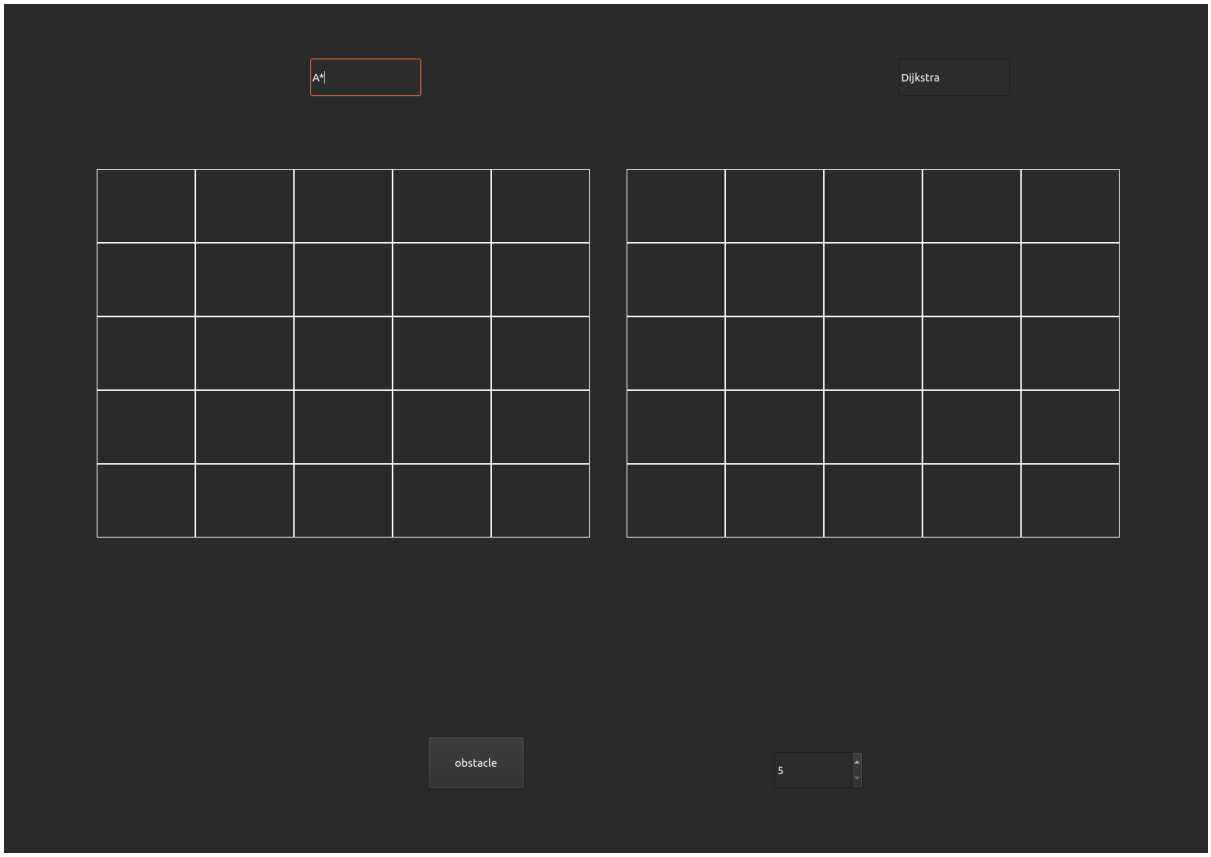
C++&QT DAY3
3번 과제 보고서

2023741024
로봇학부 박건후

목차

(1) ui 설명



현재 A*와 데이크스트라에 대한 2개의 위젯을 배치하고 아래의 obstacle이라는 버튼을 배치하여 해당 버튼을 누르면 전체 그리드의 30퍼센트를 랜덤으로 장애물로 설정하고 이를 흰색으로 색칠하도록 하였습니다. 그리고 obstacle 옆에 있는 스핀 박스를 통해 맵의 사이즈를 구현하고자 했습니다. 스핀 박스의 값이 5이면 5x5의 맵이 나오고 10이면 같은 위젯 크기 내에서 10x10의 맵이 나오는 것입니다.

