

AgentAI_OJordan_P144558

by Oliver Jordan

FILE	2911_OLIVER_JORDAN_AGENTAI_OJORDAN_P144558_110679_776738 261.DOCX		
TIME SUBMITTED	29-MAR-2018 06:09PM (UTC+0100)	WORD COUNT	4077
SUBMISSION ID	85830501	CHARACTER COUNT	27743

Search Algorithms
Agent AI
Oliver Jordan (P144558)
26/3/18

Contents

1.0	Introduction.....	3
2.0	Important Algorithms and Implementation	3
2.1	Stack.h and Stack.cpp	3
2.2	friendlyBotStates.cpp – Searching::Execute	3
2.3	Main.cpp.....	3
3.0	User Guide	4
4.0	Tests and Screenshots.....	4
5.0	References	7
6.0	Program Listing	8

1.0 Introduction

This report outlines the key algorithms used in the submission. It includes a brief user guide for the program, along with a program listing and a test schedule. Screenshots of the program working are provided.

2.0 Important Algorithms and Implementation

The following are the key algorithms and code implementations used to achieve the desired goal.

2.1 Stack.h and Stack.cpp

A depth first search (DFS henceforth) was used to let the AI search the maze efficiently and believably. This search requires the use of a stack data structure, coded in Stack.h and Stack.cpp. It contains an array of a predefined max size, and functions to push onto the stack and pop off it. It will print an error to console if the stack is full or empty.

It should be noted that a DFS has been used as opposed to a breadth first search (BFS henceforth) for one key reason. When using a BFS (and swapping out the stack structure for a queue structure) the AI would visit the maze's tiles in a way that made it appear to be teleporting around the maze. Since an AI character would not normally be able to search like that, despite the method finding the shortest route, DFS was chosen over BFS.

2.2 friendlyBotStates.cpp – Searching::Execute

Before continuing with this section, it is worth noting that the logic used to search the maze could and should be decoupled into its own function in further iterations of the code.

The AI's execute function for the searching state contains the main logic used to traverse the maze and discover its tiles. It does this by looking at the four tiles surrounding the tile the bot is currently at and updating its own map of the maze with the correct information. When it discovers a tile it hasn't seen before, it adds that tile to the stack of positions not yet visited – i.e. it carries out a DFS on all tiles it can walk on. Once it's looked at all four tiles around itself it will visit the last position added to the stack – normally a tile next to the one it is currently at. The AI achieves a believable walking route through this method. If the AI has met a dead end, it will zip back to the last junction in the maze and visit one of the routes not yet visited. Further iterations of this code should consider allowing the AI to walk back along already visited nodes.

A graph structure is filled with tile positions and information as the AI searches. While this graph is not used for any other implementation, it is the frame work for the attempted implantation of Dijkstra's Algorithm [1] [2] found in Dijkstra.h and Dijkstra.cpp.

2.3 Main.cpp

Main.cpp should be noted for its implementation of a simple and clear UI for the program. A case switch is implanted and awaits user input to direct the program to the

desired sections. The user input also acts as a placeholder for some of the AI checks – for example, the user decides if the bot should no longer be idle, whereas a full implementation of the bit could check things such as time, entities close by etc.

Main.cpp will also hold the AI after every update to allow the user to see the changes in the AI's states clearly.

3.0 User Guide

The program will take a maze as described in a text file (found in "maze.txt") and allow an AI to search for a target tile within it. Upon starting the program, options are presented. Type the number corresponding to the desired option and hit enter. At any point, to exit the program, -1 can be typed. Depending on the state of the AI different options will appear.

To compile the program, simply use the Visual Studio 2015 tools as would be standard for any other program. For reference, the program is confirmed to run on most any computer running a standard VS2015 install

4.0 Tests and Screenshots

The following tests show the program working as intended and provides screenshots of key points of the program during these tests.

Test 1: Loading the maze

Details: maze.txt contains a representation of a maze. Program should load the maze and show the full maze correctly.

Outcome: Success, as intended.

Screenshots:

```
maze - Notepad
File Edit Format View Help
#width 20
#height 16
#begin
| | | | | | | | | | | | | | | | | |
| S | f | f | f | f | f | f | f | f | f | f | f | f | f | f | f | f |
| f | f | f | | | | | | | | | | f |
| f | f | f | | | | | | | | | | f |
| f | | f | | | | | | | | | | f |
| f | | f | | | | | | | | | | f |
| f | f | f | f | | | | | | | | | f |
| f f | f | f | f | | | | | | | | f |
| f | | | | f | f f f G f f f f f |
| f f f f f f f f | | | | | | | | f |
| f | | | | f | | | | | | | | | f |
| f | | | | f | | | | | | | | | f |
| f | | | | f | | | | | | | | | f |
| f | | | | f | | | | | | | | | f |
| f f f f f f f f f f f f f f f f |
| | | | | | | | | | | | | | | | | |
```

