최종 프로젝트 보고서

전공 : 컴퓨터공학과 학년 : 2학년 학번 : 20211544 이름 : 송종서

**1. 프로젝트 목표, 실험 환경**

이 프로젝트는 사용자가 화면의 원하는 곳에 점을 그리고, 점과 점 사이를 잇는 직선을 자율적으로 그리게 하여, 이후 그린 점중 원하는 두 점을 선택하면, 점과 직선의 정보를 이용하여 선택한 두 점 사이의 최단 경로를 찾고 이를 출력하는 것을 목표로 한다.

또한 이 실험은 Windows 10에서 Visual Studio 2017을 사용하였고, C++을 기반으로 하는 openframeworks를 이용하여 진행되었다.

**2. 각 변수에 대한 설명**

해당 파트에서는 전역변수에 대해서만 다룬다. (함수 내의 변수는 알고리즘을 설명하며 다룬다)

또한 Dot 구조체에 대한 설명은 4번에서 다룬다.

**0) 매크로**

#define MAX\_DOTS 15 : 화면에 그려지는 점의 최대 개수이다.

#define DOT\_RADIUS 10 : 점의 반지름이다.

#define LINE\_WIDTH 5 : 화면에 그려지는 직선의 두께이다.

#define PATH\_WIDTH 8 : 화면에 그려지는 경로 직선의 두께이다.

#define INF 9999999 : 존재하지 않는 직선/경로 직선의 길이를 나타낸다.

**1) 화면 설정 관련 변수**

HWND hWnd : 프로그램의 창을 다루는 변수이다.

ofxWinMenu\* menu : 메뉴 객체를 저장하는 변수이다.

**2) 기능 선택 여부 및 draw 관련 변수**

int isDot, isLine, isPath : 현재 선택된 기능이 어떤 것인지 저장하는 변수이다. 예를 들어 “점 그리기” 기능이 선택되었다면 isDot에 true를 저장하고, “최단 경로 탐색” 기능이 선택되었다면 isPath에 true를 저장한다. 또한 셋 중 하나만 true가 될 수 있다(한 번에 한 기능만 선택될 수 있다).

int clickFlag : 마우스 클릭이 입력되었는지를 저장하는 변수이다. 클릭이 입력되면 true가 저장되고, 그렇지 않다면 false가 저장된다.

int isOnceHandled : 클릭이 입력된 후 점 추가/선택 처리가 이루어졌는지를 저장하는 변수이다. 점을 추가하거나 선택할 때 클릭 한 번당 추가/선택 과정이 한 번씩만 이루어지게 하기 위한 것이다.

**3) 점 정보 관련 변수**

Dot dotArr[MAX\_DOTS] : 사용자가 추가하는 점의 정보를 저장하는 배열이다.

Dot selected[2] : 직선 추가/경로 탐색 과정에서 선택되는 점의 정보를 저장하는 배열이다.

int dot\_amount : 추가된 점의 총 개수를 저장하는 변수이다.

int selected\_amount : 선택된 점의 개수를 저장하는 변수이다.

**4) 직선 정보 관련 변수**

int lineArr[MAX\_DOTS][MAX\_DOTS] : 직선의 정보를 저장하는 배열이다. 예를 들어 시작점이 점 0이고 끝점이 점 3이며 길이가 5인 직선은 lineArr[0][3] = 5와 같이 저장된다.

**5) 경로 정보 관련 변수**

int pathLen[MAX\_DOTS] : 경로 탐색의 시작점부터 각 점까지의 경로 거리를 저장하는 배열이다.

int parent[MAX\_DOTS] : 탐색한 경로를 다루기 위해, 경로에 포함된 각 점들의 이전 점을 저장하는 배열이다.

int found[MAX\_DOTS] : 경로를 탐색할 때 각 점이 이미 탐색 경로에 포함되었는지, 즉 이미 방문한 점인지를 저장하는 배열이다.

int endidx : 탐색한 경로의 끝 점의 인덱스를 저장하는 변수이다.

**3. 각 함수에 대한 설명**

void setup() : 프로그램이 실행되면 호출된다. 창의 이름, 배경색, 메뉴 등을 설정하고 전역변수들을 초기화하는 함수이다.

void appMenuFunction(string title, bool bChecked) : 메뉴의 팝업에서 각 기능이 선택되면 호출된다. 선택된 기능에 따라 isDot, isLine, isPath 변수의 값을 설정하는 함수이다.

void update() : 프로그램이 실행된 후 매 프레임마다 호출된다. “직선 그리기” 기능이 선택된 경우 addDot() 함수를 호출하고, “직선 그리기” 또는 “최단 경로 찾기” 기능이 선택된 경우에는 selectDot() 함수를 호출하는 함수이다.

void draw() : 프로그램이 실행된 후 매 프레임마다 호출된다. 화면에 안내 문구와 탐색한 경로의 총 길이를 그리고, drawLine(), drawDot(), drawPath(), drawLineLength() 함수를 호출하는 함수이다. 단, drawPath() 함수는 “최단 경로 찾기” 기능이 선택된 경우에만 호출한다.

void mousePressed(int x, int y, int button) : 마우스 클릭이 입력되었을 시 호출된다. clickFlag 변수에 클릭 여부를 저장하고, isOnceHandled 변수를 초기화하는 함수이다.

void mouseReleased(int x, int y, int button) : 마우스 클릭이 해제되었을 시 호출된다. clickFlag 변수에 클릭이 해제되었음을 저장하는 함수이다.

void addDot() : “점 그리기” 기능이 선택되었을 때 호출된다. 사용자가 화면을 클릭하면 클릭 시의 마우스 위치를 저장하는 점을 dotArr 배열에 추가하는 함수이다.

void selectDot() : “직선 그리기” 또는 “최단 경로 찾기” 기능이 선택되었을 때 호출된다. 사용자가 화면에 존재하는 점을 클릭하면 클릭된 점을 selected 배열에 추가하며, 2개의 점이 선택되었다면 선택된 기능에 따라 addLine() 함수 또는 calculatePath() 함수를 호출하는 함수이다.

void addLine() : “직선 그리기” 기능이 선택된 상황에서, selectDot() 함수를 통해 2개의 점이 선택되었다면 호출된다. selected 배열에 저장된 두 점 사이에 그려질 직선의 거리를 계산하여 정보를 lineArr 배열에 저장하는 함수이다.

void calculatePath() : “최단 경로 찾기” 기능이 선택된 상황에서, selectDot() 함수를 통해 2개의 점이 선택되었다면 호출된다. selected 배열에 저장된 두 점 사이의 최적의 경로를 탐색하는 함수이다. 그 결과는 parent 배열의 원소를 따라가며 확인할 수 있다.

void drawDot() : draw() 함수에서 호출된다. dotArr에 저장된 모든 점과 그 점의 인덱스 숫자를 화면에 그리고, lineArr에 저장된 모든 직선의 방향을 나타내는 점(직선 끝부분에 온다)을 화면에 그리는 함수이다.

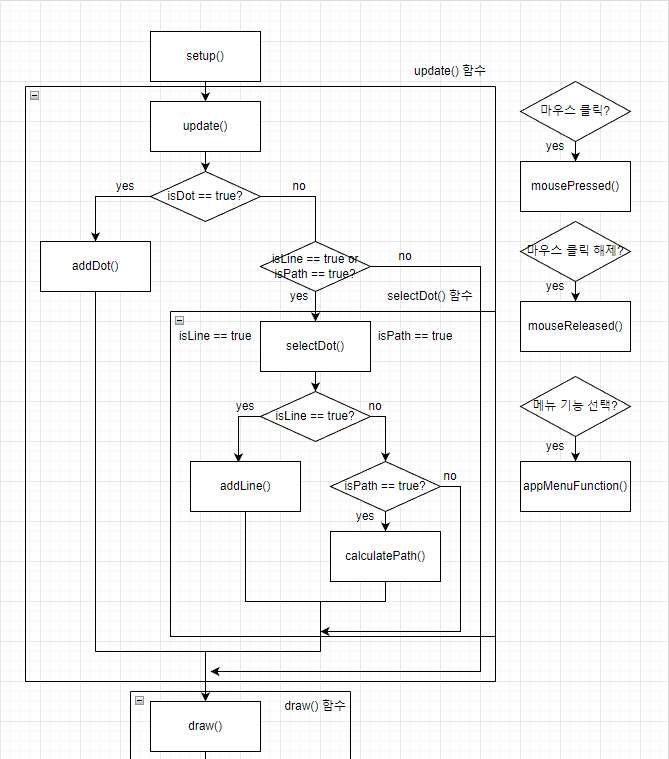
void drawLine() : draw() 함수에서 호출된다. lineArr 배열에 저장된 모든 직선을 화면에 그리는 함수이다.

void drawLineLength() : draw() 함수에서 호출된다. lineArr 배열에 저장된 모든 직선의 길이를 직선 위에 그리는 함수이다. 화면에 점/직선/직선의 길이/경로를 그리는 순서를 조절하기 위해서 drawLine()과 분리되었다.

void drawPath() : draw() 함수에서 호출된다. parent 배열을 따라가며 탐색한 경로를 구성하는 직선을 화면에 그리는 함수이다.

**4. 플로우차트, 자료구조 및 알고리즘과 시간/공간 복잡도**

**전체 플로우차트**(각 함수에 대한 플로우차트는 알고리즘 설명에서)



텍스트, 실내, 바둑판식이(가) 표시된 사진

자동 생성된 설명

**사용한 자료구조**

**구조체 :**

struct Dot

{

int x, y;

int idx;

};

점의 정보를 저장하기 위해 구조체 Dot을 선언하여, 점의 x, y좌표와 점의 인덱스를 저장한다.

**배열 :**

Dot dotArr[MAX\_DOTS];

Dot selected[2];

사용자에 의해 추가되는 각 점의 정보와 선택되는 점의 정보를 저장하기 위해 배열 dotArr과 selected를 사용한다.

int lineArr[MAX\_DOTS][MAX\_DOTS];

사용자에 의해 추가되는 각 직선의 정보와 길이를 저장하기 위해 배열 lineArr을 사용한다.

int pathLen[MAX\_DOTS];

경로 탐색에서 시작점에서 각 점까지의 경로 거리를 저장하기 위해 배열 pathLen을 사용한다.

int parent[MAX\_DOTS];

탐색한 경로에서, 경로에 포함된 각 점의 이전 점을 저장하기 위해 배열 parent를 사용한다.