(1) mainwindow.h 파일 설명

```
void on_obstacle_button_clicked();
void on_spinBox_valueChanged(int arg1);
```

slot메서드를 선언한 부분입니다. 여기서는 obstacle버튼을 누를 경우에 대한 슬롯 메서드와 spinBox의 값을 바꿀 경우 그리드의 사각형 정보를 다시 계산하고 update를 통해 paintEvent로 그리는 것까지 수행하는 메서드입니다.

```
void paintEvent(QPaintEvent* event) override;
bool eventFilter(QObject* obj, QEvent* ev) override;
void cal();
```

메서드로는 그림을 그려주는 paintEvent 메서드, 사각형을 클릭하였을 때 이벤트를 처리해줄 메서드인 eventFilter, 그리고 시작점, 끝점을 정하고 각각의 알고리즘을 통해 확정된 길들을 색칠할 cal메서드까지 선언하였습니다. 하지만 이를 시간 내에 완성하지 못하여 cal메서드를 구현하지 못했습니다.

```
void get_grid(int val);
```

get_grid라는 메서드를 통해 맵의 크기를 입력받아서 그리드의 각 사각형 위치를 계산하고 아래 사진의 A_rect, Dijk_rect를 push_back으로 초기화를 해주는 메서드입니다.

```
std::vector<QRect> A_rect;
std::vector<QRect> Dijk_rect;
```

선언한 멤버변수로는 위의 사진에서 그리드의 사각형 정보를 저장할 벡터입니다. 이는 ui에서 보셨던 2개의 위젯 내부 좌표계를 기준으로 한 사각형 정보를 저장하도록 하였습니다.

```
std::vector<bool> A_selected;
std::vector<bool> Dijk_selected;
std::vector<bool> obstacle; //장
```

위 사진의 벡터 멤버변수는 그리드의 사각형 중 어떤 사각형이 클릭된 것인지 저장하기 위해 bool형으로 선언하였습니다. 또한 전체 그리드 중 30퍼센트의 장애물이 어떤 인덱스의 사각형인지 저장하기 위해 bool형으로 선언하였습니다.

(2) mainwindow.cpp 파일 설명

```
ui->setupUi(this);
ui->spinBox->setRange(5, 30); //스핀박스
ui->widget->installEventFilter(this); /
ui->widget_2->installEventFilter(this);
```

MainWindow의 생성자입니다. 여기에서는 setRange로 스핀박스의 범위를 생성하여 너무 많은 사각형들이 생기지 않도록 30으로 제한하고 5개 이상으로 맵을 구성하도록 한 것입니다. 그리고 각 위젯에 대해 installEventFilter 메서드를 호출하여 이벤트가 발생했을 시 MainWindow의 EventFilter가 호출되도록 한 것입니다. 이렇게 한 이유는 출발점과 끝점을 위젯의 사각형을 눌러서 설정하게 되는데 그렇게 되면 해당 위젯 클래스에 해당 이벤트에 대한 메서드를 오버라이딩하여야 합니다. 하지만 이 메서드를 오버라이딩하기 위해 새로운 클래스를 생성하여 코드를 길게 하고 싶지 않았고 최대한 MainWindow 클래스 내에서 이를 해결하고자 했습니다. 그래서 이벤트가 터졌을 때 바로 해당 위젯의 이벤트 메서드로 흐름이 넘어가기 전에 MainWindow클래스에서 먼저 이 이벤트에 대한 처리를 할 수 있도록 한 것입니다. 그래서 this를 인수로 넣어주어 MainWindow에서 EventFilter 메서드를 오버라이딩하여 이벤트가 발생했을 때 MainWindow에서 바로 처리할 수 있도록 설정하였습니다.

```
void MainWindow::on_spinBox_valueChanged(int arg1)
{
    get_grid(arg1); //스핀 박스 값 바뀌면 다시 그리드 계
}
```

이는 스핀 박스 값이 바뀔 때 호출되는 슬롯 메서드입니다. 여기에서는 get_grid메서드를 호출하고 바뀐 스핀 박스 값으로 nxn 맵을 만들어야 하기에 arg1인수를 그대로 넣어준 것입니다. nxn맵의 n이 바뀌게 되면 그에 따라 사각형 하나의 크기도 바뀌기 때문에 다시 사각형 좌표를 바뀐 값에 따라 계산해주고 벡터를 초기화한 후 다시 push_back을 해줘야 합니다. 그렇기에 이 기능을 수행해줄 get_grid를 바로 호출하도록 한 것입니다.

```
QRect r = ui->widget->rect(); /
A_rect.clear(); //스핀박스 값이 바
Dijk_rect.clear();
```

