



강의실 간 최단 경로

컴퓨터알고리즘과실습

컴퓨터공학전공

2021111960 김희진 — 팀장

2021111955 신지영

2021112052 민채은

2018112180 정대용

서론

동국대학교 학생이라면 대부분 경험할 수 있는 문제가 있다. 바로 학교의 지리를 파악하지 못하여 강의 시간표를 짤 때 이동 시간을 정확히 고려하지 못하거나, 새로운 강의실을 찾아갈 때 길을 헤매는 등의 문제이다. 이는 동국대학교의 지도가 존재하나 그 길이 세세하지 못한 점 • 산 주변에 위치한 학교의 특성 상 건물과 건물 사이의 층 차 이의 존재 등의 원인으로 인해 발생하는 문제이다. 이러한 문제들을 해결하기 위하여 강의실에서 강의실을 이동할 때의 건물의 층 차이와 연결 통로 등을 고려한 최단 경로를 제공하는 프로그램을 구현하고자 한다.

이를 위하여 동국대학교의 강의실 간 거리를 측정한다. 우선적으로 컴퓨터공학과 학생들이 많이 오가는 원흥관과 신공학관의 강의실 간 거리만을 측정하여 프로그램의 test data로 사용하였다. 또한, 아직 학교 지리에 익숙해지지 못한 학부생을 해당 프로그램의 주사용자로 예측하였기 때문에, 연구실을 제외한 강의실, 실험실 등의 거리만을 측정하였다. 엘리베이터와 계단은 신공학관의 1 - 9층 직통 엘리베이터를 제외하고 모두 이동 시간을 동일시하였다. 또한 엘리베이터의 대기 시간은 시간대 • 요일 등에 따라 변하여 예측하기 어려우므로, 대기 시간을 0으로 가정하여 최단 경로를 구하였다.

본론

데이터 구성

원흥관과 신공학관에서 각 강의실 간 거리를 측정했다. 휴대폰의 “측정” 앱으로 지점간의 거리를 간편하게 구할 수 있었다. 이 데이터를 모아 정점과 간선의 데이터를 구성했다.

시작	끝	거리 M
원흥관 4층 E동 남쪽 계단, 413	원흥관 4층 411, 415	3.9
원흥관 4층 411, 415	원흥관 4층 409, 417	4.2
원흥관 4층 409, 417	원흥관 4층 407, 419	3.6
원흥관 4층 407, 419	원흥관 4층 406	2.7
원흥관 4층 406	원흥관 4층 420	0.3
원흥관 4층 420	원흥관 4층 405, 421	2.4
원흥관 4층 405, 421	원흥관 4층 404	1.5
원흥관 4층 404	원흥관 4층 403	2.7
원흥관 4층 403	원흥관 4층 여자 화장실, E동 중앙 계단	1.5
원흥관 4층 여자 화장실, E동 중앙 계단	원흥관 4층 남자 화장실	2.7
원흥관 4층 남자 화장실	원흥관 4층 E동 입구	5.22
원흥관 4층 E동 입구	원흥관 4층 F동 엘리베이터, 423	4.13
원흥관 4층 423	원흥관 4층 450	6.3
원흥관 4층 450	원흥관 4층 427	3.3
원흥관 4층 427	원흥관 4층 428	2.1
원흥관 4층 428	원흥관 4층 450	0.9
원흥관 4층 450	원흥관 4층 429	2.1
원흥관 4층 429	원흥관 4층 447	0.9
원흥관 4층 447	원흥관 4층 F동 중앙 계단	2.7
원흥관 4층 F동 중앙 계단	원흥관 4층 446	0.6
원흥관 4층 446	원흥관 4층 444	3.6
원흥관 4층 444	원흥관 4층 430	0.3
원흥관 4층 430	원흥관 4층 431	3
원흥관 4층 431	원흥관 4층 442	1.8
원흥관 4층 442	원흥관 4층 432	0.6
의총과 1층 120		4.5

사용한 알고리즘

우리 팀은 강의실 간 최단 경로를 구하기 위해 플로이드-워셜(Floyd Warshall), 데이크스트라(Dijkstra), A* 알고리즘 세 개를 사용했다.

