C++Qt_day3 hw_3 A* & 다익스트라 과제 보고서

로빛 20기 인턴 현창석

목차

1. 다익스트라 알고리즘 설명

다익스트라 알고리즘은 출발 노드에서 다른 노드까지의 최단 거리를 모두 각각 구하는 알고리즘을 말한다. 먼저 출발 노드와 최단 거리 테이블을 초기화 한다. 그 후 거치지 않은 노드 중에서 가장 짧은 최단 거리를 가진 노드를 선택한다. 이 노드를 거쳐서 다른 노드로 가는 거리를 계산하고 이 계산 값에 따라 최단 거리 테이블이 갱신된다. 모든 노드를 거칠 때까지 이 과정을 계속 반복한다.
다시 정리하여 설명하자면 우선 첫 번째로 출발 노드를 설정하고, 이 출발 노드를 기준으로 하여 각 노드까지의 최소 비용을 저장한다. 그다움 순서로 거치지 않은 노드 중에서 가장 비용이 적게 드는 노드를 선택한다. 비용은 보통 거리로 계산한다. 해당 노드를 거쳐 특정 노드로 가는 최소 비용을 계속 갱신한다. 이 과정을 계속 반복한다.
인접 행렬 또는 우선순위 대기열을 사용하여 구현할 수 있기에 모든 경로를 확인하는 무차별 접근 방식보다 효율적이고 경로를 추적하도록 알고리즘을 쉽게 수정 가능하다는 특징이 있지만 여러 단점 또한 존재한다. 음수 가중치를 간선을 가진 그래프에서 작동하지 않으며 그래프와 각 거리를 저장하기 위해서 많은 메모리가 요구된다. 또한 동일한 거리에서 여러 경로가 존재하는 경우 최상의 경로를 보장하지 않는다.

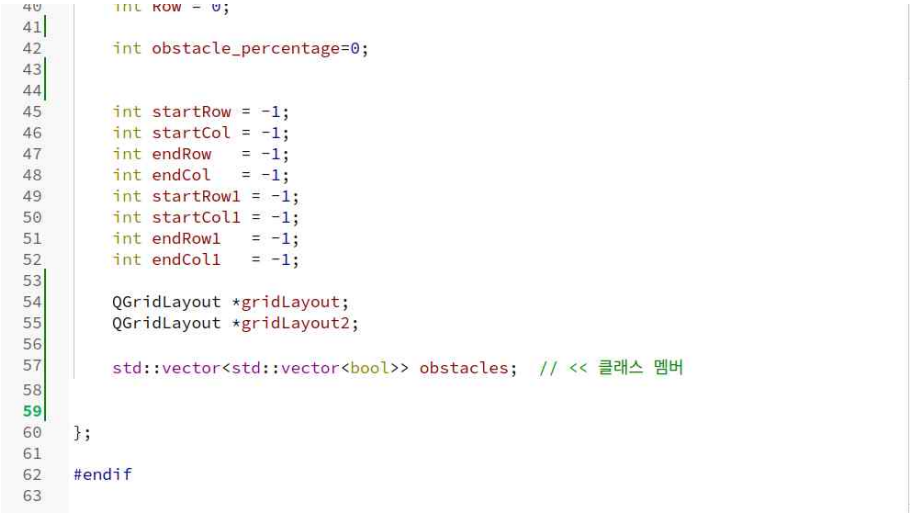
2. A* 알고리즘 설명

A* 알고리즘은 다익스트라 알고리즘과 마찬가지로 출발 노드에서부터 목적지까지의 최단 경로를 찾아내는 그래프 탐색 알고리즘 중 하나이다. 다익스트라 알고리즘과 유사하지만 A* 알고리즘은 목적지까지 휴리스틱 거리 측정값을 추가적으로 이용한다는 점에서 차이점이 존재한다.

3. 과제 코드 설명 -mainwindow.h 설명

Warning: This file is not part of any project. The code model might have issues parsing this file properly. Minimiz

```
#ifndef MYWIDGET_H
#define MYWIDGET_H

#include <QMainWindow>

#include "ui_mainwindow.h"
#include <QGridLayout>
#include <queue>
#include <vector>
#include <cmath>

class MyWidget : public QMainWindow
{
    Q_OBJECT
public:
    explicit MyWidget(QWidget *parent = nullptr);

protected:

public slots:


private slots:
    void on_pushButton_clicked();
    void on_spinBox_valueChanged(int arg);  // 슬롯 선언

    void Dijkstra_algorithm();
    void AStar_algorithm();
    void on_spinBox_2_valueChanged(int arg1);

    void on_pushButton_2_clicked(bool checked);

    void on_spinBox_3_valueChanged(int arg1);

    void on_pushButton_3_clicked(bool checked);

