hw3 보고서

20기 인턴 송수민

- 코드 분석

1. 헤더 파일

```
#pragma once
#include <QMainWindow>
#include <QGraphicsScene>
#include <QVector>
#include <QTimer>
#include <QElapsedTimer>
#include <QPoint>
#include <QSet>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

// 공유 맵
enum class CellType : unsigned char {
    Empty = 0,
    Block = 1,
    Start = 2,
    End   = 3
};

struct GridModel {
    int rows = 0;
    int cols = 0;
    QVector<CellType> cells;
    QPoint start{-1, -1};
    QPoint goal {-1, -1};

    void init(int r, int c) {
        rows = r; cols = c;
        cells.fill(CellType::Empty, rows*cols);
        start = QPoint(-1,-1);
        goal  = QPoint(-1,-1);
    }
    inline bool inBounds(int r, int c) const { return r>=0 && c>=0 && r<rows && c<cols; }
    inline int idx(int r, int c) const { return r*cols + c; }
    inline CellType at(int r, int c) const { return cells[idx(r,c)]; }
    inline void set(int r, int c, CellType t){ cells[idx(r,c)] = t; }
};
```

```cpp
// 경로 출력
struct SearchResult {
    bool found = false;
    QVector<int> visited_order;
    QVector<int> path;
    qint64 elapsed_ms = 0;
};

// 알고리즘
class Pathfinder {
public:
    virtual ~Pathfinder() = default;
    virtual SearchResult solve(const GridModel& grid) = 0;
protected:
    static inline int manhattan(int r1, int c1, int r2, int c2){
        return qAbs(r1-r2) + qAbs(c1-c2);
    }
};

class AStarPathfinder : public Pathfinder {
public:
    SearchResult solve(const GridModel& grid) override;
};

class DijkstraPathfinder : public Pathfinder {
public:
    SearchResult solve(const GridModel& grid) override;
};

// MainWindow
class MainWindow : public QMainWindow
{
    Q_OBJECT

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

```
private slots:
    void on_creat_map_clicked();
    void on_set_start_point_clicked();
    void on_set_end_point_clicked();
    void on_creat_random_block_clicked();
    void on_set_random_block_clicked();
    void on_start_clicked();
    void on_reset_clicked();

protected:
    bool eventFilter(QObject* watched, QEvent* event) override;

private:
    enum class EditMode { None, SetStart, SetEnd, ToggleObstacle } m_mode = EditMode::None;

    Ui::MainWindow *ui;
    QGraphicsScene* sceneA = nullptr;
    QGraphicsScene* sceneD = nullptr;

    GridModel grid;
    QVector<int> visitedA, visitedD, pathA, pathD;
    int stepA = 0, stepD = 0;
    QTimer timerA, timerD;

    QVector<int> visitedOrder;
    QVector<int> finalPath;
    int stepIndex = 0;
    QTimer *animTimer = nullptr;

    qreal cellSizeA = 10.0, cellSizeD = 10.0;
    qreal margin = 5.0;

    const int MIN_ROWS = 5,  MAX_ROWS = 100;
    const int MIN_COLS = 5,  MAX_COLS = 100;
    const int MIN_PCT  = 0,  MAX_PCT  = 80;

    void resetScenes();
    void redrawScenes(bool clearExtraLayers = true);
    void drawScene(QGraphicsScene* s, qreal cellSize, const QVector<int>& visited, const QVector<int>& path);
    QPoint cellFromPos(QGraphicsView* view, qreal cellSize, const QPointF& scenePos) const;
    void setStartAt(int r, int c);
```

```
    void setEndAt(int r, int c);
    void toggleObstacleAt(int r, int c);
    void setStatus(const QString& msg);
    void setupSpinRanges();
    void animateResults(const SearchResult& resA, const SearchResult& resD);
};
```

QMainWindow : Qt에서 제공하는 메인 윈도우 클래스. GUI 창 구현에 사용.
QGraphicsScene : 2D 그래픽 씬을 관리하는 클래스. 도형, 아이템, 텍스트 배치 가능.
QVectorQt : 동적 배열 컨테이너. 인덱스 접근과 크기 자동 조절 지원.
QTimer : 주기적 이벤트 신호를 발생시키는 타이머 클래스.
QElapsedTimer : 고정밀 시간 측정을 위한 클래스. 코드 실행 시간, 이벤트 간격 측정 가능.
QPoint : 2D 좌표(x, y)를 저장하고 처리하는 클래스.
QSet :중복 없는 데이터 집합을 저장하는 컨테이너. 검색, 추가, 삭제 효율적.
에이스타와 다익스트라 알고리즘은 경로를 추정할 맵을 공유하므로 맵 설정 클래스와 알고리즘 작동 클래스. 마지막으로 mainwindow ui를 설정하였다.

2. mainwindow.cpp 파일

