1. Title and Authors

Assignment 3

110704008 羅瑋翔

xianglo9121.mg10@nycu.edu.tw

2. Statement of The Problems

In this assignment, I have developed an algorithm to determine the shortest path from a designated start cell, denoted as 's,' to a destination cell, denoted as 't,' within a maze of dimensions m by n. The fundamental approach employed in solving this problem is based on the Breadth-First Search (BFS) paradigm.

The algorithm utilizes an empty queue, implemented as a class with essential functions such as pop, peek, push, and empty. The queue contains a struct called 'node,' encapsulating the x and y positions of the cell within the maze and the distance from the start point to the current node. The process involves marking visited positions to avoid redundant exploration. The algorithm continues until the queue is empty. Upon reaching the destination cell, a reverse function is invoked to retrieve the shortest path.

For the handling of various input scenarios, I have designed multiple data types:

(1) Maze with start cell, end cell, and obstacles where the start cell cannot reach the end cell.
(2) Maze without a start cell but with an end cell and obstacles.
(3) Maze with obstacles but without a start cell and end cell.
(4) A 6x5 maze with a start cell, end cell, and obstacles, wherein the end cell is reachable from the given start cell.
(5) A 4x5 maze with a start cell, end cell, and obstacles, where the end cell is reachable from the given start cell.
(6) A scenario where the start cell is initially covered by an obstacle, and an end cell is present. If the obstacle is positioned after inputting the start cell, the end cell can be reached; otherwise, it cannot.
(7) A situation with a start cell, obstacles, but without an end cell, making it impossible to reach a destination from the given start cell.

In conclusion, the algorithm is versatile and accommodates diverse maze configurations, providing a robust solution for finding the shortest path from a specified start cell to a destination cell.

3. Main Results.

(a) At the outset, I define a struct to encapsulate the x position, y position, and the distance from the start point to the current node. Additionally, a class named Queue is established to store instances of this struct, referred to as 'node,' with essential functions such as push (to add a new node at the rear), pop (to remove the node at the front), and empty (to check if the queue is empty).

The algorithm includes a function named isValid, which verifies the validity of the input data. The central function, findTheShortestPath, takes an m by n vector and two pairs indicating the positions of the start cell and the destination cell. Initially, a vector of the same size is created to record whether a position has been visited. Subsequently, a queue is employed to store the points that need processing. The algorithm iterates until all nodes in the queue are processed and the queue becomes empty. The reverse function is then called to reverse the elements of a vector path, utilizing a two-pointer approach. Finally, the function returns a pair representing the shortest path from the start cell to the destination cell. Or we can implement another method, like reverse the start cell and the end cell. By this, we will get the same result.

In the main function, the user inputs the dimensions of the maze (m and n) and initializes a maze vector with all elements set to 1. The input format involves specifying the start cell, obstacle cells, and the destination cell. A while loop with conditionals is employed to parse the input, ensuring proper handling of different data types. If 's' is encountered, a pair is created to store the position of the start cell; if 't' is encountered, another pair is created for the destination cell. Notably, when an obstacle cell is read, the corresponding position in the maze vector is assigned a value of 0. The final step involves invoking the findTheShortestPath function to obtain and display the shortest path.

(b) I used an array to save four ways to go from the point (1, 0), (-1. 0), (0, 1), (0, -1), respectively. Creating a struct to save the value of the x position, y position and the distance. And create a class queue to store the struct, node. This class provided 4 main member function (1) push: send a node to this function and save it at rear of the array. (2) pop: increase the value of "front" (which store the index of the front element), this step like removing element from the front of the array. (3) peek: used this function to get the node at the front of the array. (4) empty: to check if the queue is empty. Also, I used vector to save the data of the maze and record whether the position of the maze had been visited.

(c) List of your program with comments.

```
#include <iostream>
#include <vector>
using namespace std;
```