플로이드-워셜 알고리즘은 가중 그래프상의 모든 꼭짓점 쌍 간의 최단 경로를 모두 구하는 알고리즘이다. 모든 경로를 탐색하면서 추정 최단 경로를 점진적으로 개선시켜가며 최단 경로를 찾는다.

데이크스트라 알고리즘은 플로이드-워셜 알고리즘과는 다르게 한 시작 정점에서 다른 모든 꼭짓점까지의 최단 경로를 모두 구하는 알고리즈다. 매번 최단 경로의 정점을 선택해 탐색을 반복하며, 간선의 가중치가 음수이지 않아야 한다는 특징이 있다.

A* 알고리즘은 시작 정점에서부터 끝 정점까지의 최단 경로를 구하는 알고리즈다. 각 정점에 대해 다음 방문할 정점을 정하는데 최적의 경로에 대한 추정값인 휴리스틱을 사용한다. 이 프로젝트에 사용된 휴리스틱 함수는 많이 사용되는 휴리스틱 함수인 맨해튼 거리이다. 맨해튼 거리는 각 축에 대해 두 점 사이의 좌표 차이를 더한 것으로, 수식으로 표현하면 다음과 같다:

$$d = |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|$$

플로이드-워셜 알고리즘

가중 그래프상의 모든 꼭짓점 쌍 간의 최단 경로를 모두 구하는 알고리즘

자료 구조

```
class Vertex {
    string label;
};

class WtGraph {
    int maxSize, size;
    Vertex *vertexList;
    int *next;
    float *adjMatrix;
    float *pathMatrix;
};
```

- **Vertex** — 그래프의 정점을 나타내는 객체
 - **label** — 각 노드의 이름(신공학관 6119)을 저장하는 변수

- **WtGraph** — 그래프를 구성하는 객체
 - **vertexList** — 정점들을 저장하는 배열
 - **next** — 정점에서 다른 정점으로 갈 때 다음 경로를 저장하는 배열
 - **adjMatrix** — 정점 사이의 간선을 저장하는 배열
 - **pathMatrix** — 정점 사이의 최단 거리를 저장하는 배열

구현 설명

```

void WtGraph::floyd() {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            pathMatrix[i * maxSize + j] = adjMatrix[i * maxSize + j];
            next[i * maxSize + j] = j;
        }
    }

    for (int m = 0; m < size; m++) {
        for (int j = 0; j < size; j++) {
            for (int k = 0; k < size; k++) {
                if (getPath(j, m) == infiniteEdgeWt
                    || getPath(m, k) == infiniteEdgeWt) {
                    continue;
                }
                if (getPath(j, k) > getPath(j, m) + getPath(m, k)) {
                    setPath(j, k, getPath(j, m) + getPath(m, k));
                    next[j * maxSize + k] = next[j * maxSize + m];
                }
            }
        }
    }
}

```

1. **초기화** — pathMatrix를 간선의 데이터를 이용해서 초기화한다. next에는 현재 정점 다음에 방문할 정점을 저장한다. next[i][j]는 정점 i에서 j에 도달하기 위해 다음에 방문할 정점을 나타낸다.

2. 모든 꼭짓점 쌍 간의 최단 경로 계산

반복문을 돌면서 pathMatrix를 업데이트한다. 여기서 m , j , k 는 각각 경유 정점, 출발 정점, 도착 정점을 나타낸다. $j \rightarrow m \rightarrow k$ 경로가 $j - k$ 경로보다 가까운 경우에 pathMatrix를 업데이트한다.