```cpp
#include "mainwindow.h"
#include "./ui_mainwindow.h"
#include <QMouseEvent>
#include <QRandomGenerator>
#include <QGraphicsRectItem>
#include <QScrollBar>
#include <cmath>
#include <algorithm>

// A*
#include <queue>
#include <limits>

struct PQNode {
    int idx;
    int f, g;
    bool operator<(const PQNode& other) const { return f > other.f; }
};

SearchResult AStarPathfinder::solve(const GridModel& grid) {
    SearchResult out;
    QElapsedTimer t; t.start();

    if (grid.start.x()<0 || grid.goal.x()<0) { out.found = false; out.elapsed_ms = t.elapsed(); return out; }

    const int R = grid.rows, C = grid.cols;
    const int startIdx = grid.idx(grid.start.y(), grid.start.x());
    const int goalIdx  = grid.idx(grid.goal.y(),  grid.goal.x());

    QVector<int> g(R*C, std::numeric_limits<int>::max());
    QVector<int> parent(R*C, -1);
    QVector<bool> closed(R*C, false);
    std::priority_queue<PQNode> open;
```

```
g[startIdx] = 0;
open.push({startIdx, manhattan(grid.start.y(), grid.start.x(), grid.goal.y(), grid.goal.x()), 0});

auto neighbors = [&](int r, int c){
    static const int dr[4] = {-1,1,0,0};
    static const int dc[4] = {0,0,-1,1};
    QVector<QPoint> nb;
    for(int k=0;k<4;k++){
        int nr=r+dr[k], nc=c+dc[k];
        if (grid.inBounds(nr,nc) && grid.at(nr,nc)!=CellType::Block) nb.append(QPoint(nc,nr));
    }
    return nb;
};

while(!open.empty()){
    auto cur = open.top(); open.pop();
    if (closed[cur.idx]) continue;
    closed[cur.idx] = true;
    out.visited_order.append(cur.idx);

    if (cur.idx == goalIdx) {
        out.found = true;
        break;
    }

    int r = cur.idx / C, c = cur.idx % C;
    for (auto p : neighbors(r,c)) {
        int ni = grid.idx(p.y(), p.x());
        if (closed[ni]) continue;
        int tentative = g[cur.idx] + 1;
        if (tentative < g[ni]) {
            g[ni] = tentative;
            parent[ni] = cur.idx;
            int h = manhattan(p.y(), p.x(), grid.goal.y(), grid.goal.x());
            open.push({ni, tentative + h, tentative});
        }
    }
}
```

```
    if (out.found) {
        int cur = goalIdx;
        QVector<int> rev;
        while (cur != -1) { rev.append(cur); cur = parent[cur];
        std::reverse(rev.begin(), rev.end());
        out.path = rev;
    }
    out.elapsed_ms = t.elapsed();
    return out;
}
```

A* 경로 탐색 함수는 먼저 탐색 결과를 담을 구조체와 시간 측정기를 초기화하고, 시작점이나 목표점이 유효하지 않으면 탐색을 종료한다. 그리드의 행과 열, 시작점과 목표점의 인덱스를 계산하고, 각 노드의 최소 비용(g), 부모 노드(parent), 탐색 완료 여부(closed)를 저장할 벡터를 초기화한 뒤, 우선순위 큐(open)를 사용하여 탐색을 진행한다. 시작점의 g 값은

0으로 설정하고, f = g + h(맨해튼 거리) 값을 계산하여 큐에 삽입한다. 인접 노드를 계산하는 람다 함수는 상하좌우로 이동 가능하며, 그리드 범위 내이고 블록이 아닌 좌표만 반환한다.

메인 탐색 루프에서는 우선순위 큐에서 f 값이 가장 작은 노드를 꺼내고, 이미 탐색된 노드라면 건너뛴다. 현재 노드를 탐색 완료 처리하고 방문 순서를 기록하며, 목표 노드에 도달하면 탐색을 종료한다. 그렇지 않으면 인접 노드를 순회하며, 아직 탐색되지 않은 노드에 대해 시작점에서 현재 노드까지의 비용을 계산하고, 기존 비용보다 작으면 g 값과 부모 노드를 갱신한 뒤 f = g + h로 계산하여 큐에 삽입한다.

탐색이 완료되면 목표점부터 부모 노드를 거슬러 올라가 경로를 역순으로 기록한 뒤 정방향으로 뒤집어 최종 경로를 저장한다. 마지막으로 탐색 소요 시간을 기록하고 결과를 반환한다. 이 과정에서 우선순위 큐와 휴리스틱 비용을 활용하여 최적 경로를 효율적으로 탐색하며, 방문 순서와 경로, 소요 시간을 모두 관리한다.

