School of Data and Computer Science, Sun Yat-sen University

E01 Maze Problem

18340149 Xinmeng Sun

September 6, 2020

Contents

1 Task

- **Introduction to Pacman Rule**
  As we all know,Pacman is a popular game among kids, which is translated from an arcade game of the same name to the Atari 2600 platform, it was first launched on the arcade by Nanmong-Gong in 1980 and later released by Atari 2600 by Atari in mid-March 1982.[1]

Figure 1: Pacman Game

The rule is simple to understand: There are walls that you cannot pass and roads you can pass in a maze.The originnal game allows you to eat as many beans in the maze as you can ,while escaping from a monster.

- **Problem desctiption**

  Given a Maze.txt, which contains a matrix consists of an array of 0s and 1s, where 1 means wall and 0 means road.

  Among them we can find letter 'S' and 'E', stands for start point and end point respectively.

  This week's target is to solve the maze problem (i.e., find the shortest path from the start point to the finish point) by using BFS or DFS (Python or C++)

- **My idea**

  The maze can be modeled as an array, so what we need to do is to travel from 'S' to 'E'. The problem can be viewd as a serach problem.

  1. BFS

     First we add start point to frontier, and by visit it ,remove the elements in th frontier while place the successors of the current state at the end of the frontier.

     Because BFS has completeness and optimality, we can surely find the shortest path if it exists.

     To add successors of the current state at the end of the frontier, in C++ the data structure,**queue** can be useful. Queue is an FIFO(first in first out) structure, and once we get the front of the queue, we pop it out and add its neighbours(successors) into the queue.

  2. DFS

     First we add start point to frontier, and by visit it ,remove the elements in th frontier while place the successors of the current state at the front of the frontier.Therefore always expands the deepest node in the frontier.

[1] From Baidu Baike.

Because DFS has NO optimality, it can only return a **feasible solution** it first found. If we wannt to find the shortest path,we can use a global variable to note down the path length every time a feasible solution is found until no possible way.

DFS is implemented using **stack**, which is FILO(first in last out). Stack can help us to add neighbours(successors) at the most front of frontier, so we go deeper and deeper, until a path is found.

- **My solution**

  I choose BFS serch method to solve the problem. And the program is written in C++, output the length of the shortest path, the coordination of passing nodes, and the time duration of BFS.

  In the program , the 0&1 matrix is read from Maze.txt and stored in a matrix. Then I use C++'s STL, queue, to stand for the frontier. Everytime the most front of the queue is chosen and checked and poped, while its neighbours(successors) are added if not out-of-band. If the node is the endpoint, return the steps along the shortest path stored in Node strucure.

  We output the path by recursive function,printPath. We can use a 2D vector to store a node's father's coordinates. When we need to output the passing nodes, we check the vector to find its farther, and grandfather...until the startpoint is found. Then recurse stack is returned one by one, which form a path from start to end.

