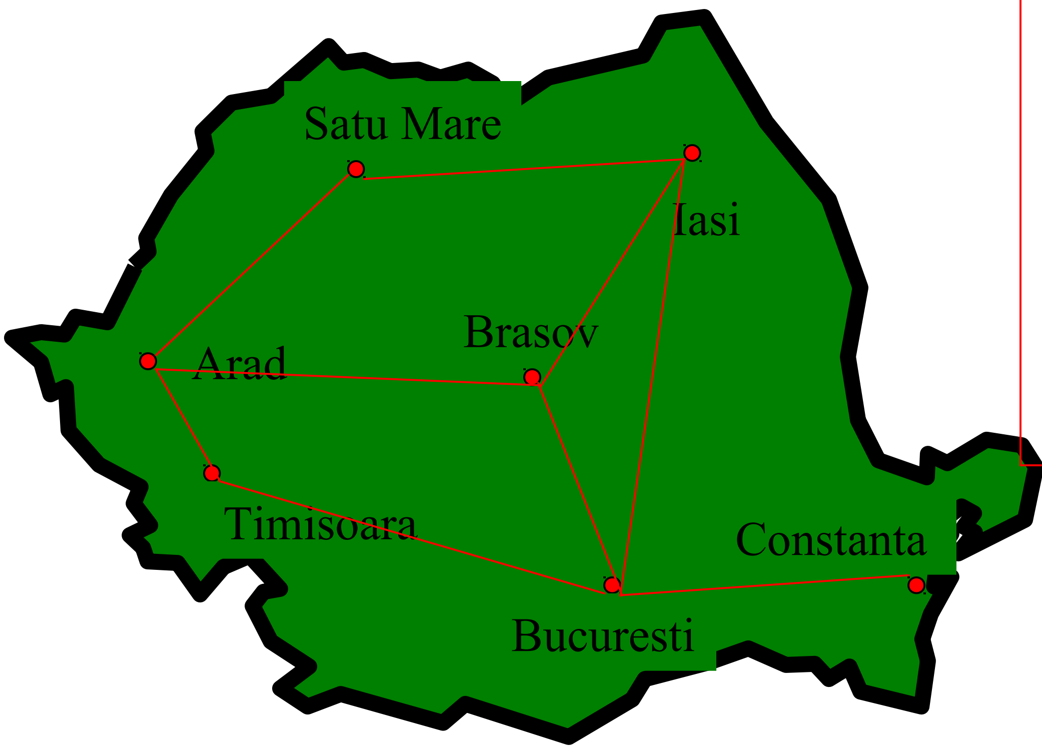
3.Bucuresti

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: path finding

Problem formalization: Let us consider that the connections are stored in a matrix C as follows:

$$C(i,j) = \begin{cases} 0 & \text{if doesn't exist a direct connection between i and j} \\ 1 & \text{if there is a direct connection between i and j} \end{cases}$$

Find all routes $S=(s_1,\ldots,s_m)$ with s_k in $\{1,\ldots,n\}$ denoting the town visited at moment k such that

s_1 is the starting town

s_m is the destination town

s_i<>s_j for all i <>j (a town is visited only once)

$C(s_i,s_{i+1})=1$ (there exists a direct connections between towns visited at successive moments)

Application: path finding

1. Solution representation

   $S=(s_1,\ldots,s_m)$ with s_k representing the town visited at moment k

1. Sets $A_1,\ldots,A_n$: {1,2,…,n}. Each set will be processed in the natural order of the elements (starting from 1 to n)

3. Continuation conditions: a partial solution $(s_1,s_2,\ldots,s_k)$ should satisfy:

   $s_k<>s_j$ for all j in {1,2,…,k-1}

   $C(s_{k-1},s_k)=1$

4. Criterion to decide when a partial solution is a final one:

   s_k = destination town

Application: path finding

Recursive algorithm

```
routes(k)
IF s[k-1]=destination town 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
```