Fig.1: The maze.txt text file components.

```
C:\Users\OI\\Desktop\AgentAI - OIORDAN\Debug\AgentAIFinal.exe
Load maze1 = 1, quit = -1
1
| | | | | | | | | | | | | | | | | |
| S | f | f | f | f | f | f | f | f | f | f | f | f | f | f | f | f |
| f | f | f | | | | | | | | | | f |
| f | f | f | | | | | | | | | | f |
| f | | f | | | | | | | | | | f |
| f | | f | | | | | | | | | | f |
| f | f | f | f | | | | | | | | | f |
| f f | f | f | f | | | | | | | | f |
| f | | | | f | f f f G f f f f f |
| f f f f f f f f | | | | | | | | f |
| f | | | | f | | | | | | | | | f |
| f | | | | f | | | | | | | | | f |
| f | | | | f | | | | | | | | | f |
| f | | | | f | | | | | | | | | f |
| f f f f f f f f f f f f f f f f |
| | | | | | | | | | | | | | | | | |
Maze loaded!
Begin search for Goal = 2, quit = -1
```

Fig.2: The result of loading the maze. Printed maze is identical, as intended.

Test 2: AI searches for goal

Details: The AI should search the maze until it reaches the goal tile.

Outcome: Success, as intended.

Screenshots:

```
C:\Users\Olvi\Desktop\AgentAI - OJORDAN\Debug\AgentAIFinal.exe
```

```
#|||||||#####  
|Sfffffffffffffff|#####  
#|f|f|f|||||||f|#####  
#####|f|#####  
#####|f|#####  
#####|f|#####  
#####|f|#####  
#####| || |f|#####  
#####G@ffff|#####  
#####| || f #####  
#####  
#####  
#####  
#####  
#####  
#####  
#####  
  
Bot update completed. Continue = 2, quit = -1  
2  
  
I'm done searching boss!  
I've found it boss!  
Bot update completed. Continue = 2, quit = -1  
2  
  
We have the goal, want to explore the rest of the maze? (1 for return to idle. 2 for continue search)2_
```

Fig.3: The maze as known by the Bot upon finding the goal is shown, along with the bot reporting its success.

Test 3: Bot searches entire maze after finding the goal

Details: The AI can be directed to search the whole maze after finding the goal. Bot should only show all floor and surrounding wall tiles, as it cannot see a wall behind a wall.

Outcome: Success, as intended.

Screenshots:

6.0 Program Listing

Dijkstra.h

```
#pragma once
#include "node.h"
#include "graph.h"

//DIJKSTRA IS NOT IMPLEMENTED IN MAIN
class Dijkstra
{
public:
    Node * unvisited[1000];
    Node * shortest[1000];
    Position shortPath[1000];
    Dijkstra();
    int srtCount2;
    bool calculated;

    Graph Calculate(Graph inGraph, Position start);
    void findShortest(Position goal, Graph inGraph);
};
```

friendlyBot.h

```
#pragma once
class State;
#include "Stack.h"
#include "Position.h"
#include "Maze.h"
#include "graph.h"
class friendlyBot
{
private:
    State* ptrCurState;

public:
    friendlyBot(Position start, Maze* ptr);
    Position actual;
    Maze* ptrMaze;
    ~friendlyBot();
    void Update();
    void ChangeState(State* ptrNextState);
    bool readyToGo();
    bool foundGoalWhatNext();
    Stack knownPositions;
    Position lastVisited;
    char discovered[50][50];
    Graph initialGraph;

    Position route[100];
    int routeCount;
    Position goalpos;
    bool foundGoal;
};
```

friendlyBotStates.h

```
#pragma once
#include "State.h"
class friendlyBot;

class Idle : public State
{
    //contains states for the ai to use, a static instance, and an enter exit and
    execute function
public:
    Idle(){};
    static Idle* Instance();
    virtual void Enter(friendlyBot*);
    virtual void Execute(friendlyBot*);
    virtual void Exit(friendlyBot*);
};

class Searching : public State
{
public:
    Searching() {};;
    static Searching* Instance();
    virtual void Enter(friendlyBot*);
    virtual void Execute(friendlyBot*);
    virtual void Exit(friendlyBot*);
};

class Success : public State
{
public:
    Success() {};;
    static Success* Instance();
    virtual void Enter(friendlyBot*);
    virtual void Execute(friendlyBot*);
    virtual void Exit(friendlyBot*);
};
```

graph.h

```
#pragma once
#include "node.h"
class Graph
{
public:
    Graph();
    Node nodelist[2500];
    int nodeCount;
};
```

Maze.h

```
#pragma once
#include <string>
#include <stdio.h>
#include "Position.h"
using namespace std;
class Maze
```

```

{
public:
    Maze(char * infile);
    ~Maze();
    char tile[50][50];
    int width;
    int height;
    Position Start;
    Position Goal;
private:
    char * file;
};

