2단계 지하철 길찾기 프로그램
결과 보고서

제출일	2021.11.4	전 공	컴퓨터공학과
과 목	어드벤처 디자인	학 번	20200679
담당교수	윤현주 교수님	이 름	안예진

kit 국립금오공과대학교
Kumoh National Institute of Technology

1. 문제

▸ 문제 내용

- 지하철 노선도 정보를 이용해 두 지점 간 모든 경로를 찾고, 이동한 역에 대한 정보를 함께 출력하여 원하는 경로를 선택할 수 있도록 하는 길찾기 프로그램
- 대구 지하철 노선도를 기준으로 길찾기 안내 노선 설정

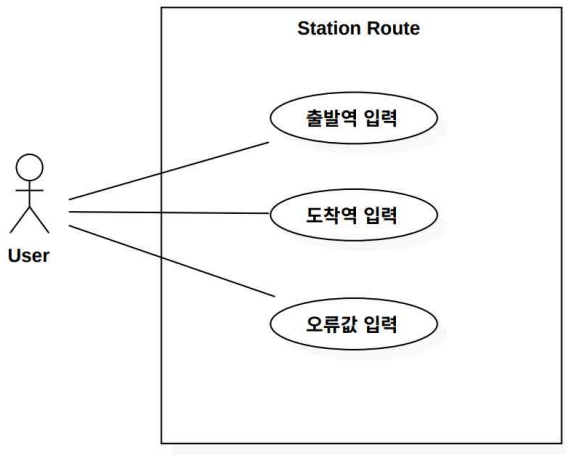
▸ 주의사항 및 고려할 점

- 대구 지하철 노선도는 웹에서 정보를 구해 파일으로 기록
- 출력물은 출발 역과 도착 역, 환승 역의 이름과 호선 정보 기재
- 이동 중 거치는 역의 개수와 이동 시간 출력
- 각 역마다 걸리는 시간은 2분으로 동일, 환승 역은 8분 추가
- 환승역에 해당하나 호선을 바꾸지 않는 경우에는 정차역으로 출력하지 않음
- 사용자의 동작마다 반응 메시지 출력
- 사용자가 잘못된 입력을 주었을 경우 오류 메시지 출력 후 프로그램 종료

▸ 문제 해결 범위

- 경로 검색에 사용할 자료 구조를 설계하여 지하철 노선 정보 파일 입출력이 가능하도록 구현
- 모든 경로 정보를 사용자가 확인할 수 있도록 출력
- text-ui로 프로그램 구현

▸ Use-Case Diagram



<Usecase Diagram>

2. 기능 및 요구 사항 분석

▸ 입출력 정의

- 입력 파일은 각 역의 호선 정보, 그래프 노드 정보, 배열 리스트 인덱스, 역 이름을 순서대로 저장
- 역 정보는 줄바꿈으로 다르게 하며 각 역의 세부 정보는 공백을 기준으로 구분
- 출력 정보는 출발역, 환승역, 도착역 순서대로 기재하며 역의 호선 정보를 포함

▸ 핵심 기능과 그들 간의 관계

- 프로그램은 파일 입력, 지하철 노선 연결 알고리즘, 환승역 정보, 이동 시간, 이동 역의 개수 출력 기능 요구
- 사용자가 출발역과 도착역을 입력한 즉시 지하철 경로 명단 출력

▸ 요구 사항 분석

- 입력한 역이 파일에 기록된 데이터와 부합하도록 함
- 프로그램 시작과 동시에 사용자 입력 주의사항 명시
  (1) 띄어쓰기와 호선은 기재하지 않음
  (2) 정차역 입력 시 '역'을 제외한 역의 이름만 기입
  (3) 출발역과 도착역을 순서에 맞게 한 개씩만 입력