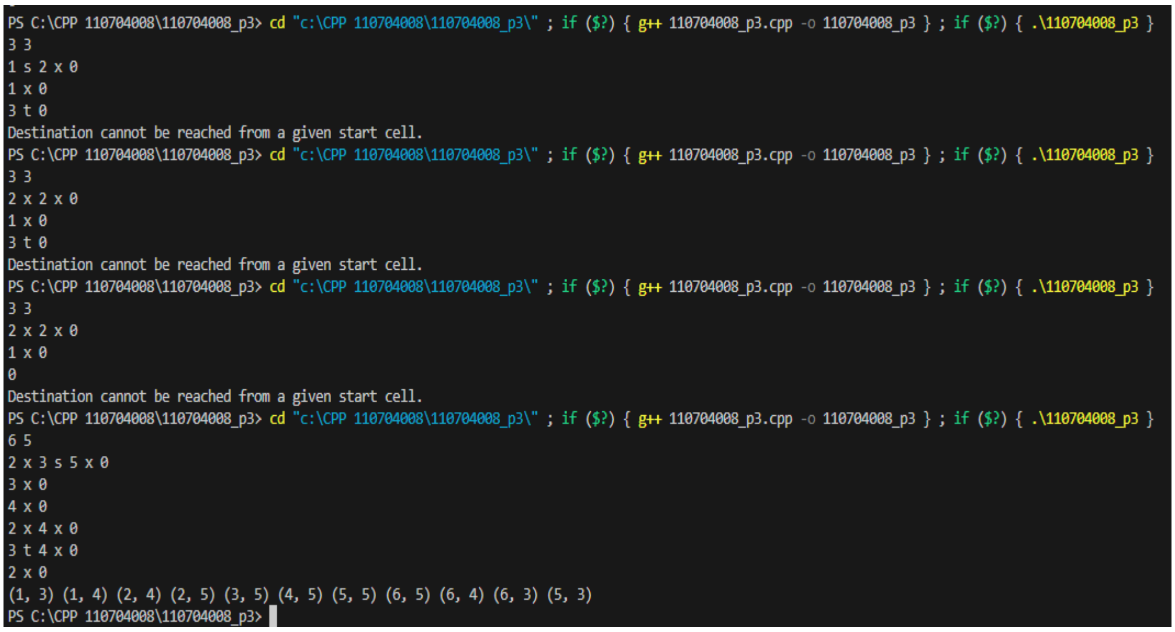
```
const int MAX_SIZE = 1000;//this can change the size of the queue,
since the maze is not too big, so i only set 1000.

struct Node{
    //this structuct is used to write down the position of the node
    //and the distance between the start point and the node
    int x, y, dist;
};

class Queue{
public:
    Queue();//constructor
    void push(Node *node);//add new node to the array.
    void pop();//subtracting the value of the index at rear like
removing the node from the array.
    Node& peek();//to get the node from the array at front.
    bool empty()const;//check if the queue is empty.
private:
    Node arr[MAX_SIZE];
    int front, rear;
};

//the possible ways of moving
int row[] = {-1, 0, 0, 1};
int col[] = {0, -1, 1, 0};

//check if it can reach the position(row, col).
//return false if the position is not valid or the value is 0 or the
positio had been visited.
bool isValid(vector<vector<int>> const &maze, vector<vector<bool>>
&visited, int r, int c);

//find the shortest path between the start cell and the end cell.
vector<pair<int, int>> findTheShortestPath(vector<vector<int>> const
&maze, pair<int, int> &src, pair<int, int> &dest);

void reverse(vector<pair<int, int>>& path);
```

```
int main(){
    int m, n;
    int posCol, posRow;
    char type;

    //construct and m by n maze
    cin >> m >> n;
    //construct an m by n metric and assign 1 as the value of the
elements inside.
    vector<vector<int>> maze(m, vector<int>(n, 1));

    pair<int, int> src = make_pair(-1, -1);
    pair<int, int> dest = make_pair(-1, -1);
    for(posRow = 0; posRow < m; posRow++){
        while(cin >> posCol && posCol != 0 && posCol <= n){
            cin >> type;
            if(type == 's')
                src = make_pair(posRow, posCol - 1);
            else if(type == 't')
                dest = make_pair(posRow, posCol - 1);
            else if(type == 'x')
                maze[posRow][posCol - 1] = 0;
            else
                maze[posRow][posCol] = 1;
        }
    }
    if(src.first == -1 || dest.first == -1)
        cout << "Destination cannot be reached from a given start cell."
<< endl;
    vector<pair<int, int>> shortestPath = findTheShortestPath(maze, src,
dest);
    if(!shortestPath.empty()){
        for(auto &path : shortestPath){
            cout << "(" << path.first << ", " << path.second << ")" << "
";
        }
        cout << endl;
    }
    else{
        cout << "Destination cannot be reached from a given start cell."
<< endl;
    }
    maze.clear();

    return 0;
}

bool isValid(vector<vector<int>> const &maze, vector<vector<bool>>
&visited, int r, int c){
    return (r >= 0 && r < maze.size()) && (c >= 0 && c < maze[0].size())
&& maze[r][c] && !visited[r][c];
}

vector<pair<int, int>> findTheShortestPath(vector<vector<int>> const
&maze, pair<int, int> &src, pair<int, int> &dest){
    if(maze.size() == 0 || maze[src.first][src.second] == 0)
        return {};
    int M = maze.size();
    int N = maze[0].size();

    //create this 2D vector to records whether the giving position had
been visited.
    vector<vector<bool>> visited(M, vector<bool>(N, false));
    vector<vector<pair<int, int>> > parent(M, vector<pair<int, int>>(N,
make_pair(-1, -1)));

    //save the node that need to be dealt
    Queue q;
    int i = src.first;
    int j = src.second;
    Node tempNode;
    tempNode.x = i;
    tempNode.y = j;
    tempNode.dist = 0;
    visited[i][j] = true;
    q.push(&tempNode);
    //if the queue q is empty, means there is no node need to be dealt.
    while(!q.empty()){
        Node *node = &q.peek();
        q.pop();//remove the first node that had already been gotten by
the above line.
        int i = node->x, j = node->y, dist = node->dist;
        //if the x and y coordinates of the node equal to the x and y
coordinates of destination, we have the shortest path.
        if(i == dest.first && j == dest.second){
            vector<pair<int, int>> path;
            while (i != -1 && j != -1) {
                path.push_back(make_pair(i + 1, j + 1));
                int new_i = parent[i][j].first;
                int new_j = parent[i][j].second;
                i = new_i;
                j = new_j;
            }
            reverse(path);
            return path;
        }
        //since there are as most four ways of the node to go, I used
for loop to deal with it.
        for(int k = 0; k < 4; k++){
            if(isValid(maze, visited, i + row[k], j + col[k])){
                Node newNode;
                visited[i + row[k]][j + col[k]] = true;
                parent[i + row[k]][j + col[k]] = make_pair(i, j);
                newNode.x = i + row[k];
                newNode.y = j + col[k];
                newNode.dist = dist + 1;
                q.push(&newNode);
            }
        }
    }
    return {};
}
```

