Backtracking, Графи, Файлове

Изготвил:
гл.ас. д-р Нора Ангелова

Задачи

Да се напише програмен фрагмент, който извежда даден файл от низове така, че всеки изведен ред да е не по-дълъг от 80 символа.

Задачи

```
const int SIZE = 81;

ifstream iFileName;
iFileName.open("C:/Users/eminor/Desktop/file.txt", ios::in);

if (!iFileName)
{
  cerr << "File couldn't be opened" << endl;
  return;
}

char str[SIZE];
while(iFileName.getline(str, SIZE))
{
  cout << str << endl;
}
iFileName.close();
```

Задачи

Зад. Да се напише програма, която чете от текстовия файл f1 реални числа, записва ги в нов текстов файл f2 и ги извежда на екрана във форматиран вид

f2

```
 +10.35000
  +9.88659
 -18.23492
+734.09000
 +91.08700
```

Задачи

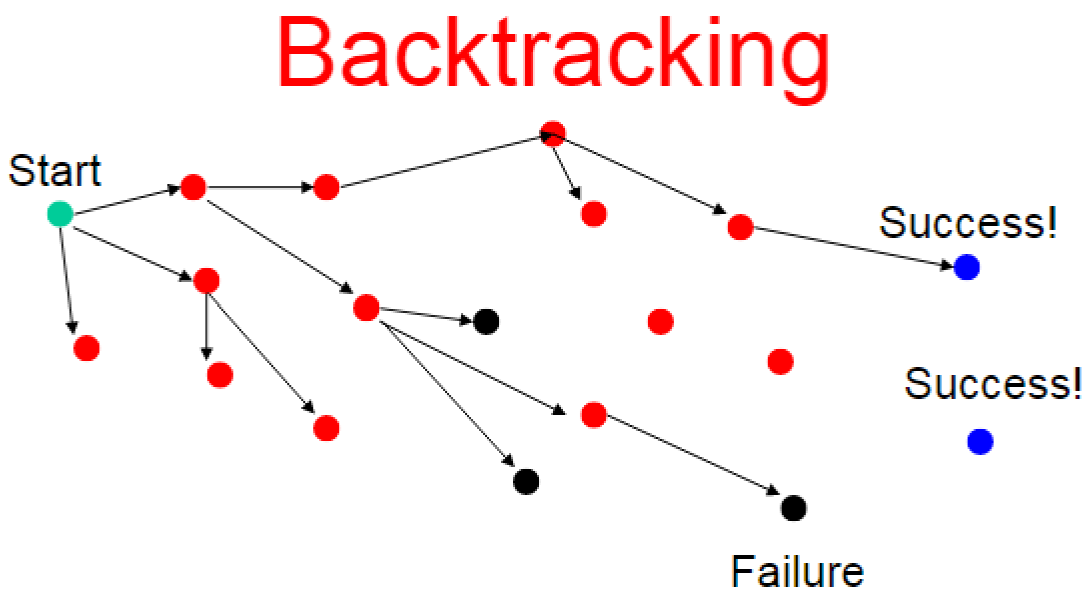
```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>

const int MAX_PATH_SIZE = 100;

void make_format(ifstream&, ofstream&, int, int);
```

Задачи

```
int main()
{
    char file_name1[MAX_PATH_SIZE];
    cout << "Име на входния файл: ";
    cin.getline(file_name1, MAX_PATH_SIZE);
    ifstream f1(file_name1);
    if(!f1)
    {
        cerr << "Не може да се отвори " << file_name1 << '\n';
        return 1;
    }
    char file_name2[MAX_PATH_SIZE];
    cout << "Име на изходния файл: ";
    cin.getline(file_name2, MAX_PATH_SIZE);
    ofstream f2(file_name2);
    if(!f2)
    {
        cerr << "Не може да се отвори " << file_name2 << '\n';
        return 1;
    }
    make_format(f1, f2, 5, 12);
    cout << "Край на програмата.\n";
    f1.close();
    f2.close();
    return 0;
}
```

Задачи

```
void make_format(ifstream& f1, ofstream& f2,
                 int num_after_decimalpoint, int field_width)
{
    f2.setf(ios::fixed);
    f2.setf(ios::showpoint);
    f2.setf(ios::showpos);
    f2.precision(num_after_decimalpoint);
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.setf(ios::showpos);
    cout.precision(num_after_decimalpoint);
    double number;
    while(f1 >> number)
    {
        f2 << setw(field_width) << number << endl;
        cout << setw(field_width) << number << endl;
    }
}
```



Problem space consists of states (nodes) and actions (paths that lead to new states). When in a node can only see paths to connected nodes.