그리고 바뀐 경로에 따라 next를 업데이트한다. $\text{next}[j][k] = \text{next}[j][m]$ 을 통해 k 로 가는 경로에 m 을 거쳐 가도록 업데이트한다.

최단 경로 탐색은 삼중 반복문에 따라 $O(N^3)$ 의 시간 복잡도를 가진다.

- 출발 정점에서 도착 정점까지의 경로 출력** — 반복문이 종료되고 최단 경로를 저장하고 있는 next 배열을 이용해서 출발 정점 u 부터 도착 정점 v 까지 경로를 출력한다. u 가 도착 정점이 될 때까지 $u = \text{next}[u][v]$ 를 반복해 경로를 탐색한다. 경로를 출력하는데는 $O(N)$ 의 시간 복잡도를 갖는다.

복잡도 분석

플로이드-워셜 알고리즘은 모든 꼭짓점 쌍 간의 최단 경로를 구하는 알고리즘이므로 $O(N^3)$ 의 시간 복잡도를 가지는 복잡한 알고리즘이라는 단점이 있다. 하지만 이미 받은 데이터로 경로를 모두 구성하는 과정을 전처리 과정으로 본다면, 이후에는 사용자에게 원하는 경로를 입력 받아 $O(N)$ 의 선형 시간 내에 사용자에게 최단 경로를 출력해줄 수 있다는 장점이 있다.

터미널 출력 결과

출발 강의실을 입력하세요 (ex 원흥관 314) : 원흥관 314
도착 강의실을 입력하세요 (ex 신공학관 6119) : 신공학관 6119

START! !

원흥관 314

- >원흥관 3층 F동 남쪽 계단
- >신공학관 7층 원흥관 방면 입/출구
- >신공학관 7층 중앙 엘리베이터
- >신공학관 7층 갈림길 G
- >신공학관 7층 화장실 갈림길 B
- >신공학관 7층 남쪽 갈림길 D
- >신공학관 7층 남쪽 갈림길 B
- >신공학관 7층 남쪽 엘리베이터(남쪽 계단)
- >신공학관 6층 남쪽 엘리베이터(남쪽 계단)
- >신공학관 6119

GOAL! !

다익스트라 알고리즘

시작 정점에서 다른 모든 꼭짓점까지의 최단 경로를 모두 구하는 알고리즘

자료 구조

```
vector<pair<int, float>> adj[MAX];
vector<string> nodeInfo;
vector<int> before;

vector<float> dijkstra(int startIdx, int V) {
    vector<float> dist(V, INF);
    priority_queue<pair<float, int> > pq;
}
```

- vector **adj[]** — 인접 리스트. $adj[n]$ 은 n 번째 정점과 이어진 정점들의 번호와 둘 사이의 가중치 쌍을 담은 벡터이다.
- vector **nodeInfo** — 정점의 이름을 저장하는 벡터이다.
- vector **before** — 방문한 정점의 인덱스를 저장한 벡터. 경로를 추적하기 위해 사용한다.
- vector **dist** — 출발 정점에 대한 모든 정점의 최단 거리를 저장하는 벡터. 크기는 정점의 수 V , 초기값은 $INF(1e9)$ 로 초기화한다.
- **pq** — 최소 비용으로 다음 방문할 정점을 구하기 위해 사용하는 우선 순위 큐. 목표 정점까지의 $dist$ 값을 기준으로 내림차순으로 정렬된다. 값이 같은 경우 목표 정점의 인덱스를 기준으로 내림차순 정렬된다.

구현

```

vector<float> dijkstra(int startIdx, int V) {
    vector<float> dist(V, INF);
    priority_queue<pair<float, int>> pq;

    dist[startIdx] = 0;
    pq.push(make_pair(0, startIdx));
    before[startIdx] = startIdx;

    while (!pq.empty()) {
        float cost = -pq.top().first;
        int cur = pq.top().second;
        pq.pop();

        for (int i = 0; i < adj[cur].size(); i++) {
            int next = adj[cur][i].first;
            float nCost = cost + adj[cur][i].second;

            if (nCost < dist[next]) {
                dist[next] = nCost;
                before[next] = cur;
                pq.push(make_pair(-nCost, next));
            }
        }
    }
    return dist;
}

```

간단히 과정을 서술하면 다음과 같다.

