

DataStructure HW10

C211123 이준선

December 2022

목차

1	개요	1
2	문제 구성	1
2.1	문제	1
2.2	입력 파일	1
2.3	입력 받기	2
2.4	그래프 구성 방식	3
3	(a) 최소 시간 경로 찾기	5
3.1	다익스트라 알고리즘 선택 이유	5
3.2	코드 구현	7
4	(b) 중간역 찾기	10
5	클래스 설계 내용 및 이유	11
6	결과값 분석	13
7	구현 시 어려웠던 점	15

그림목차

1	다익스트라의 최단 경로 생성 과정	6
2	출력 결과	14

코드목차

1	main.cpp	2
2	make_station function	3
3	subway.h 그래프 형성	3
4	insert_station and add_connection function	4
5	main:find_shortest_path	7
6	다익스트라 알고리즘 구현	8
7	코드 6 중 핵심 로직 확대	9
8	중간역 찾기	10
9	subway.h 클래스 설계 부분	11
10	destructor	13

1 개요

지하철의 노선도를 입력받고, 시작역과 도착역을 입력받아 두 역 사이의 최소 시간 경로와 두 역 간 이동 시간의 차이가 최소가 되게 하는 중간역을 출력하는 프로그램을 작성하였다. 지하철의 노선도는 “호선 출발역 호선 도착역” 형태의 파일 구조로 되어 있고, 시작역과 도착역 역시 마찬가지이다. 한편 중간역이라 함은 시작역에서 중간역까지 걸리는 시간, 도착역에서 중간역까지 걸리는 시간의 차이가 최소가 되게 하는 역을 의미한다. 모든 역간 이동 시간은 60초이며, 호선이 바뀌는 환승역에서는 30초가 추가된다.

2 문제 구성

2.1 문제

- 준회(출발역)는 현수네 집(도착역)에 지하철을 타고 가기로 했다. 이때 최소 시간 경로 및 걸리는 시간을 구하여라.
- 현수가 생각해보니 본인의 집이 더럽다며 그냥 밖에서 만나자고 했다. 이때 각자 걸리는 시간의 차이가 최소가 되는 역과 시간을 구하여라.

2.2 입력 파일

문제에서 사용된 입력 파일은 역간 정보를 구성하는 stations1.txt 파일과 출발역과 도착역 정보를 담고 있는 input1.txt 파일이다.

station1.txt 파일은 다음과 같이 구성되어 있다.

- 첫 줄의 경우 총 역의 개수이다. 예를 들어 72면, 72개의 역 정보가 있다는 의미이다.
- 둘 째줄부터 역 정보가 구성되어 있다.
띄어쓰기를 기준으로 (출발역의 호선) (출발역) (도착역의 호선) (도착역) 순으로 정보가 구성되어 있다. 출발역←→도착역 간 양방향으로 직접 연결되어 있다는 의미이다. 예를 들어 다음과 같이 구성되어 있다:

```
1 Jongno_5(o)-ga 1 Jongno_3(sam)-ga  
1 Jongno_3(sam)-ga 1 Jonggak  
1 Jonggak 1 City_Hall  
1 City_Hall 1 Seoul_Station  
1 Seoul_Station 1 Namyeong  
1 Namyeong 1 Yongsan  
1 Yongsan 1 Noryangjin  
1 Noryangjin 1 Daebang
```

- 만약 환승역의 경우, (환승 전의 호선) (환승역) (환승 후의 호선) (환승역) 순으로 정보가 구성되어 있을 수도 있다. 예를 들어 다음과 같이 구성되어 있다:

```
1 City_Hall 2 City_Hall
```

그리나 모든 환승역이 위와 같이 구성되어 있는 것은 아니며, station1.txt의 경우 1호선과 2호선을 잇는 환승역인 신도림의 경우 다음과 같이 구성되어 있다:

```
1 Yeongdeungpo 1 Sindorim
```

```
1 Sindorim 1 Guro
```

```
:
```

```
2 Daerim 2 Sindorim
```

```
2 Sindorim 2 Mullae
```

- 역 이름에는 띄어쓰기가 없다.
- 한글 이름의 역은 존재하지 않는다.
- 실제 2호선처럼 순환하는 호선이 있을 수 있다.

- 실제 1호선처럼 나뉘는 호선이 있을 수 있다.
- 호선의 개수는 1 이상 9 이하의 자연수이다¹.
- 모든 역은 양방향을 오갈 수 있어야 한다.

input1.txt 파일은 다음과 같이 구성되어 있다.

- 첫 줄은 출발역, 둘 째줄은 도착역의 정보가 입력되어 있다. 역 정보는 stations1.txt와 같이 띄어쓰기를 기준으로 (출발역의 호선) (출발역) (도착역의 호선) (도착역) 순으로 정보가 구성되어 있다. 예를 들어 다음과 같이 구성되어 있다:

```
3 Anguk
2 Chungjeongno
```

각 역간 이동은 순차적으로만 이동 가능하며, 노선의 변경은 환승역을 이용해야 한다. 각 역간 이동 시간은 60초, 환승 시간은 30초가 추가된다.

2.3 입력 받기

```

1 int main(int argc, char* argv[])
2 {
3     int numLine;
4     int line1, line2;
5     string src, dst;
6     if (argc != 3) {
7         cerr << "Argument Count is" << argc << endl << "Argument must
8         be " <<
9         argc << endl;
10    return 1;
11 }
12 fstream fin(argv[1]);
13 if (!fin) {
14     cerr << argv[1] << " open failed" << endl;
15     return 1;
16 }
17 fin >> numLine;
18
19 Subway subway;
20 // read file and insert station
21 for (int i = 0; i < numLine; i++) {
22     int line1, line2; // line1 = line of src, line2 =line of dst
23     string src, dst;
24     fin >> line1 >> src >> line2 >> dst;
25
26     subway.make_station(src, line1, dst, line2);
27 }
28 fin.close();
29
30 // input1.txt 시작역과 도착역 정보 읽기
31 fstream fin2(argv[2]);
32 if (!fin2) {
33     cerr << argv[2] << " open failed" << endl;
34     return 1;
35 }
```