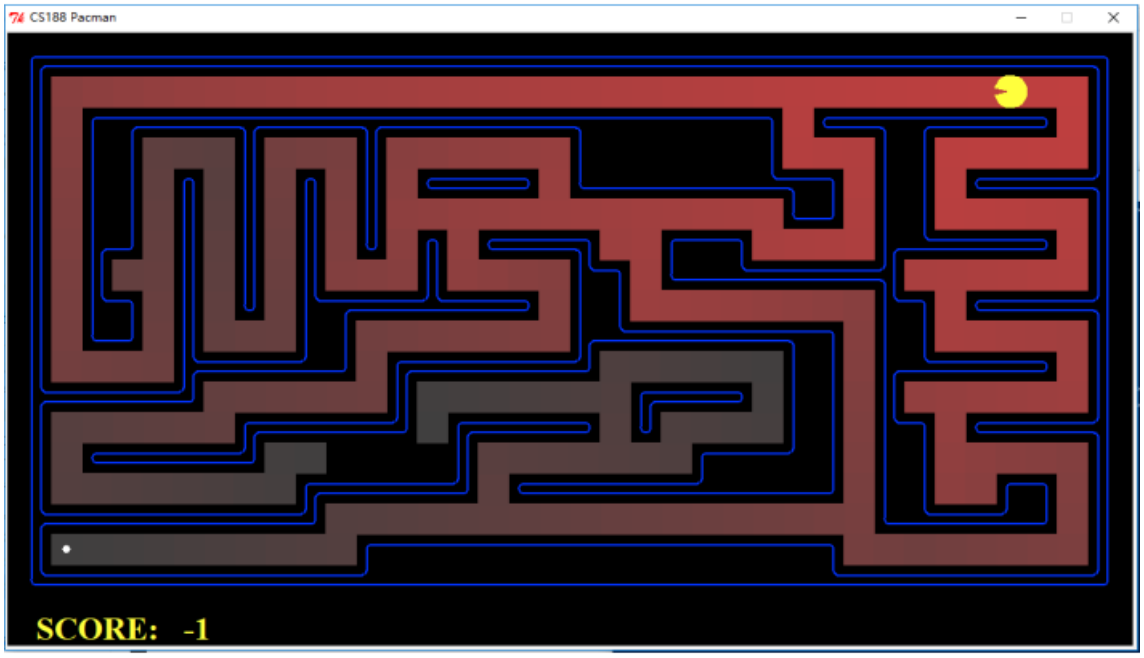

Figure 2: Searching by BFS or DFS

2 Codes

```
/*
*          Name:Sun  Xinmeng
```

```
*       ID:18340149
*       Program name:E01_Maze
*       File name:Maze.cpp
*       Task:BFSsolve Maze problem,output the shortest path length and passing noo
*       Compile:gcc -o maze maze.c
*       Run:./maze
*/

//head files
#include<iostream>
#include<fstream>
#include<queue>
#include<time.h>
#include<vector>

using namespace std;
//18 rows & 36 columns
#define ROW 18
#define COLUMN 36

//structure Node: x and y,steps,pre_x and pre_y
struct Node
{
        int step;
        int x;
        int y;
        int prex;
        int prey;
        Node(){}//default
        Node(int mx, int my, int mystep,int px,int py)
        {
                x = mx;
                y = my;
                step = mystep;
                prex = px;
                prey = py;
        }
};

//direction
int dx[4] = { 1,0,0,-1 };
int dy[4] = { 0,1,-1,0 };

//start and end coordinates
int start_x ;
int start_y;
int end_x ;
int end_y ;
```

```
char maze_matrix[ROW][COLUMN];//read maze from file and put in thw matrix
vector<vector<Node>> prePath(ROW, vector<Node>(COLUMN));//remember pre-node's vecto


/*
*       function state
*       Name: bfs
*       Arguments: void
*       Return: length of shortest path
*       Method: Use queue q, add in startpoint, when not empty, add successors,
*               each time get front of q, hudge if it's endpoint.
*               Ensure it's the shortest, because f BFS's optimality
*       Cycle detection: if a node is visited, mark it '1' in the matrix to avoid re
*/
int bfs()
{
        queue<Node> q;

        Node now_point(start_x, start_y,0,start_x,start_y);
        q.push(now_point);
        prePath[start_x][start_y] = now_point;

        Node cur = q.front();
        while (!q.empty())
        {
                //Node cur = q.front();
                cur = q.front();
                maze_matrix[cur.x][cur.y] = '1';//no re-visit
                q.pop();

                if (cur.x == end_x && cur.y == end_y)
                return cur.step;
                for (int i = 0; i < 4; i++)
                {
                        int mx = cur.x + dx[i];
                        int my = cur.y + dy[i];
                        if (mx >= 0 && mx < ROW && my >= 0 && my < COLUMN && maze_
                        {
                                Node next(mx,my,cur.step+1,cur.x,cur.y);
                                q.push(next);
                                prePath[mx][my] = cur;
                        }
                }
        }
}

/*
*       function state
*       Name: printPath
```

```
*       Arguments:start from coordinates
*       Return:void
*       Method:Recursively find the node before the node
*              until it's the start point
*              and print the path out
*/
void printPath(int x,int y)
{
        if (x == start_x && y == start_y)//end recursive
        {
                cout << " < " << x << " , " << y << " > " << endl;
                return;
        }
        int prex = prePath[x][y].prex;
        int prey = prePath[x][y].prey;
        printPath(prex, prey);
        cout << " < " << x << " , " << y << " > " << endl;
}

/*
*       function state
*       Name:solve_by_BFS
*       Arguments:void
*       Return:void
*       Method:take time before and after bfs(),calculate duration
*              output the least path length
*              call printPath
*/
void solve_by_BFS()
{
        clock_t start, finish;
        double duration;

        start = clock();//start time
        int shortest_path = bfs();
        finish = clock();
        duration = (double)(finish - start) / CLOCKS_PER_SEC;//change to seconds

        cout << "Shortest Path Length:" << shortest_path << endl;
        cout << "BFS Time Duration" << duration << " second" << endl;
        cout << "Tavelling Nodes:" << endl;
        printPath(end_x, end_y);
}

/*
*       function state
*       Name:main
*       Arguments:void
*       Return:0
```

```
*/
int main(void)
{
        cout << "----------MAZE PROBLEM----------" << endl;

        //initialize
        ifstream infile;//read-only
        infile.open("MazeData.txt", ios::in);
        if (!infile)
        {
                cout << "Error!" << endl;
                return 0;
        }

        for (int i = 0; i < ROW; i++)
        {
                for (int j = 0; j < COLUMN; j++)
                {
                infile >> maze_matrix[i][j];//read and fill in matrix

                //start and end point
                if (maze_matrix[i][j] == 'S')
                {
                        start_x = i;
                        start_y = j;
                }
                else if (maze_matrix[i][j] == 'E')
                {
                        end_x = i;
                        end_y = j;
                }
        }
}

//output
cout << "start from ( " << start_x << " , " << start_y << " ) point" << endl;
cout << "go to ( " << end_x << " , " << end_y << " ) point" << endl<<endl;

solve_by_BFS();
return 0;
}
```