If a node only leads to failure go back to its "parent" node. Try other alternatives. If these all lead to failure then more backtracking may be necessary.

Example

- Sudoku
- 9 by 9 matrix with some numbers filled in
- All numbers must be between 1 and 9
- Goal: Each row, each column, and each mini matrix must contain the numbers between 1 and 9 once each

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

*no duplicates in rows, columns, or mini matrices

Brute Force

- A brute force algorithm is a simple but general approach.
- Try all combinations until you find one that works.
- This approach isn't clever, but computers are fast.
- Then try and improve on the brute force resuts.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Brute force Sudoku Solution

- If not open cells, solved
- Scan cells from left to right, top to bottom for first open cell
- When an open cell is found start cycling through digits 1 to 9.
- When a digit is placed check that the set up is legal
- now solve the board

5	3	1		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

?

After placing a number in a cell is the remaining problem very similar to the original problem ?

Solving Sudoku – Later Steps

5	3	1		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4			
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	8		
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

uh oh!

Brute force Sudoku Solution

- We have reached a dead end in our search
- With the current set up none of the nine digits work in the top right corner

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Brute force Sudoku Solution

- When the search reaches a dead end in

backs up to the previous cell it was trying to fill and goes onto to the next digit.

- We would back up to the cell with a 9 and that turns out to be a dead end as well so we back up again

*so the algorithm needs to remember what digit to try next

- Now in the cell with the 8. We try and 9 and move forward again.

Brute force Sudoku Solution

- Now in the cell with the 8. We try and 9 and move forward again.

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	9		
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Brute force

- Brute force algorithms are slow.
- The don't employ a lot of logic.

For example we know a 6 can't go in the last 3 columns of the first row, but the brute force algorithm will plow ahead any way.

- But, brute force algorithms are fairly easy to implement as a first pass solution.
- backtracking is a form of a brute force algorithm.

Sudoku Solution

- After trying placing a digit in a cell we want to solve the new sudoku board.

Isn't that a smaller (or simpler version) of the same problem we started with?

- After placing a number in a cell, we need to remember the next number to try in case things don't work out.
- We need to know if things worked out (found a solution) or they didn't, and if they didn't try the next number.
- If we try all numbers and none of them work in our cell we need to report back that things didn't work.

Sudoku Solution

- Problems such as Suduko can be solved using recursive backtracking
- Recursive because later versions of the problem are just slightly simpler versions of the original.
- Backtracking because we may have to try different alternatives

Sudoku Solution

If at a solution, report success

for(every possible choice from current node)

- Make that choice and take one step along path. Use recursion to solve the problem for the new node
- If the recursive call succeeds, report the success to the next high level.
- Back out of the current choice to restore the state at the beginning of the loop.

