HW3 보고서

로빛 20기 예비인턴

2025407006 로봇학부 모시온

목차

1. 개요

본 프로젝트는 과제2를 통해 익힌 개념을 바탕으로 실제로 QT 환경을 통해 각각의 최단 경로 탐색 알고리즘을 실습하는 데에 목표를 두는 것으로 한다.

A*, 다익스트라 알고리즘을 각각 적용하여 서로 다른 장애물 상황에서 구동되는 모습을 시각화하여 비교할 수 있도록 합니다.

2. .h/.hpp 헤더파일 설계

```
#include <QMainWindow>
#include <QGraphicsScene>
#include <QGraphicsView>
#include <QTimer>
#include <QLineEdit>
#include <QPushButton>
#include <QMouseEvent>
#include <vector>
#include <limits>
```

1. QMainWindow : QT 소프트웨어와 창(윈도우)의 상속 모듈
   QTimer: A*, 다익스트라 경로 탐색 애니메이션 지연 시간 구현
2. vector: 사용자의 임의 지도 크기 지정에 따른 동적할당
3. limits: 다익스트라 알고리즘 구현 시 각각의 배열에 입력할 INF 가중치 구현(max)
4. QLineEdit: 지도 크기 및 장애물 비중 입력란 구현
5. QMouseEvent: 시작점, 도착점, 장애물 설치를 위하여 호출
6. QPushButton: 큐 푸시 버튼 사용을 위하여 호출

```
//픽셀 노드
struct Node {
    int x, y;
    //장애물 배치 X
    bool obs = false;
    bool visited = false;
    //다익스트라용
    double g = std::numeric_limits<double>::infinity();
    double h = 0;
    Node* parent = nullptr;
};
```

지도에 할당할 각각의 노드(그래픽상으로 픽셀 구현)를 위한 구조체
이후 해당 구조체를 동적을 할당(벡터 사용)

해당 구조체에는 각각의 노드의 위치(x,y)와 장애물 또는 알고리즘의 방문을 감지(감지시 파란색) 및 다익스트라와 휴리스틱 구현을 위하여 각각 g(소비 자원 저장-다익스트라), h(목표까지의 예상 소비 자원)를 선언하며 이를 연결할 수 있도록 부모 노드를 포인터로 선언하였습니다.

(해당 각각의 노드의 g와 h를 통해 다익스트라 및 A*를 구현합니다.)

```
class PF {
public:
    PF(int n, double obsRatio);
    void map_random(double obsRatio);
    void searching();
    std::vector<Node*> sidenode(Node* node);
    double heuristic(Node* a, Node* b);
    bool dijkstra();
    bool aStar();
    std::vector<Node*> map_way();

    int size;
    std::vector<std::vector<Node>> map;
    Node* start;
    Node* goal;

    std::vector<Node*> openD;
    std::vector<Node*> openA;
    bool finishedD, finishedA;
};
```

PF Class는 지도 생성, 장애물 배치, 경로 탐색, 탐색 경로 반환을 담당하며, 지도의 크기(size)와 2D 노드 픽셀 지도(map)를 제어하며, 시작

점과 도착점을 각각 start, goal로 지정합니다.

이때, 다익스트라와 A* 알고리즘을 위한 동적 지도 픽셀(openD, openA)와 탐색 완료 여부를 결정할 (finishedD, finishedA)도 선언합니다. 가까운 노드 픽셀을 반환하거나 휴리스틱을 계산하고, 실제 경로를 반환하여 결과적으로 경로 탐색을 이루도록 하였습니다.

```
public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

//%%마우스 클릭은 protected 영역
protected:
    bool eventFilter(QObject* obj, QEvent* ev) override;

private slots:
    void on_B_Apply_clicked();
    void on_B_Play_clicked();
    void nextPixel();

private:
    Ui::MainWindow *ui;
    QTimer* timer;
    PF* pfA;
    PF* pfD;
    QGraphicsScene* sceneA;
    QGraphicsScene* sceneD;
    int pixel;
    int click;
    bool play;

    void map_generating();
    void map_sight();
    void map_scaling();
    bool map_center(QObject* viewObj, const QPoint& pos, int &row, int &col);
```

UI 버튼 클릭 시 동작을 정의하는 함수가 슬롯으로 포함되며. 지도 제작/적용 버튼, 탐색 시작 버튼, 타이머 단계별 호출 함수 등이 있으며, 버튼과 UI 요소를 Qt 시그널/슬롯으로 연결하도록 하였습니다.

또한, MainWindow는 탐색 상태와 시각화를 관리하기 위해 다양한 멤버 변수를 사용하는데 UI 포인터, 타이머, A*와 다익스트라 PF 객체(pfA, pfD), 애니메이션 출력(sceneA, sceneD) 등이 있습니다.
현재 탐색 픽셀(pixel), 클릭 상태(click), 탐색 진행 여부(play) 등도 관리하도록 하였습니다.

