A* 알고리즘을 활용한 지하철 최단경로 제공 시스템

2019144060 신승우 2021111012 이승연

2020147051 장 건 2021171001 한준희

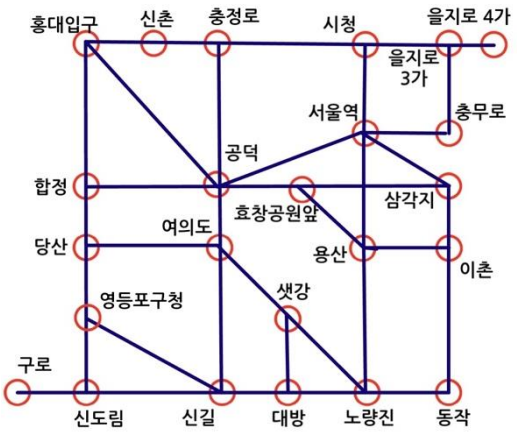
1. 동기 및 목적

'네이버 지도'나 '카카오맵'과 같은 여러 어플리케이션에서 제공하는 '길찾기' 기능은 실생활에서 널리 활용되고 있다. 특히나 사람들은 대중교통 이용에 있어 최단경로에 대한 정보를 이로부터 얻는다. 그렇다면, 그와 같은 시스템을 휴리스틱을 이용한 프롤로그를 통해 실현할 수 있는가? 이러한 맥락에서 본 프로젝트는 지하철 최단경로 제공 시스템을 구현하는 것을 목적으로 한다. 구체적으로, A* 알고리즘을 활용한 프롤로그(PROLOG)를 통하여 지하철 최단경로 제공 시스템을 실현하고자 한다.

2. Knowledge-Base 구축

프롤로그 코드에 내재된 Knowledge-Base 는 '지하철역', '호선', '역간거리', '지하철역의 위도 및 경도'에 관한 정보로 구성하였다.

<그림 1, 출처: 네이버지도>



<그림 2>

2.1 지하철역 선정

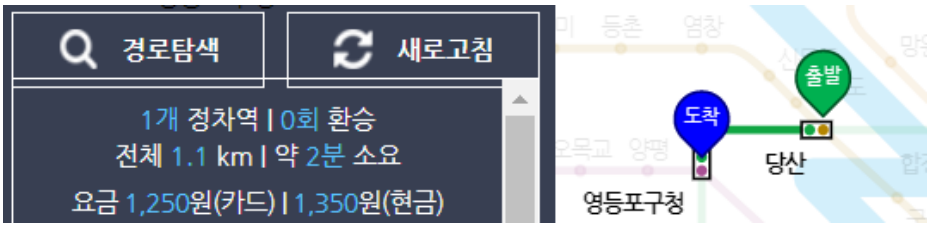
원활한 코드 실행을 위하여 알고리즘에서 활용할 지하철역을 별도로 선정하였다. 1 차적으로, 서울역을 중심으로 여러 호선이 혼재하고 있는 인근

지역을 주된 범위로 설정하였다. 이후 해당 범위 내에서 서울역을 포함한 환승역 23 곳, 추가로 연세대학교와 근접한 신촌역을 선정하였다. 그림으로 보면 위의 <그림 1>과 같다. 선정된 지하철역들의 노선도를 간소화하면 <그림 2>와 같다.

2.2 호선 및 역간거리

서울교통공사에서 제공하는 사이트를 통해 최단거리를 검색하여 24 개 역간 거리를 기록하였다. 또한 각 역과 역 사이의 선로에 해당하는 호선 또한 기입하였다.

예시)



<출처: 서울교통공사, http://www.seoulmetro.co.kr/kr/cyberStation.do>

road(영등포구청, 당산, 1.1). subway(영등포구청, 당산, '2호선').