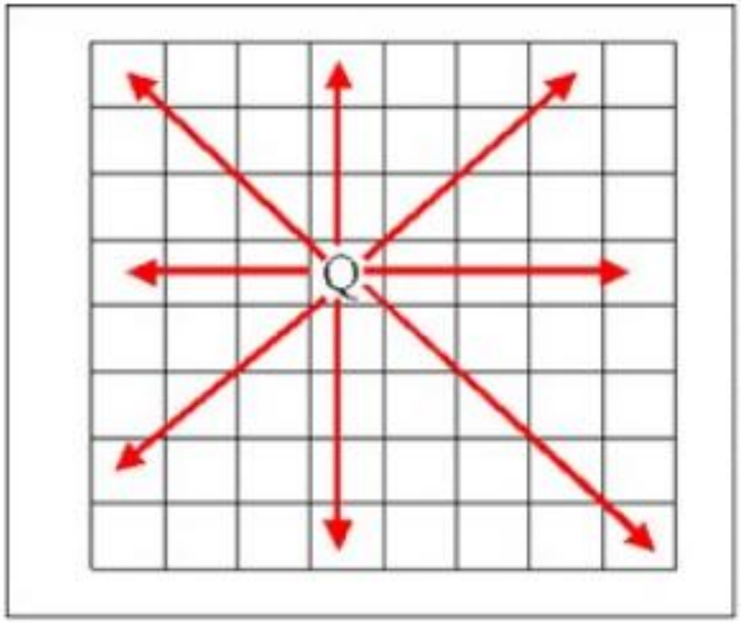
Report failure

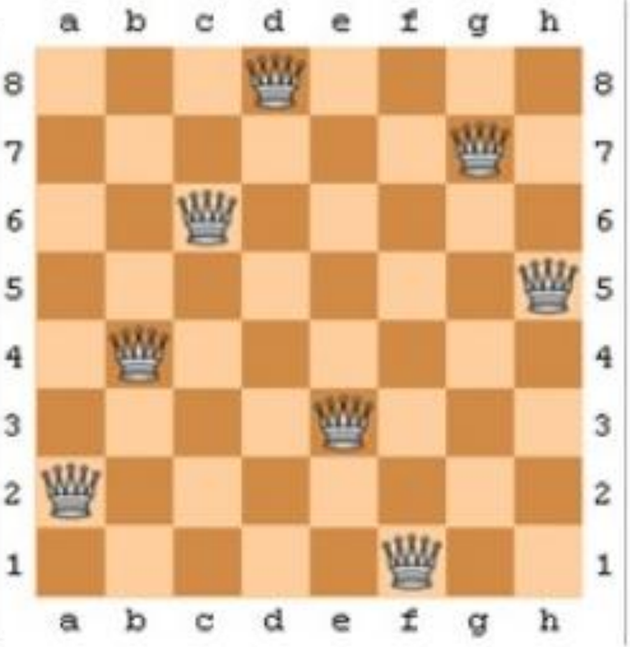
Goals of Backtracking

- Possible goals
  - Find a path to success
  - Find all paths to success
  - Find the best path to success

The 8 Queens Problem

- Place 8 queen pieces on a chess board so that none of them can attack one another





The 8 Queens Problem

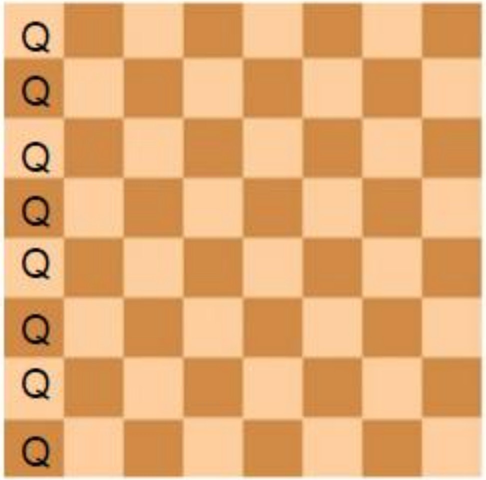
- Place N Queens on an N by N chessboard so that none of them can attack each other
- Number of possible placements?
- How many ways can you choose k things from a set of n items?
  In this case there are 64 squares and we want to choose 8 of them to put queens on.
- In 8 x 8
  64 * 63 * 62 * 61 * 60 * 59 * 58 * 57 =
  78,462, 987, 637, 760 / 8!= 4,426,165,368

The 8 Queens Problem

- For valid solutions how many queens can be placed in a give column?

The 8 Queens Problem

- The previous calculation includes set ups like this one.



- Includes lots of set ups with multiple queens in the same column.
- Number of set ups 8 * 8 * 8 * 8 * 8 * 8 * 8 * 8 = 16,777,216
  We have reduced search space by two orders of magnitude by applying some logic.