1. 한 노드에서 도달할 수 있는 노드 중 가장 적은 비용이 드는 노드를 찾는다.
2. 그 노드의 이웃 노드들에 도달하는데 드는 비용을 계산한다.
3. 1, 2단계를 그래프의 모든 노드들에 대해 반복한다.
4. 최종 경로를 계산한다.

dijkstra 함수에서 pq는 목표 정점까지의 dis값을 기준으로 내림차순, 이 값이 같으면 목표 정점의 index값을 기준으로 내림차순으로 하여 정렬된다. 먼저 dist의 startIdx번째 값은 0으로 할당하고, 시작 정점에 대한 dist 값 0과 startIdx를 짹지어 pq에 삽입하고, before의 startIdx번째 값은 startIdx로 할당하여 시작정점을 방문한다. 그리고 while문을 반복하여 최단 경로를 계산한다.

pq의 맨 위 데이터의 first 요소(방문한 정점의 dist 값)를 부호를 바꿔 cost에 할당하고, second 요소(방문한 정점의 인덱스 값)는 cur에 할당한다. 정점을 방문했으므로 pop을 하여 pq에서 빼준다. 그 다음, for문을 통해 현재 방문한 정점과 연결된 모든 정점을 조사한다. 인덱스가 cur인 시작점과 연결된 i번째 정점의 인덱스를 next에 할당하고, 현재 방문한 정점을 거쳐서 i번째 정점을 갈때의 비용을 nCost에 할당한다. 만약 인덱스가 next인 정점을 가고자 할 때, 현재 방문한 정점을 거친 비용인 nCost가 기존 비용보다 더 싸다면, dist의 next번째 값을 nCost로 할당하여 최단 비용으로 갱신해준다. 인덱스가 next인 정점으로 가는 최단경로는 직전에 cur을 거치므로 before의 next번째에 cur을 저장한다. 그리고 pq에 음수 nCost와 next를 짹지어 삽입한다. 우선순위 큐는 가장 큰 값이 맨 위에 있기 때문에, 가장 작은 cost를 구하기 위해 음수로 pq에 넣어주고, top 함수를 이용해 값을 읽을 때는 다시 마이너스를 붙여 본래의 양수 값으로 읽어준다. for문 반복이 끝나면, 다시 while문의 처음으로 돌아가 pq에 저장된 정점 중 가장 dist값이 작은 정점을 방문한다. pq가 empty 상태가 되면, 더 이상 방문할 정점이 없다는 의미이므로 while문을 끝내고 dist를 리턴한다.