```
// Dijkstra
struct DJNode {
    int idx;
    int dist;
    bool operator<(const DJNode& other) const { return dist > other.dist; }
};

SearchResult DijkstraPathfinder::solve(const GridModel& grid) {
    SearchResult out;
    QElapsedTimer t; t.start();

    if (grid.start.x()<0 || grid.goal.x()<0) { out.found = false; out.elapsed_ms = t.elapsed(); return out; }

    const int R = grid.rows, C = grid.cols;
    const int startIdx = grid.idx(grid.start.y(), grid.start.x());
    const int goalIdx  = grid.idx(grid.goal.y(),  grid.goal.x());

    QVector<int> dist(R*C, std::numeric_limits<int>::max());
    QVector<int> parent(R*C, -1);
    QVector<bool> visited(R*C, false);
    std::priority_queue<DJNode> pq;

    dist[startIdx] = 0;
    pq.push({startIdx, 0});

    auto neighbors = [&](int r, int c){
        static const int dr[4] = {-1,1,0,0};
        static const int dc[4] = {0,0,-1,1};
        QVector<QPoint> nb;
        for(int k=0;k<4;k++){
            int nr=r+dr[k], nc=c+dc[k];
            if (grid.inBounds(nr,nc) && grid.at(nr,nc)!=CellType::Block) nb.append(QPoint(nc,nr));
        }
        return nb;
    };
```

```
    while(!pq.empty()){
        auto cur = pq.top(); pq.pop();
        if (visited[cur.idx]) continue;
        visited[cur.idx] = true;
        out.visited_order.append(cur.idx);

        if (cur.idx == goalIdx) { out.found = true; break; }

        int r = cur.idx / C, c = cur.idx % C;
        for (auto p : neighbors(r,c)) {
            int ni = grid.idx(p.y(), p.x());
            if (visited[ni]) continue;
            int nd = dist[cur.idx] + 1;
            if (nd < dist[ni]) {
                dist[ni] = nd;
                parent[ni] = cur.idx;
                pq.push({ni, nd});
            }
        }
    }

    if (out.found) {
        int cur = goalIdx;
        QVector<int> rev;
        while (cur != -1) { rev.append(cur); cur = parent[cur]; }
        std::reverse(rev.begin(), rev.end());
        out.path = rev;
    }
    out.elapsed_ms = t.elapsed();
    return out;
}
```

다익스트라 탐색 함수는 먼저 탐색 결과 구조체를 초기화하고, QElapsedTimer로 탐색 소요 시간을 측정한다. 시작점이나 목표점이 유효하지 않으면 탐색을 종료한다. 그리드의 행과 열, 시작점과 목표점의 인덱스를 계산하고, 각 노드의 최소 거리(dist), 부모 노드(parent), 방문 여부(visited)를 저장할 벡터를 초기화하며, 우선순위 큐(pq)를 사용하여 탐색을 진행한다. 시작점의 거리를 0으로 설정하고, 큐에 삽입한다.

인접 노드를 계산하는 람다 함수는 상하좌우 방향으로 이동 가능한 좌표를 반환하며, 그리드 범위 내이고 블록이 아닌 좌표만 포함한다. 메인 탐색 루프에서는 우선순위 큐에서 최소 거리를 가진 노드를 꺼내고, 이미 방문된 노드라면 건너뛴다. 현재 노드를 방문 처리하고 방문 순서를 기록하며, 목표 노드에 도달하면 탐색을 종료한다. 그렇지 않으면 인접 노드를 순회하며, 아직 방문하지 않은 노드에 대해 시작점에서 현재 노드까지의 거리(nd)를 계산하

고 기존 거리보다 작으면 갱신하고 부모 노드를 저장한 뒤 큐에 삽입한다.

탐색이 완료되면 목표점부터 부모 노드를 따라 역순으로 경로를 기록하고, 최종 경로는 정방향으로 뒤집어 저장한다. 마지막으로 탐색 소요 시간을 기록하고 결과를 반환한다. 이 과정에서 우선순위 큐를 사용하여 최소 거리 기반으로 노드를 선택하며, 방문 순서, 최단 경로, 탐색 시간을 모두 관리한다.