¹1호선부터 9호선을 의미하며, 인천1호선, 수인선과 같은 호선은 고려하지 않는다.

```

35     fin2 >> line1 >> src; // start(source) station
36     fin2 >> line2 >> dst; // destination station
37     fin2.close();
38
39     return 0;
40 }

```

코드 1: main.cpp

코드 1은 stations1.txt와 input1.txt를 입력받는 main함수를 나타낸 것이다². Subway 클래스는 지하철 경로 탐색 등에 필요한 함수들과 변수들이 저장된 클래스이다. Subway 타입의 subway를 생성한 뒤, 역 정보를 make_station 함수에 전달해 그래프를 전달한다.

```

1 void Subway::make_station(string src, int line1, string dst, int
2   line2)
3 {
4     insert_station(src);
5     insert_station(dst);
6
7     // add connection
8     add_connection(src, dst, line1, line2);
9 }

```

코드 2: make_station function

코드 2은 make_station 함수로, insert_station 함수로 src(출발역)과 dst(도착역)을 생성하고, add_connection 함수로 src와 dst를 연결한다.

2.4 그래프 구성 방식

```

1 class Station {
2 public:
3     string name;
4     vector<Connector*> connections;
5     int max_time;
6     Station(string name);
7     friend ostream& operator<<(ostream& os, const Station& station);
8 };
9
10 class Connector {
11 public:
12     Station* dst;
13     int line;
14     int distance;
15     Connector(Station* dst, int line, int distance);
16 };
17
18 class Subway {
19 private:
20     map<string, Station*> station_list;
21 };

```

코드 3: subway.h 그래프 형성

²main.cpp의 전문이 아니며, 입력받는 부분만 표시되었다.

```

1 // function that insert a station into the station_list
2 void Subway::insert_station(string name) {
3     Station* station = new Station(name);
4     station_list.insert(pair<string, Station*>(name, station));
5 }
6
7 // function that add a connection between two stations
8 void Subway::add_connection(string src, string dst, int line1, int
9                             line2) {
10
11     Station* station1 = station_list[src];
12     Station* station2 = station_list[dst];
13     int distance = 60; // basic distance
14
15     // when trasferring, distance = 30
16     if (src == dst && line1 != line2) {
17         distance = 30;
18     }
19     else if (src == dst && line1 == line2) {
20         string error = "Error: " + src + " and " + dst + " are the same
21         station, but the same line";
22         throw error;
23     }
24
25     Connector* connector1 = new Connector(station2, line1, distance);
26     Connector* connector2 = new Connector(station1, line2, distance);
27     station1->connections.push_back(connector1);
28     station2->connections.push_back(connector2);
}

```

코드 4: insert_station and add_connection function

코드 3는 그래프 구성에 필요한 Station, Connector 클래스 전체와 Subway 클래스 속 `map<string, Station*> station_list`을 보여준다. 코드 4는 그래프를 형성하는 중요한 두 함수인 `insert_station`, `add_connection` 함수를 보여준다.

먼저 클래스 Station은 역의 정보를 담는 클래스로, 다음과 같이 구성된다:

- 역 이름 name
- 연결된 다른 역 Connections
이때 Connections는 클래스 Connector를 이용해 정보를 저장한다.
- 역까지 이동하는데 걸리는 최대 시간. 기본적으로 10000³으로 초기화하며, 만약 max_time보다 작은 time으로 해당 역까지 이동할 수 있다면 최단 경로와 최단 시간을 갱신한다.

클래스 Connector는 한 역에 연결된 다른 역의 정보를 담는 클래스로, 다음과 같이 구성된다:

- 목적지 dst
- 목적지의 호선 정보 line
- 목적지까지의 거리 정보 distance, 기본적으로 60초이나 환승역의 경우 30초이다. 한편 환승역의 경우, src와 dst가 같고 호선 정보가 다르면 환승역이라 간주한다.

³임의의 큰 값이며, 처음 들어온 값에 대해 무조건 갱신되게끔 하는 값이다. 만약 최소 시간이 10000이 넘어갈 경우, 이 초기값은 충분히 더 큰 값으로 갱신해야 한다.

station_list는 역 리스트로, 역 이름을 키로, 역 정보가 담긴 Station 구조체가 value로 담긴 map이다. 역 이름 string으로 Station 구조체 정보를 빠르게 찾기 위한 map이다.

한편 그래프를 구성하는 방식은, Station의 connections에 Connector 정보를 추가하는 식으로 그 래프를 구성한다. Connectors는 vector<Connector*> 타입으로, 구조체 Connector의 정보를 담는다. 예를 들어 신도림 역 Shindorim Station이 다음과 같이 구성되어 있다고 하자.

Station Shindorim:

- name = ‘Shindorim’
- connections =
 - Yeongdeungpo, line1, 60
 - Guro, line1, 60
 - Daerim, line2, 60
 - Mullae, line2, 60
- max_time = 10000

이는 Shindorim Station이 1호선 Yeongdeungpo Station과 60초 간격만큼, 1호선 Guro Station과 60초 간격만큼, 2호선 Daerim Station과 60초 간격만큼, … 연결되어 있다는 의미이다. 즉 Shindorim에서 Yeongdeungpo, Guro, Daerim, Mullae 역을 갈 수 있다. 당연히 모든 역은 양방향이기에, Guro 역의 connections에도 Shindorim, line1, 60이 포함되어 있다.

함수 insert_station은 Station 구조체를 형성한다. 역 이름을 매개 변수로 받아서, Station 구조체를 new로 동적 생성하고, 이를 station_list에 이름과 페어를 이루어 추가한다. 그러면 station_list[역이름]을 통해 그 역의 Station 구조체를 바로 검색할 수 있다.