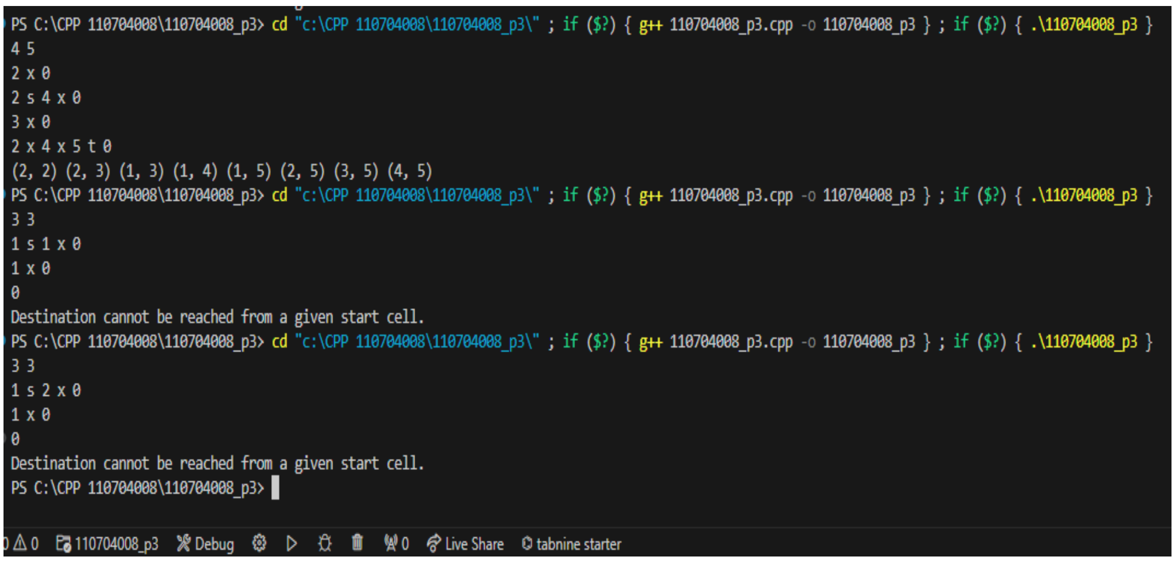
```
//by this function, i can record shortest path from the start point to
the end point
void reverse(vector<pair<int, int>>& path){
    int start = 0;
    int end = path.size() - 1;

    while(start < end){
        swap(path[start], path[end]);
        ++start;
        --end;
    }
}

Queue::Queue(){
    front = -1;
    rear = -1;
}

void Queue::push(Node* node){
    if(rear == (MAX_SIZE - 1)){
        cout << "Queue is overflow!" << endl;
        return;
    }
    if(empty()){
        front = 0;
        rear = 0;
    }
    else
        rear++;
    arr[rear] = *node;
}

void Queue::pop(){
    if(empty()){
        cout << "Queue is empty!" << endl;
        return;
    }
    if(front == rear){
```

```cpp
        front = -1;
        rear = -1;
    }
    else
        front++;
}
Node& Queue::peek(){
    if(empty())
        cout << "Queue is empty." << endl;
    return arr[front];
}

bool Queue::empty()const{
    return front == -1;
}
```