*두 역 간에 복수의 경로가 있고 역간 거리가 같은 경우, 모든 호선을 병기하였다. (예, 홍대입구-공덕 구간(경의중앙선/공항철도))

**두 역 간에 복수의 경로가 있고 역간 거리 또한 다른 경우, 최단경로 관점이기에 역간 거리가 짧은 노선만 기입함. (예, 공덕-효창공원앞(6 호선)과 공덕-효창공원앞(경의중앙선) 중 '공덕-효창공원앞(6 호선)'만 기입하였다.

2.3 지하철역의 좌표

프롤로그 코드 중, 'hvalue(Station, Destination, Distance)'에서 역 간의 직선거리를 활용할 수 있도록 각 역의 위경도 좌표를 프롤로그 코드에 기입하였다.

예시)

```
coordinates(37.503039, 126.881966, 구로).
coordinates(37.508725, 126.891295, 신도림).
coordinates(37.52497, 126.895951, 영등포구청).
coordinates(37.517122, 126.917169, 신길).
coordinates(37.53438, 126.902281, 당산).
coordinates(37.521624, 126.924191, 여의도).
coordinates(37.517274, 126.928422, 샛강).
```

<출처:https://github.com/henewsuh/subway_crd_line_info/blob/main/지하철역_좌표.csv>

3. 프롤로그 코드

3.1 A* 알고리즘

휴리스틱 기준

A* algorithm 의 evaluation function 은 'f(n) = g(n) + h(n)'이다. g(n) 함수에는 출발점에서 측정 시점 노드까지의 거리로 지하철 역간거리를 사용하였다. h(n)함수에는 측정 시점 포인트에서 목표 노드까지의 거리로 유클리드 거리를 사용하였다.

관련하여, A* 알고리즘에서의 h-value 로 활용할 수 있는 지표의 후보로 유클리드 거리와 맨해튼 거리를 두었다. 그러나 맨해튼 거리는 격자 상태에서의 최단 거리를 의미하기에 본 알고리즘에서 두 지점(두 역) 간 최단 거리를 보장할 수 없고 그로 인하여 A* 알고리즘에 오류를 유발한다. 따라서 맨해튼 거리를 제외하고 직선(최단)거리를 보장하는 유클리드 거리만을 해당 A* 알고리즘에서 활용하였다.

유클리드 거리 계산

유클리드 거리는 각 지하철 역의 위경도 좌표 및 하버사인 공식을 사용하여 거리를 측정하였다. 하버사인 공식은 다음과 같다.

*지구는 완벽한 구형이라는 가정이 있다.

$a = \sin^2(\Delta\varphi / 2) + \cos \varphi1 \times \cos \varphi2 \times \sin^2(\Delta\lambda / 2)$

$c = 2 \times \operatorname{atan2}(\sqrt{a}, \sqrt{(1-a)})$

$d = R \times c$

Δφ 는 두 역 간의 위도차, Δλ 는 두 역간의 경도차, R = 지구의 반지름(6,371 km), d 는 두 지점 사이의 거리

3.2 작동원리

```
sb1(X, Y, Z):-subway(X, Y, Z).
sb1(X, Y, Z):-subway(Y, X, Z).

r(X, Y, Z):-road(X, Y, Z).
```

```
r(X, Y, Z):-road(Y, X, Z).
```

위 sb1(X, Y, Z)와 r(X, Y, Z) predicate 는 노드간 경로의 양방향성을 위해 정의하였다.

전체적인 흐름은 다음과 같다.

```
shortest_path(Point, Destination):-
        astar([[0,Point]],Destination,ReversePath),
        reverse(ReversePath, Path),
        write('최단경로: '), print_path(Path,Subways),
        write('경로의 역간 이용 지하철 호선: '),print_subways(Subways).
```