함수 add_connection은 src, dst, line1, line2를 전달받는다. line1 src \longleftrightarrow line2 dst를 형성한다. 한편 동일한 역에 대해 호선만 다르게 입력될 수 있다⁴. 이때는 시간 정보인 distance⁵를 60이 아닌 30으로 놓고 역을 추가한다. 즉 호선이 다른 경우 같은 이름의 역에 대해서도 Connector를 가질 수 있다.

```
1 Connector* connector1 = new Connector(station2, line1, distance);  
2 Connector* connector2 = new Connector(station1, line2, distance);  
3 station1->connections.push_back(connector1);  
4 station2->connections.push_back(connector2);
```

이 부분이 각각 line1 src \rightarrow line2 dst와 line2 dst \rightarrow line1 src를 형성하는 코드이다. 양방향으로 Connector를 놓는다. 이 함수들을 이용해서 코드 2에서 역간 그래프를 추가한다.

3 (a) 최소 시간 경로 찾기

3.1 다익스트라 알고리즘 선택 이유

최단 경로 찾기에는 DFS, BFS, 다익스트라(Dijkstra), A*, θ^* 알고리즘 등 다양한 알고리즘이 존재한다. 각각의 알고리즘들은 각각의 상황과 경우에 맞추어 적합하다. 예를 들어 BFS는 거리, 시간 등의 가중치가 부여되지 않는 경우에 최단 경로를 찾기 유리하며, A* 알고리즘은 휴리스틱 기법을 도입해 성능을 크게 향상시킨 경로 찾기 알고리즘으로, 휴리스틱 기법을 적용할 수 있는 현실 세계의 지도나 실제 게임 지도 상에서의 경로 찾기에 유리하다. θ^* 알고리즘은 기계적인 움직임을 보이는 A* 알고리즘 경로를 자연스러운 시각 정보를 반영하여 부드럽게 움직일 수 있게 해주는 경로 스무딩 알고리즘으로, 기계적인 최단 경로를 찾아가는 것보다 캐릭터의 자연스러운 움직임을 구현하는데 중요한 게임 속에서 자주 사용한다. 한편 이 문제의 경우, 지하철 노선에 대한 정보만 존재하여 휴리스틱을 구현하기 힘들고, 각 역간 이동 시간과 환승 시간이라는 가중치가 존재한다. 이 경우 **다익스트라 알고리즘이** 가장 적합한 알고리즘일 것이다.

⁴src == dst이나 line1 \neq line2인 경우

⁵가중치를 의미. distance는 거리 정보를 의미하는 단위이나 여기서는 시간과 거리가 같은 가중치를 가지기에 같은 의미로 사용한다.

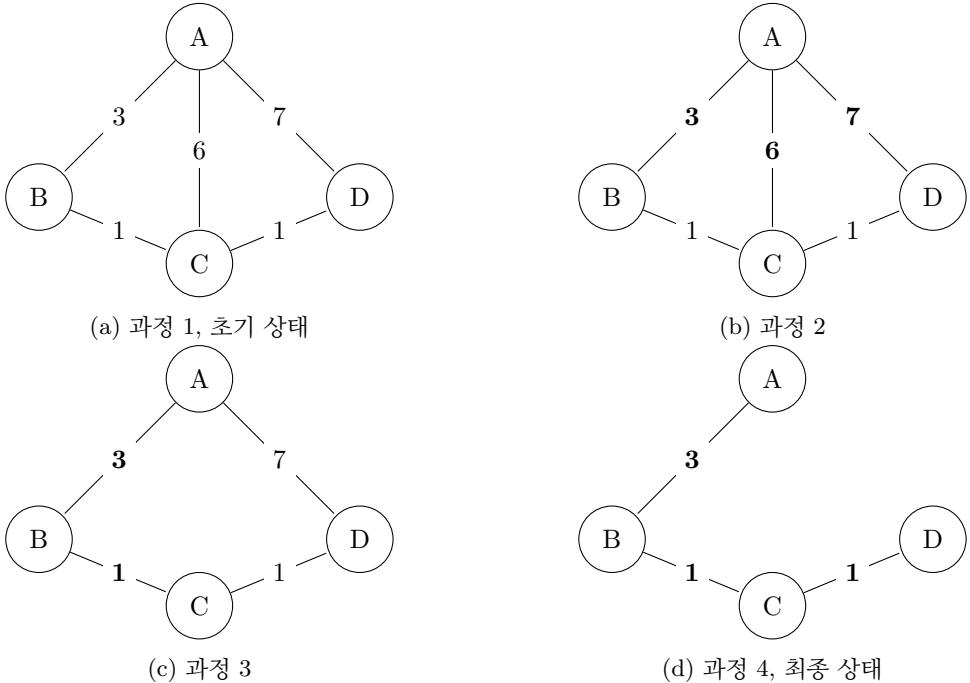


그림 1: 다익스트라의 최단 경로 생성 과정

다익스트라 알고리즘은 음의 가중치가 없는 그래프에서 한 정점과 다른 모든 정점 사이의 최단 경로를 구하는 탐욕적 방법의 알고리즘이다. 실제 음의 가중치를 갖는 경우는 매우 드물기에 널리 사용되는 기법의 알고리즘이다. 각 간선간 가중치가 모두 다를 때 유리하며, 우선순위 큐(힙)을 사용할 경우 최악의 시간 복잡도가 $O(n \log n)$ 임을 보장한다. 단, 이번 문제에서는 각 역간 이동 시간이 모두 60초로 동일한 점, 그리고 환승 시에만 환승 시간 30초가 추가된다는 점, 한 정점(Vertex)이 3개 이상의 간선(Edge)를 갖는 경우가 극히 드물다는 점을 고려하여, 우선순위 큐를 사용해도 큰 폭의 성능 향상을 기대할 수 없을 거란 판단에 하에 구현 난이도가 높은 우선순위 큐의 사용을 하지 아니하였다. 이럴 경우 훨씬 직관적인 코드 구성이 가능하다⁶.