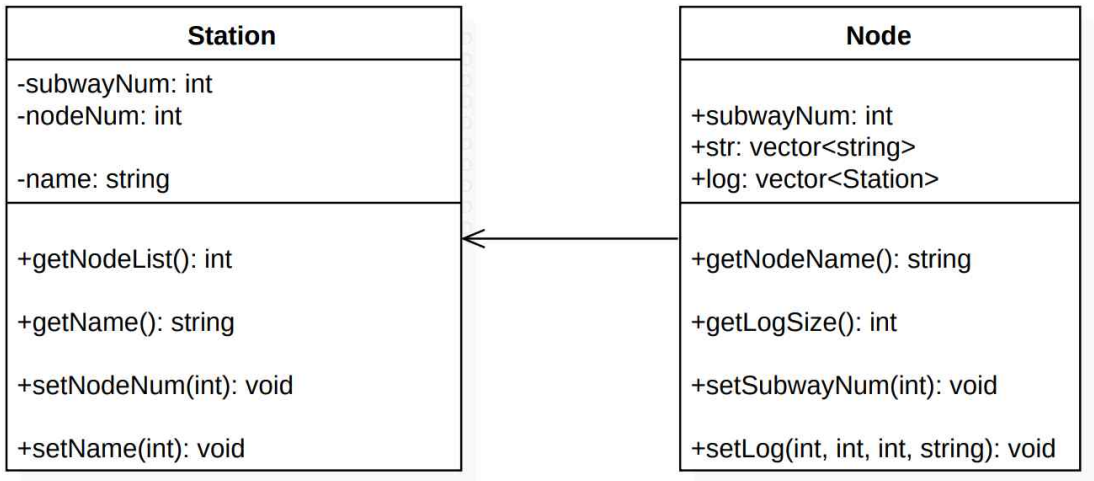
▸ 대체흐름

- 노선도에 등록되지 않은 역을 입력했을 경우 오류 메시지 출력 후 프로그램 종료 (철자, 오타, 문자열과 다른 타입의 변수, 정차역 입력으로 '역' 문자열 포함, 철자 누락 등)