Validation algorithm

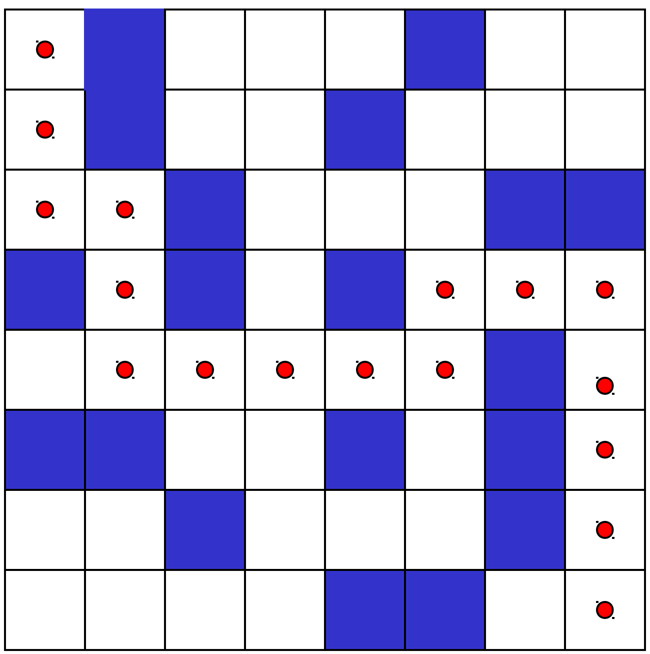
```
Valid(s[1..k])
IF C[s[k-1],s[k]]=0 THEN
      RETURN False
ENDIF
FOR i:=1,k-1 DO
  IF s[i]=s[k]
     THEN RETURN False
  ENDIF
ENDFOR
RETURN True
```

Call:

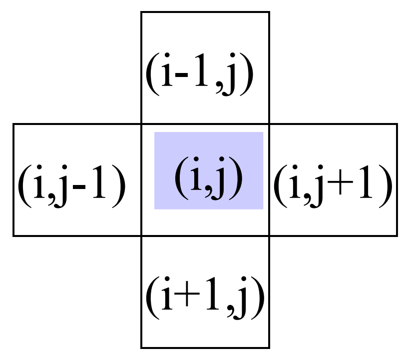
```
s[1]:=starting town
routes(2)
```

Application: maze

Maze problem. Let us consider a maze defined on a nxn grid. Find a path in the maze which starts from the position (1,1) and finishes in (nxn)



Only white cells can be accessed. From a given cell (i,j) one can pass in one of the following neighboring positions:



Remark: cells on the border have fewer neighbours

Application: maze

Problem formalization. The maze is stored as a nxn matrix

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

Find a path $S=(s_1,\ldots,s_m)$ with s_k in $\{1,\ldots,n\}x\{1,\ldots,n\}$ denoting the indices corresponding to the cell visited at moment k

- s_1 is the starting cell (1,1)
- s_m is the destination cell (n,n)
- $s_k <> s_{qj}$ for all k <>q (a cell is visited at most once)
- $M(s_k)=0$ (each visited cell is a free one)
- s_k and s_{k+1} are neighborhood cells

Application: maze

1. Solution representation

   $S=(s_1,\ldots,s_n)$ with s_k representing the cell visited at moment k

2. Sets $A_1,\ldots,A_n$ are subsets of {1,2,…,n}x{1,2,…,n}. For each cell (i,j) there is a set of at most 4 neighbors

3. Continuation conditions: a partial solution $(s_1,s_2,\ldots,s_k)$ should satisfy:

   $s_k <> s_q$ for all q in {1,2,…,k-1}

   $M(s_k)=0$

   s_{k-1} and s_k are neighbours

4. Criterion to decide when a partial solution is a final one:

   $s_k = (n,n)$

Application: maze

```
maze(k)
IF s[k-1]=(n,n) THEN WRITE s[1..k]
ELSE   // try all neighbouring cells
   s[k].i:=s[k-1].i-1; s[k].j:=s[k-1].j        // up
   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        // down
   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        // left
   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        // right
   IF valid(s[1..k])=True THEN maze(k+1) ENDIF
ENDIF
```

Application: maze

```
valid(s[1..k])
IF s[k].i<1 OR s[k].i>n OR s[k].j<1 OR s[k].j>n    // out of the grid
    THEN RETURN False
ENDIF
IF M[s[k].i,s[k].j]=1 THEN RETURN False ENDIF   // occupied cell
FOR q:=1,k-1 DO     // loop
    IF s[k].i=s[q].i AND s[k].j=s[q].j THEN RETURN False ENDIF
ENDFOR
RETURN True
```

Call of algorithm maze:

s[1].i:=1;   s[1].j:=1

maze(2)