한편 다익스트라 알고리즘의 기본 전제는 ‘최단 경로는 여러 개의 최단 경로로 이루어져 있다’라는 것이다. 즉 만약 정점 A에서 정점 D까지 가는 최단 경로가 $A \rightarrow B \rightarrow C \rightarrow D$ 라면, 경로 $A \rightarrow B$, $B \rightarrow C$, $A \rightarrow B \rightarrow C$, $B \rightarrow C \rightarrow D$ 도 모두 최단 경로일 것이라 가정하는 것이다. 물론 위 경우와 맞지 않는 특수한 경우를 반례로 구성할 경우, 위 가정은 거짓이 되므로 다익스트라에서 구한 최단 경로가 수학적으로 실제 최단 경로임을 보장하지는 않는다. 그러나 그리디 알고리즘의 특성상 수학적으로 참인 해를 보장하진 않지만 참에 가까운 해를 빠른 시간 복잡도로 구할 수 있다는 장점이 있다.

다익스트라 알고리즘의 작동 과정은 다음과 같다:

1. 출발 노드를 설정합니다.
2. 출발 노드를 기준으로 각 노드의 최소 비용을 저장합니다.
3. 방문하지 않은 노드 중에서 가장 비용이 적은 노드를 선택합니다.
4. 해당 노드를 거쳐서 특정한 노드로 가는 경우를 고려하여 최소 비용을 갱신합니다.
5. 위 과정에서 3번, 4번을 반복합니다.

그림 1a에서 정점 A를 보면, 현재 A에서 B, C, D로 가는 비용은 3, 6, 7로 확인할 수 있다(그림 1b). 그러나 정점 B에 대해 처리하면, 경로 $A \rightarrow B \rightarrow C$ 의 비용이 4로 경로 $A \rightarrow C$ 의 6보다 더 작다는 것을 알 수 있다. 이때 A에서 C로 가는 최단 경로를 $A \rightarrow B \rightarrow C$ 로 새롭게 갱신하고, $A \rightarrow C$ 를 무시하는

⁶단 이럴 경우 최악의 시간 복잡도가 $O(n)$ 인 단점이 있으나, 문제 조건 상 역의 개수가 한 호선 당 최대 100개씩 900개로 제한되어 있어, 큰 문제는 아닐 것이라 판단된다.

것이다(그림 1c). 마찬가지로 노드 C에 대해 각 최소 경로를 확인하면, A에서 D로 가는 최단 경로가 A→B→C→D 임을 알 수 있고, 이때 A→D를 무시한다(그림 1d).

3.2 코드 구현

```
1 void Subway::find_shortest_path(const string src, const string dst,
2                                 const int line)
3 {
4     // if src or dst are not in the station_list, throw an error
5     if (station_list.find(src) == station_list.end()) {
6         string error = "Error: " + src + "is not in the station_list";
7         throw error;
8     }
9     else if (station_list.find(dst) == station_list.end()) {
10        string error = "Error: " + dst + "is not in the station_list";
11        throw error;
12    }
13
14    vector<Station*> path;
15    try {
16        find_shortest_path(path, src, dst, line, 0);
17    }
18    catch (string error) {
19        std::cout << error << std::endl;
20    }
21    print_shortest_path();
22}
23 // dir_list 중 가장 짧은 시간을 가진 경로를 출력
24 void Subway::print_shortest_path()
25 {
26     set_shortest_path();
27
28     std::cout << "Shortest path : " << endl;
29     print_path(min_path);
30
31     int minute = min_time / 60;
32     int seconds = min_time % 60;
33     std::cout << "Time : " << minute << " min " << seconds << " sec"
34     << " (" << min_time << " seconds)" << endl;
35 }
36 void Subway::set_shortest_path()
37 {
38     int min_time = 10000;
39     int min_index = 0;
40     for (int i = 0; i < dir_list.size(); i++) {
41         if (dir_list[i].second < min_time) {
42             min_time = dir_list[i].second;
43             min_index = i;
44         }
45     }
46 }
```

```

47 // min_path에 최소 경로 저장
48 min_path = dir_list[min_index].first;
49 this->min_time = dir_list[min_index].second;
50 }

```

코드 5: main:find_shortest_path

코드 5에서 find_shortest_path는 public하게 접근할 수 있지만, 실제 다익스트라 알고리즘을 사용하여 최단 경로를 찾는 동명의 함수는 private하게 숨겨져 있다. 이 이유는 후에 ‘class 설계 내용 및 이유’ 파트에서 다룬다.

public find_shortest_path 함수에서 먼저 src, dst가 station_list에 존재하는지 확인하고, 빈 Station 벡터 리스트 path를 생성한다. 이 path는 deep copy로 private find_shortest_path로 전달된다. 이 path에 경로가 저장된다.

private find_shortest_path에서 list_dir에 path 정보를 저장할 것이다. 이때 주의할 점은, private find_shortest_path는 src == dst인 모든 path를 저장할 수도 있다는 것이다. 이때 list_dir 중 가장 짧은 시간을 갖고 있는 path가 최단 경로이다.

print_shortest_path() 함수는 list_dir 중 최단 경로를 출력하는 함수이다. set_shortest_path 함수를 통해 list_dir 중에서 최단 경로와 최소 시간을 각각 min_path와 min_time에 저장하고, 이를 출력한다.

```

1 // station_dir : 현재까지 지나온 경로
2 // current : 현재 위치
3 // dst : 목적지(destination)
4 // current_line : 현재 경유중인 노선 (line)
5 // time : 현재까지 걸린 시간 (기본은 0)
6 void Subway::find_shortest_path(vector<Station*> station_dir,
    string current, string dst, int current_line, int time = 0)
{
    // If the time that comes in is smaller than the max_time of the
    // current, change the max_time of the current station to time.
    if (time < station_list[current]->max_time) {
        station_list[current]->max_time = time;
        // add current station to station_dir
        station_dir.push_back(station_list[current]);
    }
    else {
        // 새로 들어온 time이 current의 max_time보다 크거나 같은 경우, 더 이상
        // 진행하지 않는다.
        return;
    }

    // 만약 current가 dst와 같은 경우, station_dir와 time을 dir_list에
    // 추가한다.
    if (current == dst) {
        dir_list.push_back(make_pair(station_dir, time));
        return;
    }

    for (int i = 0; i < station_list[current]->connections.size(); i++)
    {
        Connector* station = station_list[current]->connections[i];
        // transfer time : basically 0, but when transferring, it takes
        // 30 seconds
        // when the line of current and next station is different, it
        // is a transfer station
        int transfer_time = 0;

