2025 Data Structure & Algorithm I

Find the Best Subway Route in Seoul

(line number 1~5)

Team 12 I 21101164 곽용진 24102359 김이영

Contents

역할 분담

곽용진: 플로이드 워셜 알고리즘, y 환승 구역 최적화, 맵 시각화

김이영: 데이터 처리(수집, 처리), 다익스트라 알고리즘, 변형된 다익스트라 알고리즘

Data Preprocessing > 데이터 수집

서울교통공사_노선별 지하철역 정보:

https://data.seoul.go.kr/dataList/OA-15442/S/1/datasetView.do

서울시 역사마스터 정보(위경도): https://data.seoul.go.kr/dataList/OA-21232/S/1/datasetView.do

국가철도공단: 1~5호선 역간 거리(km) 정보:

https://www.data.go.kr/data/15081860/fileData.do?recommendDataYn=Y

서울교통공사: 1_8호선 표정속도 정보, 서울 도시철도 환승 정보

총 12개의 파일 사용.

파일은 모두 cp949 encodding으로 처리함.

Data Preprocessing

환승 정보 데이터

- 역별 환승 호선 추출

각 역마다 최대 2개의 환승 정보를

저장할 수 있도록 함

또한 1~5호선만 출력되도록 필터링

```
transfer_df = pd.read_csv("서울교통공사_서울 도시철도 환승정보_20250319.csv", encoding="cp949")
# 환승역끼리 그룹핑하여 각 역마다 환승 대상 호선을 추출
grouped = transfer_df.groupby("환승시작역")["환승종료 호선"].unique().reset_index()
grouped.columns = ["역명", "환승호선리스트"]

# 각 역별 transfer1, transfer2로 나눠 저장
grouped["transfer1"] = grouped["환승호선리스트"].apply(lambda x: x[0] if len(x) > 0 else "")
grouped["transfer2"] = grouped["환승호선리스트"].apply(lambda x: x[1] if len(x) > 1 else "")
transfer_df = grouped[["역명", "transfer1", "transfer2"]]
transfer_df.head(10)
```

	역명	transfer1	transfer2
0	가락시장	8	3
1	가산디지털단지	7	1
2	가좌	경의선	
3	강남	2	
4	강남구청	7	수인분당선
5	강동	5	
6	건대입구	7	2
7	계양	공항철도	인천1
8	고속터미널	9	7
9	고잔	수인분당선	4

```
[177] def clean_transfer(val):
        try:
            val_int = int(val)
            if 1 <= val_int <= 5: #1호선부터 5호선까지만 출력
                return str(val_int)
        except:
            pass
        return ""

input_df["transfer1"] = input_df["transfer1"].apply(clean_transfer)
input_df["transfer2"] = input_df["transfer2"].apply(clean_transfer)

input_df.to_csv("input.csv", index=False, encoding="cp949")
input_df.head()
```

```
<ipython-input-177-232ef9339370>:10: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
  input_df["transfer1"] = input_df["transfer1"].apply(clean_transfer)
<ipython-input-177-232ef9339370>:11: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
  input_df["transfer2"] = input_df["transfer2"].apply(clean_transfer)
```

	line_num	station_nm	cx	cy	link1	link2	link3	link4	transfer1	transfer2
0	1	서울	126.972533	37.553150						
1	1	시청	126.975407	37.563590					2	1
2	1	종각	126.983116	37.570203						
3	1	종로3가	126.992095	37.570429					3	5
4	1	종로5가	127.001900	37.570971						

Data Preprocessing

결과

line number, 역 이름, 위도, 경도,

link 1~3, transfer 1~2 로 구성됨.

(link 4는 의미 X 추후에 삭제함.)

```python
def remove_duplicate_transfer(row):
    line = row["line_num"]
    station = row["station_nm"]
    new_transfers = []

    for col in ["transfer1", "transfer2"]:
        val = row[col]
        if isinstance(val, str) and val.startswith("(") and ":" in val:
            try:
                content, time = val.split(":")
                name_in_val, line_in_val = content.replace("(", "").replace(")", "").split(",")
                if int(line_in_val) != line or name_in_val != station:
                    new_transfers.append(val)
                else:
                    new_transfers.append("")  # 동일 역명 + 동일 호선이면 제거
            except:
                new_transfers.append("")
        else:
            new_transfers.append("")
    return pd.Series(new_transfers, index=["transfer1", "transfer2"])

# 적용
final_full_df[["transfer1", "transfer2"]] = final_full_df.apply(remove_duplicate_transfer, axis=1)

final_full_df.to_csv("input.csv", index=False, encoding="cp949")

final_full_df.head(20)
```

	line_num	station_nm	cx	cy	link1	link2	link3	link4	transfer1	transfer2
0	1	서울	126.972533	37.553150						
1	1	시청	126.975407	37.563590	(종각,1):2.3	(서울역,1):2.5			(시청,2):5.0	
2	1	종각	126.983116	37.570203	(종로3가,1):1.8	(시청,1):2.3				
3	1	종로3가	126.992095	37.570429	(종로5가,1):2.1	(종각,1):1.8			(종로3가,3):5.0	(종로3가,5):5.0
4	1	종로5가	127.001900	37.570971	(동대문,1):1.8	(종로3가,1):2.1				
5	1	동대문	127.011383	37.571790	(동묘앞,1):1.4	(종로5가,1):1.8			(동대문,4):5.0	

(종로5가,1): 2.1

종로5가역 1호선 소요시간 2.1분

소요시간 계산 방법은,

시간(min) = 거리(km) / 속도(km/h) * 60으로 계산함.