```

node.h

```

#pragma once
#include "Position.h"
class Node
{
public:
    Node *link[4];
    int linkCount;
    char key;
    Position pos;
    bool visited;
    int distance;
    Node();
    void AddLink(Node *toLink);
};

```

Position.h

```

#pragma once
class Position
{
public:
    int x;
    int y;
    Position() { x = 0; y = 0; }
};

```

Stack.h

```

#pragma once
#define STACK_SIZE 500
#include "Position.h"
class Stack
{
public:
    Stack();
    ~Stack();
    void push(Position x);
    Position pop();
    int stackCount;
    Position * pstnList;
    bool isEmpty();
private:

```

```

        bool isFull();

};

```

State.h

```

#pragma once

class friendlyBot;

class State
{
public:
    virtual void Enter(friendlyBot*) = 0;
    virtual void Execute(friendlyBot*) = 0;
    virtual void Exit(friendlyBot*) = 0;
};

```

Dijkstra.cpp

```

#include "Dijkstra.h"
#include <iostream>
using namespace std;
//DJIKSTRA IS NOT IMPLEMENTED IN MAIN
Dijkstra::Dijkstra()
{
    calculated = false;
    int srtCount2 = 0;
}

//The main concept behind the calculate function was to look at the graph generated by
//the bot and create a new graph with one way links leading from the
//goal node back to the start node. As each node with the shortest path from a neighbour
//node was discovered it would link back to the source neighbour node. This
//is an incorrect implementation of djisktra's algorithm - the intention was to use a
//similar system but have the returning link chain be the true shortest route, as opposed
//to the shortest route at each step of the route.
Graph Dijkstra::Calculate(Graph inGraph, Position start)
{
    Graph SRT;
    int srtCount = 0;
    Node * shortestDist = nullptr;
    int unvisitedCount = 0;
    int current = 0;
    for (int i = 0; i < inGraph.nodeCount; i++)
    {
        if (inGraph.nodelist[i].pos.x == start.x && inGraph.nodelist[i].pos.y ==
start.y)
        {
            inGraph.nodelist[i].distance = 0;
        }
        else
        {
            inGraph.nodelist[i].distance = 100000;
        }
    }
}

```

```

        unvisited[i] = &inGraph.nodelist[i];
        unvisitedCount++;
    }

    while (unvisitedCount != 0)
    {
        for (int j = 0; j < unvisitedCount; j++)
        {
            if (shortestDist == nullptr)
            {
                shortestDist = unvisited[j];
                current = j;
                SRT.nodelist[srtCount] = *shortestDist;
                for (int z = 0; z < 4; z++)
                {
                    SRT.nodelist[srtCount].link[z] = nullptr;
                }
                srtCount++;
            }
            else if (unvisited[j]->distance < shortestDist->distance)
            {
                shortestDist = unvisited[j];
                current = j;
            }
        }
        unvisited[current] = nullptr;
        for (int h = current; h < unvisitedCount - 1; h++)
        {
            unvisited[h] = unvisited[h + 1];
        }
        unvisitedCount--;
        for (int k = 0; k < shortestDist->linkCount; k++)
        {
            for (int l = 0; l < unvisitedCount; l++)
            {
                if (shortestDist->link[k]->pos.x == unvisited[l]->pos.x &&
                    shortestDist->link[k]->pos.y == unvisited[l]->pos.y)
                {
                    if (shortestDist->distance + 1 < unvisited[l]->distance
                        + 1;
                    {
                        unvisited[l]->distance = shortestDist->distance
                            + 1;
                        SRT.nodelist[srtCount] = *unvisited[l];
                        for (int z = 0; z < 4; z++)
                        {
                            SRT.nodelist[srtCount].link[z] = nullptr;
                        }
                        SRT.nodelist[srtCount].link[0] = shortestDist;
                        srtCount++;
                    }
                }
            }
        }
        shortestDist = nullptr;
    }
}

```

```

        cout << "Calcs done!\n";
        calculated = true;
        srtCount2 = srtCount;
        return SRT;
    }

    //findShortest was an attempt to turn the graph resulting from Calculate into a readable
    //format. This algorithm is incomplete and will fail.
    void Djikstra::findShortest(Position goal, Graph ingraph)
    {
        int goalPos = 0;
        int routeLength = 0;
        Node * current = nullptr;
        if (calculated)
        {
            for (int i = 0; i < srtCount2; i++)
            {
                if (ingraph.nodelist[i].pos.x == goal.x && ingraph.nodelist[i].pos.y
                == goal.y)
                {
                    goalPos = i;
                    current = &ingraph.nodelist[i];
                    shortPath[routeLength] = goal;
                    routeLength++;
                }
            }
            for (int j = 0; j < srtCount2; j++)
            {
                shortPath[routeLength] = current->link[0]->pos;
                routeLength++;
                current = current->link[0];
            }
            cout << "SHORTEST ROUTE:\n";
            for (int k = 0; k < routeLength; k++)
            {
                cout << shortPath[routeLength - k].x << ", " << shortPath[routeLength
                - k].y << "\n";
            }
        }
    }
}