```

```

30     if (current_line != station->line) {
31         transfer_time = 30;
32     }
33
34     string next_station = station->dst->name;
35     int next_line = station->line;
36     int totalTime = time + station->distance + transfer_time;
37
38     // 현재 위치에서 갈 수 있는 모든 station에 대해
39     // 재귀적으로 find_shortest_path를 호출한다.
40     // we have to pass station_dir by deep copy
41     find_shortest_path(station_dir, next_station, dst, next_line,
42                         totalTime);
41 }
42 }
```

코드 6: 다익스트라 알고리즘 구현

코드 6의 함수가 다익스트라 알고리즘을 구성하는 단 하나의 함수이다. `find_shortest_path` 함수는 매개 변수로 다음을 전달받는다:

- `station_dir` : 현재까지 지나온 경로
- `current` : 현재 역 이름⁷
- `dst` : 목적지
- `current_line` : 현재 역의 호선
- `time` : 현재까지 걸린 총 시간

먼저 들어온 총 시간 `time`에 대해 현재 `max_time`보다 작다면, 더 짧은 시간으로 `max_time`을 교체하며, `station_dir`에 현재 역을 추가한다. 이로써 다익스트라의 핵심 논리 중 하나인 더 짧은 정점에 대해 경로를 구성하고, 더 긴 경로는 버린다는 논리를 가질 수 있다. 한편 `max_time`의 초기값은 10000으로 설정되어 있기에 처음 들어온 `time`에 대해서는 무조건 바뀌게 될 것이다.

한편 새로 들어온 `time`이 기존 `max_time`보다 크다면, 그것은 이미 최소 시간을 가지는 최단 경로가 아니란 뜻이다. 따라서 그 경로는 죽은 경로(가망이 없는 노드)가 되며, 해당 경로에 대한 탐색은 더 이상 진행하지 않고 버린다.

만약 경로 탐색을 하다 `current`와 `dst`가 같은 경우, 하나의 경로를 찾은 것이다. 전체 경로를 담는 `dir_list`에 지금까지 탐색한 경로 `station_dir`와 총 시간을 추가한다. 그러나 이 `station_dir`가 최소 시간 경로라는 의미는 아니다. 탐색을 더 하다 보면 더 짧은 경로를 찾을 수 있다.

```

1 for (int i = 0; i < station_list[current]->connections.size(); i++)
2 {
3     Connector* station = station_list[current]->connections[i];
4     // transfer time : basically 0, but when transferring, it takes
5     // 30 seconds
6     // when the line of current and next station is different, it
7     // is a transfer station
8     int transfer_time = 0;
9     if (current_line != station->line) {
10         transfer_time = 30;
11     }
12
13     string next_station = station->dst->name;
14     int next_line = station->line;
```

⁷출발역인 `src`가 아님에 주의한다.

```

12     int totalTime = time + station->distance + transfer_time;
13
14     // 현재 위치에서 갈 수 있는 모든 station에 대해
15     // 재귀적으로 find_shortest_path를 호출한다.
16     // we have to pass station_dir by deep copy
17     find_shortest_path(station_dir, next_station, dst, next_line,
18     totalTime);
19 }
```

코드 7: 코드 6 중 핵심 로직 확대

코드 7은 코드 6의 재귀호출문을 확대한 것이다. 이 부분이 다익스트라 알고리즘의 핵심이 된다. 현재 정점 current station의 모든 connection에 대해 다음 탐색을 수행한다.

1. 환승 시간에 대한 정보 transfer_time을 추가한다. transfer_time은 기본적으로 0이지만, 환승역일 시 30이 된다.
2. 전체 시간 totalTime = time(인자로 전달된 지금까지의 시간) + station->distance(다음 역까지의 거리) + transfer_time(환승 시간)이다.
3. station은 connector, 즉 다음 역이다. 다음 역에 대해 find_shortest_path 함수를 재귀 호출한다. 이때 인자로 현재까지의 경로 정보 station_dir와 다음 역 정보, 목적지, 다음 호선, 전체 시간을 전달한다. 주의할 점은 계속해서 강조하듯이 station_dir은 deep copy로 전달되어야 한다. 만약 참조 형식과 같이 shallow copy로 전달될 경우 경로 정보가 꼬일 수 있다.

한편 구현 과정 특성상 find_shortest_path가 재귀적으로 탐색하면서, 두 개 이상의 src와 dst를 잇는 경로를 발견할 수 있다. 때문에 발견된 모든 경로를 dir_list에 저장하여, 위에서 설명한 과정을 통해 최단 경로를 확정하는 별도의 과정이 필요하다.

4 (b) 중간역 찾기

여기서 중간역이란, 시작역에서 중간역까지 걸리는 시간, 도착역에서 중간역까지 걸리는 시간의 차이가 최소가 되게 하는 역을 의미한다. 이때 중간역은 모든 역에 대해 조사할 필요는 없다. 이미 다익스트라 알고리즘을 통해 min_path를 찾았고, 최단 경로는 여러 개의 최단 경로로 구성되어 있다⁸는 가정에 따라 중간역 M에 대해 식 (1)이 성립한다.

$$M \in \text{min_path} \text{ s.t. } \min |T_{src-M} - T_{dst-M}| \quad (1)$$

따라서 min_path의 역에 대해서만 각자 걸리는 시간의 차이가 최소가 되는 역을 찾으면 된다.

```

1 // min_path의 역 중에서, src와 dst까지의 거리의 차가 가장 짧은, 중간 지점의 역을
2     // 찾아서 출력
3 void Subway::find_middle_station(string src, string dst)
4 {
5     if (min_path.size() == 0) {
6         std::cout << "ERROR : There is no path between " << src << "
7             and " << dst << endl;
8         return;
9     }
10
11     Station* src_station = station_list[src];
12     Station* dst_station = station_list[dst];
13     int min_distance = 10000;
14     Station* min_station = min_path[0];
```

