2025

Data Structure & Algorithm I

Team Project Report

: Find the Best Subway Route in Seoul (line number 1 ~ 5)

Team 12 I 21101164 곽용진, 24102359 김이영

Contents

1. Data preprocessing

서울 지하철 경로 탐색 시스템 구축의 첫 단계는 방대한 지하철 정보를 정형화하여 알고리즘 기반으로 처리할 수 있도록 전처리하는 것이다. 본 프로젝트는 서울교통공사 및 국가철도공단에서 제공하는 역사 정보, 환승 정보, 거리 데이터 등을 수집하여 통합했음.

I. Data 구성 및 수집 경로

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.

II. Data Preprocessing & input.csv 파일 생성

1. 서울교통공사 데이터 기반 위경도 좌표 데이터 로드(= 역 좌표 정리)

```
[129] import pandas as pd
      # 서울교통공사 제공 위경도 정보 기준 정리
      latlon_df = pd.read_csv("서울교통공사_1_8호선 역사 좌표(위경도) 정보_20241031.csv", encoding="cp949")
      latlon_df["역명"] = latlon_df["역명"].str.strip()
      latlon_df["노드명"] = latlon_df["역명"] + "_" + latlon_df["호선"].astype(str)
      latlon_df.head()
```

	연번	호선	고유역번호(외부역코드)	역명	위도	경도	작성일자	노드명
0	1	1	150	서울	37.553150	126.972533	1974-02-28	서울_1
1	2	1	151	시청	37.563590	126.975407	1974-08-15	시청_1
2	3	1	152	종각	37.570203	126.983116	1974-08-15	종각_1
3	4	1	153	종로3가	37.570429	126.992095	1974-08-15	종로3가_1
4	5	1	154	종로5가	37.570971	127.001900	1974-03-31	종로5가_1

역명_호선 형식으로 노드명 생성(ex. 서울_1)

2. 우선, 4호선만 역간거리, 표정속도를 이용해 예상 소요시간 계산

```
distance_df = pd.read_csv("국가철도공단_수도권4호선_역간거리_20230516.csv", encoding="cp949")
speed_df = pd.read_csv("서울교통공사_1_8호선 표정속도 정보_20250314.csv", encoding="cp949")

# 4호선 기준 표정속도 추출
# "4호선"이 아니라 숫자 4가 들어가 있는지 확인 후 처리
if 4 in speed_df["호선"].values:
    line_speed = speed_df.loc[speed_df["호선"] == 4, "표정속도"].iloc[0]
    print("4호선 표정속도:", line_speed)
else:
    print("4호선 데이터가 표정속도 테이블에 없습니다.")

# 소요시간 계산
distance_df["예상소요시간"] = (distance_df["역간거리(km)"] / line_speed) * 60
distance_df.head()
```

```
4호선 표정속도: 40.1
```

	철도운영기관명	선명	역명	역간거리(km)	예상소요시간
0	코레일	4호선	대공원	1.0	1.496259
1	코레일	4호선	과천	1.0	1.496259
2	코레일	4호선	인덕원	3.0	4.488778
3	코레일	4호선	범계	2.6	3.890274
4	코레일	4호선	대야미	2.6	3.890274

소요시간 계산 방법은, 시간(min) = 거리(km) / 속도(km/h) * 60으로 계산함.
계산값을 예상소요시간 column을 추가해 저장함.
데이터의 거리(km)를 소요 시간(min)으로 변환하는 역할을 해줌.

3. 서울교통공사의 환승정보 데이터를 활용해 역별 환승호선 추출 단계

```python
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

환승시작역을 기준으로 grouping하여 해당 역에서 갈 수 있는 환승 호선 목록을 추출함.
각 역마다 최대 2개 환승 정보를 저장할 수 있도록 transfer1, transfer2 열에 저장함.
다대일 mapping(역 --> 여러 환승 호선)을 이대일 구조(transfer1, transfer2)로 제한함.
최대 2개까지 추출한 이유는 대부분의 역이 2개 이하의 환승지점만 보유하기 때문임.

4. 1차 전처리 파이프라인 구성, 앞서 확인한 내용 기반으로 함수 수정 후 전체 재실행

```python
import pandas as pd

# 위경도 정보 로드
latlon_df = pd.read_csv("/content/서울교통공사_1_8호선 역사 좌표(위경도) 정보_20241031.csv", encoding="cp949")
latlon_df["역명"] = latlon_df["역명"].str.strip()
latlon_df["노드명"] = latlon_df["역명"] + "_" + latlon_df["호선"].astype(str)
latlon_df.rename(columns={"위도": "cy", "경도": "cx"}, inplace=True)

# 환승 정보 로드 및 정제
transfer_df = pd.read_csv("/content/서울교통공사_서울 도시철도 환승정보_20250319.csv", encoding="cp949")
transfer_grouped = transfer_df.groupby("환승시작역")["환승종료 호선"].unique().reset_index()
transfer_grouped["transfer1"] = transfer_grouped["환승종료 호선"].apply(lambda x: x[0] if len(x) > 0 else "")
transfer_grouped["transfer2"] = transfer_grouped["환승종료 호선"].apply(lambda x: x[1] if len(x) > 1 else "")
transfer_grouped.rename(columns={"환승시작역": "역명"}, inplace=True)

# 표정속도 정보
speed_df = pd.read_csv("/content/서울교통공사_1_8호선 표정속도 정보_20250314.csv", encoding="cp949")
speed_df = speed_df[speed_df["호선"].apply(lambda x: str(x).isdigit())]
speed_df["호선"] = speed_df["호선"].astype(int)
speed_dict = speed_df.groupby("호선")["표정속도"].mean().to_dict()

# 유연한 역간 거리 정보 로드
def load_distance_data(filepath):
    df = pd.read_csv(filepath, encoding="cp949")
    for colname in ["역간거리(km)", "역간 거리(km)", "역간거리"]:
        if colname in df.columns:
            df = df[["선명", "역명", colname]]
            df.columns = ["line", "station", "distance_km"]
            return df
    raise ValueError(f"지원되지 않는 컬럼 구조: {filepath}")

distance_dfs = [
    load_distance_data("/content/국가철도공단_수도권1호선_역간거리_20241015.csv"),
    load_distance_data("/content/국가철도공단_수도권2호선_역간거리_20241015.csv"),
    load_distance_data("/content/국가철도공단_수도권3호선_역간거리_20241022.csv"),
    load_distance_data("/content/국가철도공단_수도권4호선_역간거리_20230516.csv"),
    load_distance_data("/content/국가철도공단_수도권5호선_역간거리_20241015.csv"),
]
```

```
]
distance_df = pd.concat(distance_dfs, ignore_index=True)
distance_df["line"] = distance_df["line"].str.extract(r"(\d)").astype(int)
distance_df = distance_df[distance_df["line"].isin([1, 2, 3, 4, 5])]
distance_df["time_min"] = distance_df.apply(
    lambda row: round((row["distance_km"] / speed_dict.get(row["line"], 30)) * 60, 1), axis=1
)

# 연결 정보 예시 만들기
link_data = distance_df.copy()
link_data["link"] = "(" + link_data["station"] + "," + link_data["line"].astype(str) + "):" + link_data["time_min"].astype(str)