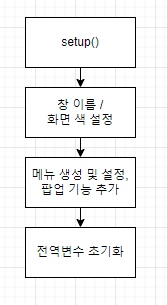
int found[MAX\_DOTS];

경로 탐색에서 각 점의 방문 여부를 저장하기 위해 배열 found를 사용한다.

**프로그램 전체 알고리즘**

우선 간략하게 전체적인 흐름을 정리해보면, addDot()으로 점을 추가하고, selectDot()으로 점을 선택하며, addLine()으로 선택한 점 사이에 직선을 추가하거나 calculatePath()로 선택한 점 사이의 최단 경로를 찾는다. 이후 drawDot(), drawLine(), drawPath(), drawLineLength()를 통해 점, 직선, 탐색 경로, 직선 길이를 그린다.

이를 참고하여 프로그램의 작동 순서를 처음부터 따라가보자. (주요 알고리즘 구현 부분 : 강조)

프로그램이 실행되면 먼저 setup() 함수가 작동한다.

**1. void setup()**

void ofApp::setup() {

ofSetWindowTitle("Shortest Path Finder");

ofBackground(ofColor::white);

setup() 함수는 우선 ofSetWindowTitle() 함수를 이용하여 창의 이름을 설정하고, ofBackground() 함수를 이용해 배경의 색을 흰색으로 설정한다.

hWnd = WindowFromDC(wglGetCurrentDC());

menu = new ofxWinMenu(this, hWnd);

menu->CreateMenuFunction(&ofApp::appMenuFunction);

HMENU hMenu = menu->CreateWindowMenu();

HMENU hPopup = menu->AddPopupMenu(hMenu, "기능");

menu->AddPopupItem(hPopup, "점 그리기", false, false);

menu->AddPopupItem(hPopup, "직선 그리기", false, false);

menu->AddPopupItem(hPopup, "최단 경로 찾기", false, false);

menu->AddPopupItem(hPopup, "종료", false, false);

menu->SetWindowMenu();

이후 WindowFromDC() 함수를 이용하여 hWnd에 현재 다루는 창을 불러오고 menu에 새로운 메뉴를 할당한 후, CreateMenuFunction() 함수를 이용하여 메뉴에 대해 작동할 함수를 appMenuFunction로 설정한다. 또한 CreateWindowMenu() 함수로 실제 창에 메뉴를 추가하며, addPopupMenu() 함수로 메뉴에 “기능” 팝업을 새롭게 추가한다. 이후 AddPopupItem() 함수를 이용하여 “기능” 팝업에 “점 그리기”, “직선 그리기”, “최단 경로 찾기”, “종료” 기능을 추가해준다. 이와 같은 과정을 통하여 메뉴에 대한 설정을 한다.

isDot = false;

isLine = false;

isPath = false;

clickFlag = false;

isOnceHandled = false;

dot\_amount = 0;

selected\_amount = 0;

endidx = -1;

Dot newDot;

newDot.idx = -1;

selected[0] = newDot; selected[1] = newDot;

for (int i = 0; i < MAX\_DOTS; i++)

for (int j = 0; j < MAX\_DOTS; j++)

{

if (i == j) lineArr[i][j] = 0;

else lineArr[i][j] = INF;

}

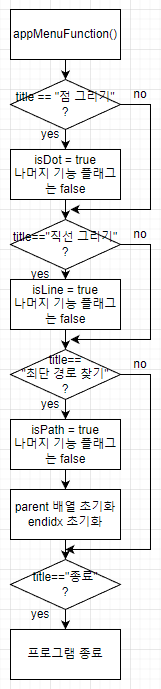
}

이후 각 전역변수들을 초기화해주는 과정을 거친다. 처음에는 선택된 기능이 없으니 isDot, isLine, isPath를 false로 초기화한다. 그리고 클릭과 클릭에 대한 처리가 이루어지면 안 되므로 clickFlag와 isOnceHandled를 false로 초기화하고, 실행 직후이니 전체 점의 개수와 선택된 점의 개수를 0으로 초기화한다. 또한 경로 탐색이 이루어지지 않은 상황이니 경로의 끝점의 인덱스를 저장하는 endidx를 -1로 초기화하며, 아직 선택된 점이 없으므로 Dot 타입의 변수 newDot을 선언하여 배열 selected을 초기화한다. 마지막으로 아직 그려진 직선이 없으므로 배열 lineArr을 모두 INF로 초기화한다. 단 직선의 시작점과 끝점이 같은 경우(i = j)면 0으로 초기화한다.

시간 복잡도 : MAX\_DOTS에 대한 이중 for문을 사용하므로 O(MAX\_DOTS \* MAX\_ DOTS)이다.

공간 복잡도 : 배열 selected와 lineArr을 이용하므로 O(sizeof(Dot) \* 2 + sizeof(int) \* MAX\_DOTS \* MAX\_DOTS) = O(MAX\_DOTS \* MAX\_DOTS)이다.

이렇게 setup() 함수를 통하여 기본적인 설정이 끝나면, update() 함수와 draw() 함수가 매 프레임마다 작동하여 프로그램의 주요 역할을 한다. 그러나 메뉴에서 기능이 선택되기 전까지(isDot 플래그 등이 true가 되기 전까지)는 두 함수가 특별한 처리를 하지 않는다. 그러므로 우선 메뉴에서 기능이 선택되면 호출되는 함수 appMenuFunction()를 먼저 보자.

**2. void appMenuFunction(string title, bool bChecked)**

void ofApp::appMenuFunction(string title, bool bChecked)

{

if (title == "점 그리기")

{

isDot = true;

isLine = false;

isPath = false;

}

appMenuFunction() 에서는 선택된 기능에 따라 각자의 처리를 한다. “점 그리기” 기능이 선택되었다면 isDot을 true로 하여 기능이 선택되었음을 저장하고, 나머지 기능들의 플래그는 false로 설정하여 update()와 draw()에서 작동하지 않게 한다.

if (title == "직선 그리기")

{

isDot = false;

isLine = true;

isPath = false;

}

앞과 동일하다. isLine을 true로 한다.

if (title == "최단 경로 찾기")

{

isDot = false;

isLine = false;

isPath = true;

for (int i = 0; i < dot\_amount; i++)

parent[i] = i;

endidx = -1;

}

이 또한 앞과 동일한데, 이 기능이 선택되었다면 parent 배열의 각 원소를 자기 자신으로 초기화하고 endidx 또한 초기화하여, 이전에 탐색한 경로의 정보를 없앤다.

if(title == "종료") ofExit();

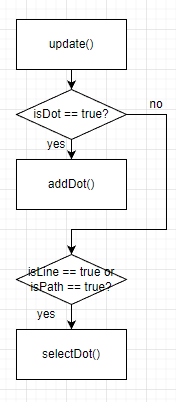
}

마지막으로 “종료” 기능이 선택되었다면 프로그램을 종료한다.

시간 복잡도 : “점 그리기”, “직선 그리기”, “종료” 기능이 선택되었을때는 O(1)이고, “최단 경로 찾기” 기능이 선택되었을때는 dot\_amount에 대한 for문을 사용하므로 점의 개수가 최대치에 도달한 상황에서는 O(MAX\_DOTS)이다.

공간 복잡도 : “점 그리기”, “직선 그리기”, “종료” 기능이 선택되었을때는 배열을 이용하지 않으니 O(1)이고 “최단 경로 찾기” 기능이 선택되었을때는 배열 parent을 이용하니 O(sizeof(int) \* MAX\_DOTS) = O(MAX\_DOTS)이다.

이 함수를 통해 isDot, isLine, isPath 플래그중 하나가 true로 설정되고, update()와 draw() 함수에서는 이 플래그에 따라 다른 처리를 한다. update() 함수를 보자.

**3. void update()**

void ofApp::update(){

if (isDot)

addDot();

update() 함수는, “점 그리기” 기능이 선택되었을 때, 즉 isDot가 true일 때에는 addDot() 함수를 호출한다.

else if (isLine || isPath)

selectDot();

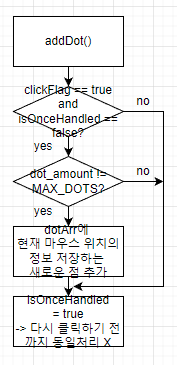
}

그렇지 않을 때, “직선 그리기” 또는 “최단 경로 찾기” 기능이 선택되었다면, 즉 isLine 또한 isPath가 true라면 selectDot() 함수를 호출한다.

시간 복잡도 : 뒤에서 계산하겠지만, addDot() 함수의 시간 복잡도는 O(1)이고 selectDot() 함수의 시간 복잡도는 O(MAX\_DOTS \* MAX\_DOTS)이므로 O(1) 또는 O(MAX\_DOTS \* MAX\_DOTS)이다.

공간 복잡도 : addDot() 함수의 공간 복잡도는 O(MAX\_DOTS)이고 selectDot() 함수의 공간 복잡도는 O(MAX\_DOTS \* MAX\_DOTS)이므로 둘 중 하나이다.

이렇게 호출된 addDot(), selectDot() 함수를 통하여, 사용자는 점을 추가하거나 선택할 수 있다. addDot() 함수를 보자.