⁸A에서 D까지 최단 경로가 A→B→C→D라면, 경로 A→B, B→C, A→B→C, B→C→D도 모두 최단 경로, 위에서 선술한 내용 참고

```

13
14     int time_src = 0;
15     int time_dst = 0;
16
17     for (int i = 0; i < min_path.size(); i++) {
18         if (min_path[i] == src_station || min_path[i] == dst_station) {
19             continue;
20         }
21
22         // find stations that has the smallest absolute value of the
23         // difference
24         int distance = abs(abs(min_path[i]->max_time - src_station->
25             max_time)
26             - abs(min_path[i]->max_time - dst_station->max_time));
27         if (distance < min_distance) {
28             min_distance = distance;
29             min_station = min_path[i];
30             time_src = abs(min_path[i]->max_time - src_station->max_time)
31             ;
32             time_dst = abs(min_path[i]->max_time - dst_station->max_time)
33             ;
34         }
35     }
36
37     std::cout << "Middle station : " << min_station->name << endl;
38     std::cout << src << " - " << min_station->name << " time : "
39     << time_src / 60 << " min " << time_src % 60 << " sec ("
40     << time_src << " seconds)" << endl;
41     std::cout << dst << " - " << min_station->name << " time : "
42     << time_dst / 60 << " min " << time_dst % 60 << " sec ("
43     << time_dst << " seconds)" << endl;
44     std::cout << "The difference in time taken : " << abs(time_src -
45         time_dst) << " seconds" << endl;
46 }
```

코드 8: 중간역 찾기

코드 8은 위에서 설명한 과정을 그대로 구현한 것이다. 최소 시간 차이를 나타내는 `min_distance`⁹보다 짧은 시간이 발견되면 그 값으로 갱신한다¹⁰.

한편 중간역은 적어도 출발역이나 종착역에 있지는 않을 것이므로, 이에 대한 비교는 수행하지 않는다.

5 클래스 설계 내용 및 이유

```

1 class Station {
2 public:
3     string name;
4     vector<Connector*> connections;
5     int max_time;
```

⁹ 계속해서 강조하지만, 이 문제에서 시간과 거리는 사실상 같은 의미로 사용되었다.

¹⁰ 초기값은 10000으로 설정하여 처음 한 번은 무조건 갱신되게끔 함

```

6   Station(string name);
7   friend ostream& operator<<(ostream& os, const Station& station);
8   void printStationInfo();
9   ~Station();
10 };
11
12 class Connector {
13 public:
14     Station* dst;
15     int line;
16     int distance;
17     Connector(Station* dst, int line, int distance);
18     void printConnectorInfo();
19 };
20
21 class Subway {
22 private:
23     map<string, Station*> station_list;
24     // vector<Station*> 의 벡터 리스트와 int 시간 정보의 페어
25     vector< pair<vector<Station*>, int> > dir_list;
26     // 최소 경로와 시간
27     vector<Station*> min_path;
28     int min_time;
29
30     void set_shortest_path();
31     void find_shortest_path(vector<Station*> station_dir, string
32         current, string dst, int current_line, int time);
33     void print_path(vector<Station*> path);
34     void insert_station(string name);
35     void add_connection(string src, string dst, int line1, int line2)
36         ;
37 public:
38     void make_station(string src, int line1, string dst, int line2);
39     void print_shortest_path();
40     void find_shortest_path(const string src, const string dst, const
41         int line);
42
43     // src와 dst 사이에 걸리는 시간의 차이가 최소가 되는 middle station을 찾는
44     // 함수
45     void find_middle_station(string src, string dst);
46     void print_station_info(string name);
47     void printAllStations();
48     ~Subway();
49 };

```

코드 9: subway.h 클래스 설계 부분

프로그램은 사용자에게 간단한 인터페이스를 제공하는 한편, 접근하기에 위험할 수 있는 정보들을 차단하는 방식으로 설계되었다.

먼저 Station과 Connector은 사실상 구조체 struct처럼 사용한다. 각각의 정보를 출력하는 print-ConnectorInfo 함수와 printStationInfo가 존재한다.

사용자가 주로 사용하는 클래스는 Subway 클래스이다. station_list, dir_list, min_path, min_time 은 민감한 내부 정보이므로 보호한다. 사용자는 이 정보에 오직 public 함수를 통해서만 간접적으로 접근할 수 있다. 정보를 가지고 접근을 제한하는 이유는 객체 지향 프로그래밍(OOP)의 기본 설계

원칙이므로 추가적인 설명은 생략한다.

한편 set_shortest_path, find_shortest_path는 public한 find_shortest_path에서 내부적으로 사용한다. 일반 사용자는 접근할 일이 없는 함수들이고, private한 변수들에 접근하므로 private로 분류하였다. insert_station, add_connection 함수 역시 public한 make_station 함수에서 내부적으로 사용하는 함수들이다. 위와 같은 이유로 private로 분류하였다. print_path 함수는 경로를 전달하면 그 경로를 출력하는 함수이다. 일반 사용자는 dir_list에 접근할 수 없기에 private로 분류하였다. print_station_info, printAllStations 함수는 역 정보를 출력하는 함수로, 내부적인 테스트 용도로 사용되었다¹¹.

한편 기타 사항으로, 파괴자 함수 ~Subway, ~Station 함수는 코드 10에 나와 있다. Station의 destructor는 자신이 해제될 때 자신과 연결된 모든 connection 객체들을 해제한다. Subway의 destructor는 station_list의 모든 station 객체들을 해제한다.

```
1 Subway::~Subway()
2 {
3     // delete all stations and Connectors
4     for (map<string, Station*>::iterator it = station_list.begin();
5          it != station_list.end(); it++) {
6         delete it->second;
7     }
8
9 Station::~Station()
10 {
11    for (int i = 0; i < connections.size(); i++) {
12        delete connections[i];
13    }
14 }
```

코드 10: destructor

6 결과값 분석

그림 2은 여러 input에 대한 출력 결과가 담겨 있다. 먼저 input1.txt에 대한 출력 결과를 보자. input1.txt는 다음과 같이 구성되어 있다.

```
3 Anguk
2 Chungjeongno
```

출력(그림 2a 참고):

```
Shortest path :
Anguk -> Jongno_3(sam)-ga -> Jonggak -> City_Hall -> Chungjeongno
Time : 5 min 0 sec (300 seconds)
```

```
Middle station : Jonggak
Anguk - Jonggak time : 2 min 30 sec (150 seconds)
Chungjeongno - Jonggak time : 2 min 30 sec (150 seconds)
The difference in time taken : 0 seconds
```

그림 2a에서도 볼 수 있듯이 Shortest path로 최단 경로가 잘 출력되어 있고, 소모 시간 역시 5분 0초(300초)로 잘 나온 것을 알 수 있다. 중간역은 Jonggak으로, Anguk에서 Jonggak까지 2분 30초, Chungjeongno에서 Jonggak까지 2분 30초가 걸려서 걸린 시간의 차가 0초로 가장 짧은 중간역임을 알 수 있다.