# 위경도 정보에서 1~5호선만 필터
latlon_df["호선"] = latlon_df["호선"].astype(int)
final_df = latlon_df[latlon_df["호선"].isin([1, 2, 3, 4, 5])].copy()
final_df["line_num"] = final_df["호선"]
final_df["station_nm"] = final_df["역명"]
final_df["link1"] = ""

# 환승 정보 병합
final_df = final_df.merge(transfer_grouped[["역명", "transfer1", "transfer2"]], on="역명", how="left")
final_df[["transfer1", "transfer2"]] = final_df[["transfer1", "transfer2"]].fillna("")

# 누락된 link2~4 채우기
for col in ["link2", "link3", "link4"]:
    final_df[col] = ""

# 컬럼 정렬 및 결과 출력
input_df = final_df[["line_num", "station_nm", "cx", "cy", "link1", "link2", "link3", "link4", "transfer1", "transfer2"]]
input_df.to_csv("input.csv", index=False, encoding="cp949")

input_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					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						
5	1	동대문	127.011383	37.571790					4	1
6	1	동묘앞	127.016459	37.573265					6	1
7	1	신설동	127.024710	37.576117					우이신설경전철	2
8	1	제기동	127.034902	37.578116						
9	1	청량리	127.045063	37.580148					경의선	1
10	2	을지로입구	126.982569	37.565998						
11	2	을지로3가	126.991773	37.566292					3	2
12	2	을지로4가	126.998122	37.566611					5	2
13	2	동대문역사문화공원	127.009113	37.565597					5	4
14	2	신당	127.019488	37.565681					6	2
15	2	상왕십리	127.028872	37.564504						
16	2	왕십리	127.035505	37.561159					수인분당선	5
17	2	한양대	127.043504	37.556580						
18	2	뚝섬	127.047413	37.547180						
19	2	성수	127.055983	37.544628					2	

1~5호선의 위경도, 연결 시간, 환승 호선을 포함해서 경로 탐색 알고리즘의 input data로 활용할 수 있도록 1차 전처리를 함.

연결정보는 (역명,호선):시간 형식의 link문자열을 생성하고자 함.

하지만 link 관련된 열에서 빈 셀을 보임. 또한 1~5호선을 제외한 호선들도 출력되고 있고, transfer의 경우 환승하려는 호선의 이름만 출력되고 소요 시간이 출력되지 않는 상태임.

5. 환승 정보 정제 - 1~5호선만 남도록 필터링

```
[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						

clean_transfer(): 문자열을 정수로 변환하여 5 이하인 경우에만 유지하도록, 즉, 환승 호선 정보를 1~5호선으로 한정하여 필터링함.

정수형 호선 번호로 변환이 불가능하거나 6호선 이상의 노선은 환승 대상에서 제외함.

6. 누락됐던 link 1~4 생성

```python
# 연결 정보 자동 생성: 각 호선별 정렬 기준으로 인접한 역을 연결하고 link1~4에 입력
from collections import defaultdict

# 호선별로 정렬된 역 리스트 구성
grouped = input_df.sort_values(["line_num", "cy", "cx"]).groupby("line_num")

# 역 연결 관계 저장용 딕셔너리
links_by_node = defaultdict(list)

# 역 이름을 'station_nm_line_num' 형태로 만들어 인접 연결
for line, group in grouped:
    stations = group.reset_index(drop=True)
    for i in range(len(stations) - 1):
        curr = stations.loc[i]
        next_ = stations.loc[i + 1]

        curr_key = f"{curr['station_nm']}_{curr['line_num']}"
        next_key = f"{next_['station_nm']}_{next_['line_num']}"

        # 거리 추정값: 기본 2.0분
        links_by_node[curr_key].append((next_key, 2.0))
        links_by_node[next_key].append((curr_key, 2.0))

# link1~4 채우기
def assign_links(row):
    key = f"{row['station_nm']}_{row['line_num']}"
    links = links_by_node.get(key, [])
    formatted = [f"({k.split('_')[0]},{k.split('_')[1]}):{v}" for k, v in links]
    out = ["", "", "", ""]
    for i in range(min(4, len(formatted))):
        out[i] = formatted[i]
    return pd.Series(out, index=["link1", "link2", "link3", "link4"])

input_df[["link1", "link2", "link3", "link4"]] = input_df.apply(assign_links, axis=1)



input_df.to_csv("input.csv", index=False, encoding="cp949")
input_df.head(20)
```

	line_num	station_nm	cx	cy	link1	link2	link3	link4	transfer1	transfer2
0	1	서울	126.972533	37.553150	(시청,1):2.0					
1	1	시청	126.975407	37.563590	(서울,1):2.0	(종각,1):2.0			2	1
2	1	종각	126.983116	37.570203	(시청,1):2.0	(종로3가,1):2.0				
3	1	종로3가	126.992095	37.570429	(종각,1):2.0	(종로5가,1):2.0			3	5
4	1	종로5가	127.001900	37.570971	(종로3가,1):2.0	(동대문,1):2.0				
5	1	동대문	127.011383	37.571790	(종로5가,1):2.0	(동묘앞,1):2.0			4	1
6	1	동묘앞	127.016459	37.573265	(동대문,1):2.0	(신설동,1):2.0				1
7	1	신설동	127.024710	37.576117	(동묘앞,1):2.0	(제기동,1):2.0				2
8	1	제기동	127.034902	37.578116	(신설동,1):2.0	(청량리,1):2.0				
9	1	청량리	127.045063	37.580148	(제기동,1):2.0					1
10	2	을지로입구	126.982569	37.565998	(신당,2):2.0	(을지로3가,2):2.0				
11	2	을지로3가	126.991773	37.566292	(을지로입구,2):2.0	(용답,2):2.0			3	2
12	2	을지로4가	126.998122	37.566611	(용답,2):2.0	(용두,2):2.0			5	2
13	2	동대문역사문화공원	127.009113	37.565597	(상왕십리,2):2.0	(신당,2):2.0			5	4
14	2	신당	127.019488	37.565681	(동대문역사문화공원,2):2.0	(을지로입구,2):2.0				2
15	2	상왕십리	127.028872	37.564504	(시청,2):2.0	(동대문역사문화공원,2):2.0				
16	2	왕십리	127.035505	37.561159	(충정로,2):2.0	(신답,2):2.0				5
17	2	한양대	127.043504	37.556580	(신촌,2):2.0	(이대,2):2.0				
18	2	뚝섬	127.047413	37.547180	(성수,2):2.0	(합정,2):2.0				
19	2	성수	127.055983	37.544628	(건대입구,2):2.0	(뚝섬,2):2.0			2	

정렬된 역들을 순차적으로 연결하여 양방향 연결 정보를 만들었고,

(역명,호선):소요시간(2.0으로 잡음) 형식으로 link1~link4에 저장함.

기본 이동 시간은 2.0분으로 설정했고, 추후 실제 거리를 기반으로 계산하여 넣을 예정임.(or 외부 데이터 이용)

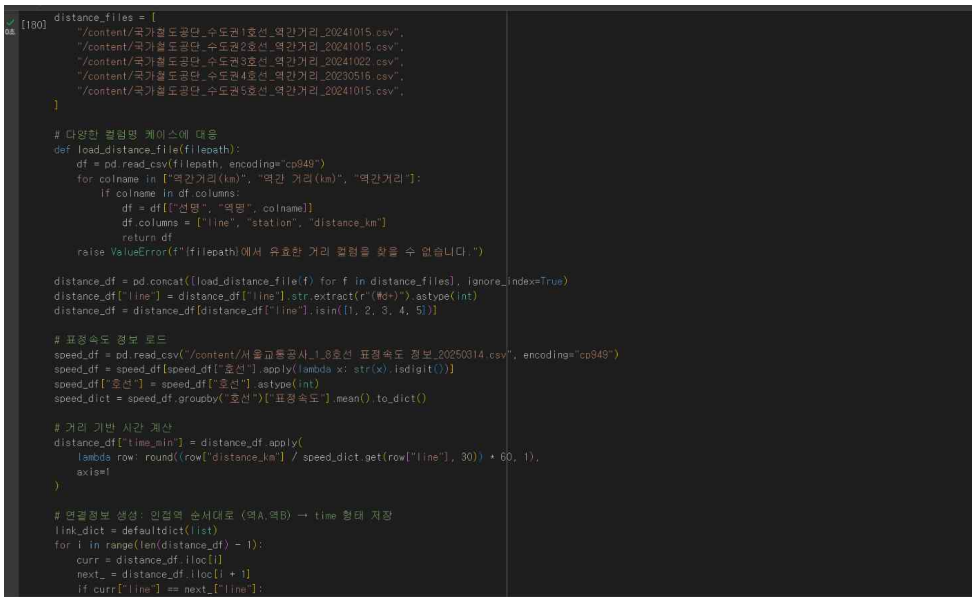
7. 환승 정보 포맷 정제 - (역명,호선):5.0형식으로