```

friendlyBot.cpp

```

#include "friendlyBot.h"
#include "friendlyBotStates.h"
#include "Djikstra.h"
#include <iostream>
using namespace std;
friendlyBot::friendlyBot(Position start, Maze* ptr)
{
    //initialise discovered as blank spaces
    for (int i = 0; i < 50; i++)
    {
        for (int j = 0; j < 50; j++)
        {
            discovered[i][j] = '#';
        }
    }
}

```

```

    }
    //initialise variables
    ptrMaze = ptr;
    ptrCurState = Idle::Instance();
    actual = start;
    lastVisited = start;
    //push starting position into stack of known positions
    knownPositions.push(actual);
    initialGraph.nodelist[initialGraph.nodeCount].pos = actual;
    initialGraph.nodelist[initialGraph.nodeCount].key = 'f';
    initialGraph.nodeCount++;
    discovered[actual.x][actual.y] = 'S';
    foundGoal = false;
    Dijkstra shortpath;
    route[0] = actual;
    routeCount = 1;
    goalpos.x = 0;
    goalpos.y = 0;
}

friendlyBot::~friendlyBot()
{
}

void friendlyBot::Update()
{
    //execute current states logic
    ptrCurState->Execute(this);
}

void friendlyBot::ChangeState(State * ptrNextState)
{
    //transition into new state
    ptrCurState->Exit(this);
    ptrCurState = ptrNextState;
    ptrCurState->Enter(this);
}

bool friendlyBot::readyToGo()
{
    //algorithm asks user if we're ready - will return result to idle state to check
    if a state change is needed
    //ideally would be a check to see if the AI needs to do something, i.e. there's an
    enemy in sight, or we have to find something ect.
    int reply;
    cout << "\nReady to go? We're still idle here! (Type 1 for yes, 2 for no)\n";
    cin >> reply;
    if (reply == 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool friendlyBot::foundGoalWhatNext()

```

```

{
    //requests action after finding a goal, in similar fashion to readyToGo()
    int reply;
    cout << "\nWe have the goal, want to explore the rest of the maze? (1 for return
to idle, 2 for continue search)";
    cin >> reply;
    if (reply == 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

friendlyBotStates.cpp

```

#include "friendlyBotStates.h"
#include "friendlyBot.h"
#include <iostream>
using namespace std;
//create static idle instance
Idle * Idle::Instance()
{
    static Idle instance;
    return &instance;
}
//when entering idle...
void Idle::Enter(friendlyBot* ptrBot)
{
    //print this
    cout << "I'm Idle Now!\n";
}

void Idle::Execute(friendlyBot* ptrBot)
{
    cout << "I'm idling here!";
    //call readyToGo to check if a state change is needed
    if (ptrBot->readyToGo())
    {
        //if a change is needed, change state to searching
        //true implementation would include a more thorough check against more
        conditions, to allow the ai to pass into many different states from idle
        ptrBot->ChangeState(Searching::Instance());
    }
}

//on idle exit
void Idle::Exit(friendlyBot* ptrBot)
{
    //print this
    cout << "\nGot stuff to do, can't be idling\n";
}
//create static searching instance
Searching * Searching::Instance()
{
    static Searching instance;
}

```

```

        return &instance;
    }
    //on enetering searching...
    void Searching::Enter(friendlyBot* ptrBot)
    {
        //print this
        cout << "Now I'm Searching!\n";
    }

    //searchiung state execute function - noted that the searching of a passed in maze can be
    decoupled from here
    void Searching::Execute(friendlyBot* ptrBot)
    {
        //if no more positions
        if (ptrBot->knownPositions.isEmpty())
        {
            //searching is complete, return to idle
            ptrBot->ChangeState(Idle::Instance());
        }
        //if goal is found
        else if (ptrBot->foundGoal)
        {
            //searching is complete, move to success state
            ptrBot->ChangeState(Success::Instance());
        }
        else
        {
            //pop position from stack, make it last visited
            ptrBot->lastVisited = ptrBot->knownPositions.pop();
            //initialise curNode, tracks node we are currently at
            int curNode = 0;
            //make the bots actual position that of the popped position - note:
            investigate wether lastvisted and actual are duplicate, could remove lastvisited and use
            actual in its place
            ptrBot->actual = ptrBot->lastVisited;
            //for each node in Graph...
            for (int i = 0; i < ptrBot->initialGraph.nodeCount; i++)
            {
                //if position is equal to position of a node...
                if (ptrBot->actual.x == ptrBot->initialGraph.nodelist[i].pos.x &&
ptrBot->actual.y == ptrBot->initialGraph.nodelist[i].pos.y)
                {
                    //node we are at is that node
                    curNode = i;
                }
            }
            //for 4 directions, do the following
            for (int i = 1; i < 5; i++)
            {
                //initialise variables
                int x;
                int y;
                Position topush;
                //depending on the direction, where numbers 1-4 represent
                direction...
                switch (i)
                {
                    //if cases are unmet break

```