shortest_path(Point, Destination)는 우리가 원하는 시작역(노드)과 도착역(노드) 사이의 최단경로를 찾기 위해 최종 실행하는 predicate 다. 이 predicate 는 크게 2 가지로 볼 수 있는데 첫 번째로 A*알고리즘을 구현한 astar predicate 와 두 번째로 astar 로부터 얻은 결과값을 원하는 형태로 출력하기 위해 필요한 predicate 들인 reverse, print_path, print_subway 가 있다. shortest_path predicate 를 실행하게 되면 순서에 따라서 astar predicate 를 먼저 실행하게 된다.

```
astar(Paths, Destination, [C,Destination|Path]):-
        member([C,Destination|Path],Paths),
        choosebest(Paths, Destination, [C1|_]),
        C1 == C.
astar(Paths, Destination, BestPath):-
        choosebest(Paths, Destination, Best),
        delete(Paths, Best, PrevPaths),
        expand(Best, NewPaths),
        append(PrevPaths, NewPaths, L),
        astar(L, Destination, BestPath).
```

첫 번째 astar predicate 는 먼저 현재까지 존재하는 리스트 형태의 경로들(Paths) 중에서 도착역인 Destination 노드를 가지고 있는 경로를 찾는다. 이때 Destination 노드를 갖는 경로가 없다면 2 번째 astar predicate 로 넘어가고, 있다면 그 경로가 갖는 Cost 를 변수 C 에 저장한다. 이후 choosebest predicate 를

실행하여 생성된 경로들(Paths)의 최근 노드에서 휴리스틱 함수값을 측정하고 그 값이 가장 적은 값을 가진 경로를 채택한다. 이 choosebest predicate 에서 채택된 경로의 Cost 인 C1 과 아까 저장해놨던 C 가 같다면 휴리스틱 함수를 이용한 최단거리를 찾은 것이다.

두번째 astar 는 현재 생성된 경로들(Paths) 중에서 Destination 노드가 포함되어 있는 경로가 없으므로 위에서처럼 똑같이 choosebest predicate 를 실행하여 기존 경로들(Paths) 중 휴리스틱 함수를 기준으로 최적인 경로 하나를 정한다. 이후 기존 경로들(Paths)중에서 delete predicate 를 사용해 뽑힌 최적의 경로인 Best 를 삭제한 나머지 경로들을 PrevPaths 로 할당한다. 뽑힌 경로인 Best 는 그 경로의 최근 노드에서 갈 수 있는 모든 경로를 expand predicate 를 사용하여 찾고 그때의 모든 cost 를 업데이트하여 NewPaths 로 할당한다. 이후 append predicate 를 사용하여 PrevPaths 와 NewPaths 를 합쳐 L 로 할당한다. 이후 L 을 가지고 재귀함수의 형태로 astar predicate 를 다시 실행시키면서 Destination 노드까지의 최단 경로를 찾을 때까지 위 과정을 반복하게 된다.

이후 Destination 노드를 포함한 최단 경로를 찾았다면 이때의 형태가 [cost, 도착역, 중간 거쳐가는 역들, 시작역]이기 때문에 reverse predicate 를 사용하여 [시작역, 중간 거쳐가는 역들, 도착역, cost]형태로 바꿔준다. 그리고 print_path predicate 를 사용하여 시작역부터 도착역까지 거쳐간 모든 역들을 출력하고, print_subways predicate 를 사용하여 모든 역간 이용한 지하철 호선이 나오게 출력한다.

아래는 직접 정의한 세부적인 predicate 들에 대한 설명이다. delete, findall predicate 는 Swish prolog 의 내장된 predicate 를 사용하였다.

```
choosebest([X],_,X):-!.
choosebest([[C1,Station1|Y],[C2,Station2|_]|Z], Destination,Best):-
        hvalue(Station1, Destination,H1),
        hvalue(Station2, Destination,H2),
        H1 + C1 =< H2 + C2,
        choosebest([[C1,Station1|Y]|Z], Destination,Best).
choosebest([[C1,Station1|_],[C2,Station2|Y]|Z], Destination,Best):-
```

```
    hvalue(Station1, Destination, H1),
    hvalue(Station2,Destination, H2),
    H1 + C1 > H2 + C2,
    choosebest([[C2,Station2|Y]|Z], Destination,Best).
```