The 8 Queens Problem

- If number of queens is fixed and we realize there can't be more than one queen per column we can iterate through the rows for each column.

```
for(int c0 = 0; c0 < 8; c0++){
      board[c0][0] = 'q';
       for(int c1 = 0; c1 < 8; c1++){
            board[c1][1] = 'q';
            for(int c2 = 0; c2 < 8; c2++){
                  board[c2][2] = 'q';
                  // a little later
                  for(int c7 = 0; c7 < 8; c7++){
                        board[c7][7] = 'q';
                        if( queensAreSafe(board) )
                              printSolution(board);
                        board[c7][7] = ' '; //pick up queen
                  }
                  board[c6][6] = ' '; // pick up queen
```

The 8 Queens Problem

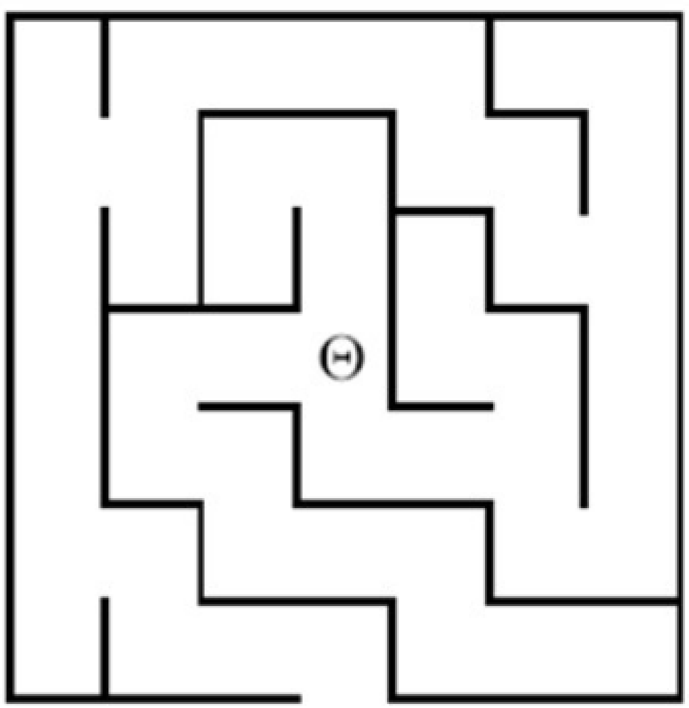
The problem with N queens is you don't know how many for loops to write.

The 8 Queens Problem

- Do the problem recursively
- Learn to recognize problems that fit the pattern.

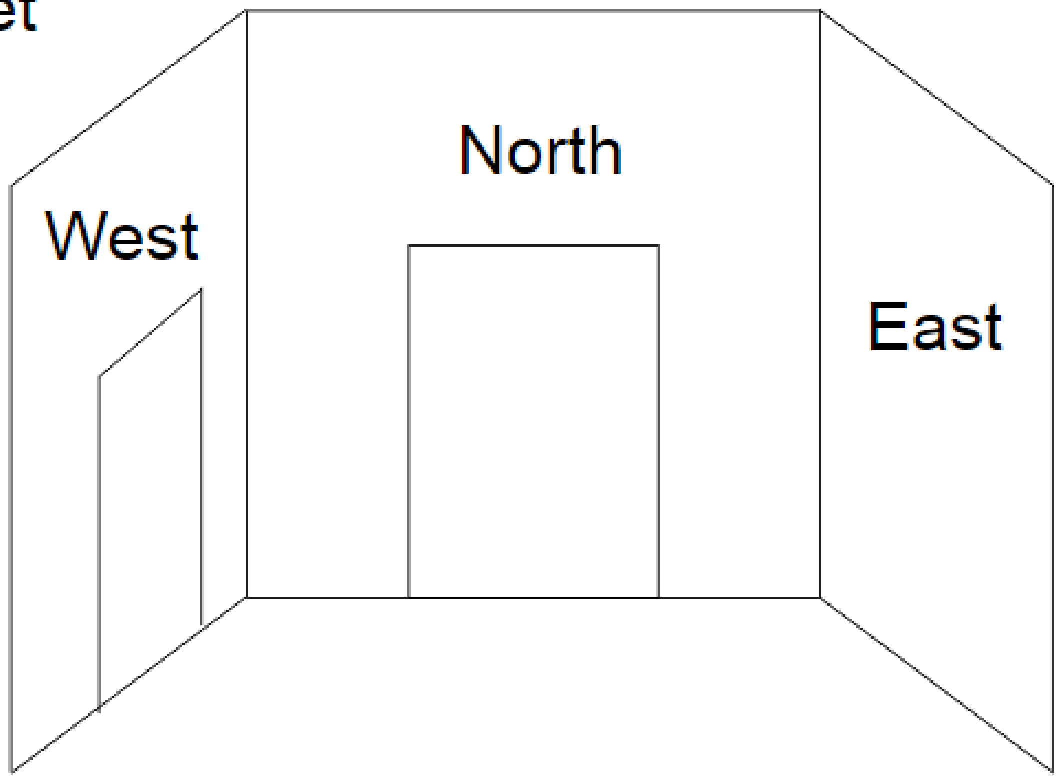
A Simple Maze

- Search maze until way out is found. If no way out possible report that.