데이터 입출력

- CSV에서 nodeInfo, adj에 저장** — 가중치 데이터파일인 "edges.csv"파일을 열어 ,을 구분자로 하여 시작 정점 이름, 도착 정점 이름, 두 정점 사이의 거리를 각각 from, to, cost에 저장한다. 그리고 from과 to 각각 nodeInfo에 동일한 이름이 존재하는지 myfind 함수를 통해 탐색하고, 없는 경우 nodeInfo에 새 정점을 삽입하는 과정을 거친다. 또한 adj의 fromIndex번째에 <toIndex, cost>쌍을, toIndex번째에도 <fromIndex, cost>쌍을 맨 뒤에 삽입하여 양방향 그래프를 만든다. 이 과정을 fs파일에 끝에 도달할 때까지 반복하면, 모든 정점과 간선에 대한 정보가 nodeInfo, adj에 저장된다.
- 사용자로부터 출발, 도착 정점 입력받고 인덱스 저장** — assign 함수를 통해 before의 크기는 정점의 수인 index만큼 지정하고 모든 값을 -1로 초기화한다. 그 다음, 사용자에게 한 문장으로 출발 강의실을 입력받아 start에 저장하고 nodeInfo의 처음부터 끝까지 start와 동일한 요소가 있는지 찾는다. 찾지 못했으면 에러 메시지를 출력하고 리턴한다. 찾았을 경우, 시작 정점의 인덱스를 fromIndex에 저장한다. 그리고 dijkstra 함수를 통해 시작 정점에서 모든 정점으로의 최단 경로를 탐색한다. 도착 강의실도 입력받아 goal에 저장하고 goal이 nodeInfo에 존재할 경우, 발견된 도착 노드의 인덱스 값을 toIndex에 할당한다.
- 출력** — dijkstra 함수의 리턴값이 메인의 dist에 할당됨으로써 시작 정점에서 모든 정점으로의 최단 거리 정보가 이에 저장된다. 시작정점에서 목표정점까지의 최단경로를 저장할 route를 선언하고, route에 먼저 목표정점 이름을 삽입한다. 그리고 while문을 통해 목표정점을 시작으로 경로를 역추적하여 route에 저장한다. 그 다음, route의 마지막 요소인 시작노드를 출력하고, for문을 통해 나머지 route를 출력하여 시작정점에서 목표정점까지의 최단 경로를 텍스트로 출력한다.

복잡도 분석

`myfind` 함수는 매개변수 `first`가 `last`와 같아지거나 `first`가 가리키는 값이 `value`와 일치하면 `for`문을 멈춘다. 메인에서 호출된 2번의 `myfind`는 모두 `first`로는 `nodeInfo`의 `begin()`을, `last`로는 `nodeInfo`의 `end()`를 인자로 넣어주기 때문에 최악의 시간 복잡도는 `nodeInfo`의 크기이자 정점의 수인 V 에 대하여 $O(|V|)$ 이다.

우선순위 큐는 이진트리 형태이므로 반복적으로 자식 노드로 내려가면서 새 데이터를 삽입할 위치나 삭제할 데이터의 위치를 찾는다. 즉, 최대 트리의 높이만큼의 비교가 일어나므로, 우선순위 큐의 삽입과 삭제의 시간 복잡도는 모두 노드(정점)의 수 V 에 대하여 $O(\log |V|)$ 이다.

`dijkstra` 함수는 `adj`에 저장된 모든 간선의 가중치 값을 읽는다. 최악의 경우, 가중치 값을 읽을 때마다 `dist`가 갱신되고 `pq`에 삽입될 것이다. 간선의 수를 E , 정점의 수를 V 라고 했을 때, 현재 정점과 연결된 주변 정점들을 확인하는 과정은 E 만큼 반복되고, `pq`의 `push`, `pop` 과정의 복잡도는 $O(\log |E|)$ 이다. 따라서 모든 간선을 우선순위 큐에 넣고 뺀다고 했을 때 시간복잡도는 $O(|E| * \log |E|)$ 이다. 이때 만약 모든 정점이 서로 연결되어있을 경우, E 는 $V * (V - 1)$ 의 값을 가질 것이다. 즉, $\log E$ 는 $\log |V|^2 = 2 * \log |V|$ 보다 작으므로 시간복잡도는 $O(|E| * \log |V|)$ 이다.

터미널 출력 결과

출발 강의실을 입력하세요 (ex 원흥관 314) : 원흥관 314
 도착 강의실을 입력하세요 (ex 신공학관 6119) : 신공학관 6119

```
***START!!***
원흥관 314
->원흥관 3층 F동 남쪽 계단
->신공학관 7층 원흥관 방면 입/출구
->신공학관 7층 중앙 엘리베이터
->신공학관 7층 갈림길 G
->신공학관 7층 화장실 갈림길 B
->신공학관 7층 남쪽 갈림길 D
->신공학관 7층 남쪽 갈림길 B
->신공학관 7층 남쪽 엘리베이터(남쪽 계단)
->신공학관 6층 남쪽 엘리베이터(남쪽 계단)
->신공학관 6119
***GOAL!!***
```