choosebest predicate 는 휴리스틱 함수를 기준으로 계산하여 그 값이 가장 작은 경로를 선택한다. 첫번째 choosebest predicate 는 리스트 형태의 경로집합에 경로가 하나만 있다면 그 경로를 결과값으로 할당한다. 두번째, 3 번째 choosebest predicate 는 2 개 이상의 경로가 존재한다면 앞에서부터 2 개씩 묶어서 휴리스틱 함수를 사용하여 비교한다. 이 때 첫번째 경로의 가장 최근 노드인 Station1 과 두번째 경로의 가장 최근 노드인 Station2 를 hvalue predicate 를 각자 사용하여 Destination 노드까지의 유클리드 거리값(h(n))인 H1 과 H2 를 얻는다. 이후 h(n)에 현재까지 경로의 cost(g(n))를 의미하는 C1, C2 를 각각 더해 각자 휴리스틱 값을 도출하고 비교한다. 2 번째 choosebest predicate 에서 Station1 의 휴리스틱 함수 값이 Station2 휴리스틱 함수 값보다 더 작다면 전체 경로에서 Station2 를 포함한 경로를 버리고 나머지 경로들의 리스트 집합에서 재귀함수의 형태로 choosebest predicate 를 실행시킨다. Station1 의 휴리스틱 함수 값이 Station2 휴리스틱 함수 값보다 크다면 3 번째 choostbest predicate 를 실행하게 되고 그 결과 Station1 을 포함한 경로를 버리고 나머지 경로들의 리스트 집합에서 재귀함수의 형태로 choose predicate 를 실행시킨다. 이후 경로들의 리스트 집합 내에 휴리스틱 함수 기준으로 가장 작은 값을 가지는 한가지 경로만 남을 때까지 위 과정을 반복한다.

```
hvalue(Station, Destination, Distance) :-
coordinates(X1, Y1, Station),
coordinates(X2, Y2, Destination),
Dlat is (X2 - X1) * pi / 180,
Dlon is (Y2 - Y1) * pi / 180,
A is sin(Dlat / 2) ** 2 + cos(X1 * pi / 180) * cos(X2 * pi / 180) * sin(Dlon / 2) ** 2,
C is 2 * atan2(sqrt(A), sqrt(1 - A)),
Distance is C * 6371.0.
```

hvalue 는 유클리드 거리 개념과 실제 두 지점의 위도와 경도를 사용하여 직선거리를 계산하는 predicate 이다. Station 노드와 Destination 노드 각각의 위도, 경도 좌표는 coordinate predicate 를 이용하여 X1, Y1 과 X2, Y2 로 불러온다. 이후 구현된 하버사인 공식을 통해 두 직선거리를 계산하여 Distance 로 할당한다.

```
expand([Cost,Station|Path],Paths):-
        findall([Cost,NewStation,Station|Path],
        (r(Station, NewStation,_),
        not(member(NewStation,Path))),L),
        update_costs(L, Paths).
```

expand predicate 는 [Cost, Station|Path]로 구성되어 있는 한 경로에서 Station 노드에 길(r(Station, NewStation,_))이 있어 갈 수 있는 노드들 중 Path 에 이미 있는 노드를 제외한 나머지 노드를 찾는다(not(member(NewStation,Path))). 이후 NewStation 에 올 수 있는 노드들을 리스트 형태인 [Cost, NewStation, Station|Path]에 하나씩 넣어 리스트들의 집합 리스트를 만들어 L 에 할당한다. 이후 update_costs predicate 를 실행하여 앞에서 찾은 각각의 새로운 경로에 대해 cost 를 바꿔준다.

```
update_costs([],[]):-!.
update_costs([[Total_Cost,Station1,Station2|Path]|Y],[[NewCost_Total,Station1,Statio
n2|Path]|Z]):-
        r(Station2, Station1, Distance),
        NewCost_Total is Total_Cost + Distance,
        update_costs(Y,Z).
```