```
// ---------------- MainWindow ----------------
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    sceneA = new QGraphicsScene(this);
    sceneD = new QGraphicsScene(this);
    ui->A_star->setScene(sceneA);
    ui->dijkstra->setScene(sceneD);

    ui->A_star->viewport()->installEventFilter(this);
    ui->dijkstra->viewport()->installEventFilter(this);

    setupSpinRanges();

    connect(&timerA, &QTimer::timeout, this, [this](){
        if (stepA < visitedA.size()) {
            stepA++;
            redrawScenes(false);
        } else if (stepA == visitedA.size()) {
            stepA++;
            redrawScenes(false);
        } else {
            timerA.stop();
        }
    });
    connect(&timerD, &QTimer::timeout, this, [this](){
        if (stepD < visitedD.size()) {
            stepD++;
            redrawScenes(false);
        } else if (stepD == visitedD.size()) {
            stepD++;
            redrawScenes(false);
        } else {
            timerD.stop();
        }
    });
```

```
    ui->start->setStyleSheet("background-color: green;");
    ui->reset->setStyleSheet("background-color: red;");

    setStatus("Ready. ROW/COL 설정 후 creat_map을 누르세요.");
}
MainWindow::~MainWindow()
{
    delete ui;
}
```

MainWindow 생성자에서는 먼저 UI를 초기화하고, A*와 다익스트라 알고리즘을 위한 QGraphicsScene을 각각 생성하여 그래픽 뷰에 연결한다. 각 뷰포트에 이벤트 필터를 설치하여 사용자의 마우스 입력이나 키보드 입력을 처리할 수 있도록 준비한다. setupSpinRanges() 함수를 호출하여 행과 열을 설정하는 스핀 박스 범위를 초기화한다.

타이머(timerA, timerD)를 연결하여 일정 시간 간격으로 redrawScenes() 함수를 호출하며 탐색 과정을 단계별로 시각화한다. 타이머는 각 알고리즘의 방문 노드 수만큼 반복하며, 모든 노드를 그린 후 자동으로 정지한다. 시작 버튼과 리셋 버튼의 배경색을 각각 초록색과 빨간색으로 설정하여 상태를 시각적으로 표시하고, 상태바에는 초기 안내 메시지를 출력한다.

MainWindow 소멸자에서는 UI 객체를 해제하여 메모리 누수를 방지한다. 이 구조를 통해, 사용자는 행/열 설정 후 맵을 생성하고, 두 알고리즘의 탐색 과정을 실시간으로 비교하며 시각적으로 확인할 수 있다.

```cpp
void MainWindow::setupSpinRanges(){
    ui->row->setRange(MIN_ROWS, MAX_ROWS);
    ui->spinBox_2->setRange(MIN_COLS, MAX_COLS);
    ui->how_many_block->setRange(MIN_PCT, MAX_PCT);
    ui->how_many_block->setSuffix("%");
}

void MainWindow::resetScenes(){
    sceneA->clear();
    sceneD->clear();
    stepA = stepD = 0;
    visitedA.clear(); visitedD.clear();
    pathA.clear(); pathD.clear();
}

void MainWindow::redrawScenes(bool clearExtraLayers){
    sceneA->setSceneRect(0,0, ui->A_star->viewport()->width(), ui->A_star->viewport()->height());
    sceneD->setSceneRect(0,0, ui->dijkstra->viewport()->width(), ui->dijkstra->viewport()->height());

    if (grid.rows>0 && grid.cols>0){
        cellSizeA = std::floor(std::min(
            (sceneA->width()-2*margin)/qreal(grid.cols),
            (sceneA->height()-2*margin)/qreal(grid.rows)
            ));
        cellSizeD = std::floor(std::min(
            (sceneD->width()-2*margin)/qreal(grid.cols),
            (sceneD->height()-2*margin)/qreal(grid.rows)
            ));

        if (cellSizeA < 3) cellSizeA = 3;
        if (cellSizeD < 3) cellSizeD = 3;
    }

    if (clearExtraLayers) { sceneA->clear(); sceneD->clear(); }

    QVector<int> curPathA = (stepA > visitedA.size()) ? pathA : QVector<int>{};
    QVector<int> curPathD = (stepD > visitedD.size()) ? pathD : QVector<int>{};

    drawScene(sceneA, cellSizeA, visitedA.mid(0, stepA), curPathA);
    drawScene(sceneD, cellSizeD, visitedD.mid(0, stepD), curPathD);
}
```

setupSpinRanges() 함수는 사용자 입력용 스핀 박스의 최소값과 최대값 범위를 설정한다. 행(row)과 열(spinBox_2)은 MIN_ROWSMAX_ROWS, MIN_COLSMAX_COLS 범위로 지정되며, 장애물 개수를 나타내는 스핀 박스(how_many_block)는 최소/최대 퍼센트와 함께 % 단위를 표시하도록 설정된다.
resetScenes() 함수는 A*와 다익스트라 장면을 모두 초기화하고, 탐색 단계(stepA, stepD)를 0으로 초기화하며, 각 알고리즘의 방문 노드(visitedA, visitedD)와 경로(pathA, pathD)를 비워 새 탐색 준비를 한다.

redrawScenes() 함수는 그래픽 뷰의 크기에 맞게 씬의 영역을 조정하고, 그리드의 행과 열에 따라 각 셀의 크기를 계산한다. 셀 크기는 화면 크기와 그리드 크기를 고려하여 자동 조

정되며, 최소 크기를 3으로 제한한다. clearExtraLayers가 true이면 이전에 그려진 모든 아이템을 제거하여 깨끗한 상태에서 다시 그릴 수 있다. 현재 탐색 단계에 따라, 각 알고리즘의 방문 노드와 경로를 잘라서 선택적으로 표시하며, drawScene() 함수를 통해 씬에 그린다.