```
[179] # 기존 transfer1, transfer2 값을 링크 형식으로 다시 구성: (역명,호선):5.0
      def transfer_links_like_link(row):
          transfers = []
          for t in ["transfer1", "transfer2"]:
              val = row[t]
              if isinstance(val, str) and "(" in val:
                  # 이미 (역명,호선):시간 형식이면 그대로
                  transfers.append(val)
              elif isinstance(val, (int, float)) or (isinstance(val, str) and val.isdigit()):
                  # 숫자 또는 숫자 문자열이면 포맷팅
                  transfers.append(f"({row['station_nm']},{int(val)}):5.0")
              else:
                  # 그 외는 빈 칸
                  transfers.append("")
          return pd.Series(transfers, index=["transfer1", "transfer2"])


      # 적용
      final_df[["transfer1", "transfer2"]] = final_df.apply(transfer_links_like_link, axis=1)

      final_df.to_csv("input.csv", index=False, encoding="cp949")
      final_df.head(20)
```

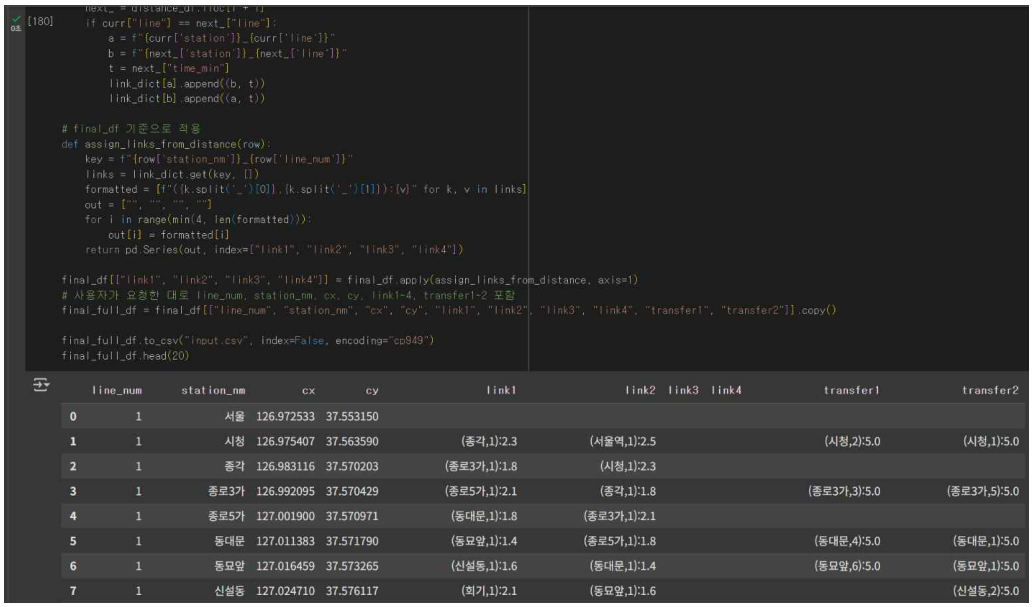
0	1	1	150	서울	37.553150	126.972533	1974-02-28	서울_1	1	서울		
1	2	1	151	시청	37.563590	126.975407	1974-08-15	시청_1	1	시청	(시청,2):5.0	(시청,1):5.0
2	3	1	152	종각	37.570203	126.983116	1974-08-15	종각_1	1	종각		
3	4	1	153	종로3가	37.570429	126.992095	1974-08-15	종로3가_1	1	종로3가	(종로3가,3):5.0	(종로3가,5):5.0
4	5	1	154	종로5가	37.570971	127.001900	1974-03-31	종로5가_1	1	종로5가		
5	6	1	155	동대문	37.571790	127.011383	1974-04-30	동대문_1	1	동대문	(동대문,4):5.0	(동대문,1):5.0

환승 대상 호선 정보를 (역명,호선):소요시간 형태로 변환하여 전체 포맷을 link 1~4와 일치시킴. 환승시간은 우선 기본값인 5.0분으로 고정하며, 추후 환승 거리 or 시간 정보 기반으로 조정할 예정임.

8. 2차 최종 데이터 전처리 - 거리 기반 연결 정보 생성



```
[180] distance_files = [
        "/content/국가철도공단_수도권1호선_역간거리_20241015.csv",
        "/content/국가철도공단_수도권2호선_역간거리_20241015.csv",
        "/content/국가철도공단_수도권3호선_역간거리_20241022.csv",
        "/content/국가철도공단_수도권4호선_역간거리_20230516.csv",
        "/content/국가철도공단_수도권5호선_역간거리_20241015.csv",
    ]

    # 다양한 컬럼명 케이스에 대응
    def load_distance_file(filepath):
        df = pd.read_csv(filepath, encoding="cp949")
        for colname in ["역간거리(km)", "역간 거리(km)", "역간거리"]:
            if colname in df.columns:
                df = df[["선명", "역명", colname]]
                df.columns = ["line", "station", "distance_km"]
                return df
        raise ValueError(f"{filepath}에서 유효한 거리 컬럼을 찾을 수 없습니다.")

    distance_df = pd.concat([load_distance_file(f) for f in distance_files], ignore_index=True)
    distance_df["line"] = distance_df["line"].str.extract(r"(\d+)").astype(int)
    distance_df = distance_df[distance_df["line"].isin([1, 2, 3, 4, 5])]