**3-1. void addDot()**

void ofApp::addDot()

{

int x, y;

Dot newDot;

우선 마우스의 위치를 저장하기 위해 변수 x, y를 선언하고, 배열 dotArr에 새로운 점을 추가하기 위해 변수 newDot을 선언한다.

if (clickFlag && !isOnceHandled)

{

점을 추가하는 처리는, 마우스 클릭이 이루어져 clickFlag가 true가 되었으며 아직 처리가 한 번도 이루어지지 않아 isOnceHandled가 false인 상황에서 이루어진다. (함수 설명에서 말했듯, clickFlag와 isOnceHandled의 설정은 mousePressed()와 mouseReleased() 함수에서 이루어지는데, 이 두 함수는 단순히 두 변수를 설정하는 부분밖에 존재하지 않으므로 설명에서 생략하겠다)

if (dot\_amount != MAX\_DOTS)

{

또한 점의 개수가 최대치에 도달하지 않았어야 한다.

x = ofGetMouseX(); y = ofGetMouseY();

newDot.x = x; newDot.y = y;

newDot.idx = dot\_amount;

dotArr[dot\_amount++] = newDot;

}

else printf("점 개수가 최대입니다.\n");

isOnceHandled = true;

}

}

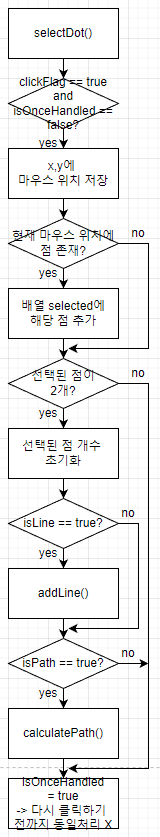
앞의 세 조건이 성립된다면, x, y에 현재 마우스의 위치를 저장하고, newDot에 이 위치를 다시 저장하며 인덱스 또한 지정해준다. 이후 배열 dotArr에 newDot을 추가한다. 마지막으로 이러한 처리가 끝났다면(점의 개수가 최대여서 추가하지 못한 상황 포함) isOnceHandled를 true로 설정하여 한 번의 클릭에 대해 여러 번 처리하지 않게 한다(이렇게 하지 않으면 클릭 버튼을 꾹 누르고 있으면 연속적인 클릭으로 판정하여 점을 그리는 처리가 반복된다).

시간 복잡도 : 반복이 일어나지 않으므로 O(1)이다.

공간 복잡도 : 배열 dotArr을 이용하니 O(sizeof(Dot) \* MAX\_DOTS) = O(MAX\_DOTS)이다.

이를 통하여 “점 그리기” 기능이 선택된 상황에서 새로운 점을 추가할 수 있다.

이렇게 추가된 점은, “직선 그리기” 또는 “최단 경로 찾기” 기능이 선택된 상황에서 selectDot() 함수를 통해 선택된다. selectDot() 함수를 보자.

**3-2. void selectDot()**

void ofApp::selectDot()

{

int x, y;

int dis;

Dot curDot;

우선 마우스의 위치를 저장하기 위해 변수 x, y를 선언하고, 반복문에서 처리하는 점과 마우스의 위치 사이의 거리를 저장하기 위해 변수 dis를 선언하며, 배열 dotArr에 저장된 점의 정보를 불러오기 위해 변수 curDot을 선언한다.

if (clickFlag && !isOnceHandled)

{

점을 선택하는 처리 또한, 마우스 클릭이 이루어져 clickFlag가 true가 되었으며 아직 처리가 한 번도 이루어지지 않아 isOnceHandled가 false인 상황에서 이루어진다.

x = ofGetMouseX(), y = ofGetMouseY();

for (int i = 0; i < dot\_amount; i++)

{

curDot = dotArr[i];

dis = (x - curDot.x) \* (x - curDot.x) + (y - curDot.y) \* (y - curDot.y);

if (dis < DOT\_RADIUS \* DOT\_RADIUS)

{

selected[selected\_amount++] = curDot;

break;

}

}

앞의 조건이 만족된다면 마우스의 위치를 x, y에 저장하고, 반복문을 이용하여 curDot에 dotArr의 각 점의 정보를 불러와, 현재 마우스의 위치와 curDot 사이의 거리를 계산한다. 계산한 거리가 점의 반지름(DOT\_RADIUS)보다 작다면 배열 selected에 curDot을 추가해주고, break를 통해 반복문을 종료한다(여러 개의 점이 겹치는 상황에 한 개의 점만 선택되게 하기 위함).

if (selected\_amount == 2)

{

selected\_amount = 0;

if (isLine) addLine();

if (isPath) calculatePath();

}

isOnceHandled = true;

}

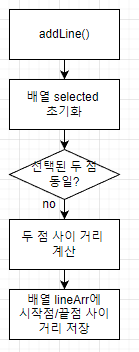
}

이후 선택된 점의 개수가 2개라면, 개수를 0으로 초기화해주고 현재 선택된 기능이 “직선 그리기”면 addLine() 함수를 호출하고, “최단 경로 찾기”면 calculatePath() 함수를 호출한다.

시간 복잡도 : dot\_amount에 대한 for문을 이용하는 것에 대해서는, 점의 개수가 최대치에 도달한 상황에서는 O(MAX\_DOTS)이고 addLine() 함수의 시간 복잡도는 O(1)이며, calculatePath() 함수의 시간 복잡도는 O(MAX\_DOTS \* MAX\_DOTS)인데, addLine() 함수나 calculatePath() 함수 중 하나만 작동하므로 시간 복잡도는 O(MAX\_DOTS + 1) = O(MAX\_DOTS)이나 O(MAX\_DOTS + MAX\_DOTS \* MAX\_DOTS) = O(MAX\_DOTS \* MAX\_DOTS)이다.

공간 복잡도 : addLine() 함수가 작동할때는 배열 selected와 lineArr을 사용하니 O(sizeof(Dot) \* 2 + sizeof(int) \* MAX\_DOTS \* MAX\_DOTS) = O(MAX\_DOTS \* MAX\_DOTS)이고, calculatePath() 함수가 작동할때는 4개의 모든 배열을 사용하니 calculatePath() 함수의 공간복잡도와 동일하게 O(MAX\_DOTS \* MAX\_DOTS)이다. 결과적으로 O(MAX\_DOTS \* MAX\_DOTS)이다.

이렇게 점을 선택할 수 있는데, addLine() 함수와 calculatePath() 함수를 통하여 선택된 점 사이에 직선이 추가되거나 그 사이의 최단 경로가 계산된다. 두 함수를 차례대로 보자.

**3-2-1. void addLine()**

void ofApp::addLine()

{

Dot d1 = selected[0], d2 = selected[1];

Dot newDot;

newDot.idx = -1;

selected[0] = newDot; selected[1] = newDot;

우선 배열 selected에서 선택된 두 점의 정보를 새로운 변수 d1, d2에 저장해주고, 변수 newDot을 선언하여 배열 selected를 초기화해준다.

if (d1.idx == d2.idx) return;

만약 선택한 두 점이 같은 점이라면, 즉 한 점을 두 번 선택했다면 직선을 추가하지 않는다.

int dis = sqrt((d1.x - d2.x) \* (d1.x - d2.x) + (d1.y - d2.y) \* (d1.y - d2.y));

lineArr[d1.idx][d2.idx] = dis;

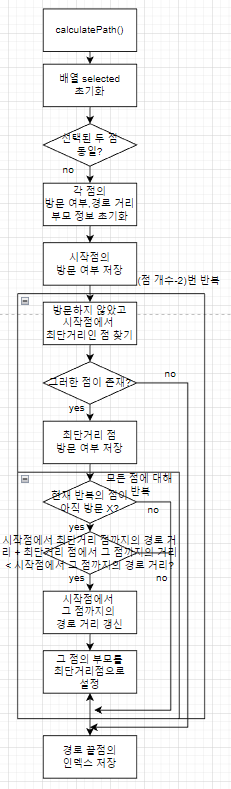
}

그렇지 않다면 두 점 사이의 거리를 계산하고, 배열 lineArr에 시작점과 끝 점 사이의 거리를 저장한다.

시간 복잡도 : 반복이 일어나지 않으니 O(1)이다.

공간 복잡도 : 배열 selected와 lineArr을 이용하니 O(sizeof(Dot) \* 2 + sizeof(int) \* MAX\_DOTS \* MAX\_DOTS) = O(MAX\_DOTS \* MAX\_DOTS)이다.

이와 같은 방식으로 직선을 추가할 수 있다. 다음은 calculatePath() 함수이다.

**3-2-2. void calculatePath()**

calculatePath() 함수는 Dijkstra's algorithm을 사용하여 구현하였다.

void ofApp::calculatePath()

{

int d1 = selected[0].idx, d2 = selected[1].idx;

int curd;

int minLen;

Dot newDot;

newDot.idx = -1;

selected[0] = newDot; selected[1] = newDot;

기본적인 세팅을 먼저 다뤄보면, 배열 selected에서 선택된 두 점의 인덱스를 변수 d1, d2에 불러온다. 이후 올 반복문에서, 아직 방문하지 않았고 시작점과 가장 가까이 있는 점(최단거리 점)의 인덱스를 저장하기 위해 변수 curd를 선언하며, 가장 가까이 있는 점을 찾는 과정에서 최단거리를 저장하기 위해 변수 minLen을 선언한다. 또한 변수 newDot을 선언하여 배열 selected를 초기화해준다.

if (d1 == d2) return;

만약 선택한 두 점이 같은 점이라면 경로를 계산하지 않는다.

for (int i = 0; i < dot\_amount; i++)

{

found[i] = false;

pathLen[i] = lineArr[d1][i];

parent[i] = i;

if (pathLen[i] < INF) parent[i] = d1;

}

found[d1] = true;

이후 반복문을 이용하여 각 점의 방문 여부를 초기화하고, 배열 pathLen(경로 거리, 시작점부터 여러 직선을 타고가며 각 점까지 도달할때까지의 거리)에 시작점부터 각 점까지의 거리(두 점 사이의 직선 길이)를 저장한다(시작점과의 점 사이에 직선이 없으면 INF가 된다). 그리고 각 부모 정보를 초기화하는데, 시작점과 각 점 사이에 직선이 있으면 시작점을 점의 부모로 하고, 그렇지 않다면 자기 자신으로 설정한다(이를 통해 이전의 경로 탐색 결과를 지운다). 이후 시작점의 방문 여부를 true로 저장한다(시작점부터 경로를 탐색하는 것이므로).