Data Preprocessing

	A	B	C	D	E	F	G	H	I
1	line_num	station_nm	cx	cy	link1	link2	link3	transfer1	transfer2
2	1	가능	127.044213	37.748577	(녹양,1):1	(의정부,1):2			
3	1	가산디지털단지	126.882343	37.481072	(구로,1):2	(독산,1):2			
4	1	간석	126.694181	37.464737	(동암,1):2.8	(주안,1):2			
5	1	개봉	126.85868	37.494594	(구일,1):2.3	(오류동,1):2			
6	1	관악	126.908706	37.419232	(석수,1):2	(안양,1):2			
7	1	광명	126.884466	37.416182	(금천구청광명,1):5				
8	1	광운대	127.061835	37.623632	(월계,1):2.5	(석계,1):2			
9	1	구로	126.881966	37.503039	(신도림,1):2.5	(가산디지털단지,1):2		(구로구일,1):1	
10	1	구로구일	126.881966	37.503039	(구일,1):2			(구로,1):1	
11	1	구일	126.870793	37.496756	(구로구일,1):2	(개봉,1):2.3			
12	1	군포	126.948462	37.35356	(금정,1):2	(당정,1):2			
13	1	금정	126.943429	37.372221	(명학,1):2	(군포,1):2		(금정,4):1	
14	1	금천구청광명	126.89398	37.455626	(광명,1):5			(금천구청,1):2	
15	1	금천구청	126.89398	37.455626	(독산,1):2.8	(석수,1):2		(금천구청광명,1):2	
16	1	남영	126.9713	37.541021	(서울,1):2	(용산,1):2			
17	1	노량진	126.942454	37.514219	(용산,1):3	(대방,1):2			
18	1	녹양	127.042292	37.75938	(양주,1):3.0	(가능,1):2.8			
19	1	녹천	127.051269	37.644799	(창동,1):3.2	(월계,1):2.5			
20	1	당정	126.948345	37.344285	(군포,1):2	(의왕,1):2			
21	1	대방	126.926382	37.513342	(노량진,1):2	(신길,1):1.8			
22	1	덕계	127.056486	37.818486	(덕정,1):3	(양주,1):3.7			
23	1	덕정	127.061277	37.843188	(지행,1):4	(덕계,1):3			
24	1	도봉	127.045595	37.679563	(도봉산,1):1	(방학,1):1			
25	1	도봉산	127.046222	37.689313	(망월사,1):2.8	(도봉,1):1			
26	1	도원	126.642706	37.468446	(제물포,1):1	(동인천,1):2			
27	1	도화	126.668672	37.46607	(주안,1):2.3	(제물포,1):2			
28	1	독산	126.889249	37.466613	(가산디지털단지,1):2	(금천구청,1):2.8			
29	1	동대문	127.009745	37.57142	(동묘앞,1):1.4	(종로5가,1):1.8		(동대문,4):2.7	

Programming

사용자의 출발역과 도착역을 기반으로 두 가지 기준 중 하나로 최적 경로를 탐색함.

1) 최단 시간 기준

2) 최소 환승 기준

사용한 최단 경로 알고리즘

1) DIjkstra 알고리즘(우선순위 Queue 기반)

2) Floyd-Warshall 알고리즘(전체 경로 사전 계산)

Programming > Pseudo-code

모든 역의 위경도 정보 불러오기
for 각 역에 대해:
  "역명_호선번호" 형태의 노드명 생성
  해당 노드의 위도, 경도 저장

환승 정보 불러오기
for 각 환승역에 대해:
  transfer1, transfer2 최대 2개의 환승 정보 추출

for 각 transfer 값에 대해:
  동일 역명 + 동일 호선이면 → 제거

for 각 transfer 값에 대해:
  "(역명,호선):5.0" 형식으로 변환

그래프 초기화
graph = {}
nodes = set()

for 각 역 in 역 목록:
  for link1 ~ link4 + transfer1 ~ transfer2:
    연결 정보가 있다면:
      출발노드 → 도착노드, 가중치 = 이동 시간 저장

if 거리 기반 데이터가 존재한다면:
  거리 / 해당 호선의 평균 속도로 이동 시간 계산
  이동 시간에 60 곱해서 분 단위로 변환
  graph와 edge 리스트에 추가

Programming > Pseudo-code

사용자 입력 받기
start_node = 입력("출발역과 호선 입력 (예: 서울역 1)")
end_node = 입력("도착역과 호선 입력 (예: 강남역 2)")
입력값이 유효하지 않으면 다시 입력

if 전체 경로를 미리 계산한다면:
    floyd_warshall(graph) 실행
    dist 배열과 next_node 행렬 저장

else:
    다익스트라 알고리즘

경로 복원
if Floyd-Warshall 사용:
    next_node 행렬 따라 경로 복원
else 다익스트라 사용:
    path_tracker (dict) 사용해 역추적