위 사진은 get_grid의 구현부입니다. 여기서는 우선 사각형을 계산하기 위해서는 위젯의 width, height라는 정보를 알아야 했기에 QWidget::rect메서드를 사용했습니다. 그리고 스핀박스 값이 여러 번 바뀌면 그에 따라 이전 정보를 지워야 하기에 clear를 통해 벡터를 초기화했습니다.

```
for (int i=0; i<val; i++) {
    for (int j=0; j<val; j++) {
        QRect rect(j*r.width()/val, i*r.height()/val, r.width()/val, r.height()/val);
        A_rect.push_back(rect);  //그리고 이를 저장해줌 이 때는 상대적인 좌표이기에, ui->widget(
        Dijk_rect.push_back(rect); //한꺼번에 처리해도 괜찮음
    }
}
```

그리고 이제 사각형의 위치를 잡아서 QRect 객체를 만들어줘야 합니다. 사각형의 좌상단 좌표와 너비, 높이를 넣어주면 되므로 위젯의 너비를 n으로 나눈 값으로 한 사각형의 너비를 구하고 높이도 이와 같은 방식으로 구하여 이를 생성자의 인수로 넣어준 것입니다. 그리고 push_back을 통해 해당 사각형을 각각의 벡터에 넣어줬습니다. ui->widget에 대한 rect()이기에 데이크스트라 위젯에는 좌표가 안 맞다고 할 수도 있겠지만 ui->widget->rect()에서는 위젯 내부의 좌표계이기 때문에 같은 크기의 2개의 위젯이므로 하나의 연산만으로 각각의 벡터에 사각형 정보를 넣어줄 수 있는 것입니다.

```
A_selected.assign(A_rect.size(), false);  //같
Dijk_selected.assign(Dijk_rect.size(), false);
update();
```

그리고 그만큼 마우스로 선택되는 인덱스를 저장하는 벡터 또한 크기가 바뀌기에 이렇게 resize를 해준 것이고 false로 모두 초기화해주는 부분입니다. 그리고 이렇게 그리드가 바뀐 것을 update하여 그 그리드를 바로 그려 사용자가 스핀박스 값을 바꿨을 때 해당 값으로 nxn맵이 만들어지는 것이 보이도록 하였습니다.

```
int size = ui->spinBox->value(); //
obstacle.resize(size*size, false);
int cnt=0;
```

이렇게 get_grid를 하고 나면 장애물을 설정해야 합니다. 위 사진은 장애물 버튼에 대한 슬롯 메서드 구현의 일부분입니다. 스핀 박스의 값을 가져오는 이유는 obstacle이라는 벡

터의 크기를 설정하기 위해서입니다.

```
while(cnt<obstacle.size()*0.3){ /
    int i=rand()%obstacle.size();
    if(obstacle[i]==false){ //만약
        obstacle[i] = true;
      cnt++;
    }
```

그 후에는 전체 사각형 개수 중 30퍼센트를 랜덤으로 하여 랜덤 값을 정하고 해당 인덱스를 true로 하여 장애물이 인덱스에 해당되는 사각형에 설정됨을 저장하는 부분입니다. 이 때 같은 랜덤값이 나올 수 있으므로 obstacle[i] ==false를 하였습니다. 그리고 이 장애물이 설정된 것을 사용자에게 보이기 위해 update를 하였습니다.

```
QPoint offA = ui->widget->mapTo(this, QPoint(0,0));
p.save(); //현 상태 저장
p.translate(offA); //해당 좌표로 이동
```