A Simple Maze

Which way do I go to get out?



Behind me, to the South is a door leading South

Modified Backtracking Algorithm for Maze

If the current square is outside, return TRUE to indicate that a solution has been found.

Mark the current square.

for (each of the four compass directions) {

if (this direction is not blocked by a wall and the current square is not marked) {

Move one step in the indicated direction from the current square. Try to solve the maze from there by making a recursive call. If this call shows the maze to be solvable, return TRUE to indicate that fact.

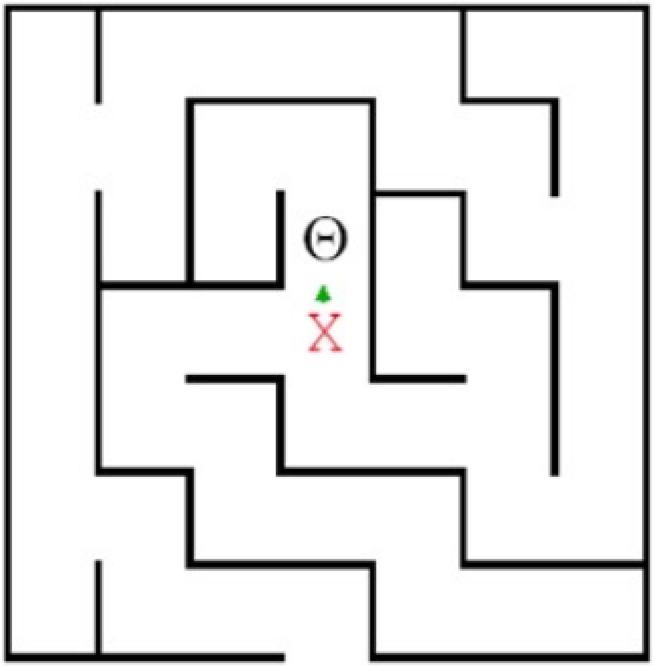
}

}

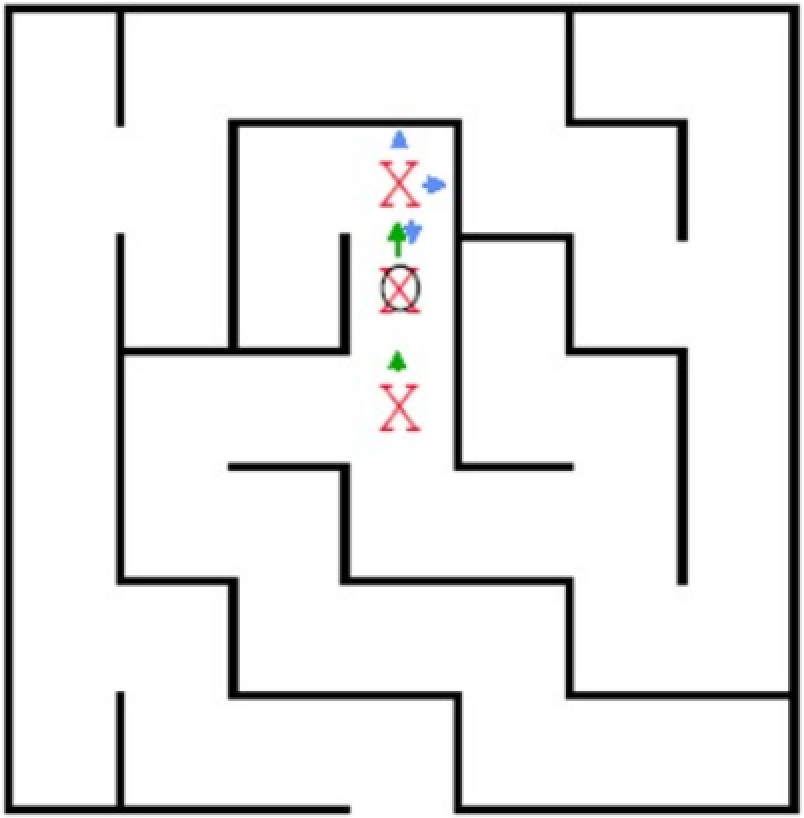
Unmark the current square.