    # 표정속도 정보 로드
    speed_df = pd.read_csv("/content/서울교통공사_1_8호선 표정속도 정보_20250314.csv", encoding="cp949")
    speed_df = speed_df[speed_df["호선"].apply(lambda x: str(x).isdigit())]
    speed_df["호선"] = speed_df["호선"].astype(int)
    speed_dict = speed_df.groupby("호선")["표정속도"].mean().to_dict()

    # 거리 기반 시간 계산
    distance_df["time_min"] = distance_df.apply(
        lambda row: round((row["distance_km"] / speed_dict.get(row["line"], 30)) * 60, 1),
        axis=1
    )

    # 연결정보 생성: 인접역 순서대로 (역A,역B) → time 형태 저장
    link_dict = defaultdict(list)
    for i in range(len(distance_df) - 1):
        curr = distance_df.iloc[i]
        next_ = distance_df.iloc[i + 1]
        if curr["line"] == next_["line"]:
```



```
[180]     if curr["line"] == next_["line"]:
            a = f"{curr['station']}_{curr['line']}"
            b = f"{next_['station']}_{next_['line']}"
            t = next_["time_min"]
            link_dict[a].append((b, t))
            link_dict[b].append((a, t))

    # final_df 기준으로 적용
    def assign_links_from_distance(row):
        key = f"{row['station_nm']}_{row['line_num']}"
        links = link_dict.get(key, [])
        formatted = [f"({k.split('_')[0]},{k.split('_')[1]}):{v}" for k, v in links]
        out = ["", "", "", ""]
        for i in range(min(4, len(formatted))):
            out[i] = formatted[i]
        return pd.Series(out, index=["link1", "link2", "link3", "link4"])

    final_df[["link1", "link2", "link3", "link4"]] = final_df.apply(assign_links_from_distance, axis=1)
    # 사용자가 요청한 대로 line_num, station_nm, cx, cy, link1~4, transfer1~2 포함
    final_full_df = final_df[["line_num", "station_nm", "cx", "cy", "link1", "link2", "link3", "link4", "transfer1", "transfer2"]].copy()

    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	(시청,1):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	(동대문,1):5.0
6	1	동묘앞	127.016459	37.573265	(신설동,1):1.6	(동대문,1):1.4			(동묘앞,6):5.0	(동묘앞,1):5.0
7	1	신설동	127.024710	37.576117	(희기,1):2.1	(동묘앞,1):1.6				(신설동,2):5.0

서울시 1~5호선의 역간 거리 정보를 기반으로, 각 역과 인접 역 간의 실제 이동시간을 반영한 연결 정보를 생성함. 거리 및 표정속도를 기반으로 정확도 높은 link 1~4 값을 설정하고자 함(거리 기반으로 소요시간을 계산하여 time_min열 추가(나중에 병합)).

for문(len(distance_df) - 1 부분)으로 각 호선 내부에서 연속된 역들 간 양방향 연결 정보(link_dict)를 저장함. 하지만 transfer 부분에서 환승이 아닌데 환승역으로 표시되는 경우 발생.(동대문 1호선에서 동대문 1호선으로 환승 이런 식으로 중복되어있음)

9. 역간거리 기반 역명 수집을 통한 누락된 역 보완

```python
[182] import os

    # 사용자가 업로드한 국가철도공단 역간거리 파일들 (1~5호선)
    korail_files = [
        "/content/국가철도공단_수도권1호선_역간거리_20241015.csv",
        "/content/국가철도공단_수도권2호선_역간거리_20241015.csv",
        "/content/국가철도공단_수도권3호선_역간거리_20241022.csv",
        "/content/국가철도공단_수도권4호선_역간거리_20230516.csv",
        "/content/국가철도공단_수도권5호선_역간거리_20241015.csv"
    ]

    # 각 파일에서 역명 추출 및 정제
    additional_stations = []
    for file in korail_files:
        try:
            df = pd.read_csv(file, encoding="cp949")
            col_names = df.columns.tolist()
            # 역명 컬럼 찾기
            from_col = next((c for c in col_names if "역명" in c and "기점" in c), None)
            to_col = next((c for c in col_names if "역명" in c and "종점" in c), None)
            line_col = next((c for c in col_names if "호선" in c), None)

            if from_col and to_col and line_col:
                line_num = int("".join(filter(str.isdigit, str(df[line_col].iloc[0]))))  # 1~5
                for station in pd.concat([df[from_col], df[to_col]]).unique():
                    additional_stations.append((line_num, station.strip()))
        except Exception as e:
            continue

    # 기존에 포함된 역명 목록
    existing_stations = set(zip(final_full_df["line_num"], final_full_df["station_nm"]))

    # 추가해야 할 역
    new_station_tuples = list(set(additional_stations) - existing_stations)

    # 위경도 알 수 없는 경우 NaN 처리
    new_df = pd.DataFrame(new_station_tuples, columns=["line_num", "station_nm"])
    new_df["cx"] = None
    new_df["cy"] = None
```

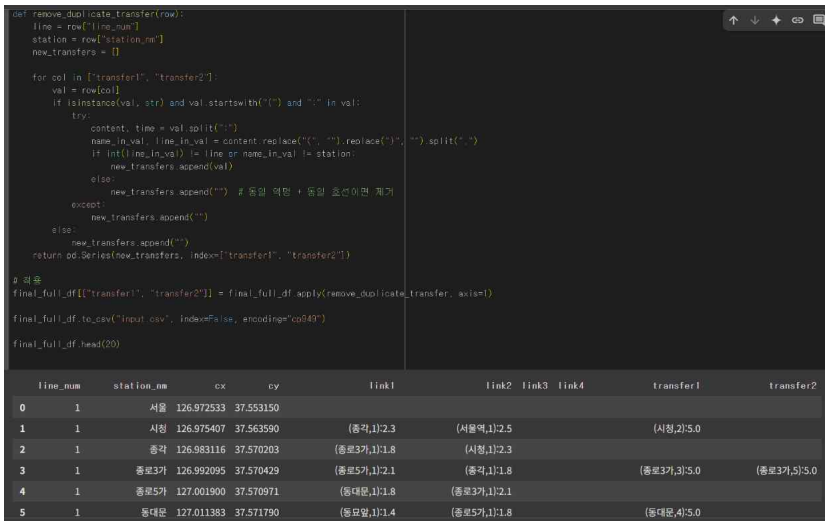
```python
# 위경도 알 수 없는 경우 NaN 처리
new_df = pd.DataFrame(new_station_tuples, columns=["line_num", "station_nm"])
new_df["cx"] = None
new_df["cy"] = None
for col in ["link1", "link2", "link3", "link4", "transfer1", "transfer2"]:
    new_df[col] = ""

# 병합
updated_df = pd.concat([final_full_df, new_df], ignore_index=True)

updated_df.to_csv("input.csv", index=False, encoding="cp949")
updated_df.head(20)
```


<ipython-input-182-595c43b63b8b>:44: FutureWarning: The behavior of DataFrame concatenation with empty or all-NA entries is deprecated. In a future version, this will n
  updated_df = pd.concat([final_full_df, new_df], ignore_index=True)