‘경로 거리’와 ‘거리’ 두 가지 표현 혼동 주의

for (int i = 0; i < dot\_amount - 2; i++)

{

이 부분부터가 주요 부분이다. 이 for문을 이용하여 (점 개수 – 2)번만큼 처리를 반복해 시작점에서 모든 점까지의 경로 거리를 계산한다.

minLen = INF;

curd = -1;

for (int j = 0; j < dot\_amount; j++)

if (pathLen[j] < minLen && !found[j])

{

minLen = pathLen[j];

curd = j;

}

if (curd == -1) break;

우선 for문을 이용하여 방문하지 않은 점중 시작점부터 해당 점까지의 경로 거리가 가장 작은 점(최단거리 점)을 찾는다. 최단거리 점을 찾는데 실패한다면, 더 이상 시작점으로부터 갈 수 있는 점이 없는 것이므로 반복문을 종료한다.

found[curd] = true;

최단거리 점을 찾았다면, 해당 점의 방문 여부를 true로 한다.

for (int j = 0; j < dot\_amount; j++)

if (!found[j])

if (pathLen[curd] + lineArr[curd][j] < pathLen[j])

{

pathLen[j] = pathLen[curd] + lineArr[curd][j];

parent[j] = curd;

}

}

endidx = d2;

}

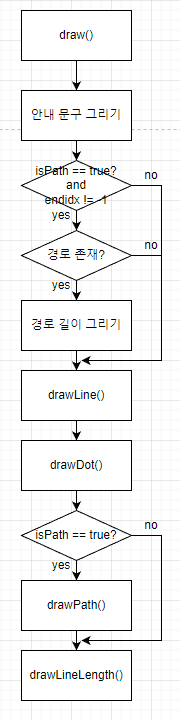
이후 for문을 이용하여, 최단거리 점과 아직 방문하지 않은 점 사이에 관한 처리를 진행한다. 만약 “시작점에서 최단거리 점까지의 경로 거리”와 “최단거리 점에서 방문하지 않은 점까지의 거리”의 합이 “시작점에서 방문하지 않은 점까지의 경로 거리”보다 작다면, 이 경로 거리를 두 개의 합으로 업데이트 해주고, 그 방문하지 않은 점의 부모를 최단거리 점으로 저장한다. 그리고 모든 반복문이 끝나면 끝점의 인덱스를 endidx에 저장한다.

시간 복잡도 : dot\_amount에 대한 for문과 dot\_amount에 대한 이중 for문을 사용하므로, 점의 개수가 최대치에 도달한 상황에서는 O(MAX\_DOTS + MAX\_DOTS \* MAX\_DOTS) = O(MAX\_DOTS \* MAX\_DOTS)이다.

공간 복잡도 : 배열 selected, pathLen, lineArr, parent 모두를 사용하므로 O(sizeof(Dot) \* 2 + sizeof(int) \* MAX\_DOTS + sizeof(int) \* MAX\_DOTS \* MAX\_DOTS + sizeof(int) \* MAX\_DOTS) = O(MAX\_DOTS \* MAX\_DOTS)이다.

이러한 과정을 통하여 시작점부터 모든 점까지의 경로 거리를 계산하고, 이후 끝점의 parent를 따라가며 탐색한 경로를 이용하는 것이다. 아무튼 이를 통하여 최단 경로를 찾을 수 있다.

이제 selectDot() 함수의 addLine() 함수, calculatePath() 함수를 모두 봤으니 **3.** update() 함수는 끝이 났다. 결과적으로 update() 함수는 점의 추가 / 점의 선택 / 직선의 추가 / 최단 경로 탐색의 기능을 처리하는 것이다. 다음으로는, 이렇게 추가된 점과 직선, 계산된 경로를 화면에 그리는 draw() 함수를 보자.

**4. void draw()**

void ofApp::draw(){

string pLen;

draw() 함수는 우선 경로의 길이를 문자열의 형태로 저장하기 위해 변수 pLen을 선언한다.

ofSetColor(ofColor::black);

ofDrawBitmapString("Click -> Draw Dot / Select Dot", 30, 30);

이후 그림의 색을 검정으로 정하고, 화면의 좌측 상단에 안내 문구를 그린다.

if (isPath && endidx != -1)

if (pathLen[endidx] != INF)

{

pLen = "Path Length : " + ofToString(pathLen[endidx]);

ofDrawBitmapString(pLen, 30, 60);

}

다음으로는 “최단 경로 찾기” 기능이 선택되었고 경로 탐색이 이루어진 상황에서, 선택된 두 점 사이에 경로가 존재하면 pLen에 경로의 길이를 문자열 형태로 저장하고 화면의 좌측 상단에 그린다.

drawLine();

drawDot();

if (isPath) drawPath();

drawLineLength();

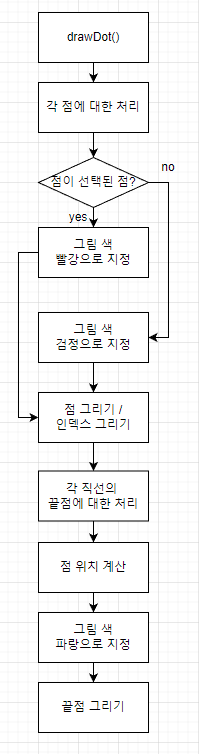
}

이후 drawLine(), drawDot(), drawPath(), drawLineLength() 함수를 순서대로 호출하여, 직선 -> 점 -> 경로 -> 직선 길이 순서로 화면에 그린다(이러한 순서로 그려야 결과가 잘 보인다). 단 점, 직선을 추가하는 상황에서 이전에 탐색한 경로가 그려져 있으면 불편하므로 경로는 “최단 경로 찾기” 기능이 선택된 상황에서만 그린다.

시간 복잡도 : drawLine(), drawDot(), drawLineLength() 함수는 O(MAX\_DOTS \* MAX\_DOTS), drawPath() 함수는 O(MAX\_DOTS)이니 시간 복잡도는 O(MAX\_DOTS \* MAX\_DOTS)이다.

공간 복잡도 : drawLine(), drawDot(), drawLineLength() 함수는 O(MAX\_DOTS \* MAX\_DOTS), drawPath() 함수는 O(MAX\_DOTS)이니 공간 복잡도는 O(MAX\_DOTS \* MAX\_DOTS)이다.

이러한 과정을 통해 화면의 요소들이 그려진다. 호출되는 각 함수들을 보자.

**4-1. void drawDot()**

void ofApp::drawDot()

{

Dot curDot;

string dotNum;

drawDot() 함수는 우선 반복문에서 점의 정보를 저장하기 위해 변수 curDot을 선언하고, 점의 인덱스 숫자를 문자열로 저장하기 위해 변수 dotNum을 선언한다.

for (int i = 0; i < dot\_amount; i++)

{

curDot = dotArr[i];

dotNum = ofToString(i);

if (curDot.idx == selected[0].idx || curDot.idx == selected[1].idx) ofSetColor(ofColor::red);

else ofSetColor(ofColor::black);

ofDrawCircle(curDot.x, curDot.y, DOT\_RADIUS);

이후 for문을 이용하여 각 점을 그리는 처리를 하는데, 만약 점이 선택된 점이라면 그림의 색을 빨강으로 정하고, 그렇지 않다면 검정으로 정한다. 색을 정하면 점의 위치에 반지름이 DOT\_RADIUS인 원을 그린다.

if (i < 10) ofDrawBitmapString(dotNum, curDot.x - DOT\_RADIUS / 2, curDot.y - DOT\_RADIUS \* 1.5);

else ofDrawBitmapString(dotNum, curDot.x - DOT\_RADIUS, curDot.y - DOT\_RADIUS \* 1.5);

}

이후 점의 위쪽 적당한 위치에 점의 인덱스를 그려주는 처리를 한다. 인덱스의 자릿수에 따라 위치를 조정해준다.

Dot d1, d2;

float t;

float c;

float s;

다음으로는 각 직선의 방향을 나타내는 점을 그리기 위해, 반복문에서 점의 정보를 저장할 변수 d1, d2를 선언한다. 또한 점의 위치를 결정하기 위해 직선이 x축과 이루는 각도에 대한 sin, cos, tan값을 계산해야 하므로, 변수 t, c, s를 선언한다.

for (int i = 0; i < dot\_amount; i++)

for (int j = 0; j < dot\_amount; j++)

{

이후 이중 for문을 이용하여 각 직선에 대한 처리를 진행한다.

if (lineArr[i][j] < INF && lineArr[i][j] != 0)

{

d1 = dotArr[i]; d2 = dotArr[j];

인덱스가 i인 점부터 j인 점까지의 직선이 존재한다면, 우선 직선의 양 끝점의 정보를 d1, d2에 저장한다.

t = ((float)d2.y - d1.y) / (d2.x - d1.x);

c = d2.x - d1.x > 0 ? sqrt(1 / (t \* t + 1)) : -sqrt(1 / (t \* t + 1));

s = d2.y - d1.y > 0 ? sqrt(t \* t / (t \* t + 1)) : -sqrt(t \* t / (t \* t + 1));

이후 직선의 양 끝점 사이의 기울기를 계산하여 tan값을 구한다. 이후 두 점의 x, y좌표의 대소를 고려하며 sin^2 + cos^2 = 1, tan = sin / cos의 두 공식을 이용하여 cos, sin값을 구해준다.

ofSetColor(ofColor::blue);

ofDrawCircle(d2.x - DOT\_RADIUS \* c, d2.y - DOT\_RADIUS \* s, DOT\_RADIUS / 2);

}

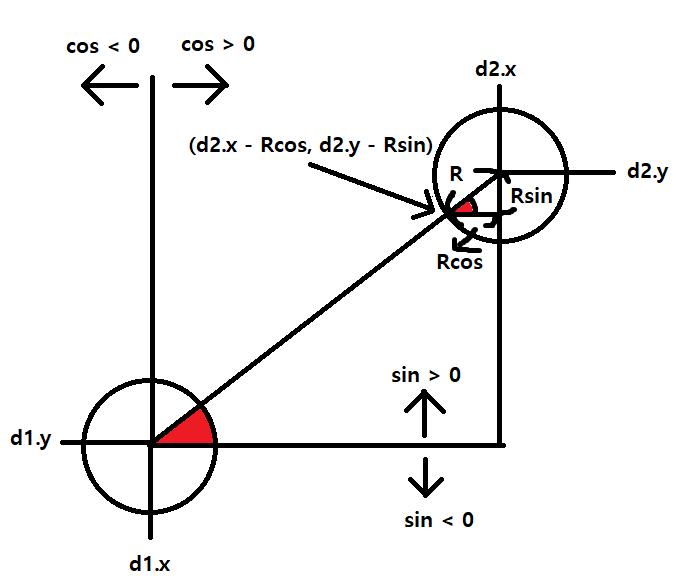
}

}

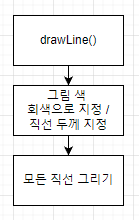
이후 그림의 색을 파랑으로 정하고, 구한 cos, sin값을 이용하여 직선의 방향을 나타낼 수 있는 적당한 위치에 반지름이 DOT\_RADIUS / 2인 원을 그려 점을 나타낸다. (하단 그림 참고)

시간 복잡도 : dot\_amount에 대한 for문과 이중 for문을 사용하므로, 점의 개수가 최대치에 도달한 상황에서 O(MAX\_DOTS + MAX\_DOTS \* MAX\_DOTS) = O(MAX\_DOTS \* MAX\_DOTS)이다.