```

default:
    break;
    //searching down
case 1:
{
    //set searched variables as one tile down from bot
    x = ptrBot->actual.x + 1;
    y = ptrBot->actual.y;
    //depending on tile character
    switch (ptrBot->ptrMaze->tile[x][y])
    {
    default:
        break;
        //if pipe, it's a wall, set corresponding tile position
in diuscovered to this, does not need a node
    case '|':
        ptrBot->discovered[x][y] = '|';
        break;
        //if f it's a floor tile
    case 'f':
    {
        //set position to push to stack
        topush.x = x;
        topush.y = y;
        //if bot has already seen this tile from a different
tile, node will already exist, need to link to existing node,
        //position will have either been visited already or in
stack waiting to be visited

        if (ptrBot->discovered[x][y] != '#')
        {
            // search all nodes for node position. Should
not have two nodes with same position
            for (int i = 0; i < ptrBot->initialGraph.nodeCount; i++)
            {
                if (ptrBot->initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y ==
topush.y)
                {
                    if (ptrBot->initialGraph.nodelist[curNode].linkCount < 4)
                    {
                        ptrBot->initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                    }
                }
            }
        }
        //if bot has not seen tile
    else
    {
        //set tile as seen as floor by bot
        ptrBot->discovered[x][y] = 'f';
        //push position onto stack
        ptrBot->knownPositions.push(topush);
        //set relevant node information
    }
}
}

```

```

        ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount].pos = topush;
        ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount].key = 'f';
        //add link to node we are at
        ptrBot-
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount]);
        //move node position to next slot
        ptrBot->initialGraph.nodeCount++;
    }
}
break;
//for case G, see ccase f for identical logic - additional
logic is commented on here
case 'G':
{
    topush.x = x;
    topush.y = y;
    if (ptrBot->discovered[x][y] != '#')
    {
        for (int i = 0; i < ptrBot-
>initialGraph.nodeCount; i++)
        {
            if (ptrBot-
>initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y ==
topush.y)
            {
                if (ptrBot-
>initialGraph.nodelist[curNode].linkCount < 4)
                {
                    ptrBot-
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                }
            }
        }
    }
    else
    {
        ptrBot->discovered[x][y] = 'G';
        ptrBot->knownPositions.push(topush);
        ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount].pos = topush;
        ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount].key = 'G';
        ptrBot-
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount]);
        //goal has been seen, add goal info to bot
        ptrBot->foundGoal = true;
        ptrBot->goalpos = topush;
        ptrBot->initialGraph.nodeCount++;
    }
}
break;
//for case G, see ccase f for identical logic - additional
logic is commented on here
case 'S':

```

```

        {
            topush.x = x;
            topush.y = y;
            if (ptrBot->discovered[x][y] != '#')
            {
                for (int i = 0; i < ptrBot->initialGraph.nodeCount; i++)
                {
                    if (ptrBot->initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y == topush.y)
                    {
                        if (ptrBot->initialGraph.nodelist[curNode].linkCount < 4)
                        {
                            ptrBot->initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                        }
                    }
                }
            }
            else
            {
                ptrBot->discovered[x][y] = 'S';
                ptrBot->knownPositions.push(topush);
                ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount].pos = topush;
                ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount].key = 'S';
                ptrBot->initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount]);
                ptrBot->initialGraph.nodeCount++;
            }
        }
        break;
    }
    break;
    //searching left
    //for case 2, see case 1 for identical logic. No additional logic
appears
case 2:
{
    x = ptrBot->actual.x;
    y = ptrBot->actual.y - 1;
    switch (ptrBot->ptrMaze->tile[x][y])
    {
        default:
            break;
        case '|':
            ptrBot->discovered[x][y] = '|';
            break;
        case 'f':
        {
            topush.x = x;

```

```

        topush.y = y;
        if (ptrBot->discovered[x][y] != '#')
        {
            for (int i = 0; i < ptrBot->initialGraph.nodeCount; i++)
            {
                if (ptrBot->initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y == topush.y)
                {
                    if (ptrBot->initialGraph.nodelist[curNode].linkCount < 4)
                    {
                        ptrBot->initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                    }
                }
            }
        }
        else
        {
            ptrBot->discovered[x][y] = 'f';
            ptrBot->knownPositions.push(topush);
            ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount].pos = topush;
            ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount].key = 'f';
            ptrBot->initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount]);
            ptrBot->initialGraph.nodeCount++;
        }
    }
    break;
    case 'G':
    {
        topush.x = x;
        topush.y = y;
        if (ptrBot->discovered[x][y] != '#')
        {
            for (int i = 0; i < ptrBot->initialGraph.nodeCount; i++)
            {
                if (ptrBot->initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y == topush.y)
                {
                    if (ptrBot->initialGraph.nodelist[curNode].linkCount < 4)
                    {
                        ptrBot->initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                    }
                }
            }
        }
        else
    }

```