	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	(시청,1):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	(동대문,1):5.0
6	1	동묘앞	127.016459	37.573265	(신설동,1):1.6	(동대문,1):1.4			(동묘앞,6):5.0	(동묘앞,1):5.0
7	1	신설동	127.024710	37.576117	(회기,1):2.1	(동묘앞,1):1.6			(신설동,2):5.0	
8	1	제기동	127.034902	37.578116	(외대앞,1):2.3	(회기,1):3.2				
9	1	청량리	127.045063	37.580148					(청량리,1):5.0	
10	2	을지로입구	126.982569	37.565998	(시청,2):1.2	(을지로3가,2):1.4				
11	2	을지로3가	126.991773	37.566292	(을지로입구,2):1.4	(을지로4가,2):1.1			(을지로3가,3):5.0	(을지로3가,2):5.0
12	2	을지로4가	126.998122	37.566611	(을지로3가,2):1.1	(동대문역사문화공원,2):1.8			(을지로4가,5):5.0	(을지로4가,2):5.0

국가철도공단의 역간 거리 파일을 기준으로 기존 서울교통공사 좌표 파일에 포함되지 않은 역들을 식별하여 input.csv파일에 보완하여 추가했음. 위경도 및 연결 정보가 없을 경우 해당 항목은 비워두되, 기존 노선을 유지하기 위해(완전성 확보) 역명과 호선 정보는 그대로 유지했음. (파일 내 "기점", "종점" 등의 column에서 역명을 수집)

10. 최종 input 파일 데이터 정리 - 동일 역 + 동일 호선 환승 제거



```
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	

환승 정보 중 (역명,호선):시간 포맷에서 실제 환승이 아닌 자기자신(본인 호선, line_num)에 대한 환승 정보가 잘못 포함된 경우를 정제함.

문자열에서 역명과 호선을 파싱하여 현재 row의 station_nm과 line_num을 비교하고 역명이 다르거나 호선이 다르면 환승 정보 유지, 역명과 호선이 모두 동일하다면 환승이 아닌 단순 반복이기에 제거 처리(“”)함.

이러한 방식으로 데이터 수집 & 정제를 진행하였으며, 누락된 정보나 잘못된 정보가 발견되면 그 셀은 수동으로 수정함. 또한 정확도를 위해 네이버 지도나 카카오맵, 구글맵을 참고하여 일부 셀 수동으로 수정함.

수정한 최종 데이터 파일은 input2로 저장함.

2. Programming

I. 지하철 경로 탐색 알고리즘 파트 정리

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)

주요 알고리즘

Dijkstra Algorithm: 최단 시간 경로 탐색을 위한 기본 알고리즘. heapq를 이용하여 시간 복잡도 $O((V + E) \log V)$를 만족하도록 구현

Modified Dijkstra: 환승 횟수를 고려한 우선 탐색. key를 (노드, 호선), value를 (환승 횟수, 소요 시간)으로 설정하여 방문 체크의 granularity를 세분화함.

Floyd-Warshall Algorithm: 모든 노드 쌍 간 최단 경로를 미리 계산. 이후 $O(1)$에 가까운 쿼리 응답이 가능하며, 반복 쿼리 상황에서 유리

Y환승 최적화: 물리적 환승이 필요한 동일역(금천구청, 병점, 구로, 강동 등)의 비효율적인 경로를 탐지하여 중간 노드 제거 및 직접 연결 경로로 최적화

경로 복원: Floyd-Warshall에서는 next_node 행렬을, Dijkstra에서는 prev_route 딕셔너리를 따라가며 경로를 구성

folium을 활용한 시각화: 노선별 색상 및 마커, 환승 지점 등을 시각적으로 표현

II. Pseudo-code & Flowchart

1) Psudo-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 = {}
edges = []

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

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 or 최소 환승 알고리즘 실행

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

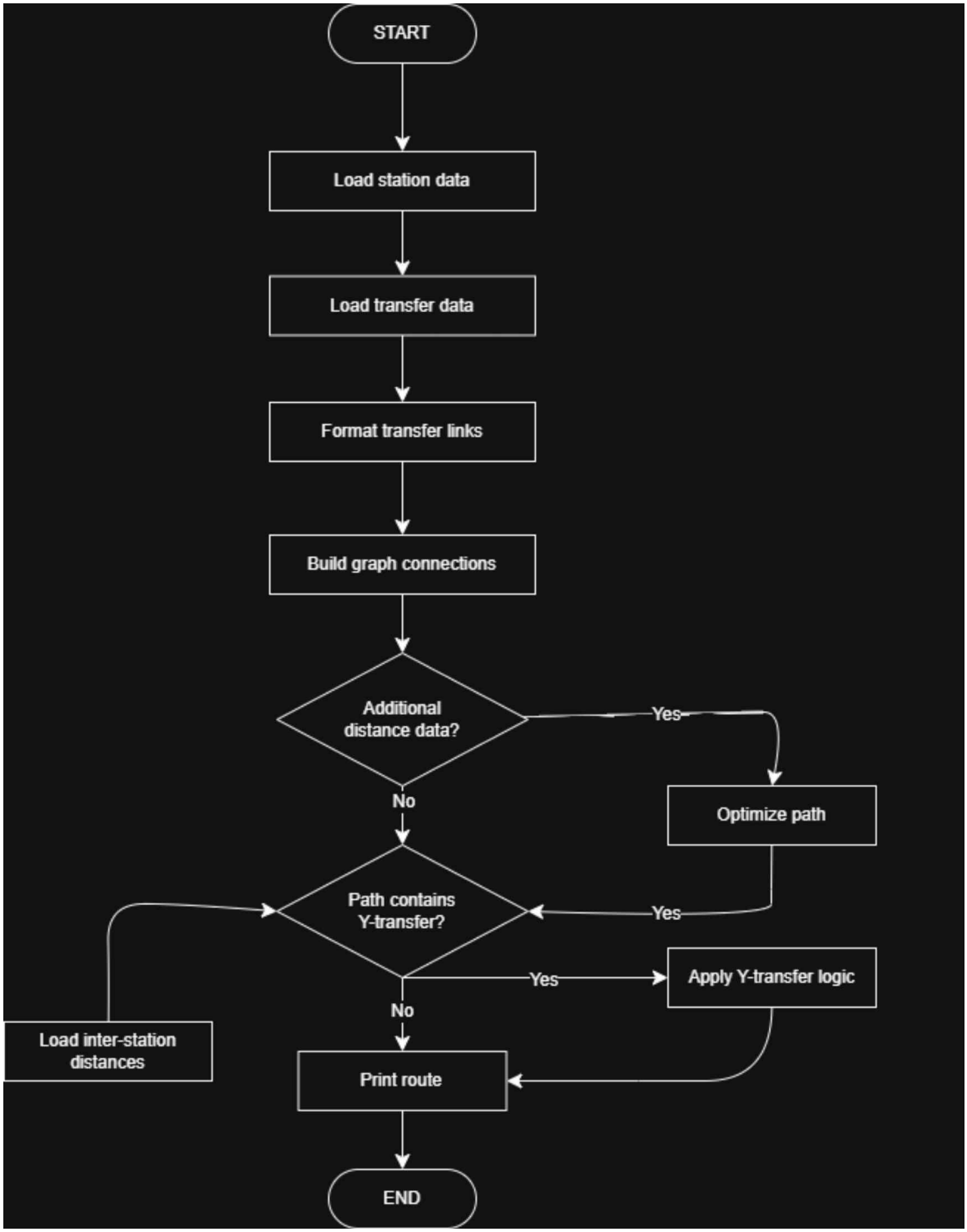
Y자 환승 최적화
for 경로 중간 역에 대해:
동일 역명 + 반대 방향 환승이 있으면:
중간 역 제거
시간 최적화 적용

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:
마커 표시 (시작역, 도착역 구분)
호선마다 색상 다르게 라인 연결

프로그램 종료

2) Flowchart


START
Load station data
Load transfer data
Format transfer links
Build graph connections
Additional distance data?
Yes
No
Optimize path
Path contains Y-transfer?
Yes
Yes
Apply Y-transfer logic
No
Load inter-station distances
Print route
END

III. 지하철 경로 탐색 알고리즘 로직

(셀 저장을 utf-8로 변경 후 utf-8로 encoding)

1. 그래프 구성 과정