```
void MainWindow::drawScene(QGraphicsScene* s, qreal cellSize, const QVector<int>& visited, const QVector<int>& path){
    s->clear();

    const int R = grid.rows, C = grid.cols;
    for(int r=0;r<R;r++){
        for(int c=0;c<C;c++){
            QRectF rect(margin + c*cellSize, margin + r*cellSize, cellSize, cellSize);
            QBrush brush(Qt::white);
            CellType t = grid.at(r,c);
            if (t == CellType::Block) brush = QBrush(Qt::black);
            else if (t == CellType::Start) brush = QBrush(Qt::green);
            else if (t == CellType::End)   brush = QBrush(Qt::red);
            auto *item = s->addRect(rect, QPen(Qt::lightGray), brush);
            item->setZValue(0);
        }
    }
    for (int idx : visited){
        int r = idx / C, c = idx % C;
        if (grid.at(r,c)==CellType::Start || grid.at(r,c)==CellType::End) continue;
        QRectF rect(margin + c*cellSize, margin + r*cellSize, cellSize, cellSize);
        auto *item = s->addRect(rect.adjusted(1,1,-1,-1), QPen(Qt::NoPen), QBrush(Qt::magenta));
        item->setOpacity(0.6);
        item->setZValue(1);
    }
    if (!path.isEmpty()){
        for (int idx : path){
            int r = idx / C, c = idx % C;
            QRectF rect(margin + c*cellSize, margin + r*cellSize, cellSize, cellSize);
            auto *item = s->addRect(rect.adjusted(1,1,-1,-1), QPen(Qt::NoPen), QBrush(Qt::blue));
            item->setOpacity(0.8);
            item->setZValue(2);
        }
    }
}
```

drawScene() 함수는 지정된 QGraphicsScene에 현재 그리드 상태와 알고리즘 탐색 진행 상황을 그린다. 먼저 씬을 초기화(clear)하고, 그리드의 행(R)과 열(C)을 기준으로 각 셀의 위치와 크기를 계산한다. 각 셀은 기본적으로 흰색으로 그려지며, 블록(CellType::Block)은 검은색, 시작점(CellType::Start)은 녹색, 목표점(CellType::End)은 빨간색으로 채워 시각적으로 구분된다. 모든 셀은 낮은 z값(Z-value 0)으로 설정되어 기본 레이어를 만든다.
그 다음, 방문한 노드(visited)를 순회하며 시작점과 목표점을 제외한 셀을 자홍색으로 표시하고, 불투명도를 0.6으로 설정하여 이전 레이어 위에 표시한다. z값을 1로 지정해 방문 노드 레이어가 기본 맵 위에 나타나도록 한다.
최종 경로(path)가 존재하면 각 셀을 파란색으로 표시하고 불투명도를 0.8로 설정하며, z값 2로 지정하여 방문 노드보다 위 레이어에 표시한다.

```
QPoint MainWindow::cellFromPos(QGraphicsView* view, qreal cellSize, const QPointF& scenePos) const {
    qreal x = scenePos.x() - margin;
    qreal y = scenePos.y() - margin;
    int c = int(x / cellSize);
    int r = int(y / cellSize);
    if (!grid.inBounds(r,c)) return QPoint(-1,-1);
    return QPoint(c,r);
}

void MainWindow::setStartAt(int r, int c){
    if (!grid.inBounds(r,c)) return;
    if (grid.start.x() >= 0) grid.set(grid.start.y(), grid.start.x(), CellType::Empty);
    if (grid.at(r,c)==CellType::Block || grid.at(r,c)==CellType::End) return;
    grid.set(r,c, CellType::Start);
    grid.start = QPoint(c,r);
    setStatus(QString("Start: (%1,%2)").arg(r).arg(c));
    redrawScenes();
}