```

        {
            ptrBot->discovered[x][y] = 'G';
            ptrBot->knownPositions.push(topush);
            ptrBot->initialGraph.nodelist[ptrBot->
>initialGraph.nodeCount].pos = topush;
            ptrBot->initialGraph.nodelist[ptrBot->
>initialGraph.nodeCount].key = 'G';
            ptrBot->
            >initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot->
>initialGraph.nodeCount]);
            ptrBot->foundGoal = true;
            ptrBot->goalpos = topush;
            ptrBot->initialGraph.nodeCount++;
        }
    }
    break;

    case 'S':
    {
        topush.x = x;
        topush.y = y;
        if (ptrBot->discovered[x][y] != '#')
        {
            for (int i = 0; i < ptrBot->
>initialGraph.nodeCount; i++)
            {
                if (ptrBot->
>initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y ==
topush.y)
                {
                    if (ptrBot->
>initialGraph.nodelist[curNode].linkCount < 4)
                    {
                        ptrBot->
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                    }
                }
            }
        }
        else
        {
            ptrBot->discovered[x][y] = 'S';
            ptrBot->knownPositions.push(topush);
            ptrBot->initialGraph.nodelist[ptrBot->
>initialGraph.nodeCount].pos = topush;
            ptrBot->initialGraph.nodelist[ptrBot->
>initialGraph.nodeCount].key = 'S';
            ptrBot->
            >initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot->
>initialGraph.nodeCount]);
            ptrBot->initialGraph.nodeCount++;
        }
    }
    break;
}
}

```

```

        break;
        //searching up
        //for case 3, see case 1 for identical logic. No additional logic
        case 3:
        {
            x = ptrBot->actual.x - 1;
            y = ptrBot->actual.y;
            switch (ptrBot->ptrMaze->tile[x][y])
            {
            default:
                break;
            case '|':
                ptrBot->discovered[x][y] = '|';
                break;
            case 'f':
            {
                topush.x = x;
                topush.y = y;
                if (ptrBot->discovered[x][y] != '#')
                {
                    for (int i = 0; i < ptrBot->initialGraph.nodeCount; i++)
                    {
                        if (ptrBot->initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y == topush.y)
                        {
                            if (ptrBot->initialGraph.nodelist[curNode].linkCount < 4)
                            {
                                ptrBot->initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                            }
                        }
                    }
                }
                else
                {
                    ptrBot->discovered[x][y] = 'f';
                    ptrBot->knownPositions.push(topush);
                    ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount].pos = topush;
                    ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount].key = 'f';
                    ptrBot->initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount]);
                    ptrBot->initialGraph.nodeCount++;
                }
            }
            break;
            case 'G':
            {
                topush.x = x;
                topush.y = y;
                if (ptrBot->discovered[x][y] != '#')

```

```

        {
            for (int i = 0; i < ptrBot->
>initialGraph.nodeCount; i++)
            {
                if (ptrBot->
>initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y ==
topush.y)
                {
                    if (ptrBot->
>initialGraph.nodelist[curNode].linkCount < 4)
                    {
                        ptrBot->
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                    }
                }
            }
        }
    else
    {
        ptrBot->discovered[x][y] = 'G';
        ptrBot->knownPositions.push(topush);
        ptrBot->initialGraph.nodelist[ptrBot->
>initialGraph.nodeCount].pos = topush;
        ptrBot->initialGraph.nodelist[ptrBot->
>initialGraph.nodeCount].key = 'G';
        ptrBot->
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot->
>initialGraph.nodeCount]);
        ptrBot->initialGraph.nodeCount++;
        ptrBot->foundGoal = true;
        ptrBot->goalpos = topush;
    }
}
break;

case 'S':
{
    topush.x = x;
    topush.y = y;
    if (ptrBot->discovered[x][y] != '#')
    {
        for (int i = 0; i < ptrBot->
>initialGraph.nodeCount; i++)
        {
            if (ptrBot->
>initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y ==
topush.y)
            {
                if (ptrBot->
>initialGraph.nodelist[curNode].linkCount < 4)
                {
                    ptrBot->
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                }
            }
        }
    }
}

```