```
[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)))
```

역명과 호선을 조합해 '역명_호선'형태의 노드를 생성하고 연결 정보 (link, transfer)를 파싱하여 인접 리스트 기반의 graph를 구성함. 또한 각 역의 위도, 경도 정보를 추출해 station_coords에 저장하여 시각화 단계에서 활용할 수 있도록 설정함.

파싱된 연결 정보는 (출발노드, 도착노드, 소요시간) 형태로 edges list에 저장되어 보다 전체 네트워크 구조를 다루기 수월하게 함.

1-2. 연결 정보와 환승 정보 기반 그래프 구조 구성

```python
# preprocessing 함수를 실행하는 코드 블럭

graph = {}          # 인접 리스트 구현을 위해서 dict 사용
nodes = set()       # 중복 허용을 막기 위해서
edges = []          # 간선 담을 리스트
station_coords = {} # 시각화 할 상대좌표 받기

link_cols = [f'link{i}' for i in range(1, 4)]
transfer_cols = [f'transfer{i}' for i in range(1, 3)]

for idx, row in df.iterrows():
    preprocessing(row, link_cols, graph, nodes, edges, station_coords)
    preprocessing(row, transfer_cols, graph, nodes, edges, station_coords)


# 리스트로 변환
nodes = list(nodes)   # 다루기 쉽게 리스트로 변환
```

지하철 그래프 자료구조 초기화 과정임.

link 1~3, transfer 1~2 열 이름을 리스트로 정의한 뒤 각 행에 대해 preprocessing()을 두 번 호출하여 연결 정보와 환승 정보를 모두 파싱함. 이 때 중복된 node는 set()을 통해 제거되고, 마지막에 리스트로 변환됨.

2. 실제 지도에 노선 그려 시각화 구현(folium 활용)

```python
# 직접 실제 지도에 노선 그려서 시각화

def plot_path_folium(path, station_coords, zoom_start=13):
    line_colors = {"1": "blue", "2": "green", "3": "orange", "4": "#00a2e8", "5": "purple"} # #00a2e8은 css 색상 코드임 : folium에서 지원하는 lightblue가 너무 연함
    latlons = [station_coords[station] for station in path if station in station_coords]
    start_lat, start_lon = latlons[0]
    m = folium.Map(location=[start_lat, start_lon], zoom_start=zoom_start)

    # 역마다 마커 및 CircleMarker(호선 색상) 추가
    for station in path:
        if station in station_coords:
            lat, lon = station_coords[station]
            _, line = station.split("_", 1)
            color = line_colors.get(line, "blue")
            folium.Marker(
                [lat, lon],
                popup=station,
                tooltip=station,
                icon=folium.Icon(color='red' if station == path[0] or station == path[-1] else 'blue')
            ).add_to(m)
            folium.CircleMarker(
                location=[lat, lon],
                radius=5,
                color=color,
                fill=True,
                fill_color=color,
                fill_opacity=0.7
            ).add_to(m)

    # 호선이 바뀌는 구간마다 PolyLine을 따로 그림
    sub_path = [latlons[0]]
    prev_line = path[0].split("_", 1)[1]
    for idx, station in enumerate(path[1:], 1):
        if station in station_coords:
            curr_line = station.split("_", 1)[1]
            if curr_line != prev_line:
                folium.PolyLine(sub_path, color=line_colors.get(prev_line, "blue"), weight=5, opacity=0.7).add_to(m)
                sub_path = []
            sub_path.append(station_coords[station])
```

```
    # 호선이 바뀌는 구간마다 PolyLine을 따로 그림
    sub_path = [latlons[0]]
    prev_line = path[0].split("_", 1)[1]
    for idx, station in enumerate(path[1:], 1):
        if station in station_coords:
            curr_line = station.split("_", 1)[1]
            if curr_line != prev_line:
                folium.PolyLine(sub_path, color=line_colors.get(prev_line, "blue"), weight=5, opacity=0.7).add_to(m)
                sub_path = []
            sub_path.append(station_coords[station])
            prev_line = curr_line
    # 마지막 구간 PolyLine 그리기
    if sub_path:
        folium.PolyLine(sub_path, color=line_colors.get(prev_line, "blue"), weight=5, opacity=0.7).add_to(m)

    return m
```

각 지하철 호선에 맞는 색상(line_colors)을 지정한 후, 출발점 좌표를 중심으로 Folium 지도를 초기화함. 각 지하철 역마다 Marker와 CircleMarker를 지도에 추가했으며, 출발지와 도착지는 빨간 아이콘으로 구분됨.

경로 중간에서 호선이 바뀌는 지점마다 다른 색상의 PolyLine을 사용하여 한 눈에 구분 가능하도록 함.

3. 사용자로부터 출발역/도착역 입력 받기

```
# 사용자로부터 입력받는 부분
# 사용자가 잘못된 입력을 할 수 있으므로 예외처리

def get_valid_node(prompt, nodes):
    while True:
        try:
            print(prompt)
            name, num = input().split()
            node = f"{str(name).strip()}_{str(num).strip()}"       # 공백, 개행, 타입 불일치 등 예방
            if node not in nodes:
                print(f"'{name} {num}'은(는) 존재하지 않는 역/호선입니다. 다시 입력해주세요.")
                continue
            return node
        except ValueError:
            print("입력 형식이 잘못되었습니다. 예시처럼 '서울역 1'과 같이 입력해주세요.")
```

입력값은 역명과 호선번호로 구성되고, 역명_호선 형태의 문자열로 통합되어 내부 그래프에서 사용되는 노드 명칭과 일치시킴.

존재하지 않는 역이나 호선을 입력할 경우 안내 메세지를 출력한 후 재입력을 유도하며 입력형식이 잘못된 경우(ValueError)도 예외처리를 통해 대응함.

4. 경로 복원 - 최단 경로 추적을 통해

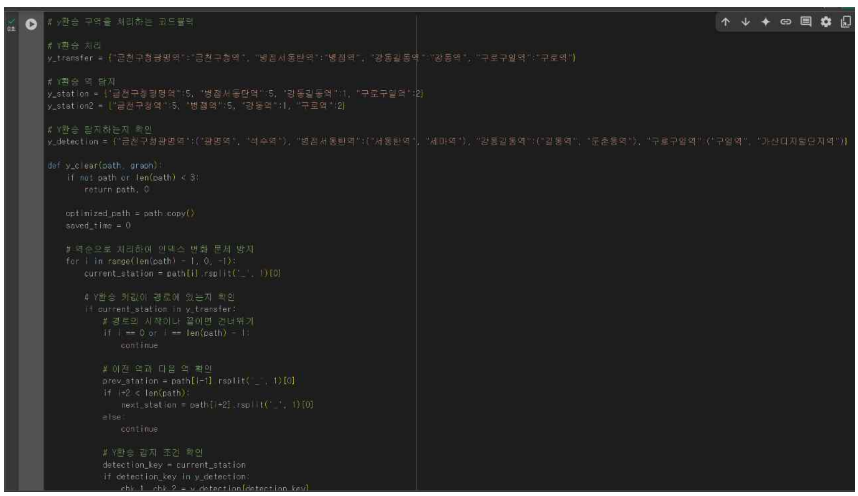
```python
# 경로 복원하기 위한 함수