Y자 환승 최적화
for 경로 중간 역에 대해:
    동일 역명 + 반대 방향 환승이 있으면:
        중간 역에서 반대 방향 환승

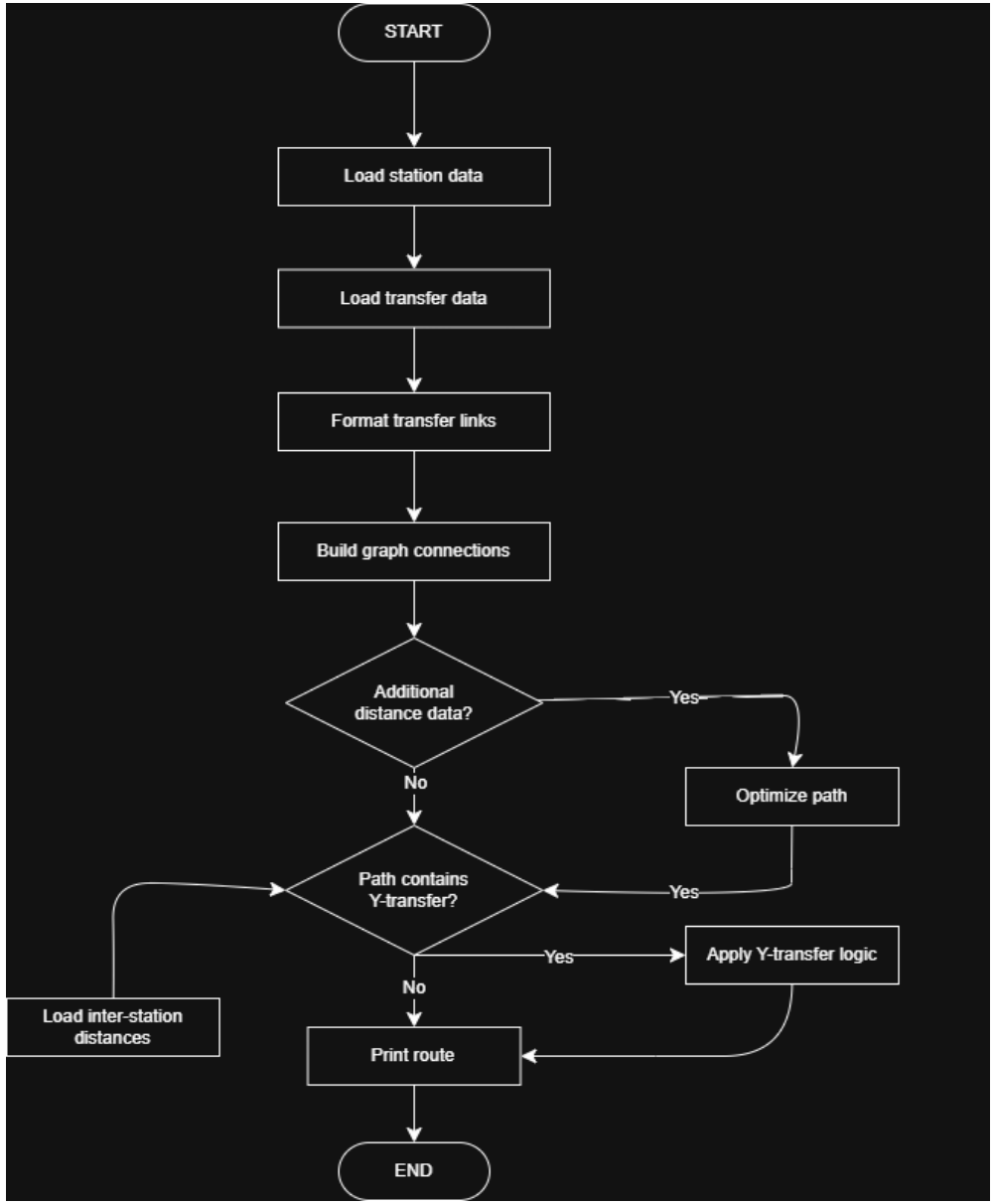
Programming > Pseudo-code

출력
for 각 구간 in 최종 경로:
    역 이름과 호선 출력
    환승 발생 시점에 환승 정보 출력
    구간별 이동 시간 출력

folium 지도 시각화
for 각 역 in 경로:
    호선마다 색상 다르게 라인 연결

프로그램 종료

Flow chart



Programming

(출발노드, 도착노드, 소요시간) 형태

```
[5]  # df에 있는 데이터를 통해 graph와 node, edge의 그래프를 구축
     # 파싱, 정제, 그래프화의 기능을 모두 포함하므로 preprocessing으로 이름 지음
     # 시각화를 위해 역 이름을 key값으로 (cx, cy) 튜플을 값으로 저장하는 딕셔너리도 생성

     def preprocessing(row, col_names, graph, nodes, edges, station_coords):
         station_nm = str(row['station_nm']).strip()                    # 공백, 개행, 타입 불일치 등 예방
         line_num = str(row['line_num']).strip()                        # 공백, 개행, 타입 불일치 등 예방
         from_node = f"{row['station_nm']}역_{row['line_num']}"
         cx = float(row['cx'])
         cy = float(row['cy'])
         station_coords[from_node] = (cy, cx)
         nodes.add(from_node)
         for col in col_names:
             val = row.get(col)
             if pd.notna(val) and val:
                 items = val.split()
                 for item in items:
                     left, travel_time = item.split(':')
                     left = left.strip('(')
                     to_station, to_line = left.split(',')
                     to_node = f"{to_station}역_{to_line}"
                     nodes.add(to_node)
                     if from_node not in graph:
                         graph[from_node] = []
                     graph[from_node].append((to_node, float(travel_time)))
                     edges.append((from_node, to_node, float(travel_time)))
```

Programming

최단경로 탐색 방법 1)

다익스트라 알고리즘

O((E+V)logV)

```python
# 다익스트라
# O(E + VlogV) (E: 간선 수, V: 노드 수)
# 쿼리가 적을 때 유리 + 메모리 사용량이 적고 사전 계산이 필요 없음

def dijkstra(graph, start, end):
    # (누적 시간, 노드) 형태로 저장
    heap = []
    heapq.heappush(heap, (0, start))

    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    prev_route = {node: None for node in graph}          # 경로 복원을 위한 사전

    while heap:
        cost, current_node = heapq.heappop(heap)

        if cost > distances[current_node]:
            continue
        next_step = graph[current_node]
        for next_node, weight in graph[current_node]:
            next_distance = cost + weight
            if next_distance < distances[next_node]:
                distances[next_node] = next_distance
                prev_route[next_node] = current_node
                heapq.heappush(heap, (next_distance, next_node))

    return distances[end], prev_route
```

Programming

최소 환승 알고리즘

heap에는 (환승횟수, 소요시간, 현재노드, 현재 호선, 경로리스트) 튜플이 저장되고, 파이썬의 튜플 비교는 사전순이기에 **자동으로 환승이 적고 시간도 짧은 경로가 우선적으로 선택**됨.

동일 노드와 호선 조합에 대해 더 나은 경로(환승 수나 시간 적게 걸리는 경우)가 이미 존재하면 가지 치기함.

인접한 노드가 다른 호선일 경우 환승 횟수를 증가시켜 환승 처리함.

```python
def min_transfer_and_time(start_node, end_node, graph, nodes):
    # 1. 환승 정보 구축 (같은 역, 다른 호선)
    station_to_nodes = defaultdict(list)
    for node in nodes:
        station, line = node.rsplit('_', 1)
        station_to_nodes[station].append(node)

    # 2. 우선순위 큐 초기화(환승 횟수, 걸린 시간, 역 이름_호선, 호선, 경로)
    heap = []
    start_station, start_line = start_node.rsplit('_', 1)
    end_station, end_line = end_node.rsplit('_', 1)
    for node in station_to_nodes[start_station]:
        line = node.rsplit('_', 1)[1]
        if line == start_line:
            heapq.heappush(heap, (0, 0, node, line, [node]))
        else:
            # graph에서 start_node와 node(환승노드) 사이의 환승시간을 찾아서 넣기
            transfer_time = None
            for neighbor, weight in graph.get(start_node, []):
                if neighbor == node:
                    transfer_time = weight
                    break
            if transfer_time is not None:
                heapq.heappush(heap, (1, transfer_time, node, line, [node]))
            # 만약 환승 간선이 없다면 추가하지 않음

    # 3. 방문 체크
    visited = dict()

    while heap:
        transfer_cnt, total_time, current, cur_line, path = heapq.heappop(heap)

        # 현재가 도착역이면
        if current == end_node:
            return transfer_cnt, total_time, path
```

```python
    # 현재가 도착역이면
    if current == end_node:
        return transfer_cnt, total_time, path

    key = (current, cur_line)
    if key in visited:
        prev_transfer, prev_time = visited[key]
        if (transfer_cnt, total_time) >= (prev_transfer, prev_time): # 파이썬에서 튜플끼리의 비교는 사전식 비교
            continue

    visited[key] = (transfer_cnt, total_time)

    for neighbor, weight in graph.get(current, []):
        neighbor_line = neighbor.rsplit('_', 1)[1]
        if neighbor_line == cur_line:
            # 같은 호선 이동
            heapq.heappush(heap, (transfer_cnt, total_time + weight, neighbor, cur_line, path + [neighbor]))
        else:
            # 환승
            heapq.heappush(heap, (transfer_cnt + 1, total_time + weight, neighbor, neighbor_line, path + [neighbor]))


return -1, -1, []
```