첫번째 update_costs predicate 는 빈리스트일 경우 빈리스트를 그대로 할당해준다. 두번째 update_costs predicate 는 리스트형태의 경로들이 모인 리스트 집합에서 앞에서부터 리스트형태의 경로를 선택한다. 이 경로에서 expand predicate 로부터 새로 추가된 Station1 노드와 바로 그전의 Station2 노드사이의 cost 인 Distance 를 불러와서 기존 Total_Cost 에 더한 값을 NewCost_Total 로 할당해준다. 이후 재귀함수의 형태로 update_costs predicate 를 실행시켜 남은 리스트형태 경로들의 cost 를 앞에서부터 순서대로 모두 바꿔준다.

```
print_path([Station_Destination,Cost],[]):- write(Station_Destination), write('.'), write('
'), nl, write('경로의 총 거리: '), write(Cost), write(' km'),nl.
print_path([Station1,Station2|Y],Subways):-
        sb1(Station1,Station2,Subway),
        append([Subway],R,Subways),
        write(Station1),write(', '),
        print_path([Station2|Y],R).
```

첫번째 print_path 는 경로 리스트 안에 순서상 도착역 노드가 할당된 Station_Destination 과 그 경로의 전체 cost 인 Cost 만 있을 경우 print_path 왼쪽에 양식에 맞춰 Station_Destination 과 Cost 를 출력하라는 predicate 이다.

두번째 print_path 는 경로 리스트 안에 역 노드가 2 개 이상 있을 경우 앞의 2 개 노드인 Station1 과 Station2 노드를 잇는 호선을 찾은 후 이를 Subway 에 할당한다. 이 Subway 를 변수 R 과 함께 리스트 형태인 Subways 에 할당하고 양식에 맞게 Station1 을 출력한다. 이후 Station1 노드를 제외한 경로리스트에 대해 재귀함수의 형태로 Print_path predicate 를 실행시켜 경로리스트 안에 역 노드가 1 개있을 때까지 과정을 반복한다. 앞에서 reverse predicate 를 사용한 경로 리스트 형태가[시작역, 중간 거쳐가는 역들, 도착역, cost]이기 때문에 역 노드가 1 개 남았을 때는 그 노드가 무조건 도착역(Station_Destination)이 된다. 이렇게 역 노드가 1 개만 남겨졌을 땐, 첫번째 print_path predicate 가 실행되고 변수 R 에 빈리스트를 할당한다. 결국 Subways 는 시작역 노트부터 역간 이용한 지하철 호선을 순서대로 모아 놓은 리스트가 된다.

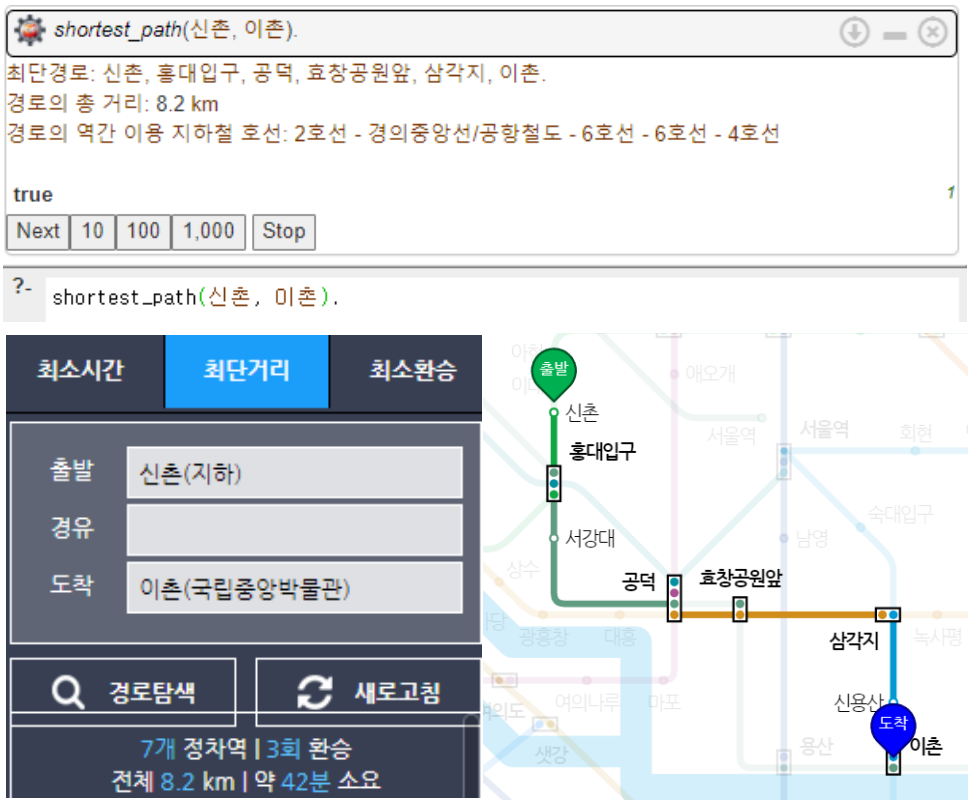
```
print_subways([X]):- write(X), nl, nl.
print_subways([X|Y]):-
        write(X),write(' - '),
        print_subways(Y),!.
```