3. 설계

▸ 자료구조 및 클래스 설계

- Class Diagram



- 각 정차역 정보가 저장될 클래스 Station 생성

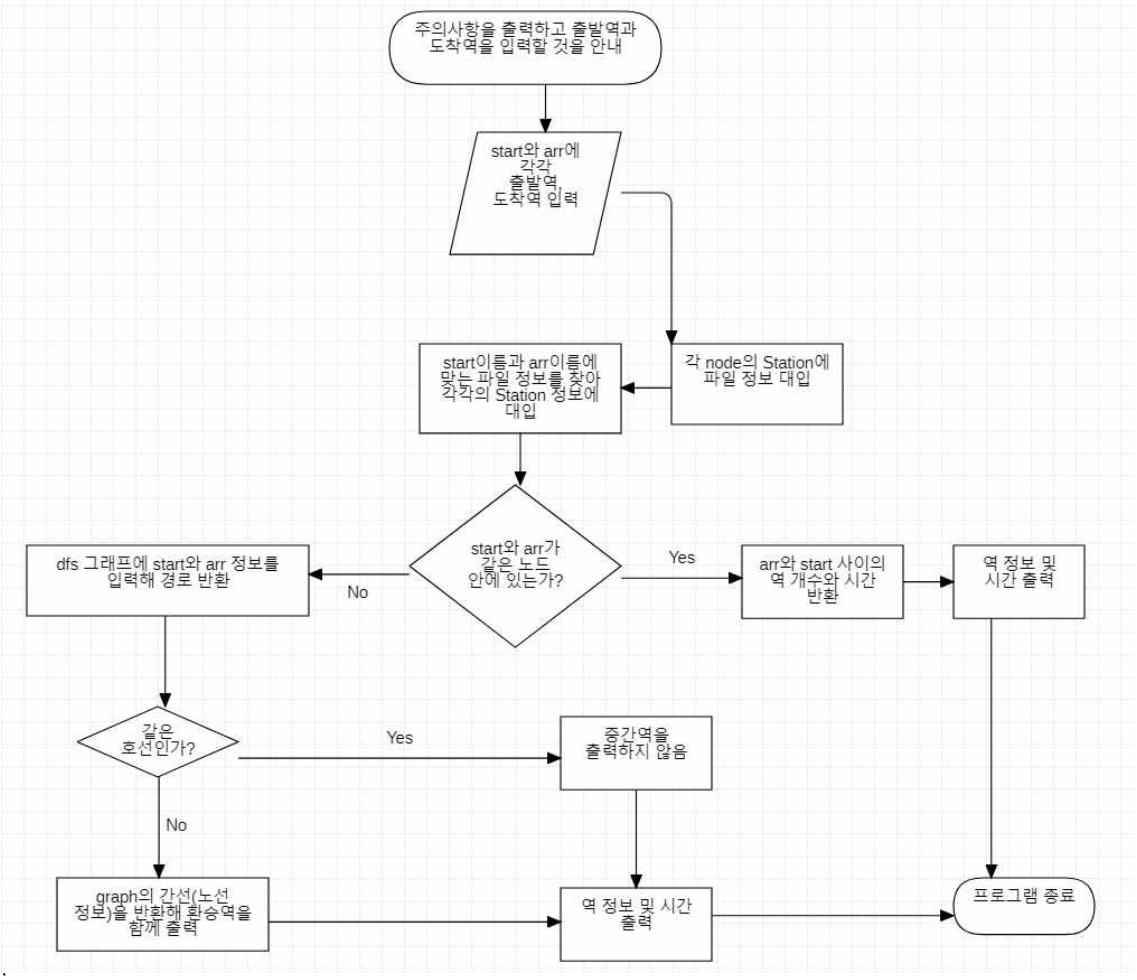
```
class Station
{
	int subwayNum;
	int nodeNum;
	int nodeList;
	string name;
}
```

- 그래프의 정점이 될 Node 클래스 생성

```
struct Node {
	int id;
	int subwayNum;
	vector<string> str;
	vector<Station> log;
}
```

▸ 알고리즘 설계

- Flowchart Diagram



- 그래프 생성 : vector<int>를 사용한 인접리스트 구현

```
vector<pair<int, int>> graph
for 첫 번째 노드에서 마지막 노드까지 순회
graph[기준 노드 번호].push_back(make_mair(인접 노드 번호, 인접 노드와 연결된 호선 번호))
```

- 그래프 순회 : 깊이 우선 탐색(DFS) 알고리즘

```
input : int '기준역 노드 번호', int '도착역 노드 번호'
output : vector<int> 'dfs를 통해 모든 경로 탐색이 이루어진 노드 정보'

vector<int> dfs(int 기준역, int 도착역)
```

```
{
    visited[기준역] <- True // 기준역 방문으로 set
    line.push_back(기준역)

    if(기준역 == 도착역)
        count<- line.size();
    for i <- 0 to count do
        route[i] <- line[i]

    for i <- 0 to graph[기준역].size() do
        temp <- graph[기준역][i].인접노드
        dfs(temp, 도착역)
        visited[temp] <- false
}
```

- 파일 입력

```
input : 입력할 파일, 공백으로 나눌 문자열, 출발역과 도착역의 이름
output : 각 노드마다 저장된 Station 정보, 출발역과 도착역의 나머지 지하철 정보

while( getline(입력파일, 문자열) )
    istringstream( buffer(문자열) )
    buffer -> 호선번호 -> 노드 번호 -> 배열 인덱스 -> 지하철 이름

    if(노드 번호 = count)
        node(count) <- setLog(호선번호, 노드번호, 인덱스, 지하철이름)
    if(호선 번호 == 0)
        count++
    if(지하철이름 == 출발역.이름)
        출발역.setSubNum(호선번호)
        출발역.setNodeNum(노드번호)
        출발역.setNodeList(인덱스)
    if(지하철이름 == 도착역.이름)
        도착역.setSubNum(호선번호)
        도착역.setNodeNum(노드번호)
        도착역.setNodeList(인덱스)
```