```



```
40      int Row = 0;
41
42      int obstacle_percentage=0;
43
44
45      int startRow = -1;
46      int startCol = -1;
47      int endRow   = -1;
48      int endCol   = -1;
49      int startRow1 = -1;
50      int startCol1 = -1;
51      int endRow1   = -1;
52      int endCol1   = -1;
53
54      QGridLayout *gridLayout;
55      QGridLayout *gridLayout2;
56
57      std::vector<std::vector<bool>> obstacles;  // << 클래스 멤버
58
59
60  };
61
62  #endif
63
```

위 사진은 mainwindow.h의 코드를 캡쳐한 사진이다.
class mywidget을 Qmainwindow를 상속받아서 메인 윈도우 형태의 위젯을 출력하기 위해 선언하였다. 헤더파일이 중복 포함되는 것을 막기 위해 MYWIDGET_H을 선언하였다.
on_spinBox_valueChanged provate slot에서 Row와 Col 좌료를 만들기 위해 slot 값을 읽는 기능의 함수이다 또한 좌표 맵 생성이나 알고리즘 스타트 장애물 생성 버튼에 해당하는 기능의 함수들을 선언해주었다. 멤버 변수로는 mainwindow의
좌표 맵을 생성하기 위한 Col, ROw 장애물 생성할 때 장애물의 수를 계산하는 변수, startRow/Col 은 gridlayer1과 2의 시작점과 끝점의 좌표를 변수로 저장하기 위해 선언한 것이다. 맵을 출력하기 위한 gridlayout 창을 2개 생성하기 위한 객체를 가리키는 포인터를선언해주었다. std::vector<std::vector<bool>> obstacles; 는 2차원 벡터 형태로 장애물이 있는가 없는가에 대한 정보를 true 또는 false로 저장한다. 이 값들을 통해 경로를 찾을 때 지나갈 수 있는 칸과 없는 칸을 구분할 수 있다.

과제 코드 설명 -mainwindow.cpp 설명

```
28
29  void MyWidget::on_pushButton_clicked()
30  {
31      QLayoutItem *child;
32      while ((child = gridLayout->takeAt(0)) != nullptr) {
33          delete child->widget(); // 버튼 삭제
34          delete child;
35      }
36      while ((child = gridLayout2->takeAt(0)) != nullptr) {
37          delete child->widget();
38          delete child;
39      }
40
41      for (int r = 0; r < Row; r++) {
42          for (int c = 0; c < Col; c++) {
43              QPushButton *cell = new QPushButton(QString("(%1,%2)").arg(r).arg(c));
44              cell->setFixedSize(40, 40);
45              gridLayout->addWidget(cell, r, c);
46
47              connect(cell, &QPushButton::clicked, this, [=]() {
48                  if (startRow == -1 && startCol == -1) {   // 시작점 선택
49                      startRow = r;
50                      startCol = c;
51                      cell->setStyleSheet("background-color: green;");
52                  }
53                  else if (endRow == -1 && endCol == -1) {  // 끝점 선택
54                      endRow = r;
55                      endCol = c;
56                      cell->setStyleSheet("background-color: red;");
57                  }
58
59                  qDebug() << "시작: (" << startRow << "," << startCol << ")"
60                           << "끝: (" << endRow << "," << endCol << ")";
61              });
62
63          }
64      }
65      for (int r = 0; r < Row; r++) {
66          for (int c = 0; c < Col; c++) {
67              QPushButton *cell = new QPushButton(QString("(%1,%2)").arg(r).arg(c));
68              cell->setFixedSize(40, 40);
69              gridLayout2->addWidget(cell, r, c);
```

```
void MyWidget::on_spinBox_valueChanged(int arg)
{

    Col = arg;
}

void MyWidget::on_spinBox_2_valueChanged(int arg1)
{
    Row = arg1;
}


void MyWidget::on_pushButton_2_clicked(bool checked)
{
    AStar_algorithm();
    Dijkstra_algorithm();
}

struct Node {
    int row, col;
    int g, h, f;
    Node* parent;
};

struct CompareNode {
    bool operator()(Node* a, Node* b) {
        return a->f > b->f;
    }
};
```