Programming

최단경로 탐색 방법 2)

플로이드-워셜 알고리즘

O(N^3) --> O(1)

```
[15]  # 플로이드-워셜
      # 실행 할 때마다 다익스트라 알고리즘을 계속 수행해야 함
      # 플로이드-워셜 알고리즘을 통해 미리 최단거리와 최단 경로를 파일로 저장한 후
      # 새롭게 사용할 때마다 파일을 불러와서 사용하는 방식으로 구현
      # 처음에만 O(V^3)이지만 이후의 쿼리에는 배열 인덱싱만 하면 되므로 O(1) (매우 빠름)
      # 메모리 사용량이 증가하지만 속도가 빠름(쿼리가 많을수록 유리)

      def floyd_warshall(nodes, edges):
          node_indices = {node: idx for idx, node in enumerate(nodes)}
          N = len(nodes)
          INF = float('inf')
          dist = [[INF] * N for _ in range(N)]
          next_node = [[None] * N for _ in range(N)]

          for i in range(N):
              dist[i][i] = 0

          for from_node, to_node, weight in edges:
              i, j = node_indices[from_node], node_indices[to_node]
              dist[i][j] = weight
              next_node[i][j] = j

          for k in range(N):
              for i in range(N):
                  for j in range(N):
                      if dist[i][j] > dist[i][k] + dist[k][j]:
                          dist[i][j] = dist[i][k] + dist[k][j]
                          next_node[i][j] = next_node[i][k]

          return dist, next_node, node_indices
```

Programming

Path_tracker가 dictionary 형식

--> 다익스트라 알고리즘으로 간주

역추적 방식으로 거슬러 올라감.

2차원 list(next_node) 형식

--> 플로이드-워셜 알고리즘으로 간주

index 기준 순차적 경로 구성

```python
# 경로 복원하기 위한 함수

def reconstruct_path(start, end, path_tracker, nodes=None, node_indices=None):
    if isinstance(path_tracker, dict):
        # 다익스트라용 (prev_route 딕셔너리)
        path = []
        node = end
        while node is not None:
            path.append(node)
            node = path_tracker[node]
        path.reverse()
        return path
    else:
        # 플로이드-워셜용 (next_node 2D 리스트)
        i = node_indices[start]
        j = node_indices[end]
        if path_tracker[i][j] is None:
            return []
        path = [start]
        while i != j:
            i = path_tracker[i][j]
            path.append(nodes[i])
        return path
```

Programming

Y자 환승 구간 최적화

문제가 발생하는 Y환승 구조를 정의하고 경로 탐색 결과에서 Y환승 구간을 탐지함. 만약 Y환승이 일어나야 하는 경로면 환승. 그렇지 않은 역이면 환승하는 시간을 제외.

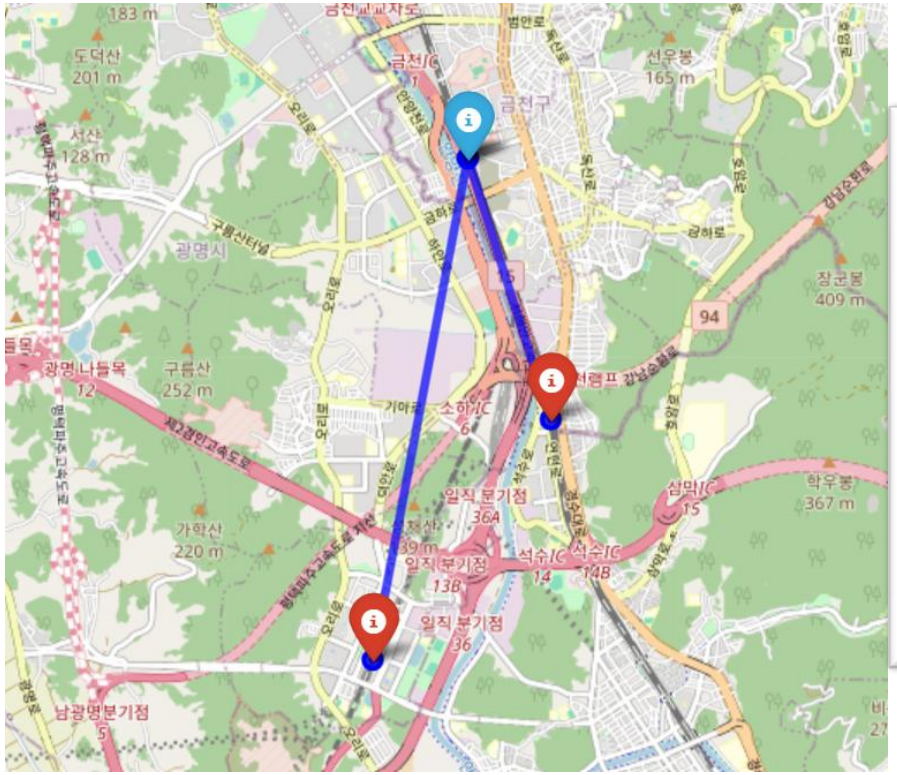
```python
# y환승 구역을 처리하는 코드블럭

# Y환승 처리
y_transfer = {"금천구청광명역":"금천구청역", "병점서동탄역":"병점역", "강동길동역":"강동역", "구로구일역":"구로역"}

# Y환승 역 탐지
y_station = {"금천구청광명역":5, "병점서동탄역":5, "강동길동역":1, "구로구일역":2}
y_station2 = {"금천구청역":5, "병점역":5, "강동역":1, "구로역":2}

# Y환승 탐지하는지 확인
y_detection = {"금천구청광명역":("광명역", "석수역"), "병점서동탄역":("서동탄역", "세마역"), "강동길동역":("길동역", "둔촌동역"), "구로구일역":("구일역", "가산디지털단지역")}

def y_clear(path, graph):
    if not path or len(path) < 3:
        return path, 0

    optimized_path = path.copy()
    saved_time = 0

    # 역순으로 처리하여 인덱스 변화 문제 방지
    for i in range(len(path) - 1, 0, -1):
        current_station = path[i].rsplit('_', 1)[0]

        # Y환승 키값이 경로에 있는지 확인
        if current_station in y_transfer:
            # 경로의 시작이나 끝이면 건너뛰기
            if i == 0 or i == len(path) - 1:
                continue

            # 이전 역과 다음 역 확인
            prev_station = path[i-1].rsplit('_', 1)[0]
            if i+2 < len(path):
                next_station = path[i+2].rsplit('_', 1)[0]
            else:
                continue

            # Y환승 감지 조건 확인
            detection_key = current_station
            if detection_key in y_detection:
                chk_1, chk_2 = y_detection[detection_key]
                # 실제 Y환승이 발생하지 않는 경우
                if not ((prev_station == chk_1 and next_station == chk_2) or (prev_station == chk_2 and next_station == chk_1)):

                    # Y환승 중간역 제거 및 시간 계산
                    removed_time = 0

                    # 이전 역 -> Y환승역 시간
                    for neighbor, weight in graph.get(path[i-1], []):
                        if neighbor == path[i]:
                            removed_time += weight
                            break

                    # Y환승역 -> 다음 역 시간
                    for neighbor, weight in graph.get(path[i], []):
                        if neighbor == path[i+1]:
                            removed_time += weight
                            break

                    # 직접 연결 시간 확인 (이전 역 -> 다음 역)
                    direct_time = y_station[current_station]

                    saved_time += (removed_time - direct_time)
                    optimized_path.pop(i)

    return optimized_path, saved_time
```