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
```

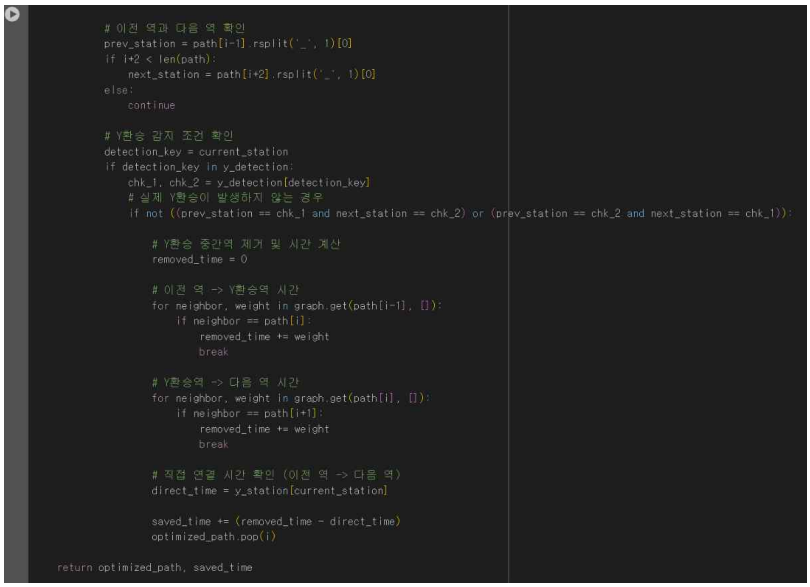
최단 경로 알고리즘의 결과를 바탕으로 실제 이동 경로를 문자열 list 형태로 복원해주는 함수(reconstruct_path()) 생성

path_tracker가 dictionary 형식이면 다익스트라 알고리즘의 결과로 간주하고, 역추적 방식으로 end --> start로 거슬러 올라간 후 list를 반전시킴.

반면, 2차원 list(next_node)의 형식인 경우, 플로이드-워셜 알고리즘의 결과로 간주하며 index를 기준으로 순차적으로 경로를 구성함.

5. Y자 환승 구간 최적화 알고리즘





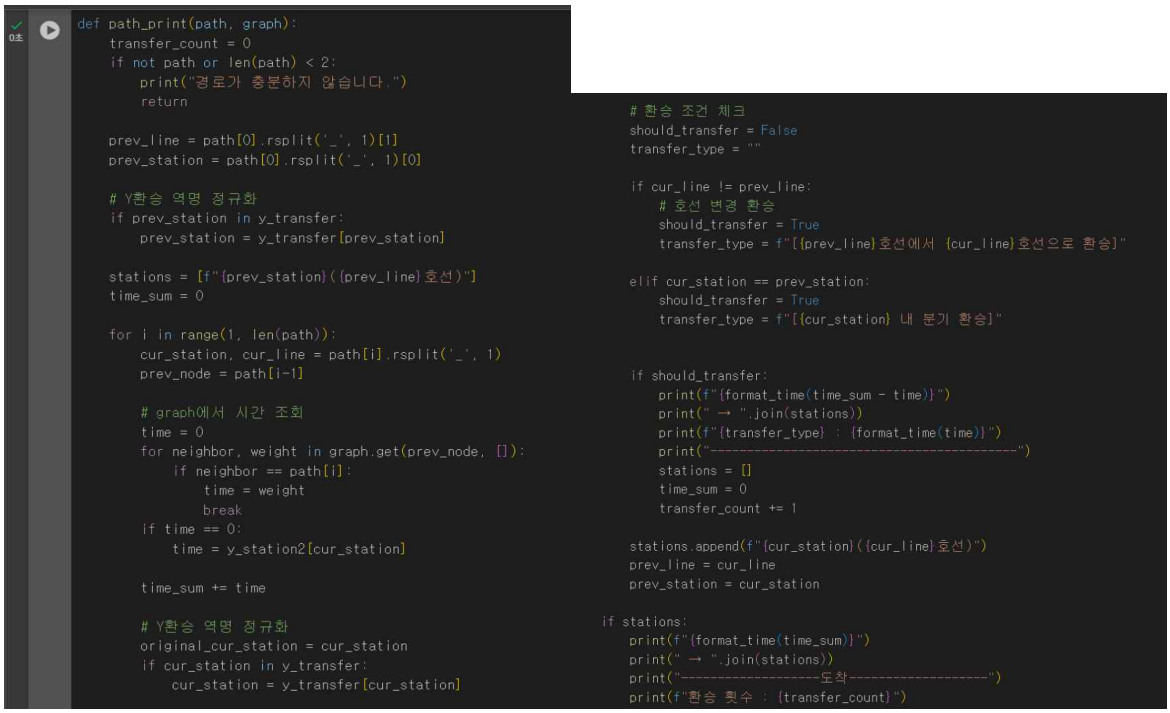
```
            # 이전 역과 다음 역 확인
            prev_station = path[i-1].rsplit('_', 1)[0]
            if i+2 < len(path):
                next_station = path[i+2].rsplit('_', 1)[0]
            else:
                continue

            # 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
```

서울 지하철 경로 탐색 중, 같은 호선 내에서 반대방향으로 갈아타야 이동할 수 있는 특수한 역이 존재함을 발견함. 1호선 금천구청역, 병점역, 구로역, 5호선 강동역이 이에 해당함.

Y환승을 같은 역 이름이지만 승강장 위치가 다른 경우(이동 시 반대 방향 열차를 타야 하는 경우)라고 정의함.

전체적인 로직은 다음과 같음. 문제가 발생하는 Y환승 구조를 정의하고 경로 탐색 결과에서 중복된 환승 구간을 탐지함. 중복 경로가 시간적으로 손해인 경우, 환승역을 제거하고 직접 연결 시간으로 대체함.

6. 환승여부, 소요시간 등 전체 이동 경로 시각화



```
def path_print(path, graph):
    transfer_count = 0
    if not path or len(path) < 2:
        print("경로가 충분하지 않습니다.")
        return

    prev_line = path[0].rsplit('_', 1)[1]
    prev_station = path[0].rsplit('_', 1)[0]