void MainWindow::setEndAt(int r, int c){
    if (!grid.inBounds(r,c)) return;
    if (grid.goal.x() >= 0) grid.set(grid.goal.y(), grid.goal.x(), CellType::Empty);
    if (grid.at(r,c)==CellType::Block || grid.at(r,c)==CellType::Start) return;
    grid.set(r,c, CellType::End);
    grid.goal = QPoint(c,r);
    setStatus(QString("End: (%1,%2)").arg(r).arg(c));
    redrawScenes();
}
```

cellFromPos() 함수는 그래픽 뷰에서 받은 장면 좌표(scenePos)를 기준으로 그리드의 셀 좌표를 계산한다. 장면 좌표에서 마진을 제외하고 셀 크기로 나누어 행(r)과 열(c)을 구하며, 계산된 좌표가 그리드 범위 밖이면 (-1,-1)을 반환하여 유효하지 않음을 표시한다.

setStartAt() 함수는 지정된 행과 열에 시작점을 설정한다. 먼저 좌표가 그리드 범위 내인지 확인하고, 기존 시작점이 존재하면 해당 셀을 비워 초기화한다. 새 위치가 블록이나 목표점일 경우에는 설정을 무시한다. 조건을 만족하면 그리드에 시작점(CellType::Start)을 표시하고, 시작점 좌표를 갱신하며 상태바에 정보를 출력한다. 마지막으로 장면을 다시 그려 변경 사항을 보여준다.

setEndAt() 함수도 setStartAt()과 유사하게 동작하며, 목표점을 설정한다. 기존 목표점이 있으면 초기화하고, 새 위치가 블록이나 시작점이면 무시한다. 유효한 경우 그리드에 목표점(CellType::End)을 표시하고 좌표를 갱신하며 상태바에 정보를 출력하고 장면을 다시 그린다.

```cpp
void MainWindow::toggleObstacleAt(int r, int c){
    if (!grid.inBounds(r,c)) return;
    auto t = grid.at(r,c);
    if (t == CellType::Empty) {
        if (grid.start == QPoint(c,r) || grid.goal == QPoint(c,r)) return;
        grid.set(r,c, CellType::Block);
    } else if (t == CellType::Block) {
        grid.set(r,c, CellType::Empty);
    } else {
        return;
    }
    redrawScenes();
}

void MainWindow::setStatus(const QString& msg){
    ui->state_text->setText(msg);
}
```

toggleObstacleAt() 함수는 지정된 행(r)과 열(c) 좌표의 셀을 장애물로 설정하거나 기존 장애물을 제거한다. 먼저 좌표가 그리드 범위 내인지 확인하고, 셀이 비어있을 경우 시작점이나 목표점이 아니라면 블록(CellType::Block)으로 설정한다. 이미 블록으로 설정되어 있는 셀은 다시 비어있는 상태(CellType::Empty)로 되돌린다. 시작점이나 목표점은 변경하지 않고 무시한다. 변경 후에는 redrawScenes()를 호출하여 장면을 다시 그려 즉시 보여준다.

setStatus() 함수는 상태 메시지를 받아 UI의 상태 텍스트 영역(state_text)에 출력한다.

```
// 연결
void MainWindow::on_creat_map_clicked(){
    int r = ui->row->value();
    int c = ui->spinBox_2->value();
    grid.init(r,c);
    resetScenes();
    redrawScenes();
    setStatus(QString("맵 생성: %1 x %2 격자").arg(r).arg(c));
}

void MainWindow::on_set_start_point_clicked(){
    m_mode = EditMode::SetStart;
    ui->set_start_point->setStyleSheet("background-color: blue;");
    ui->set_end_point->setStyleSheet("");
    ui->set_random_block->setStyleSheet("");
    setStatus("시작 지점을 클릭하여 설정하세요.");
}

void MainWindow::on_set_end_point_clicked(){
    m_mode = EditMode::SetEnd;
    ui->set_start_point->setStyleSheet("");
    ui->set_end_point->setStyleSheet("background-color: blue;");
    ui->set_random_block->setStyleSheet("");
    setStatus("끝 지점을 클릭하여 설정하세요.");
}