A* 알고리즘

시작 정점에서부터 끝 정점까지의 최단 경로를 휴리스틱을 사용해 구하는 알고리즘

데이터 구성

위의 휴리스틱 함수를 사용하기 위해 지점의 이름만을 포함하고 있던 정점 데이터에 x, y, z 좌표를 추가로 저장하도록 변경했다. 이를 위해 측정한 값을 3차원 좌표계에 옮기는 전처리 과정을 거쳤다.

좀 더 쉽게 처리하기 위해 신공학관 좌표계와 원흉관 좌표계를 나누어 각각의 원점에 대해 상대적인 좌표를 작성한 뒤, 원흉관 좌표계를 신공학관 좌표계와 실제적인 관계가 일치하도록 변환하는 과정을 거쳤다. 최종적으로 하나의 좌표계 상의 x , y , z 좌표를 구했다.



자료 구조

```

struct Coord3D {
    double x, y, z;
    Coord3D(double x, double y, double z) : x(x), y(y), z(z){};
};

struct Vertex {
    int id;
    string name;
    Coord3D coord;
};

struct Edge {
    int vid1, vid2;
    double cost;
};

struct Graph {
    vector<Vertex> vertices;
    vector<vector<Edge>> edges;
};

A_star::A_star(Graph g, int sid, int gid) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>
open_queue;
    set<int> open_set;

    int came_from[vertices_.count];
    double g_score[vertices_.count];
    double f_score[vertices_.count];
    double tentative_g_score;
}

```

- **Coord3D** — 3차원 좌표를 표현하는 구조체이다.
- **Vertex** — 정점의 인덱스(id), 이름, 좌표를 담은 구조체이다.
- **Edge** — 간선을 담고 있는 구조체이다.
- **Graph** — 정점과 간선을 담고 있는 구조체이다. g.vertices[id]로 접근하면 해당 정점에 접근할 수 있다. 간선을 추가하는 함수는 두 정점의 id를 입력으로 받는다.

- **A_star**

- **sid, gid** — A* 알고리즘에서 탐색 대상이 되는 시작 정점과 끝 정점의 id이다.
- **open_queue** — 앞으로 탐색할 대상을 담는 우선 순위 큐이다. 큐에 자료 삽입시 아래서 다를 f_score가 작은 순서대로 정렬된다.
- **open_set** — C++은 우선 순위 큐에 대해 find와 같은 함수를 제공하지 않는다. 따라서 큐에 특정 원소가 들어있는지 추적하기 위해 보조적으로 사용하는 집합이다.
- **came_from[]** — came_from[n]은 n 바로 직전에 거쳐온 정점의 id를 담고 있다.
- **g_score[]** — 시작 정점부터 정점 n까지 오는데 가진 가장 최소의 cost를 담는 변수이다.
- **f_score[]** — $f(n) = g(n) + h(n)$ 으로, 현재 정점까지의 가장 최소의 cost와 끝 정점까지의 추정 cost의 합으로, 여러 경로 중 이 값이 작은 값을 우선 탐색하는 것이 A* 알고리즘의 핵심이다.
- **tentative_g_score** — 다음에 진행할 여러 노드에 대해 f_score을 구해보는데 사용되는 중간 변수이다.