Programming

Y 환승 최적화 실행

금천구청역을 예시로 들자면, 역을 2개로 나눠서 기존의 금천구청역, 그리고 새로운 임의의 역인 금천구청광명역을 생성함. 경로 복원을 통해 Y환승을 탐지하도록 로직 구성

```
['광명역_1', '금천구청광명역_1', '금천구청역_1', '석수역_1']
['광명역_1', '금천구청광명역_1', '금천구청역_1', '석수역_1']
9.0 0
-----------------최단경로-----------------
총 소요 시간: 9분
-------------------출발-------------------
5분
광명역(1호선) → 금천구청역(1호선)
[금천구청역 내 분기 환승] : 2분
------------------------------------------
2분
금천구청역(1호선) → 석수역(1호선)
-------------------도착-------------------
환승 횟수 : 1

-----------------최소환승-----------------
최소 환승 경로 소요 시간: 9분
-------------------출발-------------------
5분
광명역(1호선) → 금천구청역(1호선)
[금천구청역 내 분기 환승] : 2분
------------------------------------------
2분
금천구청역(1호선) → 석수역(1호선)
-------------------도착-------------------
환승 횟수 : 1
```



Programming

```
[16]  start_node = get_valid_node("출발 역과 호선을 입력해주세요 (예: 서울역 1): ", nodes)
      end_node = get_valid_node("도착 역과 호선을 입력해주세요 (예: 강남역 2): ", nodes)
```

```
출발 역과 호선을 입력해주세요 (예: 서울역 1):
서울역 1
도착 역과 호선을 입력해주세요 (예: 강남역 2):
강남역2
입력 형식이 잘못되었습니다. 예시처럼 '서울역 1'과 같이 입력해주세요.
도착 역과 호선을 입력해주세요 (예: 강남역 2):
강남역 2
```