- 결과 출력 : 출발역과 도착역이 같은 node에 포함

```
input : 출발역의 노드 번호, 도착역의 노드 번호
output : X

if (출발역.getNodeNum == 도착역.getNodeNum)
    count <- abs(출발역.getList – 도착역.getList)
    print 출발역 정보 && 도착역 정보
```

- 출발역과 도착역의 노드 번호가 같을 경우, dfs알고리즘을 수행하지 않고 배열 인덱스를 이용해 두 역 간의 거리와 시간 계산

- 결과 출력 : 출발역과 도착역이 다른 node에 포함

```
input : dfs를 수행한 벡터 route
output : X

for i = 0 to route.size do
    if ( route[i] == 출발역.getNodeNum().getId )     //출발지점
        total += abs(출발역.getNodeList – 출발역.getLogSize())

    if(route[i] == 도착역.getId)    //도착지점
        total += ( node(도착역).size – 도착역.getNodeList )
        print 출발역 정보

        for j = 0 to 이름저장vector.size() do
            if( graph[route[i-1]][j].first == route[i] )
                tmp <- j;

        for j = 0 to 이름저장vector.size() do
            if(graph[route[j]][tmp].second != route[j]][tmp].second)
                print 환승역 정보
                trans++

        print 도착역 정보
        vector<string>().swap(이름저장vector)
        total <- 0
        trans <- 0

    total += node[route[i]].getLogSize()
    if( (route[i-1] == 7 and route[i] == 8 ) || (route[i-1] == 8 and route[i] == 8 ))
        total--
    route_name.push_back(node[route[i]]).getNodeName
```

- 중간에 노드를 거칠 경우, 우선 환승역이 모두 존재한다고 가정
- graph의 직전 노드와 다음 노드를 통해 간선 정보를 구함
- 만약 현재 간선과 다음 간선이 일치한다면 중간 노드를 출력하지 않음
- dfs가 모든 경로를 같은 벡터에 저장하므로 한 경로를 도착역까지 모두 출력하고 나면 이전에 기록된 이름 저장 벡터 및 total과 trans를 모두 초기화
- if문에 route[i]가 걸리지 않을 경우, 환승역이라고 생각하고 total에 node에 저장된 배열 수만큼을 저장

4. 구현

▸ 자료구조 또는 클래스 코드

- Station 클래스

```
class Station
{
private:
	int subwayNum;
	int nodeNum;
	int nodeList;
	string name;

public:
	Station(int _subwayNum, int _nodeNum, int _nodeList, string _name) {
		subwayNum = _subwayNum;
		nodeNum = _nodeNum;
		nodeList = _nodeList;
		name = _name;
	}
	Station() {
		subwayNum = -1;
		nodeNum = -1;
		nodeList = -1;
		name = "";
	}
	int getNodeNum() {
		return nodeNum;
	}
	int getNodeList() {
		return nodeList;
	}
	int getSubwayNum() {
		return subwayNum;
	}
	string getName() {
		return name;
	}
	void setSubwayNum(int _subwayNum){
		subwayNum = _subwayNum;
	}
	void setNodeNum(int _nodeNum) {
```

```
        nodeNum = _nodeNum;
    }
    void setNodeList(int _nodeList) {
        nodeList = _nodeList;
    }
    void setName(string _name) {
        name = _name;
    }
};
```