구현

```

A_star::A_star(Graph g, int sid, int gid) : g(g), sid(sid), gid(gid)
{
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>
    open_queue;
    set<int> open_set;
    open_queue.push(pair<int, int>(0, sid));
    open_set.insert(sid);

    const int vertices_count = g.vertices.size();
    int came_from[vertices_count];
    for (int i = 0; i < vertices_count; ++i)
        came_from[i] = -1;

    double g_score[vertices_count];
    for (int i = 0; i < vertices_count; ++i)
        g_score[i] = MAXFLOAT;
    g_score[sid] = 0;

    double f_score[vertices_count];
    for (int i = 0; i < vertices_count; ++i)
        f_score[i] = MAXFLOAT;
    f_score[sid] = heuristic(sid);

    while (!open_set.empty())
    {

```

```
int current = open_queue.top().second;
if (current == gid)
    reconstruct_path(came_from, current);

open_queue.pop();
open_set.erase(current);
vector<int> current_neighbors = g.getNeighbors(current);

for (int neighbor : current_neighbors)
{
    double tentative_g_score = g_score[current]
        + g.getCost(current, neighbor);
    if (tentative_g_score < g_score[neighbor])
    {
        came_from[neighbor] = current;
        g_score[neighbor] = tentative_g_score;
        f_score[neighbor] = tentative_g_score + heuristic(neighbor);
        if (open_set.find(neighbor) == open_set.end())
        {
            open_queue.push(pair<int, int>(f_score[neighbor], neighbor));
            open_set.insert(neighbor);
        }
    }
}
```

BFS와 비슷한 방법으로, 각 정점에서 이어진 정점들을 큐에 집어 넣으면서 시작 지점부터 범위를 넓혀가면서 탐색을 진행한다. 이때 큐는 우선 순위 큐를 사용하고, 우선 순위는 f_score 가 낮을 수록 높도록 설계되었다. f_score 는 시작 정점부터 현재 정점까지의 $cost$ 중 최소($g(n)$)와 현재 정점에서 목표 정점까지의 추정 $cost(h(n))$ 의 합을 말한다. f_score 가 낮을 수록 이 경로를 지날 때 $cost$ 가 낮다고 추정되기 때문에 먼저 탐색해보는 것이다.

먼저 시작 지점을 큐에 넣고, 인접한 정점들을 구한다. 인접한 정점으로 진행했다고 가정하고 그 지점에서 f_score 를 구해보는 작업을 진행한다. 먼저 인접한 정점으로 진행했다고 가정하는 일은 현재 정점의 g_score 에 인접한 정점까지의 $cost$ 를 더하는 일, 즉 $tentative_g_score$ 를 구하는 것과 같다. 그리고 $heuristic(neighbor)$ 를 통해 인접 정점에서 목표 정점까지의 추정 $cost$ 를 구한다. 그리고 이 둘을 더했을 때 f_score 가 나오는데, 이 값에 대해 큐에서의 우선 순위가 결정되게 된다. 즉, 인접 정점이 가지는 추정 $cost$ 가 낮은 순서대로 큐에 집어 넣고, 이후 과정은 너비 우선 탐색과 비슷하게 이어진 정점에 대해서 위 과정을 반복하게 된다. 그리고 목표 정점에 도달하면 탐색을 멈춘다.

휴리스틱

휴리스틱 함수를 구성하는데 중요한 점은 휴리스틱 함수가 추정하는 비용이 현재 지점에서 가능한 가장 낮은 비용보다는 작거나 같아야 한다는 점이다. 이 조건이 만족되어야만 휴리스틱 함수를 사용해 구한 경로가 최적임을 항상 보장할 수 있다.

복잡도 분석

우선 A* 알고리즘에 사용되는 그래프는 인접 리스트를 사용해서 구현했기 때문에 공간 복잡도는 대략 $O(|V|)$ 이다. 그리고 함수 내에서 사용되는 `came_from` 등과 같은 변수의 공간 복잡도 또한 $O(|V|)$ 를 갖는다.

시간 복잡도는 휴리스틱 함수에 따라 달려있다. 좋은 휴리스틱은 진행해야 하는 정점들 중 가능성성이 낮은 다수의 가지들을 잘라낼 수 있다. 맨해튼 거리를 사용한 A* 알고리즘의 시간 복잡도는 최악의 경우 $O(|E|)$ 라고 알려져 있다. 그리고 그 경우는 시작 경로부터 끝 경로까지 일자로 나열된 경우로, 모든 노드를 방문해야 한다.