3 Results

```
----------MAZE PROBLEM----------
start from ( 1, 34 ) point
go to ( 16, 1 ) point

Shortest Path Length:68
BFS Time Duration0.001 second
Tavelling Nodes:
 < 1 , 34 >
 < 1 , 32 >
 < 1 , 30 >
 < 1 , 28 >
 < 1 , 26 >
 < 2 , 25 >
 < 3 , 26 >
 < 4 , 27 >
 < 6 , 27 >
 < 6 , 25 >
 < 5 , 24 >
 < 5 , 22 >
 < 5 , 20 >
 < 7 , 20 >
 < 8 , 21 >
 < 8 , 23 >
 < 8 , 25 >
 < 8 , 27 >
 < 10 , 27 >
 < 12 , 27 >
 < 14 , 27 >
 < 15 , 26 >
 < 15 , 24 >
 < 15 , 22 >
 < 15 , 20 >
 < 15 , 18 >
 < 15 , 16 >
 < 15 , 14 >
 < 15 , 12 >
 < 15 , 10 >
 < 16 , 9 >
 < 16 , 7 >
 < 16 , 5 >
 < 16 , 3 >
 < 16 , 1 >
```

Analysis Through the output, we got that the travel of Pacman from start point(1,34) to end point(16,1) , the shortest path's length is 68. And time cost is 0.001 second.

The points passed are listed in the picture.

4 What I learnt

This is the first time that we try our little AI experiment, and also it's my first time to use LaTeX typesetting. Although there's still a lot to learn, I'm really happy to grab this new skill. Latex typesetting is reaaly convenient and helpful.

Search helps us to solve many AI problems. And what deserves our attention is how to abstract the real problem into a serach problem . After that ,the search method like BFS and DFS and Iteratively deeping serch can perform their properties.

BFS search is a kind of greedy algorithm, which ensures it can always find the shortest path. On the contrast, DFS can not, but it outperforms at space complexity