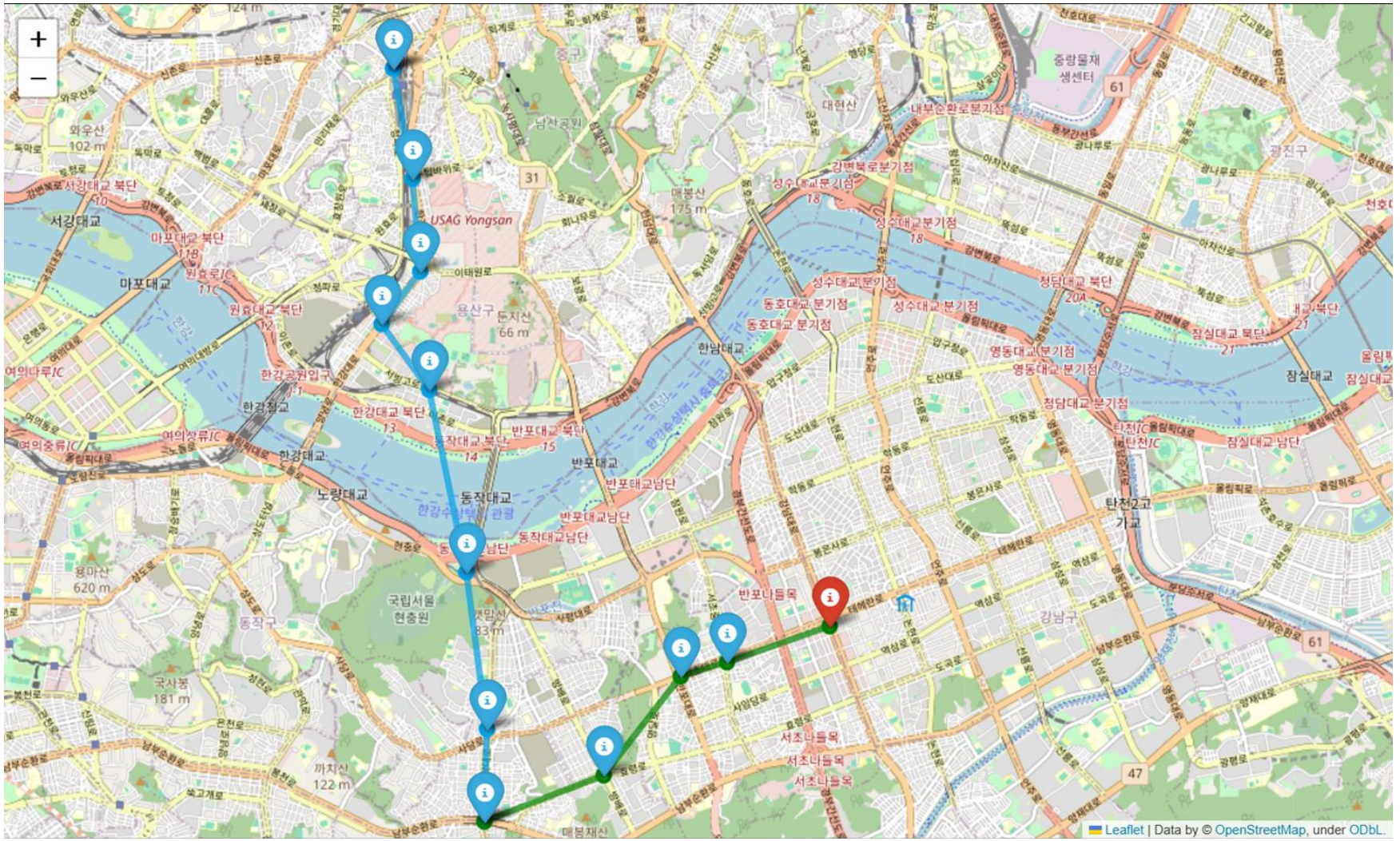
Programming

다익스트라 실행 결과 출력(최단경로, 최소환승)

```
----------------최단경로----------------
총 소요 시간: 25분
------------------출발------------------
0초
서울역(1호선)
[1호선에서 4호선으로 환승] : 2분 12초
----------------------------------------
13분
서울역(4호선) → 숙대입구역(4호선) → 삼각지역(4호선) → 신용산역(4호선) → 이촌역(4호선) → 동작역(4호선) → 총신대입구역(4호선) → 사당역(4호선)
[4호선에서 2호선으로 환승] : 1분
----------------------------------------
8분 48초
사당역(2호선) → 방배역(2호선) → 서초역(2호선) → 교대역(2호선) → 강남역(2호선)
------------------도착------------------
환승 횟수 : 2

----------------최소환승----------------
최소 환승 경로 소요 시간: 40분 24초
------------------출발------------------
2분
서울역(1호선) → 시청역(1호선)
[1호선에서 2호선으로 환승] : 1분 24초
----------------------------------------
37분
시청역(2호선) → 을지로입구역(2호선) → 을지로3가역(2호선) → 을지로4가역(2호선) → 동대문역사문화공원역(2호선) → 신당역(2호선) → 상왕십리역(2호선) → 왕십리역(2호선) → 한양대역(2호선) → 뚝섬역(2호선) → 성수역(2호선) → 건대입구역(2호선) → 구의역(2호선) → 강변역(2호선) → 잠실나루역(2호선) → 잠실역(2호선) → 잠실새내역(2호선) → 종합운동장역(2호선) → 삼성역(2호선) → 선릉역(2호선) → 역삼역(2호선) → 강남역(2호선)
------------------도착------------------
환승 횟수 : 1
```