한편 다른 input을 더 만들어 테스트해보자. input2.txt는 다음과 같이 구성되어 있다.

¹¹최종 과제 제출 시 없어도 되는, 온전히 내부적 테스트 용도의 함수여서 지울까 말까 고민하다가, 클래스는 오직 문제 풀이를 위한 함수 이외의 기능을 포함하여 그 자체로 완성도를 갖춰야 한다는 생각, 내부 테스트 용도의 함수 역시 문제 풀이 과정 중 일부라는 점을 근거로 남겨두기로 하였다. 마찬가지 이유로 Station과 Connector의 printConnectorInfo 함수와 printStationInfo 함수 역시 남겨두었다.

```
[C211123@linux2 hw10]$ hw10 stations1.txt input1.txt
Shortest path :
Anguk -> Jongno_3(sam)-ga -> Jonggak -> City_Hall -> Chungjeongno
Time : 5 min 0 sec (300 seconds)

Middle station : Jonggak
Anguk - Jonggak time : 2 min 30 sec (150 seconds)
Chungjeongno - Jonggak time : 2 min 30 sec (150 seconds)
The difference in time taken : 0 seconds
```

(a) input1.txt 결과

```
[C211123@linux2 hw10]$ hw10 stations1.txt input2.txt
Shortest path :
Jongno_5(o)-ga -> Jongno_3(sam)-ga -> Euljiro_3(sam)-ga -> Chungmuro -> Dongguk_Univ. ->
Yaksu -> Geumho -> Oksu -> Apgujeong -> Sinsa -> Jamwon ->
Express_Bus_Terminal -> Seoul_Nat'l_Univ._of_Education -> Nambu_Bus_Terminal
Time : 13 min 30 sec (810 seconds)

Middle station : Geumho
Jongno_5(o)-ga - Geumho time : 6 min 30 sec (390 seconds)
Nambu_Bus_Terminal - Geumho time : 7 min 0 sec (420 seconds)
The difference in time taken : 30 seconds
```

(b) input2.txt 결과

```
[C211123@linux2 hw10]$ hw10 stations1.txt input3.txt
Shortest path :
Guro -> Sindorim -> Mullae -> Yeongdeungpo-gu_Office -> Dangsan ->
Hapjeong -> Hongik_Univ.
Time : 6 min 30 sec (390 seconds)

Middle station : Yeongdeungpo-gu_Office
Guro - Yeongdeungpo-gu_Office time : 3 min 30 sec (210 seconds)
Hongik_Univ. - Yeongdeungpo-gu_Office time : 3 min 0 sec (180 seconds)
The difference in time taken : 30 seconds
```

(c) input3.txt 결과

그림 2: 출력 결과

1 Jongno_5(o)-ga
3 Nambu_Bus_Terminal

출력(그림 2b 참고):

Shortest path :

Jongno_5(o)-ga -> Jongno_3(sam)-ga -> Euljiro_3(sam)-ga -> Chungmuro -> Dongguk_Univ. -> Yaksu -> Geumho -> Oksu -> Apgujeong -> Sinsa -> Jamwon -> Express_Bus_Terminal -> Seoul_Nat'l_Univ._of_Education -> Nambu_Bus_Terminal
Time : 13 min 30 sec (810 seconds)

Middle station : Geumho

Jongno_5(o)-ga - Geumho time : 6 min 30 sec (390 seconds)

Nambu_Bus_Terminal - Geumho time : 7 min 0 sec (420 seconds)

The difference in time taken : 30 seconds

그림 2b에서도 볼 수 있듯이 Jongno_5(o)-ga에서 Nambu_Bus_Terminal까지의 최단 경로와, 중간 역인 Geumho가 잘 출력된 것을 확인할 수 있다.

input3.txt는 다음과 같이 구성되어 있다.

1 Guro
2 Hongik_Univ.

출력(그림 2c 참고):

[C211123@linux2 hw10]\$ hw10 stations1.txt input3.txt

Shortest path :

Guro -> Sindorim -> Mullae -> Yeongdeungpo-gu_Office -> Dangsan -> Hapjeong -> Hongik_Univ.

Time : 6 min 30 sec (390 seconds)

Middle station : Yeongdeungpo-gu_Office

Guro - Yeongdeungpo-gu_Office time : 3 min 30 sec (210 seconds)

Hongik_Univ. - Yeongdeungpo-gu_Office time : 3 min 0 sec (180 seconds)

The difference in time taken : 30 seconds

그림 2b에서도 볼 수 있듯이 Guro에서 Hongik_Univ.까지의 최단 경로와, 중간역인 Yeongdeungpo-gu_Office 잘 출력된 것을 확인할 수 있다¹².

다익스트라 알고리즘은 전체 경로를 조사하는 게 아닌, 탐욕법(Greedy Method)으로 최단 경로를 탐색하기에 구한 경로가 반드시 수학적인 최단 경로가 됨을 보장하지는 않는다. 그러나 최적의 시간 복잡도 내에 유의미한 최단 경로에 근접함은 보장할 수 있다.

7 구현 시 어려웠던 점

먼저 그래프의 구현을 어떻게 해야 할지 가장 많이 고민하였다. 제일 처음에 생각했던 것은 양방향 연결 리스트로 포인터를 사용해 구현하는 것이었다. 그러나 본격적으로 구현을 시작하자마자, 복잡한 포인터를 다루는데 있어 이 방법은 다소 어려울 것이라 생각하여 포기하였다.