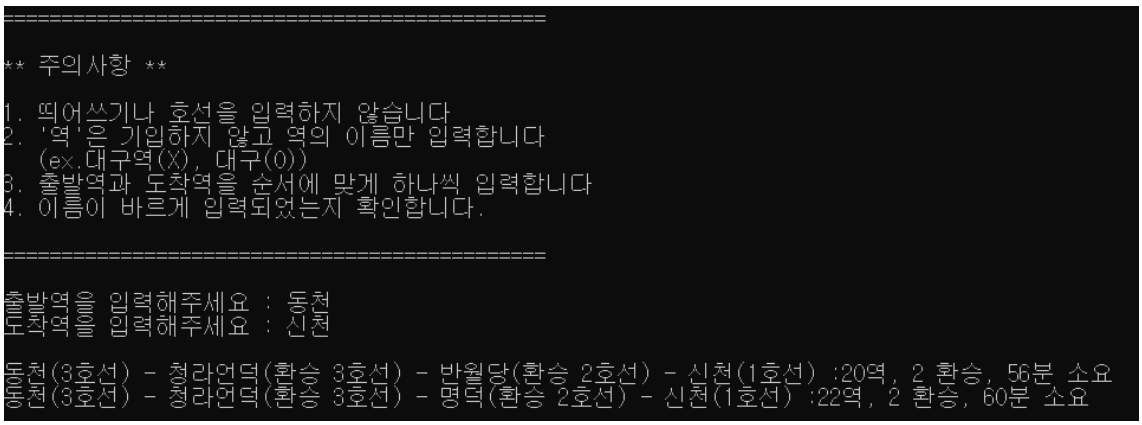
- Node 클래스

```
struct Node {
private:
    int id;
    int subwayNum;
    vector<string> str;
    vector<Station> log;

public:
    Node(int _id, vector<Station> _log) {
        id = _id;
        subwayNum = 0;
        log = _log;
    }
    Node() {
        id = -1;
        int subwayNum = -1;
    }
    ~Node() {};

    int getId() {
        return id;
    }
    string getNodeName() {
        string str = log[0].getSubwayName();
        return str;
    }
    int getNodeNum() {
        int n = 0;
        for (int i = 0; i < log.size(); i++) {
            n = log[i].getNodeNum();
        }
        return n;
```

```
    }
    int getLogSize() {
        return log.size();
    }
    vector<string> getLogName() {
        for (int i = 0; i < log.size(); i++) {
            str.push_back(log[i].getName());
        }
        return str;
    }
    void setSubwayNum(int _subwayNum) {
        subwayNum = _subwayNum;
    }
    void setId(int _id) {
        id = _id;
    }
    void setLog(int _subwayNum, int _nodeNum, int _nodeList, string _name) {
        Station temp(_subwayNum, _nodeNum, _nodeList, _name);
        log.push_back(temp);
    }
};
```

- 그래프의 정점이 될 node 객체 선언을 위해 구현
- 노드 번호, 지하철 호선 번호, vector형의 정차역 리스트 저장
- 그래프 순회와 출발역과 도착역의 정보 출력이 가능하도록 구현
- 각 변수들은 getter/setter 함수를 통해 접근 가능
- setLog는 vector의 특성을 고려해 변수값 입력 대신 push_back()을 이용한 원소 추가

4. 구현



- **핵심 알고리즘의 코드 부분**

  - dfs 알고리즘

```
vector<int> dfs(int x, int goal) {
    visited[x] = true;

    line.push_back(x);

    if (x == goal) {
        int count = line.size();

        for (int i = 0; i < count; i++) {
            route.push_back(line[i]);
        }
    }

    for (int i = 0; i < graph_[x].size(); i++) {
        int y = graph_[x][i].first;
        if (!visited[y]) {
            dfs(y, goal);
            visited[y] = false;
        }
    }
    line.pop_back();
    return route;
}
```

- 노드들의 방문 여부를 알기 위해 전역변수로 bool visited[10] 선언
- 기준역이 방문처리 되었을 경우, vector클래스 객체 line에 각 기준역 저장