터미널 실행 결과

출발 강의실을 입력하세요 (ex 원흥관 314) : 신공학관 5145

도착 강의실을 입력하세요 (ex 신공학관 6119) : 원흥관 350

신공학관 5145

신공학관 5층 북쪽 갈림길

신공학관 5147

신공학관 5층 중앙 엘리베이터

신공학관 6층 중앙 엘리베이터

신공학관 7층 중앙 엘리베이터

신공학관 7층 원흥관 방면 입/출구

원흥관 4층 F동 남쪽 계단

원흥관 411

원흥관 409

원흥관 407

원흥관 406

원흥관 420

원흥관 421

원흥관 404

원흥관 403

원흥관 4층 F동 중앙 계단

원흥관 3층 F동 중앙 계단

원흥관 3층 엘리베이터(계단)

원흥관 325

원흥관 350

이미지 출력

텍스트로 표현된 결과에 더해 해당 층의 이미지와 경로를 보여주도록 했다. 이를 위해 이미지와 해당 노드의 이미지 상 좌표 값을 구하고, OpenCV를 이용해 이미지와 경로를 합성해 보여주도록 했다.



결론

플로이드-워셜 알고리즘의 시간 복잡도는 $O(|V|^3)$, 데이크스트라 알고리즘의 복잡도는 $O(|E| * \log |V|)$, 맨해튼 거리를 휴리스틱으로 사용한 A* 알고리즘의 시간 볍잡도는 최악의 경우 $O(|E|)$ 로, A* 알고리즘의 성능이 가장 나은 것으로 보여진다.

또한 데이크스트라 알고리즘은 하나의 정점에서부터 다른 모든 정점까지의 최단 경로를 구하기 때문에 최적의 경로를 보장하지 않는다. 플로이드-워셜 알고리즘은 모든 꼭짓점 쌍 간의 최단 경로를 계산하는데 사용할 수 있지만 단순히 두 지점 간 최단 거리를 구하는데는 $O(|V|^3)$ 의 시간 복잡도가 소요되는 단점이 있다. A* 알고리즘은 시작점과 목표점의 최단거리를 구하며, 최적 경로의 근사값을 보장한다. 따라서, 시작 강의실에서 목표 강의실까지의 최단 경로를 구하는 문제에서는 휴리스틱 함수만 잘 정의되어있다면 알고리즘 자체는 A* 알고리즘이 가장 적합하다고 할 수 있다.

한계점

하지만 A* 알고리즘에 사용되는 정확한 좌표 데이터를 근거리에서는 얻기 힘들었고, 데이터를 변환하는데 시간이 많이 소요되었다. 그리고 강의실 간 거리를 사람이 직접 측정하였기 때문에 거리의 오차가 발생할 수 있다. 예를 들어 1층 계단의 x, y 좌표와 2층 계단의 x, y 좌표에 차이가 있을 수 있다. 이는 맨해튼 거리를 사용하는 휴리스틱 함수에 영향을 끼칠 수 있지만 이번 프로젝트에서 한 정점에서 경로가 최대 3가지 존재하므로, 영향이 미미할 것이다. 거리가 정확히 표시된 도면이 주어진다면 A* 알고리즘을 사용하는데 더 수월할 것으로 보인다.

기대 효과

현재 원흥관과 신공학관에 대해서만 거리 측정을 완료하여 동국대학교의 전교생에게 서비스를 제공하기에는 부족함이 있다. 추후 교내 전체의 data를 추가하면 매년 동국대학교에 새로이 입학하는 신입생, 익숙하지 않은 건물에 방문해야 하는 학부생, 동국대학교에 방문하는 자 등 동국대학교의 지리에 익숙치 않은 모든 사람들이 유용하게 사용할 수 있을 것이다.