Return FALSE to indicate that none of the four directions led to a solution.

Algorithm for Maze

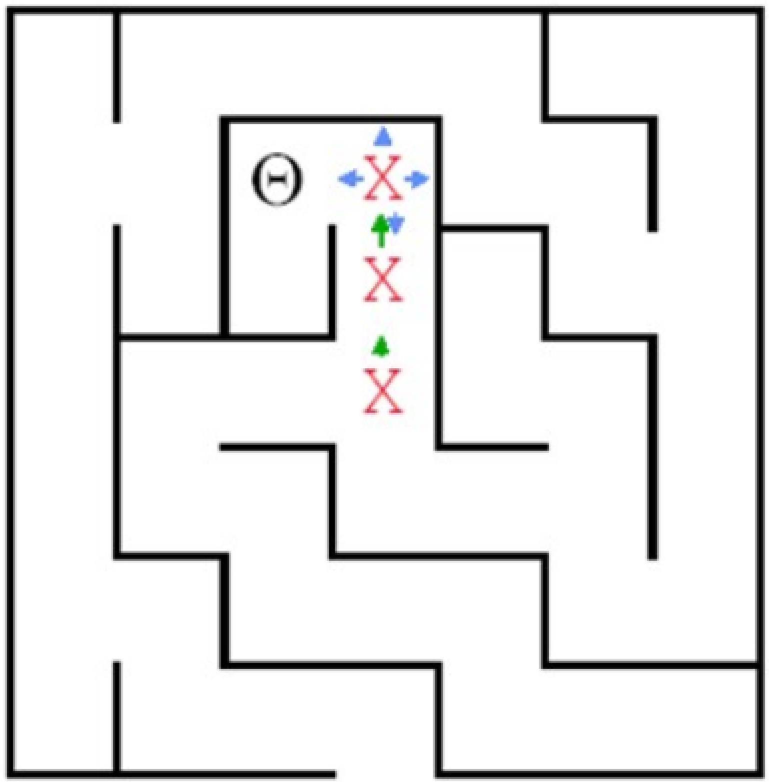


Algorithm for Maze



Here we have moved North again, but there is a wall to the North . East is also blocked, so we try South. That call discovers that the square is marked, so it just returns.