첫번째 print_subways 는 리스트 안에 원소가 X 하나만 있을 경우 주어진 양식대로 X 를 출력하는 predicate 다. 두번째 print_subways 는 리스트 안에 원소가 2 개이상 있을 경우 제일 앞의 원소 X 를 주어진 양식에 맞게 출력하고 이후 X 를 제외한 나머지 Y 를 대상으로 재귀함수의 형태로 print_subways predicate 를 실행시킨다. 이 과정은 리스트 안에 원소 하나만 남을 때까지

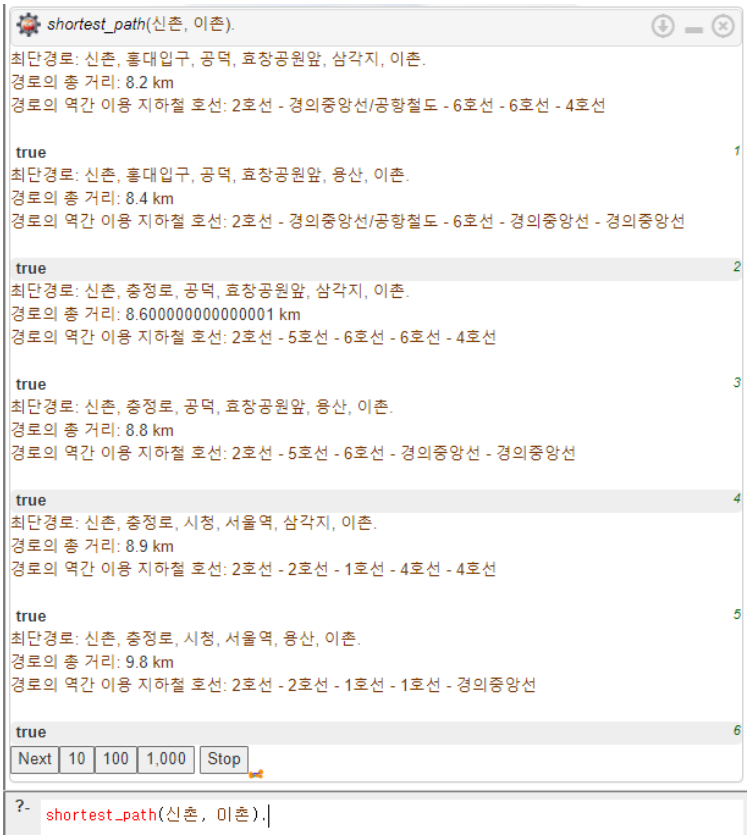
반복하게 된다. Shortest_path predicate 에서 print_subways predicate 는 변수로 위에서 설명한 Subways 리스트를 받게되므로 결국 Subways 리스트 속 원소들인 역간 이용한 지하철 호선을 앞에서부터 하나씩 출력시킨다.