Connector 클래스(구조체)를 사용해 연결하는 것은 본인의 고유한 생각은 아니었다. Dijkstra 알고리즘에 대해 공부하던 중 파이썬을 이용해 구현한 것을 보았는데, 그 코드에선 현재 Vertex에 연결되어, 갈 수 있는 유향 Vertex를 리스트 형태로 구현하였다. 파이썬은 리스트의 사용에 있어 자유롭지만, C++는 자유로운 리스트 사용에 있어 다소 제약을 받는다. 하지만 곧바로 STL의 벡터를 사용하면 리스트를 대체할 수 있음을 생각했고, 곧 Connector를 이용하는 방식을 선택했다. 이 방식을 이용할 경우 Connector를 통해 노선 정보와 가중치 등의 정보를 추가하기 편하다는 장점이 있었다.

¹²여기서는 실제 시간이 맞는지 계산하기 쉬운데, 역을 6번 이동했므로 360초에다가, 신도림에서 1호선→2호선으로 환승했으므로 환승 시간 30초를 더해 390초가 잘 출력되었음을 알 수 있다.

단 가중치의 경우 생각보다 문제는 있었는데, 환승역에서 환승할 경우 그 30초의 추가 가중치가 잘 반영되지 않는다는 점이었다. 원래 계획은 환승역에서 환승하는 것 역시 하나의 30초 가중치 간선으로 간주할 생각이었다. 즉 1 Shindorim \leftrightarrow 2 Shindorim 역시 30초의 하나의 간선인 것으로 간주하였다¹³. 그래서 걸린 시간을 전부 다 합쳐서 dijkstra 알고리즘 함수에서 인자로 전달하는 방식을 택했다.

한편 Connector를 통한 연결이 초기 의도대로 훌려가진 않았기에, 우선순위 큐를 이용한 다익스트라 알고리즘 계획도 일찍이 무산되었다. 우선순위 큐를 이용하려면 비교 함수 Compare()을 만들어야 하는데, 모든 역의 가중치가 60초로 똑같고, 30초의 환승 간선이 제대로 작동하지 않았기에 우선순위 큐를 이용하여 다음 탐색 노드를 선택할 이유가 사라졌다.

지금 생각해보면, 환승역끼리의 가중치를 30초를 더해 있는 것이 아니라, 만약 호선이 바뀌는 역간 이동의 경우 이동 시간을 60초가 아닌 90초로 저장했다면 우선순위 큐를 이용할 수 있었을지도 모르겠다. 예를 들어 1 Guro \rightarrow 1 Shindorim \rightarrow 2 Mullae의 경우 1 Shindorim \rightarrow 2 Mullae의 이동 시간을 60초가 아닌 90초로 저장하는 것이다¹⁴. 그러나 이 경우 2 Shindorim \rightarrow 2 Mullae의 경우를 어떻게 구성해야 할지¹⁵에 대한 고민이 남아 있어 다소 복잡한 구현 난이도가 있을 것으로 예상된다.

그래프 구성을 결정하고 나자 다익스트라 알고리즘의 구현은 생각보다 어렵지는 않았다. 다만 이를 코드로 구현하는 것은 다소 어려웠는데, 특히 map, vector, Station과 Connector 구조체에 이들이 포인터와 실제 value들로 섞여 있어 섞여 있어 많이 헷갈렸다. 다만 Modern C++에서는 auto¹⁶, foreach¹⁷ 등의 간편하고 강력한 기능을 제공하여, 이를 이용해 구현하였다. 이후 리눅스로 옮길 때 auto를 실제 타입으로 바꾸기만 하였다¹⁸.

한편 dir_list 역시 원래의 의도는 아니었는데, 다익스트라 알고리즘을 우선순위 큐 등의 정상적인 방식으로 구현하지 않고 조금 이상하게 구현하여, 실제로는 최단 경로가 아닌 경로와 최단 경로가 같이 출력되었다. 즉 원래는 탐색 과정에서 src == dst이면 무조건 출력하게끔 구현했는데, 그러자 두 개의 경로가 같이 출력되었다. 탐색 과정 중에 src==dst를 만나면 무조건 출력하고, 재귀 함수이므로 계속해서 탐색을 이어나가기 때문이다. 보진 못 했지만, 아마 stations의 구성에 따라 경로가 3개 이상 출력되는 경우도 있을 것이다.

그래서 dir_list에 모든 폐순환 경로를 시간과 함께 페어로 저장한 뒤, 이 중 가장 짧은 시간을 갖는 최소 시간 경로를 출력하도록 구현했다. 한편 이렇게 구현하자 오히려 b의 문제는 구현하기 쉬워졌다. b는 이미 저장된 min_path만 가져다 쓰면 되기 때문이다.

¹³ 환승역이 어떤 경우는 처음에 의도대로 잘 구현되었지만, 어떤 경우는 의도한 대로 구현되지 않았다.

¹⁴ 처음의 의도는 1 Guro \rightarrow 1 Shindorim \rightarrow 2 Shindorim \rightarrow 2 Mullae로 경로를 구성해, 1 Shindorim \rightarrow 2 Shindorim의 이동 시간(환승 시간)을 30초로 만드는 것이었다.

¹⁵ 현재로써는 1 Shindorim과 2 Shindorim에 대한 두 개의 Connector를 구성하는 방식이 떠오르는데, 이 경우 1 Shindorim에 대한 Station과 2 Shindorim에 대한 Station, 그리고 둘을 잇는 1 Shindorim \leftrightarrow 2 Shindorim의 Connector를 구성해야 하는 등 생각해야 할 것들이 많아질 것이다.

¹⁶ 자동 타입 추론

¹⁷ 범위 기반 반복

¹⁸ Visual Studio에서는 auto로 선언된 타입의 실제 타입을 알려주고, foreach문 역시 이를 기준이 for문으로 바꾸거나 iterator로 쉽게 변환할 수 있도록 도와주는 몇 가지 강력한 기능을 제공한다.