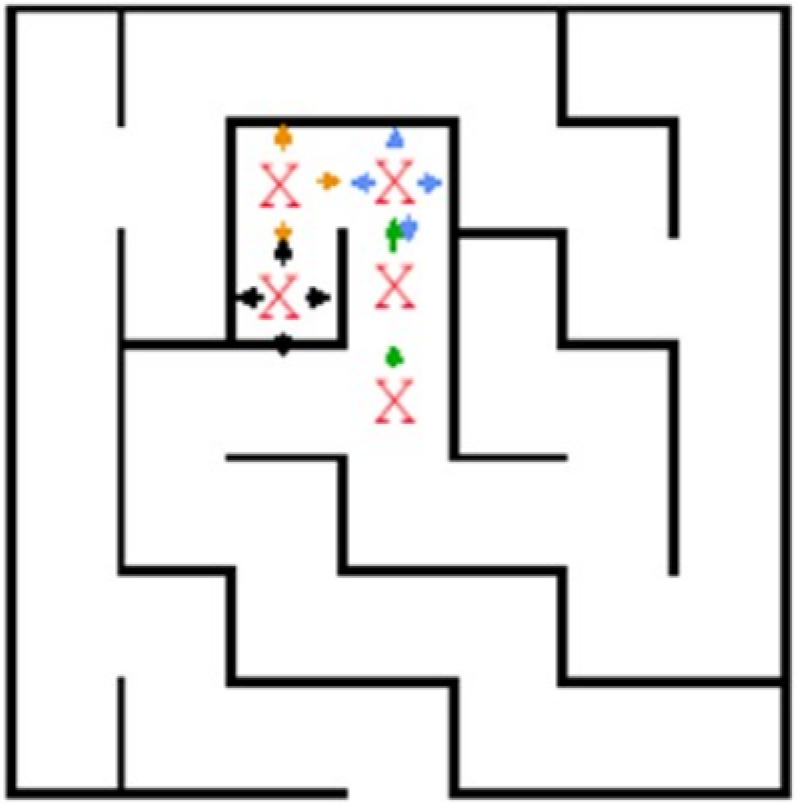
Algorithm for Maze



So the next move we can make is West.

Where is this leading?

Algorithm for Maze

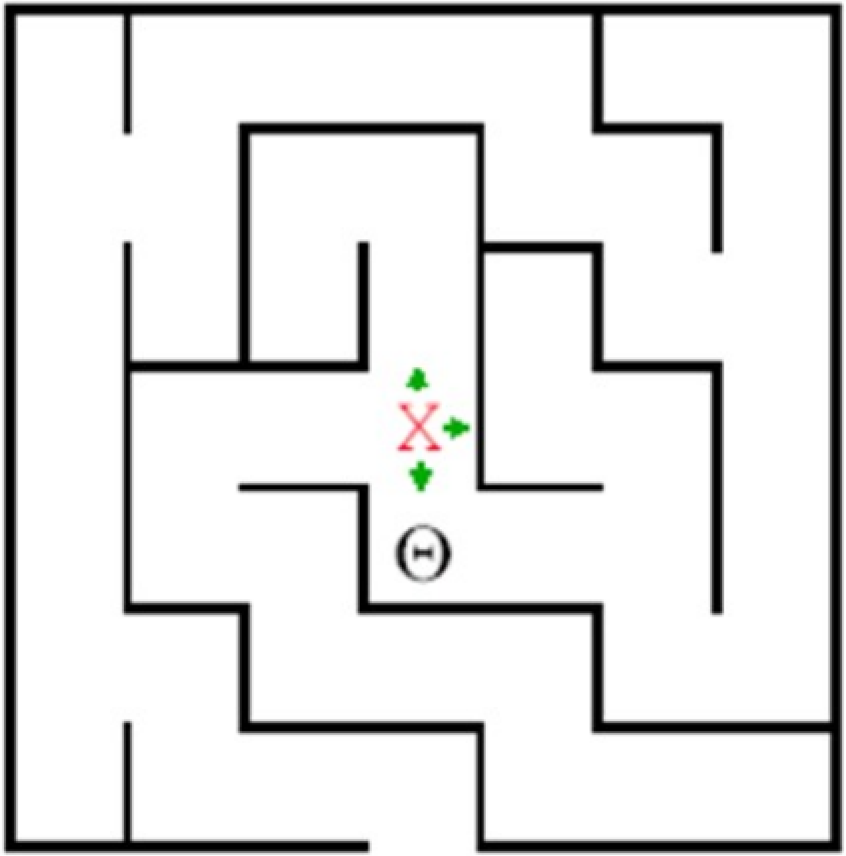


This path reaches a dead end.

Time to backtrack!

Remember the program stack!