3. .c/.cpp 소스파일 설계

```
//PF 생성자
PF::PF(int n, double obsRatio) : size(n) {
    map.resize(size, std::vector<Node>(size));
    for (int r = 0; r < size; ++r) {
        for (int c = 0; c < size; ++c) {
            map[r][c].x = c;
            map[r][c].y = r;
        }
    }
    map_random(obsRatio);
}
```

PF Class 생성자를 통해 지도 크기, 장애물 비중을 초기화합니다.
먼저 2차원 벡터(지도)를 지정 크기로 할당한후, 각 노드 픽셀의 좌표(x, y)를 설정합니다. 이후, map_random 함수를 호출해 맵 전체에 장애물을 랜덤으로 배치하고 탐색 초기 상태를 설정하여 준비를 마칩니다.

```
//장애물 배치
void PF::map_random(double obsRatio) {
    std::srand(static_cast<unsigned int>(std::time(nullptr)));
    for (int r = 0; r < size; ++r) {
        for (int c = 0; c < size; ++c) {
            map[r][c].obs = ((std::rand() % 100) < static_cast<int>(obsRatio * 100.0));
            map[r][c].visited = false;
            map[r][c].g = std::numeric_limits<double>::infinity();
            map[r][c].h = 0;
            map[r][c].parent = nullptr;
        }
    }
    start = nullptr;
    goal = nullptr;
}
```

사용자에게 미리 입력받은 수를 바탕으로 각 노드픽셀을 순회하며 장

애물을 srand를 통해 무작위로 배치합니다.
이때, obsRatio 비율은 앞서 언급한 것과 같이 사용자의 입력에 따라 결정되며, 알고리즘이 방문한 여부와 탐색 비용(g, h), 부모 노드 픽셀을 초기화하여, 시작점(start)와 도착점(goal)도 nullptr로 초기화하여 탐색 전 상태로 만듭니다.

```
void PF::searching() {
    openD.clear();
    openA.clear();
    finishedD = finishedA = false;
    for (int r = 0; r < size; ++r) {
        for (int c = 0; c < size; ++c) {
            map[r][c].visited = false;
            map[r][c].g = std::numeric_limits<double>::infinity();
            map[r][c].h = 0;
            map[r][c].parent = nullptr;
        }
    }
    if (start && goal) {
        start->g = 0;
        start->h = heuristic(start, goal);
        openD.push_back(start);
        openA.push_back(start);
    }
}
```

searching 함수에서는 각각의 지도 초기화와, 완료 여부을 0으로 하여 지도 전체 노드의 방문 상태, 비용, 부모를 초기화, 시작점과 목표점이 설정되어 있다면 시작 노드를 오픈리스트에 추가합니다.

시작점의 g 값은 0으로, h 값은 휴리스틱 계산으로 초기화됩니다.

```
std::vector<Node*> PF::sidenode(Node* node) {
    std::vector<Node*> out;
    static const int dx[4] = {0, 0, 1, -1};
    static const int dy[4] = {1, -1, 0, 0};
    for (int k = 0; k < 4; ++k) {
        int nr = node->y + dy[k];
        int nc = node->x + dx[k];
        if (nr >= 0 && nr < size && nc >= 0 && nc < size) {
            if (!map[nr][nc].obs) out.push_back(&map[nr][nc]);
        }
    }
    return out;
}
```

현재 노드 주변의 상하좌우 4개 방향 노드픽셀을 각각 확인한 후, 반복문을 통해 각각의 가까운 노드의 좌표값을 nr, nc에 저장합니다
이때 저장한 nr, nc를 통해 주변 노드 픽셀이 지도를 벗어나는지 판별하며 장애물을 설치할 때, 초과하지 않도록 합니다.

이를 통해 다익스트라와 A*에서 다음 확장 방향 여부를 결정할 수 있습니다.

```
double PF::heuristic(Node* a, Node* b) {
    int dx = a->x - b->x;
    if (dx < 0) dx = -dx;
    int dy = a->y - b->y;
    if (dy < 0) dy = -dy;

    return dx + dy;
}
```

두 노드가 픽셀로서 정의되는 현재 지도상에서는 맨해튼 거리(dx + dy)를 계산하여 A* 알고리즘에서 휴리스틱 값으로 사용하여 원활하게 거리 계산이 이루어질 수 있도록 하였습니다.

x와 y 좌표 차이를 절댓값으로 계산해 예상 비용을 산출합니다

```
bool PF::dijkstra() {
    if (finishedD) return true;
    if (openD.empty()) return true;

    auto it = std::min_element(openD.begin(), openD.end(),
                               [](Node* A, Node* B){ return A->g < B->g; });
    Node* cur = *it;
    openD.erase(it);

    if (cur->visited) return false;
    cur->visited = true;

    if (cur == goal) { finishedD = true; return true; }

    for (Node* nb : sidenode(cur)) {
        if (nb->visited) continue;
        double ng = cur->g + 1.0;
        if (ng < nb->g) {
            nb->g = ng;
            nb->parent = cur;
            openD.push_back(nb);
        }
    }
    return false;
}
```

다익스트라 알고리즘 한 단계 수행을 처리합니다.
openD에서 g 값이 가장 작은 노드를 선택하고, 방문하지 않았다면 방문 처리 후 도착점인지 확인하며 이후, 도착점이 아니면 주변 노드의 비용을 갱신 후, 더 작은 비용이면 부모 노드를 갱신, 배열에 추가합니다.