공간 복잡도 : 배열 dotArr과 lineArr을 사용하므로 O(sizeof(Dot) \* MAX\_DOTS + sizeof(int) \* MAX\_DOTS \* MAX\_DOTS) = O(MAX\_DOTS \* MAX\_DOTS)이다.



이와 같은 과정을 통하여 화면에 점이 그려진다. 다음은 drawLine() 함수를 보자.

**4-2. void drawLine()**

void ofApp::drawLine()

{

Dot d1, d2;

drawLine() 함수는 반복문에서 직선의 양 끝점의 정보를 저장하기 위해 변수 d1, d2를 선언한다.

ofSetLineWidth(LINE\_WIDTH);

ofSetColor(ofColor::gray);

직선의 두께를 LINE\_WIDTH로 정하고, 그림의 색을 회색으로 정한다.

for(int i = 0; i < dot\_amount; i++)

for (int j = 0; j < dot\_amount; j++)

{

if (lineArr[i][j] < INF && lineArr[i][j] != 0)

{

d1 = dotArr[i]; d2 = dotArr[j];

ofDrawLine(d1.x, d1.y, d2.x, d2.y);

}

}

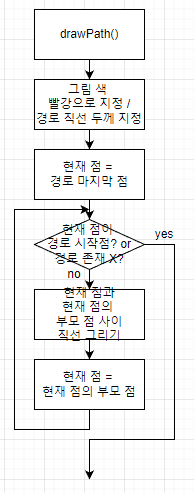
}

이후 이중 for문을 이용하여 인덱스가 i인 점과 j인 점 사이에 직선이 존재한다면 그 두 점을 끝점으로 하는 직선을 그린다.

시간 복잡도 : dot\_amount에 대한 이중 for문을 사용하니, 점의 개수가 최대치에 도달한 상황에서 O(MAX\_DOTS \* MAX\_DOTS)이다.

공간 복잡도 : 배열 dotArr과 lineArr을 사용하니 drawDot() 함수와 동일하게 O(MAX\_DOTS \* MAX\_DOTS)이다.

이를 통해 화면에 직선이 그려진다. 다음은 drawPath() 함수를 보자.

**4-3. void drawPath()**

void ofApp::drawPath()

{

Dot d1, d2;

drawPath() 함수는 반복문에서 경로에 포함된 직선의 양 끝점의 정보를 저장하기 위해 변수 d1, d2를 선언한다.

ofSetColor(ofColor::red);

ofSetLineWidth(PATH\_WIDTH);

직선의 두께를 PATH\_WIDTH로 정하고, 그림의 색을 빨강으로 정한다.

for (int i = endidx; parent[i] != i && i != -1; i = parent[i])

{

d1 = dotArr[i]; d2 = dotArr[parent[i]];

ofDrawLine(d1.x, d1.y, d2.x, d2.y);

}

}

이후 for문을 이용하여 경로의 마지막 점부터 시작점까지 따라가며, 해당 반복에서의 현재 점과 그 점의 부모 점 사이에 직선을 그리는 처리를 반복한다. 시작점에 도달한 것을 판단하는 것은 parent[i] != i를 이용한다. 시작점의 부모는 자기 자신으로 설정되어있기 때문이다. 또한 경로 탐색이 이루어지지 않은 경우(i = -1)에도 반복이 일어나지 않게 한다.

시간 복잡도 : for문을 이용하여 경로의 마지막 점부터 시작점까지 따라가니, 모든 점이 연결되어있는 상황에서 반복 수는 약 MAX\_DOTS이다. 따라서 O(MAX\_DOTS)이다.

공간 복잡도 : 배열 dotArr과 parent를 사용하니 O(sizeof(Dot) \* MAX\_DOTS + sizeof(int) \* MAX\_DOTS) = O(MAX\_DOTS)이다.

이를 통해 화면에 경로가 그려진다. 마지막으로 drawLineLength() 함수를 보자.

**4-4. void drawLineLength()**

void ofApp::drawLineLength()

{

Dot d1, d2;

string lineLen;

drawLineLength() 함수도 반복문에서 직선의 양 끝점의 정보를 저장하기 위해 변수 d1, d2를 선언한다. 또한 반복문에서 직선의 길이를 문자열로 저장하기 위해 변수 lineLen을 선언한다.

ofSetColor(ofColor::blue);

그림의 색을 파랑으로 정한다.

for (int i = 0; i < dot\_amount; i++)

for (int j = 0; j < dot\_amount; j++)

{

if (lineArr[i][j] < INF && lineArr[i][j] != 0)

{

d1 = dotArr[i]; d2 = dotArr[j];

lineLen = ofToString(lineArr[i][j]);

ofDrawBitmapString(lineLen,(d1.x + d2.x)/2,(d1.y+d2.y)/2);

}

}

}

이후 이중 for문을 이용하여 인덱스가 i인 점과 j인 점 사이에 직선이 존재한다면, 두 점의 중점 위치에 직선의 길이를 문자열로 변경하여 그려준다. (이 함수가 drawLine과 분리된 이유는 앞에서도 말했듯 화면에 점/직선/직선의 길이/경로를 그리는 순서를 조절하기 위해서이다)

시간 복잡도 : drawLine() 함수와 동일하게 O(MAX\_DOTS \* MAX\_DOTS)이다.

공간 복잡도 : drawLine() 함수와 동일하게 O(MAX\_DOTS \* MAX\_DOTS)이다.

이를 통해 화면에 직선의 길이가 그려진다.

이 모든 과정을 통하여 점을 그리고 선택하며, 선택한 점 사이에 직선을 그리거나 최단 경로를 찾을 수 있다. 프로그램의 전체 알고리즘의 설명을 마친다.

+ 사실 점과 직선을 추가하는 과정에서 계속 draw() 함수가 호출되나, 프로그램의 순서를 순차적으로 설명하기 위해 위와 같은 순서로 설명하였다.

**시간 / 공간 복잡도 총정리**

addLine(), calculatePath() 함수는 selectDot() 함수에 포함되며, addDot(), selectDot() 함수는 update() 함수에 포함되고, drawLine(), drawDot(), drawPath(), drawLineLength() 함수는 draw() 함수에 포함된다.

setup() / 시간 복잡도 : O(MAX\_DOTS \* MAX\_DOTS), 공간 복잡도 : O(MAX\_DOTS \* MAX\_DOTS)

appMenuFunction() / 시간 복잡도 : O(MAX\_DOTS), 공간 복잡도 : O(MAX\_DOTS)

update() / 시간 복잡도 : O(1) or O(MAX\_DOTS \* MAX\_DOTS), 공간 복잡도 : O(MAX\_DOTS) or O(MAX\_DOTS \* MAX\_DOTS)

draw() / 시간 복잡도 : O(MAX\_DOTS \* MAX\_DOTS), 공간 복잡도 : O(MAX\_DOTS \* MAX\_DOTS)

-> 프로그램 전체의 시간, 공간 복잡도를 정리해보면

시간 복잡도 : O(MAX\_DOTS \* MAX\_DOTS)

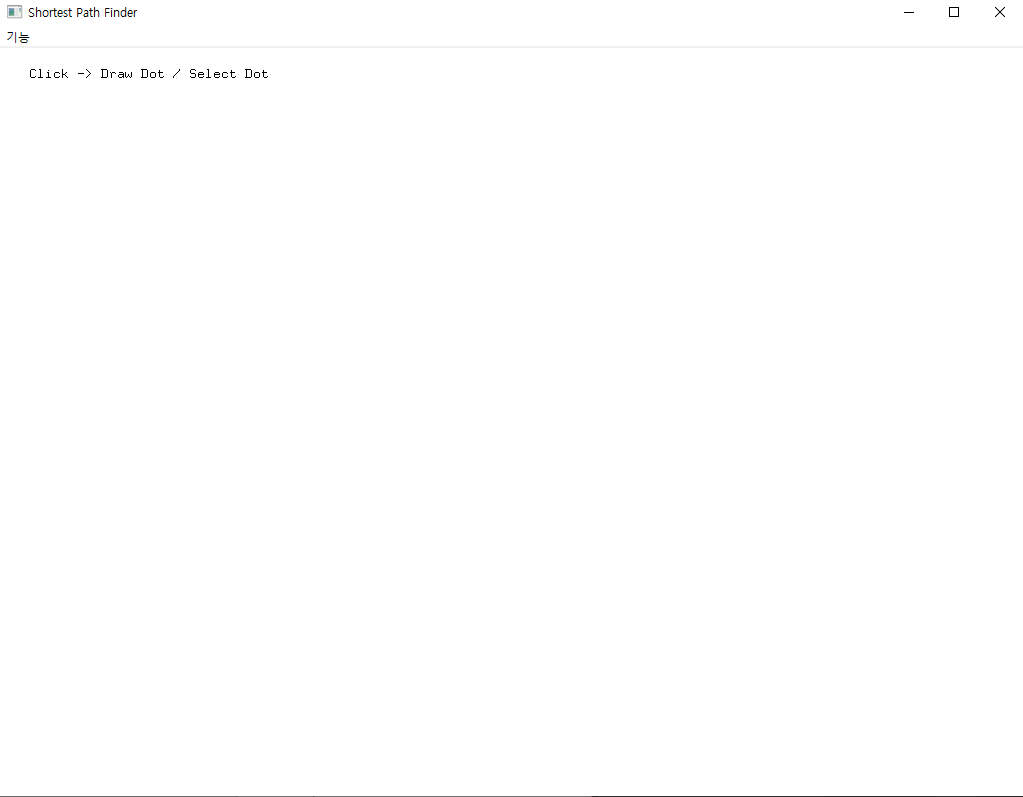
공간 복잡도 : O(MAX\_DOTS \* MAX\_DOTS)이다.