Programming

Folium 시각화, 각 호선 별 다른 색



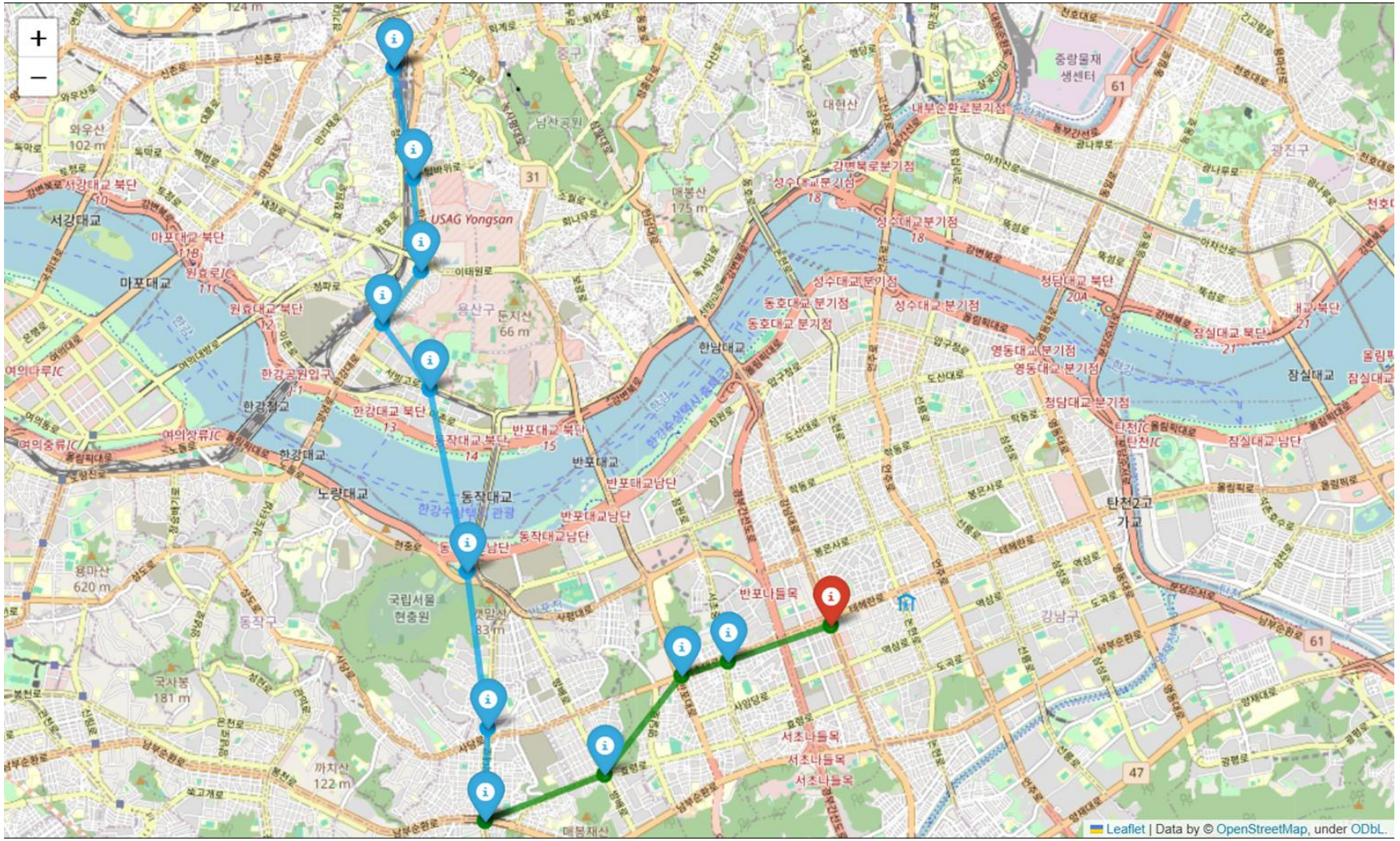
Programming

플로이드-워셜 실행 결과 출력(최단경로, 최소환승)

```
----------------최단경로----------------
총 소요 시간: 25분
-----------------출발-----------------
0초
서울역(1호선)
[1호선에서 4호선으로 환승] : 2분 12초
---------------------------------------
13분
서울역(4호선) → 숙대입구역(4호선) → 삼각지역(4호선) → 신용산역(4호선) → 이촌역(4호선) → 동작역(4호선) → 총신대입구역(4호선) → 사당역(4호선)
[4호선에서 2호선으로 환승] : 1분
---------------------------------------
8분 48초
사당역(2호선) → 방배역(2호선) → 서초역(2호선) → 교대역(2호선) → 강남역(2호선)
-----------------도착-----------------
환승 횟수 : 2

----------------최소환승----------------
최소 환승 경로 소요 시간: 40분 24초
-----------------출발-----------------
2분
서울역(1호선) → 시청역(1호선)
[1호선에서 2호선으로 환승] : 1분 24초
---------------------------------------
37분
시청역(2호선) → 을지로입구역(2호선) → 을지로3가역(2호선) → 을지로4가역(2호선) → 동대문역사문화공원역(2호선) → 신당역(2호선) → 상왕십리역(2호선) → 왕십리역(2호선) → 한양대역(2호선) → 뚝섬역(2호선) → 성수역(2호선) → 건대입구역(2호선) → 구의역(2호선) → 강변역(2호선) → 잠실나루역(2호선) → 잠실역(2호선) → 잠실새내역(2호선) → 종합운동장역(2호선) → 삼성역(2호선) → 선릉역(2호선) → 역삼역(2호선) → 강남역(2호선)
-----------------도착-----------------
환승 횟수 : 1
```

Programming

Folium 시각화, 각 호선 별 다른 색



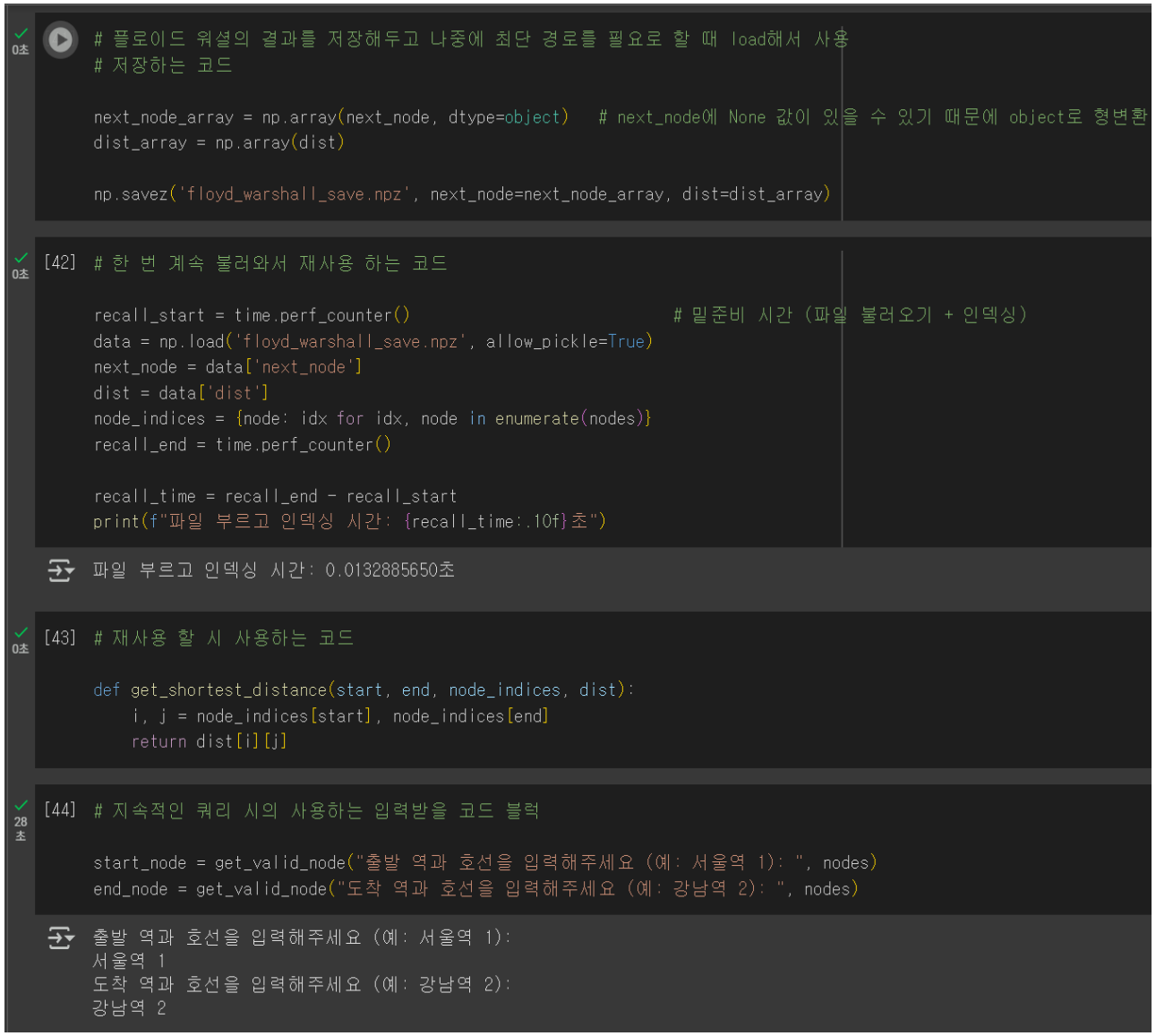
Programming

플로이드-워셜 결과 저장 & 재사용

플로이드-워셜 알고리즘의 계산 결과(dist, next_node)를 파일로 저장하고, 이후 동일한 쿼리를 빠르게 처리하기 위해 메모리로 불러오는 구조