```

else
{
    ptrBot->discovered[x][y] = 'S';
    ptrBot->knownPositions.push(topush);
    ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount].pos = topush;
    ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount].key = 'S';
    ptrBot-
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount]);
    ptrBot->initialGraph.nodeCount++;
}
}
break;
}
}
break;
//searching right
//for case 3, see case 1 for identical logic. No additional logic
appears
case 4:
{
    x = ptrBot->actual.x;
    y = ptrBot->actual.y + 1;
    switch (ptrBot->ptrMaze->tile[x][y])
    {
        default:
            break;
        case '|':
            ptrBot->discovered[x][y] = '|';
            break;
        case 'f':
            {
                topush.x = x;
                topush.y = y;
                if (ptrBot->discovered[x][y] != '#')
                {
                    for (int i = 0; i < ptrBot-
>initialGraph.nodeCount; i++)
                    {
                        if (ptrBot-
>initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y ==
topush.y)
                        {
                            if (ptrBot-
>initialGraph.nodelist[curNode].linkCount < 4)
                            {
                                ptrBot-
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                            }
                        }
                    }
                }
            }
        else
        {
            ptrBot->discovered[x][y] = 'f';
            ptrBot->knownPositions.push(topush);

```

```

        ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount].pos = topush;
        ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount].key = 'f';
        ptrBot-
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount]);
        ptrBot->initialGraph.nodeCount++;
    }
    break;

    case 'G':
    {
        topush.x = x;
        topush.y = y;
        if (ptrBot->discovered[x][y] != '#')
        {
            for (int i = 0; i < ptrBot-
>initialGraph.nodeCount; i++)
            {
                if (ptrBot-
>initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y ==
topush.y)
                {
                    if (ptrBot-
>initialGraph.nodelist[curNode].linkCount < 4)
                    {
                        ptrBot-
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                    }
                }
            }
        }
        else
        {
            ptrBot->discovered[x][y] = 'G';
            ptrBot->knownPositions.push(topush);
            ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount].pos = topush;
            ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount].key = 'G';
            ptrBot-
>initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot-
>initialGraph.nodeCount]);
            ptrBot->initialGraph.nodeCount++;
            ptrBot->foundGoal = true;
            ptrBot->goalpos = topush;
        }
    }
    break;

    case 'S':
    {
        topush.x = x;
        topush.y = y;
        if (ptrBot->discovered[x][y] != '#')
        {

```

```

        for (int i = 0; i < ptrBot->initialGraph.nodeCount; i++)
        {
            if (ptrBot->initialGraph.nodelist[i].pos.x == topush.x && ptrBot->initialGraph.nodelist[i].pos.y ==
topush.y)
            {
                if (ptrBot->initialGraph.nodelist[curNode].linkCount < 4)
                {
                    ptrBot->initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[i]);
                }
            }
        }
    }
    else
    {
        ptrBot->discovered[x][y] = 'S';
        ptrBot->knownPositions.push(topush);
        ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount].pos = topush;
        ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount].key = 'S';
        ptrBot->initialGraph.nodelist[curNode].AddLink(&ptrBot->initialGraph.nodelist[ptrBot->initialGraph.nodeCount]);
        ptrBot->initialGraph.nodeCount++;
    }
    }
    break;
}
break;
}
}
}
//prints bots known positions in a readable format
cout << "\n\n\n";
for (int a = 0; a < ptrBot->ptrMaze->height + 3; a++)
{
    for (int b = 0; b < ptrBot->ptrMaze->width + 10; b++)
    {
        if (ptrBot->actual.x == a && ptrBot->actual.y == b)
        {
            cout << "@";
        }
        else
        {
            cout << ptrBot->discovered[a][b];
        }
    }
    cout << "\n";
}
cout << "\n\n\n";

```

```

    }
}
//on exit...
void Searching::Exit(friendlyBot* ptrBot)
{
    //print this
    cout << "\nI'm done searching boss!\n";
}
//create static success instance
Success * Success::Instance()
{
    static Success instance;
    return &instance;
}
//on enter...
void Success::Enter(friendlyBot * ptrBot)
{
    //print this
    cout << "I've found it boss!\n";
}

void Success::Execute(friendlyBot * ptrBot)
{
    //check if Bot should return to idle or continue searching the whole maze
    if (ptrBot->foundGoalWhatNext())
    {
        //if idle, go idle
        ptrBot->ChangeState(Idle::Instance());
    }
    else
    {
        //if not, switch found goal off and search maze
        //this has the effect of the bot not searching for a goal, but for the rest
of the maze
        //if multiple goals exist, bot should stop for second goal still
        ptrBot->foundGoal = false;
        ptrBot->ChangeState(Searching::Instance());
    }
}

void Success::Exit(friendlyBot * ptrBot)
{
}

```

graph.cpp

```

#include "graph.h"