**5. 어떻게 창의적 구현을 하였는지**

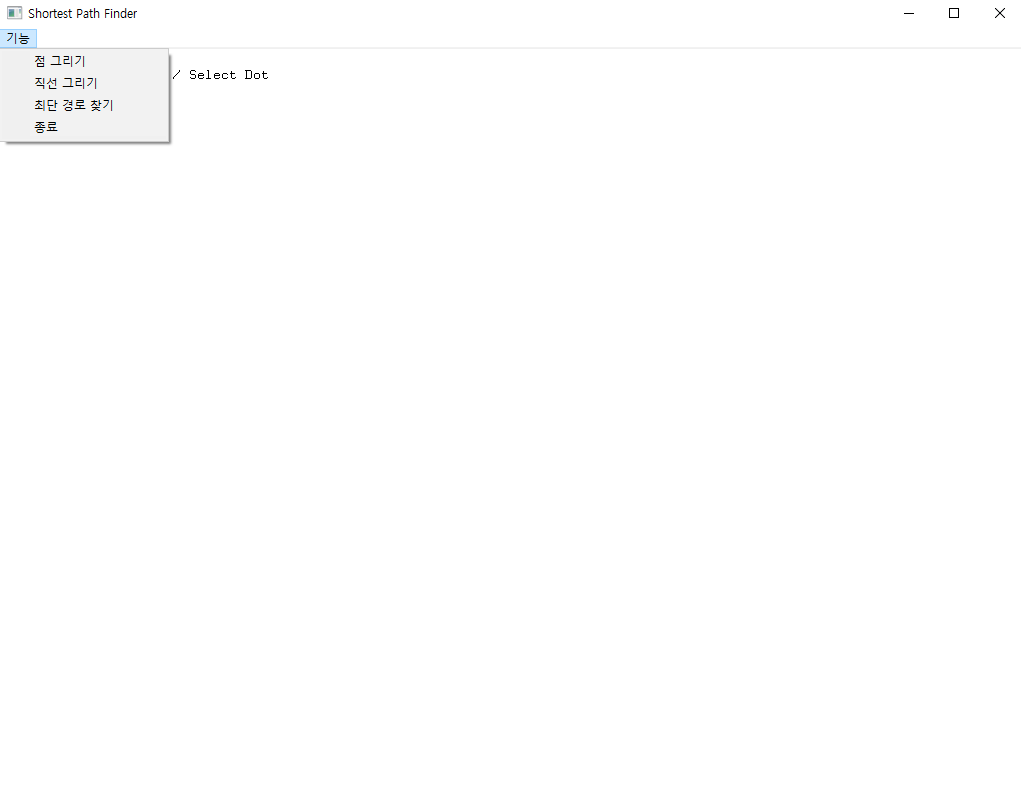
원래는 11~13주차의 미로 프로젝트에서, 방마다 가중치를 두고, 선택한 두 방 사이의 경로중 가중치가 가장 적은 경로를 찾는 알고리즘을 구현할까 생각했었다. 그러나 이는 미로 프로젝트에서 구현했던 미로의 형태에 적합하지 않고, 13주차에 구현한 DFS, BFS를 적당히 수정하여 이용하면 구현할 수 있었기에 단조롭다고 느껴져 생각을 바꿨다. 이후 미로 프로젝트와 달리 사용자가 직접 정점과 경로를 그릴 수 있게 하고, 그렇게 그려진 점과 직선의 정보를 기반으로 Dijkstra's algorithm를 이용하여 미로와 다른 방식으로 최단 경로를 찾는 프로그램을 구현해봐야겠다는 생각이 들어 이러한 구현을 하였다.