연산 결과를 .npz포맷으로 저장하고, 이후 반복적인 질의 처리 시 재사용할 수 있도록 구성

계산 시간과 resource를 절약



```
0초
# 플로이드 워셜의 결과를 저장해두고 나중에 최단 경로를 필요로 할 때 load해서 사용
# 저장하는 코드

next_node_array = np.array(next_node, dtype=object)   # next_node에 None 값이 있을 수 있기 때문에 object로 형변환
dist_array = np.array(dist)

np.savez('floyd_warshall_save.npz', next_node=next_node_array, dist=dist_array)
```

```
0초
[42] # 한 번 계속 불러와서 재사용 하는 코드

recall_start = time.perf_counter()                          # 밑준비 시간 (파일 불러오기 + 인덱싱)
data = np.load('floyd_warshall_save.npz', allow_pickle=True)
next_node = data['next_node']
dist = data['dist']
node_indices = {node: idx for idx, node in enumerate(nodes)}
recall_end = time.perf_counter()

recall_time = recall_end - recall_start
print(f"파일 부르고 인덱싱 시간: {recall_time:.10f}초")

파일 부르고 인덱싱 시간: 0.0132885650초
```

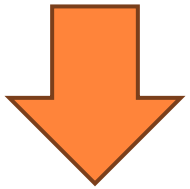
```
0초
[43] # 재사용 할 시 사용하는 코드

def get_shortest_distance(start, end, node_indices, dist):
    i, j = node_indices[start], node_indices[end]
    return dist[i][j]
```

```
28초
[44] # 지속적인 쿼리 시의 사용하는 입력받을 코드 블럭

start_node = get_valid_node("출발 역과 호선을 입력해주세요 (예: 서울역 1): ", nodes)
end_node = get_valid_node("도착 역과 호선을 입력해주세요 (예: 강남역 2): ", nodes)

출발 역과 호선을 입력해주세요 (예: 서울역 1):
서울역 1
도착 역과 호선을 입력해주세요 (예: 강남역 2):
강남역 2
```

Programming

```
# 플로이드-워셜 실행 시간

print(f"플로이드-워셜 실행 시간: {end_time - start_time:.10f}초")
```

```
플로이드-워셜 실행 시간: 4.0921214510초
```



0초

```
reuse_spend_time = reuse_end_time - reuse_start_time
print(f"배열 재사용 실행 시간: {reuse_spend_time:.10f}초")
```

```
배열 재사용 실행 시간: 0.0002882440초
```

Programming

다익스트라 1번 = 0.001s

플로이드-워셜 1번 = 4.0s

파일 불러오는 시간 = 0.01s

불러온 파일을 기반으로 최단거리를 계산하는 시간 = 0.0002s

Conclusion & Differentiator

알고리즘 / 처리 과정	시간복잡도	공간복잡도	비고
위경도 및 환승 정보 불러오기	$O(N)$	$O(N)$	N: 역 개수 (단순 순회 및 병합)
그래프 구축 (링크 + 환승 정보 기반)	$O(E)$	$O(N + E)$	E: 간선 수
Dijkstra (우선순위 큐 기반)	$O((V + E) \log V)$	$O(V)$	V: 노드 수 (역 개수), E: 간선 수
Modified Dijkstra (최소 환승 고려)	$O((V + E) \log V)$	$O(V)$	환승 수를 key로 추가 비교만 함
Floyd-Warshall	$O(V^3)$	$O(V^2)$	전 정점 쌍 최단거리. 경로 추적 위한 next_node 포함
Floyd 결과 저장 및 불러오기	$O(V^2)$	$O(V^2)$	np.savez/ np.load처리
경로 복원 (Dijkstra)	$O(V)$	$O(V)$	역추적 prev_route
경로 복원 (Floyd-Warshall)	O(경로 길이)	$O(1)$	next_node[i][j]를 따라가며 추적
Y환승 최적화 (경로 수정)	$O(P)$	$O(1)$	P: 경로 내 정점 수 (길이), 조건부 분기 제거만 수행
지도 시각화 (folium)	$O(P)$	$O(P)$	마커 & 선 반복 출력

Dijkstra는 단일 쿼리에 빠르고 메모리 효율적

Floyd-Warshall은 다중 쿼리 시 유리하지만
초기 계산량과 공간 부담이 큼

Y환승 최적화 및 시각화는
전체 흐름상 시간에 큰 영향을 주지 않음.

Conclusion & **Differentiator**

1. Y환승 최적화 로직

서울 지하철에는 동일 역명, 동일 호선이더라도 상행-하행 간 환승이 필요한 구조가 존재함(e.g. 금천구청역, 병점역, 구로역, 강동역). 해당 문제는 기존 알고리즘으로는 감지 불가하며, 본 시스템은 이를 탐지 및 최적화하는 알고리즘(y_clear)을 독자적으로 설계함. 따라서 해당 패턴을 사전 정의된 y_detection사전을 통해 탐지한 후, 중간 환승역 제거 및 직접 연결 간선으로 재구성함으로써 경로 최적화를 수행함.
(자세한 로직은 하단 참고)

2. Floyd-Warshall 기반 이중 모드 지원

대부분의 경로 탐색은 Dijkstra 기반이지만, 본 시스템은 선계산 후반복 쿼리 대응을 위한 Floyd-Warshall 기반도 지원함. 이를 통해 수십~수백건의 경로 탐색이 반복될 때 $O(1)$ 수준의 쿼리 응답이 가능함.
즉, 단건 실시간 탐색에는 Dijkstra 기반, 대규모 반복 탐색에는 Floyd-Warshall 기반 사전 계산 방식을 선택 할 수 있도록 함.
초기 계산은 O(V^3)이지만 이후의 쿼리에는 배열 인덱싱만 하면 되므로 O(1)으로 매우 빨라짐.
메모리 사용량은 증가하지만 역과 경로 수가 많지 않으므로 많은 쿼리시 속도 측면에서 유리함.

Conclusion & **Differentiator**

3. 최소 환승 Dijkstra 설계
기존의 시간 기반 Dijkstra 알고리즘 외에, 환승 수를 우선적으로 고려한 Modified Dijkstra를 통해 환승을 꺼리는 사용자에게 유리한 경로 탐색 결과를 제공할 수 있도록 함.
이 구조는 dictionary 자료형으로 visited의 key를 (노드, 호선)으로 구성하고 value를 (환승 횟수, 소요 시간)으로 구성하여 불필요한 경로 재탐색 방지를 통해 효율성을 높임.

4. 맵 시각화 (folium 기반)
텍스트 기반 경로 출력에 더해, folium 지도 시각화를 통해 출발~도착 마커, 호선 별 색상 라인을 통해 환승 시 위치 변화 등을 직관적으로 파악할 수 있도록 구성함.