3.3 실행 예시

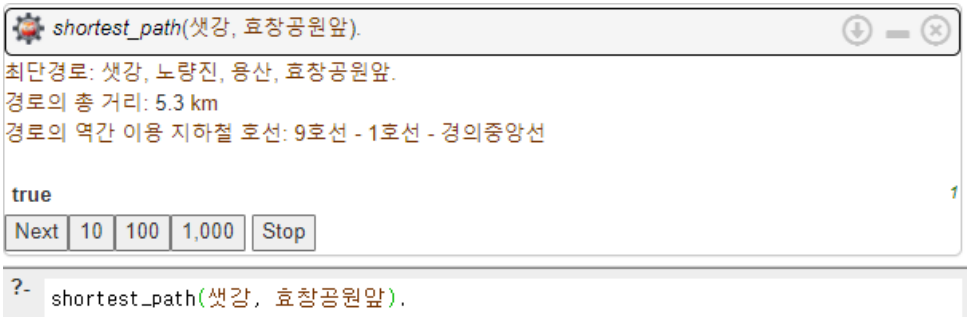
'신촌역 - 이촌역'에 대한 최단경로는 '신촌-홍대입구-공덕-효창공원앞-삼각지-이촌'으로, 서울교통공사의 정보와 동일하다. 또한 그 최단거리 및 이용 호선도 동일하다.



또한 프롤로그 결과창의 Next 를 활용하여, 최단거리 뿐만 아니라 출발역에서 도착역까지의 가능한 모든 경로 중 N 번째로 짧은 경로와 그때의 총 이동거리도 확인할 수 있었다. 아래 사진은 이 알고리즘이 '신촌역-이촌역'에 대해 최단경로 이외의 경로들을 총 거리가 작은 순서대로 찾아주는 것을 일부 캡쳐한 것이다.



구현한 코드에 의하면, '샛강 - 효창공원앞'에 대한 최단경로는 '샛강-노량진-용산-효창공원앞'이다. 반면 서울교통공사는 '샛강-여의도-공덕-효창공원앞'을 최단거리를 가지는 경로로 나타낸다. 그러나, 이 두 가지의 경로의 거리의 합은 5.3(km)으로 동일하다. 따라서 작성된 코드가 정상적으로 작동하고 있음을 확인할 수 있다.



4. 시사점 및 한계점

본 프로젝트에서는 A* 알고리즘을 활용하며 h-value 를 역간 직선거리로 설정하고, 역간 실제 거리를 활용함으로써 최단거리에 기반한 최단경로를 제공하는 시스템을 구현하였다. 기존에 존재하는 여러 다양한 길 찾기 기능들은 다익스트라 알고리즘(Dijkstra-algorithm)에 그 기반을 두고 있는 것에 반하여, 본 프로젝트를 통해 A* 알고리즘으로도 최단경로 길 찾기 기능을 실현할 수 있음을 보였다. 또한 프롤로그의 Next 기능을 통해 경로의 총 거리를 기준으로 출발지로부터 목적지까지 가능 경로 중 최단 경로뿐만 아니라 N 번째로 짧은거리 경로 또한 찾을 수 있음을 보였다. 본 프로젝트는 그러한 맥락에서 시사점과 의의를 가진다.

그럼에도, 한계는 남아있다. 최단경로 제공에 있어 최단거리만을 고려한 것에 관한 문제이다. 실제 길 찾기에 있어 최단 거리를 갖는 경로를 찾았다고 해도 중요한 것은 실제 소요 시간이다. 본 프로젝트에서는 해당 부분에 대한 고려가 다소 부족하였다고 판단한다. 이와 같은 문제를 보완하기 위해 역간 환승 소요 시간, 배차 간격, 실제 역간 열차 운행속도, 역의 혼잡도, 시계열 등을 최단 소요시간의 판단 근거로 활용해야 할 것이다.