탐색 완료 시 finishedD를 true로 설정하여 종료될 수 있도록 하였습니다.

```
bool PF::aStar() {
    if (finishedA) return true;
    if (openA.empty()) return true;

    auto it = std::min_element(openA.begin(), openA.end(),
                               [](Node* A, Node* B){ return (A->g + A->h) < (B->g + B->h); });
    Node* cur = *it;
    openA.erase(it);

    if (cur->visited) return false;
    cur->visited = true;

    if (cur == goal) { finishedA = true; return true; }

    for (Node* nb : sidenode(cur)) {
        if (nb->visited) continue;
        double ng = cur->g + 1.0;
        if (ng < nb->g) {
            nb->g = ng;
            nb->h = heuristic(nb, goal);
            nb->parent = cur;
            openA.push_back(nb);
        }
    }
    return false;
}
```

A* 알고리즘 담당하는 함수입니다.

openA에서 g+h 값이 가장 작은 노드를 선택, 방문 처리 후 도착점 여부를 확인합니다. 주변 노드 중 비용이 갱신 가능한 경우 g와 h를 새로 계산하고 부모 노드를 설정하는 것으로 openA에 추가합니다.

이또한, 탐색 완료 시 finishedA를 true로 설정한 후 종료됩니다.

```
std::vector<Node*> PF::map_way() {
    std::vector<Node*> path;
    if (!goal) return path;
    Node* cur = goal;
    while (cur) {
        path.push_back(cur);
        if (cur == start) break;
        cur = cur->parent;
    }
    std::reverse(path.begin(), path.end());
    return path;
}
```

알고리즘 실행 중 각 노드는 부모 노드의 포인터를 가지며, 이는 해당 노드로 도달하기 직전의 노드를 가리키게되어 도착 노드에서 시작해서 부모 노드를 누적하여 따라가면, 탐색이 목표점까지 어떻게 진행되었는지 역방향으로 이동해 경로 산출이 가능합니다.

과정을 통해 경로를 저장할 임시 경로 리스트를 구성하고, 마지막으로 리스트를 뒤집어 start 노드에서 goal 노드까지의 올바른 순서를 반환하여 완성됩니다.

(알고리즘 특성상 큐가 활용됩니다.)

```
void MainWindow::map_sight() {
    if (!pfA || !pfD) return;
    int n = pfA->size;

    for (int r=0;r<n;++r) {
        for (int c=0;c<n;++c) {
            Node* na = &pfA->map[r][c];
            Node* nd = &pfD->map[r][c];

            if (na->visited && na != pfA->start && na != pfA->goal)
                sceneA->addRect(c*pixel,r*pixel,pixel,pixel,QPen(Qt::gray),QBrush(QColor(173,216,230)));
            if (nd->visited && nd != pfD->start && nd != pfD->goal)
                sceneD->addRect(c*pixel,r*pixel,pixel,pixel,QPen(Qt::gray),QBrush(QColor(255,228,181)));
        }
    }

    auto pathA = pfA->map_way();
    for (auto node : pathA) {
        if (node != pfA->start && node != pfA->goal)
            sceneA->addRect(node->x*pixel,node->y*pixel,pixel,pixel,QPen(Qt::gray),QBrush(Qt::yellow));
    }
    auto pathD = pfD->map_way();
    for (auto node : pathD) {
        if (node != pfD->start && node != pfD->goal)
            sceneD->addRect(node->x*pixel,node->y*pixel,pixel,pixel,QPen(Qt::gray),QBrush(Qt::yellow));
    }
}

void MainWindow::map_scaling() {
    if (!sceneA->sceneRect().isEmpty()){
        ui->A->fitInView(sceneA->sceneRect(), Qt::KeepAspectRatio);
    }
    if (!sceneD->sceneRect().isEmpty()){
        ui->D->fitInView(sceneD->sceneRect(), Qt::KeepAspectRatio);
    }
}
```

탐색 과정 중 방문 노드와 최종 경로를 시각화합니다. 방문한 노드는 연한 색으로 칠해 탐색 확장을 직관적으로 보여줍니다.
경로는 노란색으로 표시, 시작점과 목표점은 색상을 변경하지 않으며 사용자는 최단 경로 탐색이 어떻게 확장되고 있는지, 알고리즘이 어떤 경로를 최종적으로 선택했는지 애니메이션을 통한 시각화를 통해 쉽게 확인할 수 있습니다.

4. 실행 사진