paintEvent의 구현부분입니다. paintEvent는 MainWindow에 대한 것이기에 QPainter p(this);로 선언한 상태입니다. 그렇기에 여기서는 위젯의 좌표계가 아닌 MainWindow창의 좌표계인 상태입니다. 그렇기에 mapTo를 통해 해당 위젯이 MainWindow 좌표계에서 0,0에 해당 되는 좌표를 구한 것입니다. 이를 통해 MainWindow의 좌표계의 0,0과 대응되는 QWidget의 좌상단 좌표값을 MainWindow 좌표계를 기준으로 구할 수 있었습니다. 그리고 이 점을 저장하였습니다. 그리고 위젯의 내부 좌표계처럼 사용하고자 translate를 사용하였고 이 때 좌표계가 이동됨에 따라 문제가 발생할 수 있으니 p.save()를 해둔 것입니다.

```
for (int i=0; i<(int)A_rect.size(); ++i) {
    if(i<obstacle.size()&&obstacle[i]){ //장애물인 경우 검은색으로 칠하기
        p.fillRect(A_rect[i].adjusted(1,1,-1,-1), QColor(255,255,255));
    }
```

translate를 하고 위젯의 내부 좌표계처럼 사용할 수 있기에 A_rect라는 사각형 좌표를 그대로 넘겨서 장애물 사각형을 흰색으로 쉽게 칠할 수 있었습니다.

```
    if (A_selected[i]) {  //마우스 클릭 받은 사각형의 경우
        p.fillRect(A_rect[i].adjusted(1,1,-1,-1), QColor(255,255,0,120));
    }
    p.drawRect(A_rect[i].adjusted(0,0,-1,-1)); //그리드 사각형 정보대로 네모 그
}
```

그리고 마우스를 클릭하여 사각형을 선택한 경우 그 사각형을 색깔로 칠하도록 했고, 기본적으로 grid 사각형 선들을 그려야 하므로 if문 없이 모든 사각형을 drawRect로 그렸습니다. 이는 데이크스트라 위젯에 대해서도 같은 방식의 작업을 해주었습니다.

```
if (ev->type() == QEvent::MouseButtonPress) {
    auto* me = static_cast<QMouseEvent*>(ev);
```

eventFilter의 구현부분입니다. 받은 이벤트 타입이 마우스 클릭인 경우에 이를 QMouseEvent로 판단하여 새로운 포인터를 할당하는 부분입니다.

```
if (obj == ui->widget) { //A*의 위젯을 누른 경우
    QPoint pt = me->pos(); // 이벤트가 발생한 객체
    for (int i=0; i<(int)A_rect.size(); ++i) {
```

그리고 클릭당한 객체가 A*위젯인 경우에 그 객체의 pos라는 메서드를 사용하여 그 객체의 내부 좌표계에 대한 마우스 클릭된 점을 받아올 수 있었습니다.

```
    if (A_rect[i].contains(pt)) { //QRect의 멤
        A_selected[i] = !A_selected[i]; // 토
        Dijk_selected[i] = !Dijk_selected[i];
        if(start_index<0) start_index=i; //시
        else if(end_index<0) end_index=i;
```

그리고 이 점을 포함한 사각형을 contains메서드를 통해 찾은 후에 해당 인덱스의 사각형이 눌린 것이므로 A_selected를 반전하였습니다. 그렇게 한 이유는 같은 사각형을 두 번 누르는 것에 대해 예외처리를 하기 위함입니다. 그리고 이번 과제의 예시 영상에서도 그랬듯 한 쪽을 누르면 그 반대쪽 위젯에서도 그 사각형에 대응되는 사각형에도 색이 바뀌었기에 이를 구현하기 위해 해당 사각형의 Dijk_selected 또한 반전해주었습니다.

그리고 시작 사각형과 끝 사각형의 인덱스를 저장하기 위해 값을 저장하는 코드 또한 넣어줬습니다.

```
update(); //마우스 클릭된 거 색칠
if(start_index>=0 &&end_index>=0&& start_index!= end_index){
    cal(); //만약 시작점, 끝점이 다 정해지면 이제 cal메서드로 알고리즘 연산
}
```

그리고 이렇게 사각형이 눌린 것에 대해 색을 바꿔야 하기에 update를 호출했고 만약 2개의 사각형이 눌렸다면 각각의 알고리즘을 수행하고 확정된 노드들의 사각형의 색을 바꿀 cal을 호출하였습니다. 이는 데이크스트라 위젯에 대해 같은 방식으로 적용시켰습니다.