Graph::Graph()
{
    nodeCount = 0;
}

```

main.cpp

```
#include <iostream>
using namespace std;
#include "Stack.h"
#include "Maze.h"
#include "friendlyBot.h"
#include "Dijkstra.h"
int main()
{
    Dijkstra shortest;
    int mainChoice = 0;
    int loopcount = 500;
    cout << "Load maze1 = 1, quit = -1\n";
    cin >> mainChoice;
    if (mainChoice == -1)
    {
        return 0;
    }
    Maze maze1("maze.txt");
    friendlyBot Bot(maze1.Start, &maze1);
    while(mainChoice != -1)
    {
        switch (mainChoice)
        {
        default:
            cout << "INCORRECT VALUE ENTERED\n";
            cin >> mainChoice;
            break;
        case 1:
            for (int i = 0; i < maze1.height; i++)
            {
                cout << maze1.tile[i] << "\n";
            }
            cout << "Maze loaded!\nBegin search for Goal = 2, quit = -1\n";
            cin >> mainChoice;
            break;
        case 2:
            Bot.Update();
            cout << "Bot update completed. Continue = 2, quit = -1\n";
            cin >> mainChoice;
            break;
        }
    }

    return 0;
}
```

Maze.cpp

```
#include "Maze.h"
#include <iostream>
Maze::Maze(char * infile)
{
    char buffer[50];
    file = infile;
    FILE *mazeFile;
    mazeFile = fopen(file, "r");
    fscanf(mazeFile, "%s", buffer);
    while (!feof(mazeFile))
    {
        if (strcmp(buffer, "#width") == 0)
        {
            char tmp[10];
            fscanf(mazeFile, "%s", tmp);
            width = atoi(tmp);
        }
        else if (strcmp(buffer, "#height") == 0)
        {
            char tmp[10];
            fscanf(mazeFile, "%s", tmp);
            height = atoi(tmp);
        }
        else if (strcmp(buffer, "#begin") == 0)
        {
            if (width <= 50 && height <= 50)
            {
                fscanf(mazeFile, "%s", buffer);
                for (int i = 0; i < height; i++)
                {
                    for (int j = 0; j < width; j++)
                    {
                        tile[i][j] = buffer[j];
                        if (buffer[j] == 'S')
                        {
                            Start.x = i;
                            Start.y = j;
                        }
                        if (buffer[j] == 'G')
                        {
                            Goal.x = j;
                            Goal.y = i;
                        }
                    }
                    fscanf(mazeFile, "%s", buffer);
                }
            }
            fscanf(mazeFile, "%s", buffer);
        }
    }
}

Maze::~Maze()
{
}
```

Node.cpp

```
#include "node.h"

Node::Node()
{
    linkCount = 0;
    distance = 0;
    visited = false;
}

void Node::AddLink(Node * toLink)
{
    link[linkCount] = toLink;
    linkCount++;
}
```

Stack.cpp

```
#include "Stack.h"
#include <iostream>
using namespace std;
Stack::Stack()
{
    stackCount = -1;
    pstnList = new Position[STACK_SIZE];
}

Stack::~~Stack()
{
}

void Stack::push(Position x)
{
    if (!isFull())
    {
        stackCount++;
        pstnList[stackCount] = x;
    }
    else
    {
        cout << "Stack is full!"; //need error catch here, what to do when stack is
full? start new stack? Stack size is set in stone, so could allow recursive creation of a
stack...
        //consider how AI will react when stack is full, perhaps First in First
Out, only remember back a certain way
    }
}

Position Stack::pop()
{
    if (!isEmpty())
    {
        Position temp = pstnList[stackCount];
        stackCount--;
        return temp;
    }
}
```

```

        else
        {
            cout << "Stack is empty!"; //same as push, work out how AI will react
            Position blank;
            blank.x = 0;
            blank.y = 0;
            return blank;
        }
    }

    bool Stack::isFull()
    {
        if (stackCount >= STACK_SIZE - 1)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    bool Stack::isEmpty()
    {
        if (stackCount < 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}

```

FINAL GRADE

GENERAL COMMENTS

Instructor

76/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

GRADING FORM: SMU GENERIC_GJ

OLIVER JORDAN

75.50

RESEARCH-INFORMED LITERATURE



Refs included - including Dijkstra's original, well done. You've reviewed required SE principles related to code dev but include a ref to these.

3

KNOWLEDGE AND UNDERSTANDING



Good use of state machines - well done and solution is scalable - excellent. Good understanding of dfs and its implementation in C++.

12

ANALYSIS



Analysis of problem and selection of appropriate algorithm in evidence. Use of state machines exceptional - well done. Solution could have been better animated . Potentially "over-engineered. Keep design as simple as necessary.

12

PRACTICAL APPLICATION AND DEP.



Some good work here. Plenty of evidence of SE principles here, multiple patterns used. Algorithms implemented correctly. Good commenting in places; evidence of separation of interface from implementation; possibly over-engineered.

45

SKILLS FOR PROFESSIONAL PRAC.



A well present report, together with some refs. SEP practices v. good.

3.5