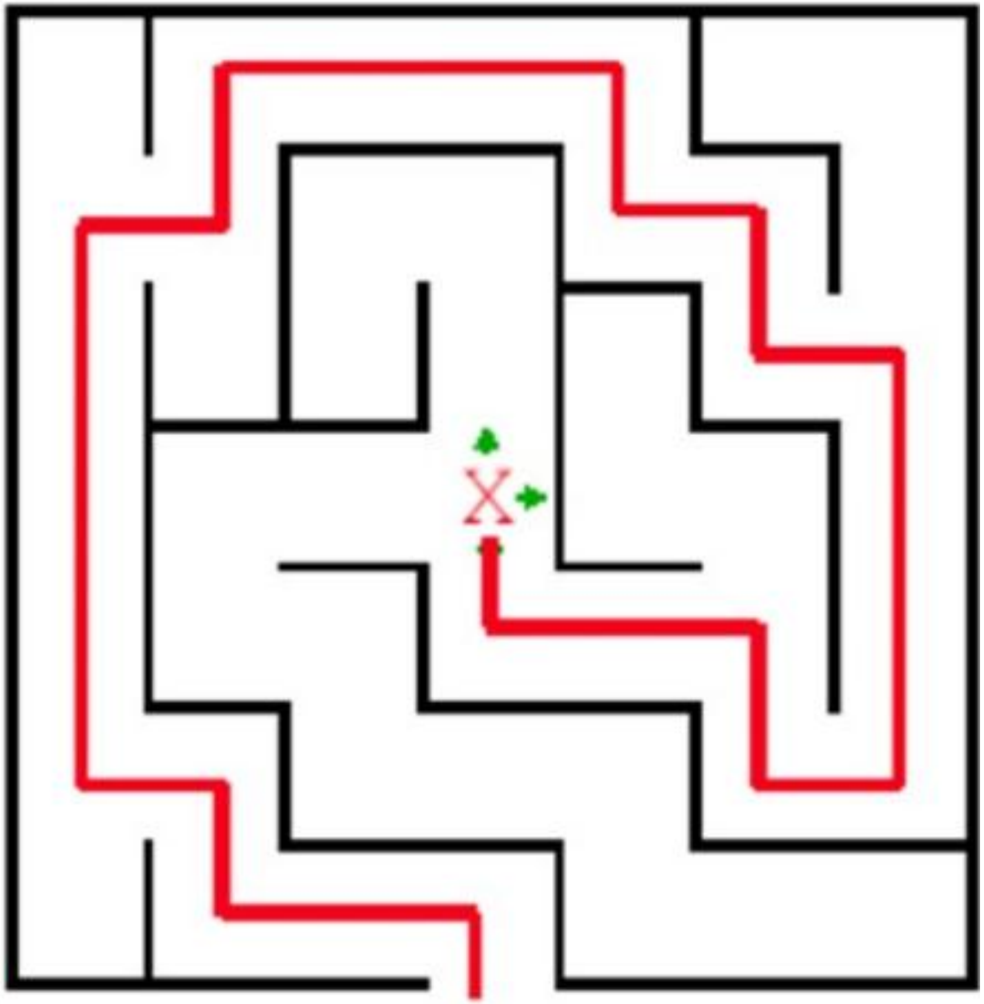
Algorithm for Maze



And now we try
South

Algorithm for Maze

- One Path found



Графи
(свързано представяне)

```
bool way(int a, int b, IntGraph &g, IntList &l)
{
  l.ToEnd(a);
  if (a == b)
  {
    return true;
  }

  elem_link<int> * q = Point(a, g);
  q = q->link;
  while (q)
  {
    if (!member(q->inf, l) && way(q->inf, b, g, l))
      return true;
    q = q->link;
  }

  DeleteLast(l); // връщане назад
  return false;
}
```

O(n!*m)

Графи
(свързано представяне)

- Сложност

- n е брой на върховете на графа.

Намира всички ациклични пътища => сложността може да достигне до n!

- member – линейна сложност

m <u>не</u> е брой на ребрата.

O(n!*m)

Графи

(свързано представяне)

```
bool findPathDFSrec(IntGraph &g, unsigned u, unsigned v,
IntList& visited, LList<unsigned>& path) {
   // обхождаме u
   path.ToEnd(u);
   visited.ToEnd(u);

   if (u == v)
     return true;

  elem_link<int> * q = Point(u, g);
  q = q->link;
  while (q)
  {
     if (!member(q->inf, visited) &&
       findPathDFSrec(g, q->inf, v, visited, path))
       return true;
     q = q->link;
   }
   // отказваме се от u
   DeleteLast(path);
   // visited.remove(u);
   return false;
}
```

O(n*m)

Графи
(свързано представяне)

- Сложност

- n е брой на върховете на графа.

Търсим дали съществува път
Не повтаряме върхове => сложността може да достигне до n

- member – линейна сложност

m <u>не</u> е брой на ребрата.

O(n*m)

Графи
(свързано представяне)

- Сложност

- n е брой на върховете на графа.
- m е брой на ребрата на графа.

Търсенето в дълбочина обхожда върховете и ребрата по веднъж => O(n + m)

!За практическата реализация се използват и други операции, които разглеждат върховете