void MainWindow::on_set_random_block_clicked(){
    m_mode = EditMode::ToggleObstacle;
    ui->set_start_point->setStyleSheet("");
    ui->set_end_point->setStyleSheet("");
    ui->set_random_block->setStyleSheet("background-color: blue;");
    setStatus("빈 칸 클릭: 장애물 추가 / 장애물 클릭: 제거");
}
```

“맵 생성” 버튼을 누르면 스핀 박스에서 행과 열 값을 가져와 그리드를 초기화하고(grid.init) 이전 탐색 결과와 장면을 초기화(resetScenes())한 후 장면을 다시 그려(redrawScenes()) 새로운 맵을 화면에 보여준다. 상태바에는 생성된 맵의 크기를 표시한다.

“시작 지점 설정” 버튼을 누르면 편집 모드를 시작점 설정으로 바꾸고(m_mode =

EditMode::SetStart), 버튼 색을 파란색으로 바꿔 현재 모드를 시각적으로 표시한다. 동시에 다른 버튼들은 원래 색으로 초기화하며, 상태바에는 “시작 지점을 클릭하여 설정하세요.”라는 안내 메시지를 띄운다.

“끝 지점 설정” 버튼도 비슷하게 동작한다. 편집 모드를 목표점 설정으로 바꾸고, 버튼 색을 파란색으로 표시하며 다른 버튼 색은 초기화한다. 상태바에는 “끝 지점을 클릭하여 설정하세요.”라는 안내 메시지가 나온다.

“랜덤 블록 설정” 버튼을 누르면 편집 모드를 장애물 추가/제거 모드로 바꾸고, 버튼 색을 파란색으로 표시한다. 다른 버튼 색은 초기화하며, 상태바에는 “빈 칸 클릭: 장애물 추가 / 장애물 클릭: 제거”라는 안내 메시지를 띄워 사용자가 클릭 동작으로 장애물을 토글할 수 있음을 알려준다.

```
void MainWindow::on_creat_random_block_clicked(){
    if (grid.rows==0) { setStatus("먼저 creat_map으로 맵을 만드세요."); return; }
    int pct = ui->how_many_block->value();
    pct = qBound(MIN_PCT, pct, MAX_PCT);
    int total = grid.rows * grid.cols;
    int toPlace = (total * pct) / 100;

    for (int r=0;r<grid.rows;r++){
        for(int c=0;c<grid.cols;c++){
            if (grid.at(r,c)==CellType::Block) grid.set(r,c, CellType::Empty);
        }
    }

    int placed = 0;
    while (placed < toPlace) {
        int r = QRandomGenerator::global()->bounded(grid.rows);
        int c = QRandomGenerator::global()->bounded(grid.cols);
        if (grid.at(r,c)==CellType::Empty && QPoint(c,r)!=grid.start && QPoint(c,r)!=grid.goal){
            grid.set(r,c, CellType::Block);
            placed++;
        }
    }
    redrawScenes();
    setStatus(QString("무작위 장애물 배치: %1% (%2개)").arg(pct).arg(toPlace));
}
```

“무작위 블록 생성” 버튼을 누르면 먼저 맵이 생성되어 있는지 확인한다. 맵이 없으면 상태바에 “먼저 creat_map으로 맵을 만드세요.”를 표시하고 함수를 종료한다.

그 다음, 스핀 박스에서 설정한 장애물 비율(pct)을 가져와 최소·최대 값 범위(MIN_PCT, MAX_PCT)로 제한한다. 전체 칸 수(total)를 계산하고, 그 비율에 맞춰 배치할 장애물 개수(toPlace)를 정한다.

기존에 배치되어 있는 장애물은 모두 지워 초기화하고, 그 후 무작위로 빈 칸을 선택해 장애물을 하나씩 배치한다. 단, 시작점과 목표점에는 장애물이 배치되지 않도록 체크한다. 배치가 완료되면 장면을 다시 그려(redrawScenes()) 새로운 장애물 상태를 화면에 표시하고,

상태바에는 “무작위 장애물 배치: XX% (YY개)”와 같이 몇 퍼센트, 몇 개의 장애물이 배치되었는지 보여준다.

```
void MainWindow::on_start_clicked(){
    if (grid.rows==0) { setStatus("맵이 없습니다. creat_map을 먼저 누르세요."); return; }
    if (grid.start.x()<0 || grid.goal.x()<0) {
        setStatus("시작/끝 지점이 설정되지 않았습니다.");
        return;
    }

    AStarPathfinder astar;
    DijkstraPathfinder dijkstra;
    auto resA = astar.solve(grid);
    auto resD = dijkstra.solve(grid);

    visitedA = resA.visited_order;
    visitedD = resD.visited_order;
    pathA = resA.found ? resA.path : QVector<int>{};
    pathD = resD.found ? resD.path : QVector<int>{};
    stepA = stepD = 0;
    redrawScenes();

    QString msg;
    if (!resA.found && !resD.found) {
        msg = QString("길이 막혔습니다. (A*: %1ms, Dijkstra: %2ms)").arg(resA.elapsed_ms).arg(resD.elapsed_ms);
    } else {
        msg = QString("시작!  A*: %1ms (%2칸 방문), Dijkstra: %3ms (%4칸 방문)")
                  .arg(resA.elapsed_ms).arg(resA.visited_order.size())
                  .arg(resD.elapsed_ms).arg(resD.visited_order.size());
        if (resA.found) msg += QString("₩nA* 경로 길이: %1").arg(resA.path.size());
        if (resD.found) msg += QString("₩nDijkstra 경로 길이: %1").arg(resD.path.size());
    }
    setStatus(msg);