(d)Input data and the Output



```
PS C:\CPP 110704008\110704008_p3> cd "c:\CPP 110704008\110704008_p3\" ; if ($?) { g++ 110704008_p3.cpp -o 110704008_p3 } ; if ($?) { .\110704008_p3 }
3 3
1 s 2 x 0
1 x 0
3 t 0
Destination cannot be reached from a given start cell.
PS C:\CPP 110704008\110704008_p3> cd "c:\CPP 110704008\110704008_p3\" ; if ($?) { g++ 110704008_p3.cpp -o 110704008_p3 } ; if ($?) { .\110704008_p3 }
3 3
2 x 2 x 0
1 x 0
3 t 0
Destination cannot be reached from a given start cell.
PS C:\CPP 110704008\110704008_p3> cd "c:\CPP 110704008\110704008_p3\" ; if ($?) { g++ 110704008_p3.cpp -o 110704008_p3 } ; if ($?) { .\110704008_p3 }
3 3
2 x 2 x 0
1 x 0
0
Destination cannot be reached from a given start cell.
PS C:\CPP 110704008\110704008_p3> cd "c:\CPP 110704008\110704008_p3\" ; if ($?) { g++ 110704008_p3.cpp -o 110704008_p3 } ; if ($?) { .\110704008_p3 }
6 5
2 x 3 s 5 x 0
3 x 0
4 x 0
2 x 4 x 0
3 t 4 x 0
2 x 0
(1, 3) (1, 4) (2, 4) (2, 5) (3, 5) (4, 5) (5, 5) (6, 5) (6, 4) (6, 3) (5, 3)
PS C:\CPP 110704008\110704008_p3>
```



```
PS C:\CPP 110704008\110704008_p3> cd "c:\CPP 110704008\110704008_p3\" ; if ($?) { g++ 110704008_p3.cpp -o 110704008_p3 } ; if ($?) { .\110704008_p3 }
4 5
2 x 0
2 s 4 x 0
3 x 0
2 x 4 x 5 t 0
(2, 2) (2, 3) (1, 3) (1, 4) (1, 5) (2, 5) (3, 5) (4, 5)
PS C:\CPP 110704008\110704008_p3> cd "c:\CPP 110704008\110704008_p3\" ; if ($?) { g++ 110704008_p3.cpp -o 110704008_p3 } ; if ($?) { .\110704008_p3 }
3 3
1 s 1 x 0
1 x 0
0
Destination cannot be reached from a given start cell.
PS C:\CPP 110704008\110704008_p3> cd "c:\CPP 110704008\110704008_p3\" ; if ($?) { g++ 110704008_p3.cpp -o 110704008_p3 } ; if ($?) { .\110704008_p3 }
3 3
1 s 2 x 0
1 x 0
0
Destination cannot be reached from a given start cell.
PS C:\CPP 110704008\110704008_p3>
```

(4) Conclusions.

Through this assignment, I have gained a solid understanding of the Breadth-First Search (BFS) method. This technique proves valuable in maintaining a systematic order of exploration, ensuring that nodes are visited in layers and allowing for efficient traversal of vertices. The concept of BFS becomes particularly handy in scenarios involving layered exploration, shortest path determination, and similar problem-solving contexts.

Moreover, this assignment has provided me with a deeper familiarity with the data structure 'queue.' By constructing my own queue class and implementing essential functions such as push, pop, and empty, I delved into the inner workings and properties of this fundamental data type. This hands-on experience allowed me to comprehend the underlying mechanisms of a queue, enhancing my overall understanding of data structures.

Furthermore, the assignment necessitated the implementation of functions, such as the reverse function, which is typically available in libraries. This requirement prompted me to investigate and comprehend the functionality and underlying principles of these functions. While working without the convenience of pre-existing libraries might pose challenges, it has proven to be a valuable learning experience, as it encourages a deeper understanding of the core concepts behind the algorithms and data structures used. Overall, this assignment has equipped me with practical knowledge and problem-solving skills that will prove beneficial in future endeavors involving layered exploration and pathfinding algorithms.