**6. 프로젝트 실행 결과 캡쳐**

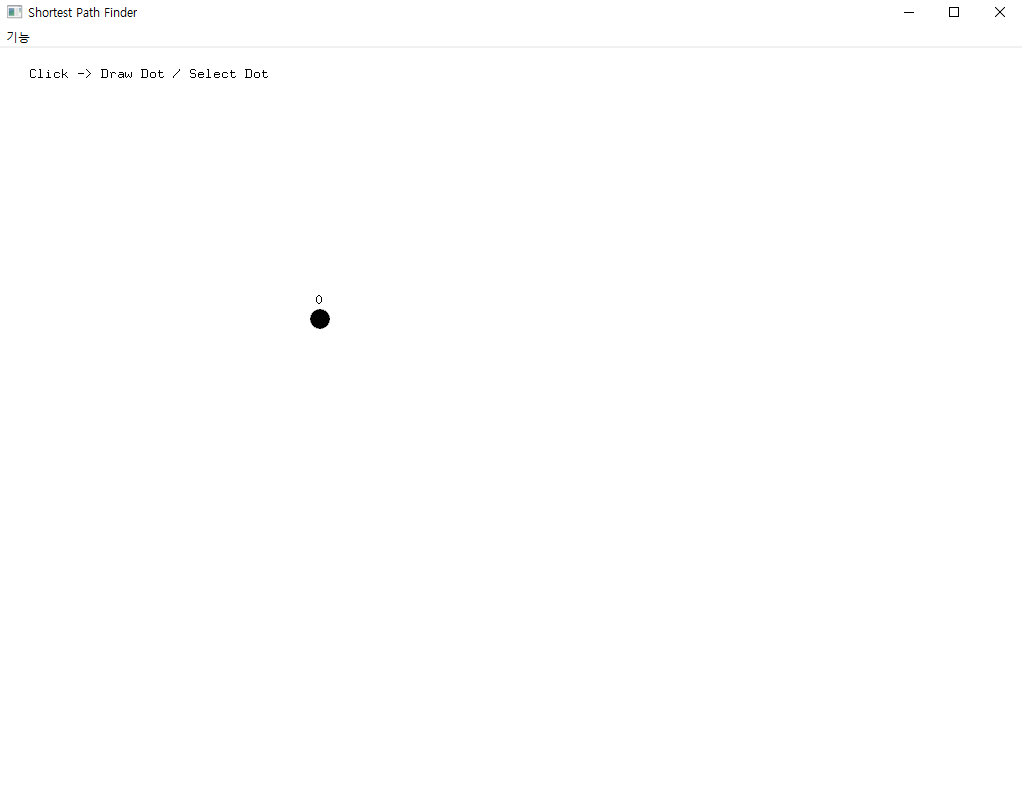
최초 실행시



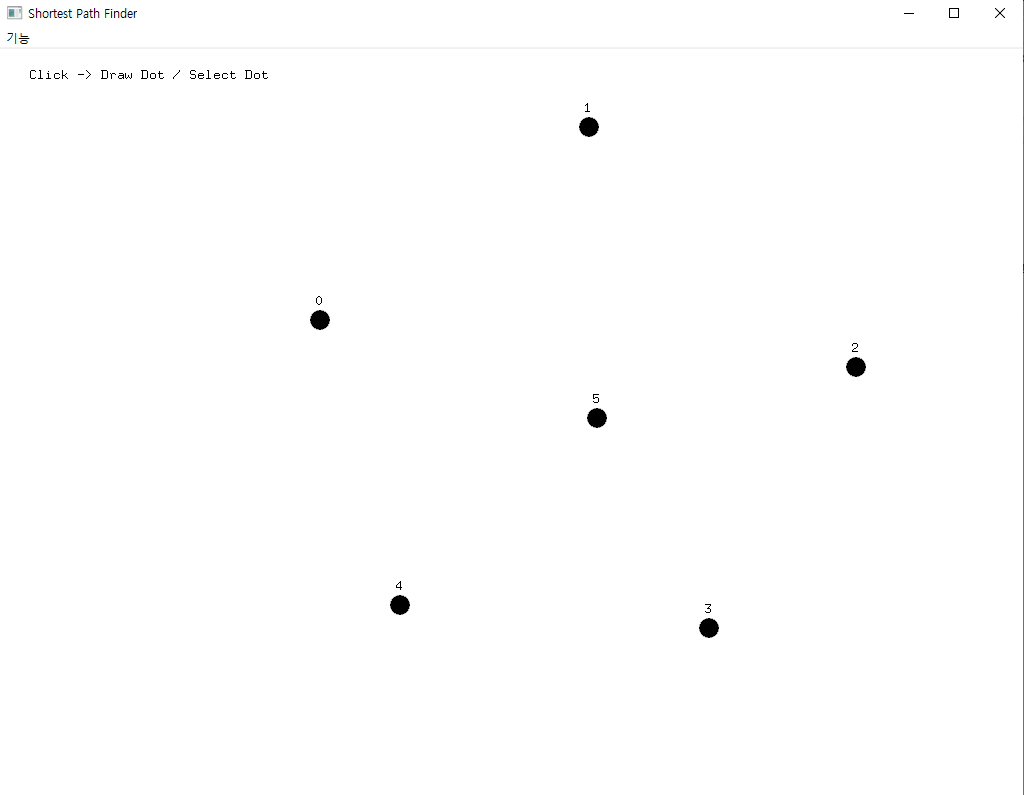
“기능” 팝업의 여러 기능들



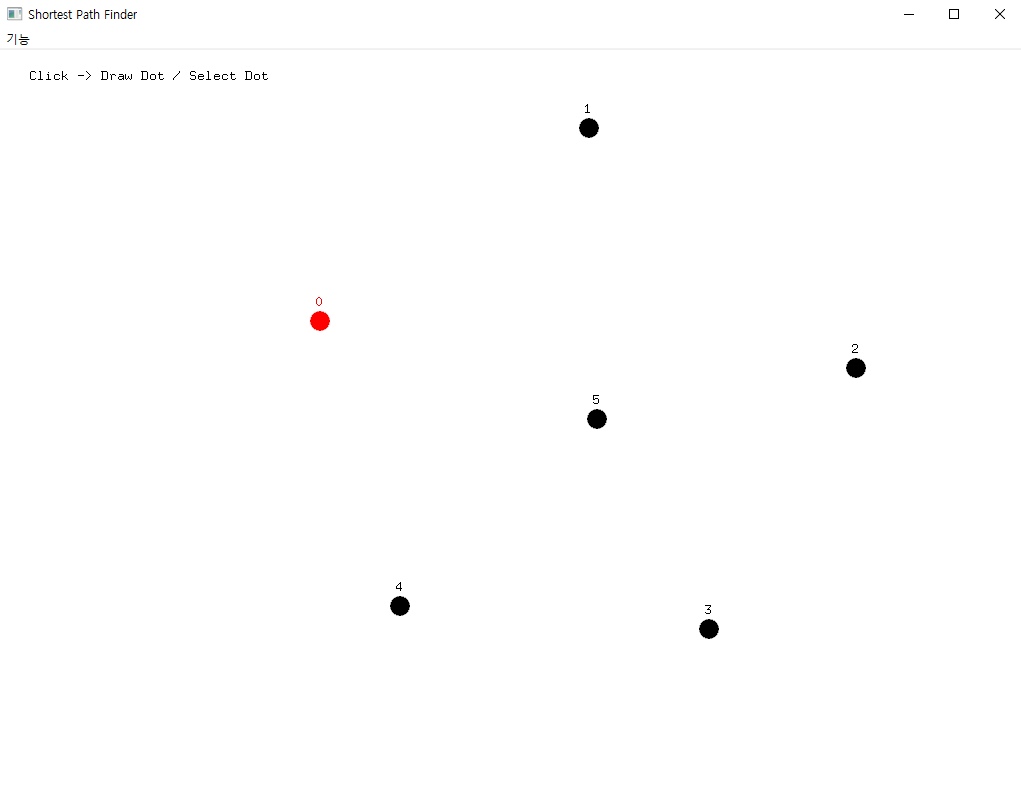
“점 그리기” 기능을 선택하고 창을 클릭시 점이 그려지는 모습



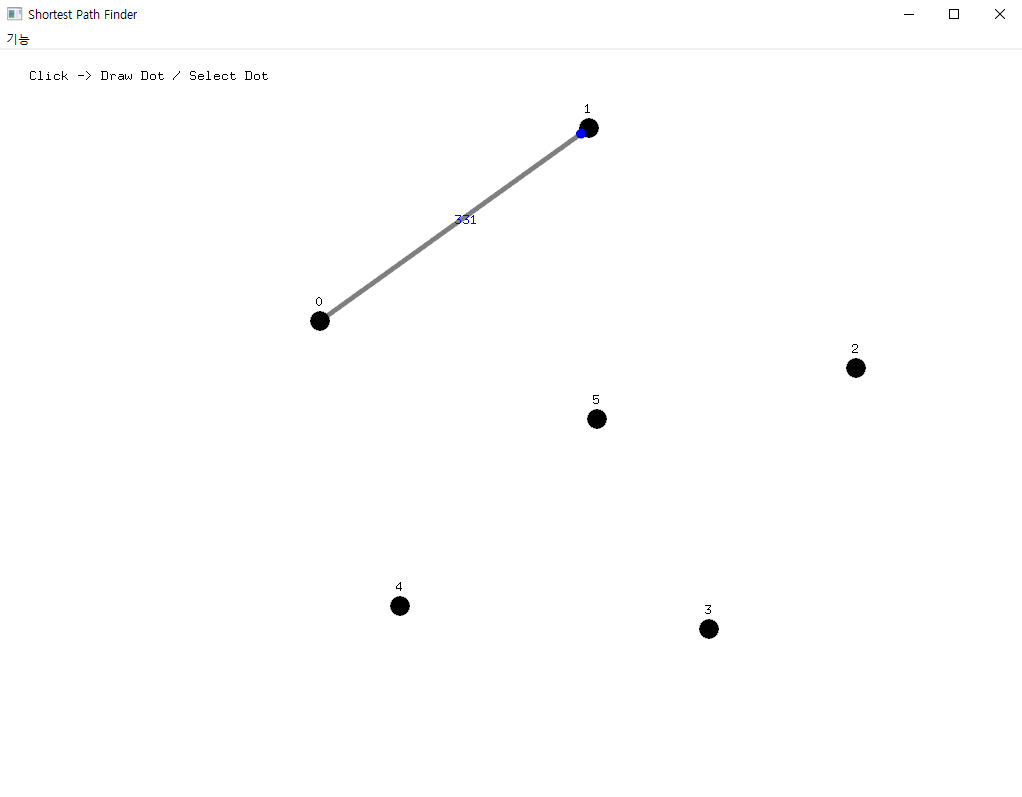
여러 개의 점을 그린 모습



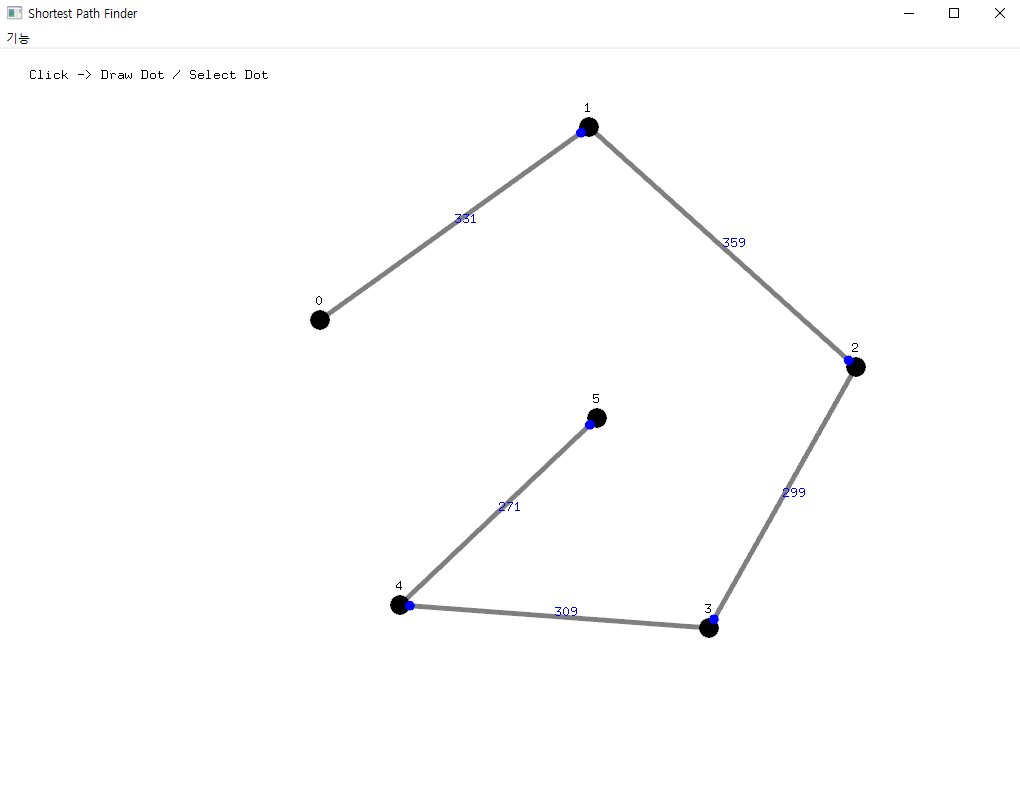
“직선 그리기” 기능을 선택하고 점중 하나를 클릭하여 선택한 모습



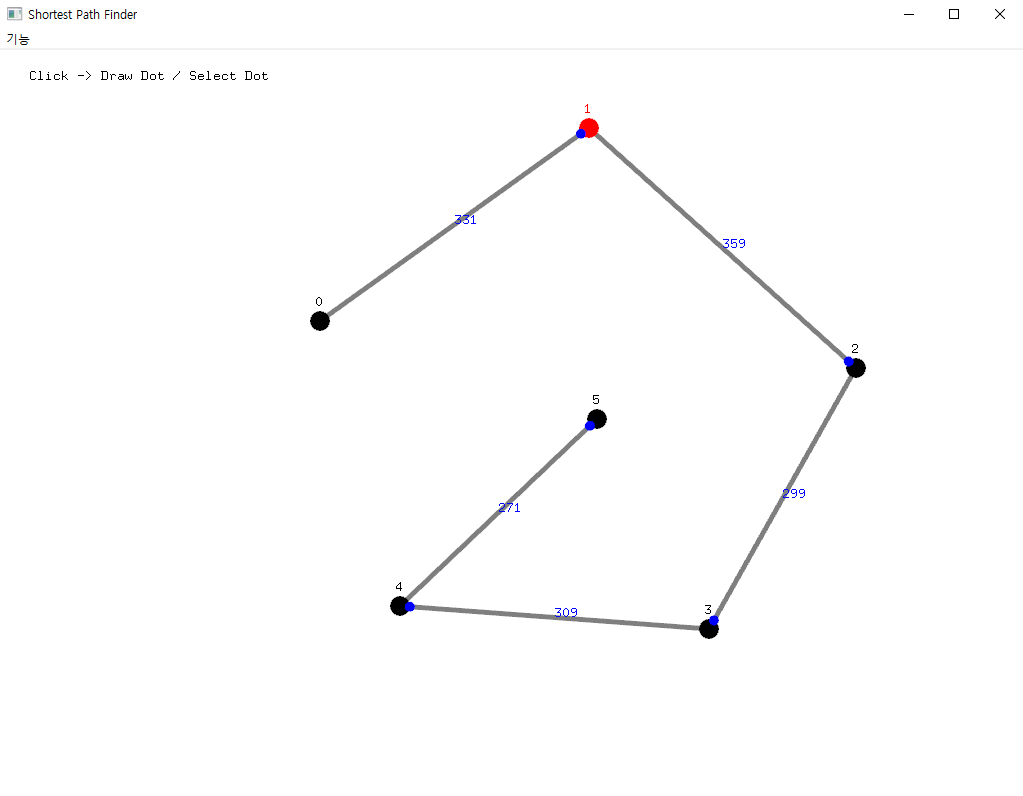
이후 다른 한 점을 선택하여, 선택한 두 점 사이의 직선과 직선의 방향, 길이가 그려진 모습