- 기준역의 인접 노드 개수만큼 반복문 이용하여 인접노드마다 재귀함수 수행
- 모든 경로를 출력해야 하므로 한번 순회를 하고 나면 인접노드의 방문을 false로 초기화
- dfs탐색을 통해 기준역이 도착역에 도달했다면 저장된 벡터를 반환

- 인접리스트로 노드 연결

```
//노드에 정보입력
for (int i = 0; i < 10; i++) {
    node.push_back(Node(i, log));
    node[i].setSubwayNum(sub[i]);
}
//int 그래프 연결(인접리스트)
graph_[1].push_back(make_pair(7, 2));
graph_[2].push_back(make_pair(7, 3));
graph_[3].push_back(make_pair(8, 1));
graph_[4].push_back(make_pair(8, 2));
graph_[5].push_back(make_pair(9, 3));
graph_[6].push_back(make_pair(9, 1));

graph_[7].push_back(make_pair(1, 2));
graph_[7].push_back(make_pair(2, 3));
graph_[7].push_back(make_pair(8, 2));
graph_[7].push_back(make_pair(9, 3));

graph_[8].push_back(make_pair(3, 1));
graph_[8].push_back(make_pair(4, 2));
graph_[8].push_back(make_pair(7, 2));
graph_[8].push_back(make_pair(9, 1));

graph_[9].push_back(make_pair(5, 3));
graph_[9].push_back(make_pair(6, 1));
graph_[9].push_back(make_pair(7, 3));
graph_[9].push_back(make_pair(8, 1));
```

- int형 노드가 연결된 인접리스트 벡터 graph 구현
- pair을 사용해 first 변수에 인접한 노드의 번호 저장
- pair의 second에는 호선 번호를 간선의 가중치로 저장
- second값인 가중치는 환승역의 호선 정보를 구하기 위해 반환
- 지하철이 양 방향으로 이동할 수 있으므로 양방향 그래프 구현
  (graph[x]의 인접 노드로 y를 저장하고, graph[y]의 인접 노드로 x를 저장)

- 파일 입력

```
if (!fin.is_open()) {
    cout << "경로 파일이 존재하지 않습니다." << endl;
}

int count = 1;

while (getline(fin, line)) {
    istringstream buffer(line);

    buffer >> subwayNum >> nodeNum >> nodeList >> name;

    if (nodeNum == count) {
        node[count].setLog(subwayNum, nodeNum, nodeList, name);
    }

    if (subwayNum == 0)
        count++;

    if (name == arr.getName()) {
        arr.setSubwayNum(subwayNum);
        arr.setNodeNum(nodeNum);
        arr.setNodeList(nodeList);
    }

    if (name == start.getName()) {
        start.setSubwayNum(subwayNum);
        start.setNodeNum(nodeNum);
        start.setNodeList(nodeList);
    }
```

- 파일 입력 시 각 노드를 구분할 수 있도록 count를 설정
- 노드 번호와 count가 일치할 경우 해당 노드에 지하철 정보 저장
- 해당 노드의 log벡터에 setLog로 접근해 push_back을 통한 원소 추가
- 지하철 정보 파일의 노드 번호가 바뀌기 직전 문장에 0을 첨부, 호선이 0이라면 count를 증가시켜 다음 노드에 정보 저장
- 만약 name이 출발역이나 도착역의 이름과 동일하다면 해당 역의 객체에 나머지 Station 정보 저장