화면에 로그를 찍기 위한 헤더파일 Qdebug와 queue, vector를 불러와 경로 탐색 알고리즘(A*, Dijkstra)에서 우선순위 큐와 2차원 벡터 자료구조를 활용하였다. 랜덤으로 장애물을 생성하 기위한 난수를 설정하기 위해 cstdlib, ctime를 인클루드 하였다. 맨해튼 거리를 계산하기 위해서 cmath 헤더파일을 불러왔다.
qmainwindow를 상속받아서 mywidget클래스로 창 위젯을 생성한다.
UI를 초기화한 후 gridLayout과 gridLayout2 각각 두 개의 gridlayout을 만들었다. 이 두개의 그리드 레이아웃에 A* 알고리즘과 다익스트라 알고리즘을 실행하는 화면을 출력할 것이다.
on_pushButton_clicked 함수는 맵 생성 버튼이 눌리면 실행되고 Row, Col 값을 결정하는 스핀박스 위젯의 값에 따라 Row * Col 수만큼의셀을생성한다.이 셀에는 QString("(%1,%2)").arg(r).arg(c)) 으로 각 좌표 텍스트가 출력되어있다. connect (cell, &QPushButton::clicked 를 이용해 셀을 클릭하면 처음 클릭한 셀일 경우에 (start row와 startcol이 초기값이라면) 시작점으로 선택한다. 이 시작점의 색은 녹색으로 지정하였다. 끝점을 설정하는 알고리즘도 이와 같다. 탐색 알고리즘에 사용할 NOde 구조체를 설정하였다. g는 현재 노드에서 끝점까지의 거리를 나타낸다.
h는 현재 노드에서 끝점까지의 휴리스틱 값을 나타낸다. 여기서 맨해튼 거리가 사용된다. g+h를 저장하는 변수가 f이며 A* 알고리즘에서는 이 f 값이 가장 작은 노드를 우선 탐색한다.
탐색 알고리즘은 경로 자체를 저장하지 않기 때문에 parent 노드를 만들어 현재 셀에 오기 바로 직전 셀을 가리켜 경로를 추적할 수 있도록 한다.
CompareNode 구조체는 A*알고리즘을 사용할 때 우선순위를
하지만 A* 알고리즘은 f 값이 작을수록 먼저 탐색해야 하기에
비교 연산자를 반대로 정의해서 f 값이 작은 노드가 먼저 나오게 하도록 하였다.

```cpp
void MyWidget::AStar_algorithm() {
    if (startRow == -1 || endRow == -1) {
        qDebug() << "시작점/끝점이 지정되지 않았습니다.";
        return;
    }

    int dr[4] = {1, -1, 0, 0};
    int dc[4] = {0, 0, 1, -1};

    std::priority_queue<Node*, std::vector<Node*>, CompareNode> openList;
    std::vector<std::vector<bool>> closed(Row, std::vector<bool>(Col, false));

    Node* start = new Node{startRow, startCol, 0, 0, 0, nullptr};
    start->h = abs(endRow - startRow) + abs(endCol - startCol);
    start->f = start->g + start->h;
    openList.push(start);

    Node* found = nullptr;

    while (!openList.empty()) {
        Node* current = openList.top();
        openList.pop();

        if (current->row == endRow && current->col == endCol) {
            found = current;
            break;
        }

        closed[current->row][current->col] = true;

        for (int i = 0; i < 4; i++) {
            int nr = current->row + dr[i];
            int nc = current->col + dc[i];

            if (nr < 0 || nr >= Row || nc < 0 || nc >= Col) continue;
            if (closed[nr][nc]) continue;
            if (obstacles[nr][nc]) continue;  // 장애물 무시

            Node* neighbor = new Node{nr, nc, current->g + 1, 0, 0, current};
            neighbor->h = (endRow - nr) + abs(endCol - nc);
            neighbor->f = neighbor->g + neighbor->h;
```

```cpp
163
164             openList.push(neighbor);
165
166
167             QWidget* widget = gridLayout->itemAtPosition(nr, nc)->widget();
168             if (widget) widget->setStyleSheet("background-color: lightgreen;");
169         }
170     }
171
172
173     if (found) {
174         Node* asd = found;
175         while (asd->parent) {
176             QWidget* widget = gridLayout->itemAtPosition(asd->row, asd->col)->widge
177             if (widget) widget->setStyleSheet("background-color: yellow;");
178             asd = asd->parent;
179         }
180     }
181 }
```