    timerA.start(10);
    timerD.start(10);
}
```

“시작” 버튼을 누르면 먼저 맵이 존재하는지 확인한다. 맵이 없으면 상태바에 “맵이 없습니다. creat_map을 먼저 누르세요.”를 표시하고 함수를 종료한다. 다음으로 시작점과 목표점이 설정되어 있는지 확인하고, 설정되지 않았다면 “시작/끝 지점이 설정되지 않았습니다.”라는 메시지를 띄우고 종료한다.

조건이 모두 만족되면, A* 알고리즘과 다익스트라 알고리즘 객체를 생성하고 각각의 solve() 함수를 호출하여 그리드에 대한 탐색을 수행한다. 탐색 결과로 각 알고리즘이 방문한 노드 순서(visited_order)와 경로(path)를 받아와 저장하고, 단계 변수(stepA, stepD)를 0으로 초기화한 뒤 장면을 다시 그려(redrawScenes()) 현재 상태를 화면에 표시한다.

탐색 결과를 분석하여 상태바 메시지를 만든다. 두 알고리즘 모두 경로를 찾지 못하면 “길이 막혔습니다.”라는 메시지와 탐색 소요 시간을 표시하고, 하나라도 경로를 찾으면 방문한 칸 수, 탐색 소요 시간, 경로 길이를 함께 표시한다.

마지막으로 타이머를 시작하여(timerA, timerD) 단계별 탐색 경로가 화면에 순차적으로 나

타나도록 한다.

```
void MainWindow::on_reset_clicked(){
    if (grid.rows==0) { setStatus("초기화할 맵이 없습니다."); return; }
    int r = grid.rows, c = grid.cols;
    grid.init(r,c);
    resetScenes();
    redrawScenes();
    setStatus("리셋 완료. 맵이 비워졌습니다.");
}

bool MainWindow::eventFilter(QObject* watched, QEvent* event){
    if (event->type() == QEvent::MouseButtonPress){
        if (grid.rows==0) return false;
        auto *view = (watched == ui->A_star->viewport()) ? ui->A_star :
                     (watched == ui->dijkstra->viewport()) ? ui->dijkstra : nullptr;
        if (!view) return false;
        auto *me = static_cast<QMouseEvent*>(event);
        QPointF scenePos = view->mapToScene(me->pos());
        qreal cs = (view == ui->A_star) ? cellSizeA : cellSizeD;
        QPoint cell = cellFromPos(view, cs, scenePos);
        if (cell.x()<0) return false;
        int r = cell.y(), c = cell.x();

        switch (m_mode){
        case EditMode::SetStart: setStartAt(r,c); break;
        case EditMode::SetEnd:   setEndAt(r,c);   break;
        case EditMode::ToggleObstacle: toggleObstacleAt(r,c); break;
        default: break;
        }
        return true;
    }
    return QMainWindow::eventFilter(watched, event);
}
```

“리셋” 버튼을 누르면 먼저 맵이 존재하는지 확인한다. 맵이 없으면 상태바에 “초기화할 맵이 없습니다.”를 표시하고 함수를 종료한다.
맵이 존재하면 현재 맵의 행과 열 크기를 가져와 grid.init()으로 그리드를 초기화한다. 그리고 내부 상태(resetScenes())를 초기화하고, 장면(redrawScenes())을 다시 그려서 화면을 비운다. 마지막으로 상태바에 “리셋 완료. 맵이 비워졌습니다.”를 표시한다.

장면 위에서 마우스 버튼이 눌리면 먼저 맵이 존재하는지 확인한다. 맵이 없으면 이벤트를 무시한다.
그 다음, 이벤트가 발생한 뷰포트가 A* 시각화 장면인지, 다익스트라 시각화 장면인지 확인하고, 해당 뷰를 가져온다. 뷰가 아니라면 이벤트를 무시한다.
마우스 좌표를 장면 좌표로 변환한 후, cellFromPos()를 이용해 어떤 셀(격자)에 클릭했는

지 계산한다. 셀이 맵 범위를 벗어나면 이벤트를 무시한다.
마지막으로 현재 편집 모드(m_mode)에 따라 클릭한 셀을 처리한다.
SetStart 모드라면 시작점을 설정하고,
SetEnd 모드라면 끝점을 설정하고,
ToggleObstacle 모드라면 장애물을 토글(추가/제거)한다.
처리가 완료되면 이벤트를 처리했다고 true를 반환하고, 그렇지 않으면 기본 QMainWindow 이벤트 처리로 넘긴다.

3. main.cpp 파일

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.setWindowTitle("길 찾기");
    w.show();
    return a.exec();
}
```

창 이름 설정 후 실행한다.