- 환승 처리 알고리즘

```
route_ = dfs(node[start.getNodeNum()].getId(), node[arr.getNodeNum()].getId());

for (int i = 0; i < route_.size(); i++)
{
    if (route_[i] == node[start.getNodeNum()].getId())
    {
        total += abs(start.getNodeList() - node[start.getNodeNum()].getLogSize());
        continue;
    }

    if (route_[i] == node[arr.getNodeNum()].getId())        //도착지점
    {
        total += (node[arr.getNodeNum()].getLogSize() - arr.getNodeList());

        cout << start.getName() << "(" << start.getSubwayNum() << "호선) - ";


        for (int j = 0; j < graph_[route_[i]].size() - 1; j++) {
            if (graph_[route_[i]][j].first == route_[i]) {
                tmp = j;
            }

        }
        for (int j = 0; j < route_name.size() -1 ; j++) {
            if (j > route_name.size()) {
                if (graph_[route_[j]][tmp].second != arr.getSubwayNum())
                    cout << route_name[j] << "(환승 " << graph_[route_[j]][tmp].second << "호선) - ";
            }
            if (graph_[route_[j]][tmp].second != graph_[route_[j-1]][tmp].second) {
                cout << route_name[j] << "(환승 " << graph_[route_[j]][tmp].second << "호선) - ";
                trans++;
            }
```

- 정차역의 정보를 배열으로 저장하기 위한 Station 클래스
- 각 변수마다 호선(int), 노드 번호(int), 배열 인덱스(int), 정차역의 이름(string) 저장
- 생성자는 모든 정보가 Station에 입력되었을 때 해당 정보로 객체를 초기화
- Station 객체에 입력값이 없을 경우 int형은 –1, string형은 공백으로 초기화
- 각 변수들은 getter/setter 함수를 통해 접근 가능

5. 결과 분석 및 검토

▸ 실행 결과로부터 도출된 결론

- 입력 파일에 어떠한 정보가 들어가야 간결하고 복잡도가 낮은 프로그램을 구현할 수 있는지 고민해보아야 함
- 인접행렬보다 인접리스트의 시간 및 공간 복잡도가 낮음
- 그래프 구현을 대구 지하철에 맞추지 말고 다른 지하철 노선도를 파일으로 입력했을 때에도 프로그램이 동작할 수 있도록 방안을 모색해보아야 함.

▸ 문제 해결 과정에서 검토했거나 시도할 의미가 있는 기술요소

- vector의 데이터를 클래스로 받아 다양한 조작을 할 수 있음.

▸ 프로그램 구조의 명쾌함, 시간/공간 복잡도, readability 등 코드 평가

- 환승역을 구현할 때 벡터 선언이 과하고 수정하기 복잡함
- dfs알고리즘을 인접리스트로 구현해 시간 및 공간 복잡도가 인접행렬에 비하여 간단함
- 코드를 메인 함수에 통째로 구현해 코드가 간결하지 않음. 각 구현 분야 별로 세분화하여 함수를 나눌 필요가 있음

6. 개발 일지 및 후기

▸ 후기

이번 문제는 내가 지금까지 어떠한 알고리즘을 알고 있고 자료구조를 어떻게 작성해왔는가, 되돌아보는 계기가 되었다. 정보가 담긴 노드를 그래프로 구현하고 dfs 알고리즘을 사용하기까지 일주일이 넘는 시간이 걸렸다. 책이나 강의를 통해 알고리즘이나 행렬, 그래프의 존재를 아는 것은 크게 가치가 없다는 것을 체감했다. 컴퓨터를 배우는 공학도의 입장에서 가장 중요한 것은 알고 있는 지식을 코드로 표현하는 것이기 때문이다. 알고리즘 자체는 어려운 것이 아니었어서, 더더욱 그동안 코딩을 소홀히 한 여파를 실감하게 되었다.

특히, 환승역에 대한 정보를 구현하면서 코딩의 중요성을 내가 알고 있던 알고리즘 지식을 코드로 작성하는 것 이외에도 직관적인 코드를 작성할 수 있도록 부단히 연습을 해보아야 하겠다. 2단계 과제를 구현하는 도중에는 그저 막막하기만 했는데, 보고서 작성을 마치고 나니 이번 학기에서 배운 전공 공부 중 가장 의미있는 시간이었다는 생각이 들었다. 코드 작성 능력과 명쾌한 설계도 작성을 위해 지난 1단계 과제나 2단계 과제 중, 내가 풀어보지 않은 다른 문제들도 구현해보고 싶다.