AStar_algorithm 함수에서는 먼저 시작점이나 끝점이 선택되지 않아 초기값을 가지고 있다면 경로 찾기를 실행하지 않는다. 오류 메세지를 출력하고 함수를 종료한다.
dr과 dc는 행과 열을 이동하는 배열이다. 다음 이동 위치를 나타낼 때 사용된다. openlist라는 방문할 셀들을 넣을 우선순위 큐를 만들고 이미 방문한 셀인지 정보를 저장하기 위한 closed 벡터를 선언하였다.
선택한 시작점 셀의 좌표값을 불러와 시작점에 해당하는 노드를 만든다. 도착 끝 점까지의 거리를 절댓과 거리계산 공식을 통해 계산해주고 f애 g+h 를 하여 초기 설정을 완료해준다.
그 후 openlist가 빌 때까지 가장 f가 작은 노드를 꺼내 현재 노드로 사용한다. 만약 그 노드가 사용자가 선택한 끝 점의 좌표와 같다면 종료하고 그렇지 않으면 해당 노드를 true로 바꾸어 closed에 저장한다. true로 변경되었으므로 한 번 방문했음을 나타내었다. 그 다음 현재 노드에서 반복문을 통해 dr,dc 배열 값을 불러와 4방향으로 탐색을 진행한다. 이때 만약 탐색한 노드가 좌표에서 벗어났거나 혹은 이미 방문하였거나 혹은 장애물에 해당한다면 continue를 통해 다음 반복순서로 넘어간다.
새로운 노드를 생성해 탐색하는 위치와 시작점부터 이동한 거리를 설정하고 끝점까지 얼마나 남았는지 맨해튼 거리 계산한 결과를 h에 저장한다. 전체 경로의 거리는 h에 저장한다.
탐색 중인 칸을의 색을 setStyleSheet으로 밝은 녹색으로 설정하여 시각적으로 나타내었다. 경로를 찾으면 found를 가리키는 임시변수 asd를 생성한 후 parent가 존재하는 동안 계속하여 반복해 경로를 노란색으로 표시한다.

```
187
188 void MyWidget::on_pushButton_3_clicked(bool checked)
189 {
190     if (Row <= 0 || Col <= 0) return;
191
192     std::srand(std::time(nullptr));
193
194     // 장애물 배열 초기화 → 클래스 멤버 사용
195     obstacles.assign(Row, std::vector<bool>(Col, false));
196
197     int obstacleCount = obstacle_percentage;
198     qDebug() << "장애물 개수:" << obstacleCount;
199
200     int placed = 0;
201     while (placed < obstacleCount) {
202         int r = std::rand() % Row;
203         int c = std::rand() % Col;
204
205         // 시작점과 끝점은 제외
206         if ((r == startRow && c == startCol) || (r == endRow && c == endCol))
207             continue;
208
209         if (!obstacles[r][c]) {
210             obstacles[r][c] = true;
211             QWidget* w1 = gridLayout->itemAtPosition(r, c)->widget();
212             QWidget* w2 = gridLayout2->itemAtPosition(r, c)->widget();
213             if (w1) w1->setStyleSheet("background-color: black;");
214             if (w2) w2->setStyleSheet("background-color: black;");
215             placed++;
216         }
217     }
218 }
219
```

on_spinBox_3_valueChanged 함수에서는 스핀 박스 안의 값이 바뀔 때마다 그 값에 따라 (Col * Row * arg1 * 0.01)장애물의 수를 계산하였다.

on_pushButton_3_click 함수에서는 버튼3 이 클릭되어 true가 되고 맵이 생성된 상태라면 다음 코드가 실행된다. 장애물의 위치를 랜덤으로 지정하기 위해 srand로 초기화해주었고 obstacles Row x Col 크기 2차원 벡터로 초기화하였다.
그 후 생성해야할 장애물 개수만큼 while문으로 반복하며
랜덤으로 x,y좌표를 설정한 후 시작과 끝점에 해당하지 않고 이미 장애물이 생성되어있지 않으면 해당 좌표에 해당하는 벡터를 true로 설정하여 장애물로 표시하였다. setStyleSheet으로 검은색을 칠해 장애물을 시각적으로 나타내었다. 장애물 배치가 되면 placed가 1 증가한다.

```
271
272             // 탐색 중인 셀 표시
273             QWidget* w = gridLayout2->itemAtPosition(nr, nc)->widget();
274             if (w) w->setStyleSheet("background-color: lightgreen;");
275         }
276     }
277
278
279     if (found) {
280         DNode* cur = found;
281         while (cur->parent) {
282             QWidget* w = gridLayout2->itemAtPosition(cur->row, cur->col)->widget();
283             if (w) w->setStyleSheet("background-color: yellow;");
284             cur = cur->parent;
285         }
286     }
287 }
288
289
```

에이스타 알고리즘이 다익스트라 알고리즘을 기반으로 하고 있기에 두 알고리즘의 코드 구조는 비슷한다. 다만 차이점이 있다. 에이스타 알고리즘에서는 휴리스틱(맨해튼 거리) 를 사용해 가장 빠른 최적 경로를 탐색하지만 다익스트라 알고리즘은 이를 사용하지 않는다.

따라서 다익스트라 알고리즘은 dist가 가장 작은 값부터 즉 단순히 시작점에서 가장 가까운 노드부터 탐색한다고 할 수 있다. 이러한 점에서 코드 상 차이점이 있다.