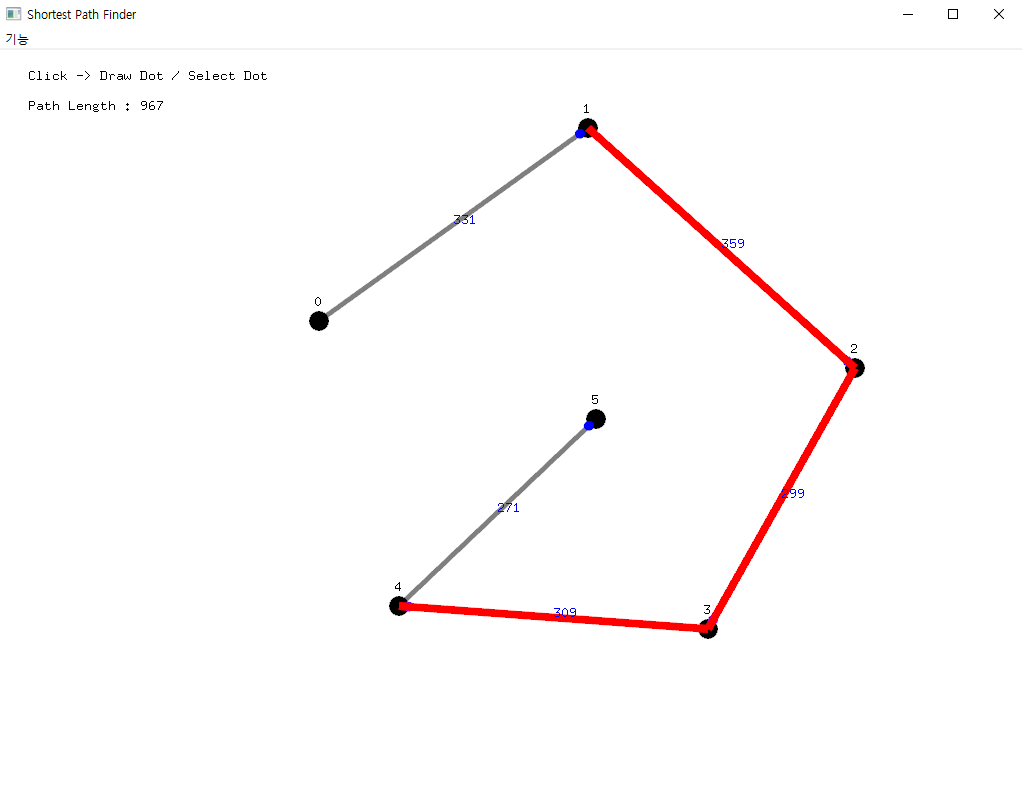
동일하게 반복하여 여러 개의 직선을 그린 모습



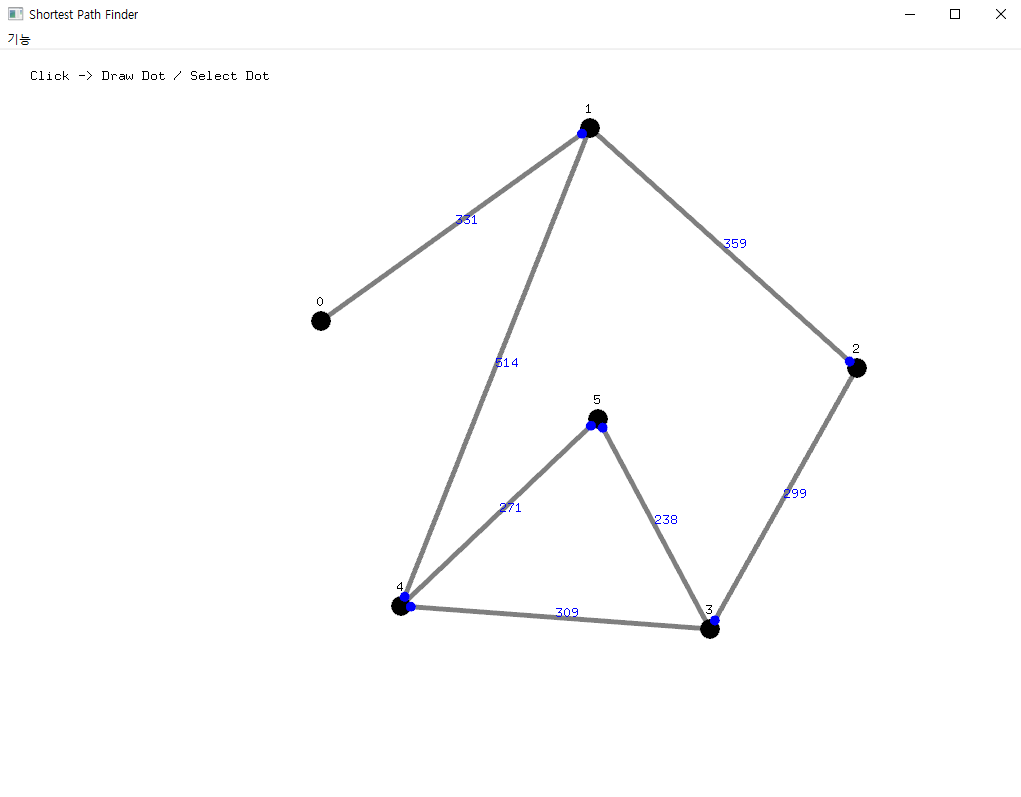
“최단 경로 찾기” 기능을 선택하고 점중 하나를 선택한 모습



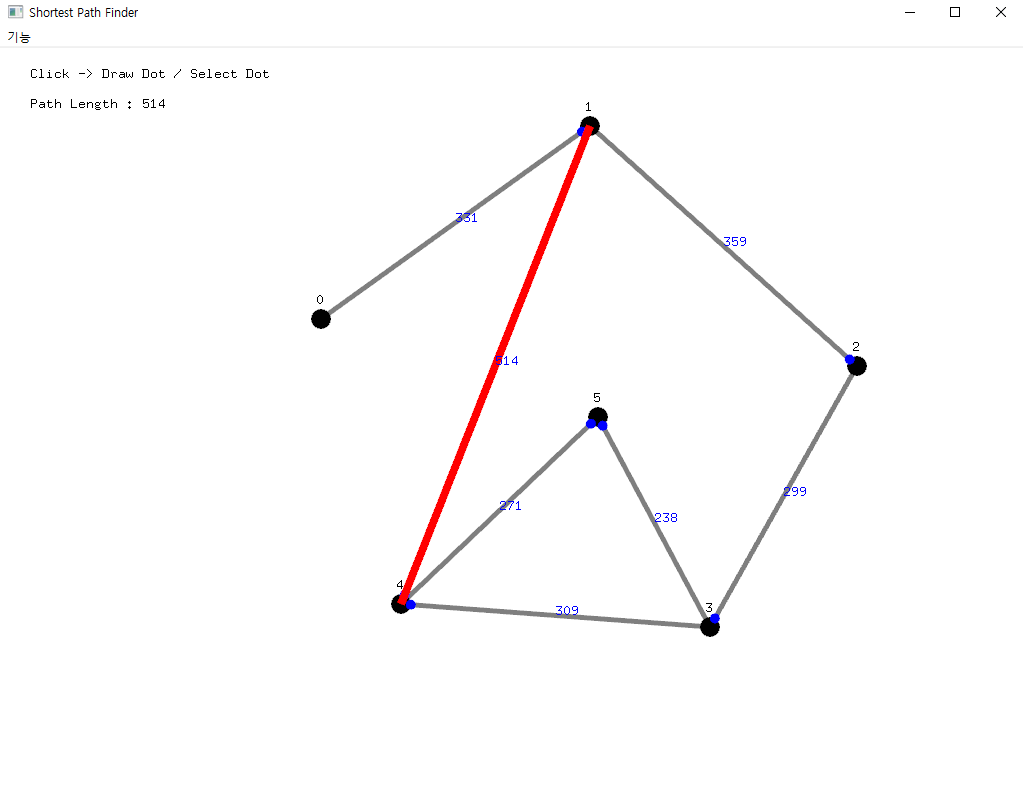
이후 다른 점을 선택하여 두 점 사이의 최단 경로와 그 길이가 그려진 모습



다시 “직선 그리기” 기능을 선택하여 직선을 추가한 모습



다시 “최단 경로 찾기” 기능을 선택하고 앞과 동일하게 1번, 4번 점을 선택한 모습



**7. 느낀 점 및 개선 사항**

느낀 점 :

사용자가 직접 점과 직선을 그리는 시스템의 구현을 통해, openframeworks의 GUI 기능의 설계 방법을 배울 수 있었고, 최단 경로 탐색 기능을 구현하며 Dijkstra's algorithm을 직접 활용해봄으로써 이에 대한 이해도를 높일 수 있었다. 또한 내가 만들어보고 싶은 프로그램을 직접 설계하고 제작하는 것에는 고려해야 할 사항이 생각보다 더 많다는 것을 느꼈다. 그리고 직선의 방향을 나타내는 점을 나타내는 점을 그리는 부분을 구현하며, 수학적 지식과 프로그램 구현 사이의 연관성도 느낄 수 있었다.

개선 사항 :

점을 그리는 것 말고도 사용자의 편의를 위해 점을 삭제하는 기능이나, 점과 점 사이의 경로를 직선이 아닌 곡선으로도 그릴 수 있는 기능을 추가하면 좋을듯 하다. 아니면 실제로 이 프로그램을 지도의 길찾기 기능과 유사하게 사용할 수 있게 창 위에 지도의 이미지를 불러와서 사용자가 그 위에 점을 그리고 선택하여 실제 경로를 찾을 수 있게 만들어보는 것도 좋은 개선점이라고 느껴진다.