    # Y환승 역명 정규화
    if prev_station in y_transfer:
        prev_station = y_transfer[prev_station]

    stations = [f"{prev_station}({prev_line}호선)"]
    time_sum = 0

    for i in range(1, len(path)):
        cur_station, cur_line = path[i].rsplit('_', 1)
        prev_node = path[i-1]

        # graph에서 시간 조회
        time = 0
        for neighbor, weight in graph.get(prev_node, []):
            if neighbor == path[i]:
                time = weight
                break
        if time == 0:
            time = y_station2[cur_station]

        time_sum += time

        # Y환승 역명 정규화
        original_cur_station = cur_station
        if cur_station in y_transfer:
            cur_station = y_transfer[cur_station]

        # 환승 조건 체크
        should_transfer = False
        transfer_type = ""

        if cur_line != prev_line:
            # 호선 변경 환승
            should_transfer = True
            transfer_type = f"[{prev_line}호선에서 {cur_line}호선으로 환승]"

        elif cur_station == prev_station:
            should_transfer = True
            transfer_type = f"[{cur_station} 내 분기 환승]"


        if should_transfer:
            print(f"{format_time(time_sum - time)}")
            print(" → ".join(stations))
            print(f"{transfer_type} : {format_time(time)}")
            print("-----------------------------------------")
            stations = []
            time_sum = 0
            transfer_count += 1

        stations.append(f"{cur_station}({cur_line}호선)")
        prev_line = cur_line
        prev_station = cur_station

    if stations:
        print(f"{format_time(time_sum)}")
        print(" → ".join(stations))
        print("-------------------도착-------------------")
        print(f"환승 횟수 : {transfer_count}")
```

y_transfer에 따라 분기 환승 가능한 역이면 이름을 정규화하고, 현재깢 이어진 경로를 저장하는 stations list, 환승하지 않고 이어진 이동시간을 합산하는 time_sum을 설정함.

graph에서 이전역 --> 현재역 소요시간을 조회하고, 없다면 y_station2의 직접 연결 시간 값을 사용함.

cur_line != prev_line: 일반적인 호선 간 환승

cur_station == prev_station: Y자 환승 감지

따라서 출력 형태는

00:05

서울역(1호선) → 시청(1호선)

[1호선에서 2호선으로 환승] : 02:00

02:30

을지로입구(2호선) → 을지로3가(2호선) → 을지로4가(2호선)

-------------------도착-------------------

환승 횟수 : 1

이런식으로 진행됨.

7. 총 소요시간을 분, 초, 시간 단위로 환산(문자열 형태로 반환)

```python
# 소요 시간을 시간, 분, 초 단위로 환산 후 출력

def format_time(total_time):
    hours = int(total_time // 60)
    minutes = int(total_time % 60)
    seconds = int(round((total_time - int(total_time)) * 60))

    # 초가 60이 되는 경우(반올림 등) 보정
    if seconds == 60:
        seconds = 0
        minutes += 1
    if minutes == 60:
        minutes = 0
        hours += 1

    result = []
    if hours > 0:
        result.append(f"{hours}시간")
    if minutes > 0:
        result.append(f"{minutes}분")
    if seconds > 0:
        result.append(f"{seconds}초")

    # 만약 모두 0이면 0초 출력
    if not result:
        return "0초"
    return " ".join(result)
```

8. 최소 환승 알고리즘

```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

        # 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, []
```

min_transfer_and_time: 시작역에서 도착역까지 경로 중 환승 횟수와 이동 시간을 동시에 최소화하는 경로를 탐색함.

station_to_nodes = defaultdict(list)는 같은 역 이름을 갖고 있지만 다른 호선에 존재하는 node들을 mapping함.

초기 우선순위 Queue 같은 경우, 같은 역 이름과 출발 호선에 속하는 node들의 가중치를 0으로 초기화하고 Y환승이 가능한 경우라면, 시작 node와 다른 node간 환승 소요 시간을 탐색해 경로 후보로 추가함.

heapq.heappush(heap, (transfer_cnt, total_time, current, cur_line, path)) : 다익스트라로 환승 횟수 우선 탐색을 진행하는 코드임.

heap에는 (환승횟수, 소요시간, 현재노드, 현재호선, 경로리스트) 튜플리 저장되고, 파이썬의 튜플 비교는 사전순이기에 자동으로 환승이 적고 시간도 짧은 경로가 우선적으로 선택됨.

또한 visited[(current, cur_line)] = (transfer_cnt, total_time)을 통해 동일 노드와 호선 조합에 대해 더 나은 경로(환승 수나 시간 적게 걸리는 경우)가 이미 존재하면 가지치기함.

if neighbor_line != cur_line: (엔터 생략) transfer_cnt + 1을 통해 인접한 노드가 다른 호선일 경우 환승 횟수를 증가시켜 환승 처리함.

8. 최단경로 탐색 방법 1 - 다익스트라 알고리즘

```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
```

우선순위 큐 heapq를 활용하여 누적 시간이 가장 짧은 노드를 우선적으로 탐색하며, 각 노드별 누적 시간은 distances딕셔너리에 저장됨. 또한 경로 복원을 위해 prev_route딕셔너리를 별도로 관리하여, 도착 노드에서 역추적이 가능하도록 구성함. 이 방식은 간선 수(E), 노드 수(V)에 따라 O((E + V) log V)의 시간 복잡도를 가지며, 쿼리가 적고 사전 계산이 불필요한 지하철 경로 탐색에 적합함.

주요 로직은 초기화, 탐색반복, 종료조건 정도로 이야기할 수 있음.
모든 node에 대해 시간은 무한대로 초기화하고 시작 node의 시간은 0으로 설정함.
현재 node의 누적 시간이 이미 기록된 거리보다 크다면 더 나은 경로가 있기에 무시하고, 그렇지 않다면 이웃 node들을 검사하고 더 짧은 경로가 발견되면 갱신함.
우선순위 Queue가 빌 때까지 반복하면 됨.

9. 최단경로 탐색 방법 2 - 플로이드 워셜

```
[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
```

지하철 노드 간 최단 거리와 경로 정보를 미리 계산해두기 위한 플로이드-워셜(Floyd-Warshall)알고리즘을 구현한 것임. 모든 노드 쌍 간의 거리 행렬 dist와 다음 경유지를 기록한 next_node행렬을 기반으로, 중간 노드를 반복적으로 거쳐가며 최소 이동 시간을 계산한다. **연산 복잡도는 O(N^3)**이며, 초기에 계산 비용은 높지만, 이후 특정 노드 간 최단 경로를 매우 빠르게 질의할 수 있다는 장점이 있음. 따라서 경로 탐색 요청이 빈번한 경우, 한 번의 사전 계산을 통해 실시간 경로 탐색 성능을 높일 수 있음.

주요 로직은 초기화, 간선 입력 처리, 3중 루프 정도로 이야기할 수 있음.
node 수 N만큼 거리 행렬 dist, 경로 행렬 next_node를 생성하고, 자기자신으로 가는 거리 dist[i][i] = 0으로 초기화함. 이후 주어진 edges list를 통해 초기 거리와 연결 정보를 설정하고, 모든 노드 쌍에 대해 중간 node k를 거쳤을 때 더 짧은 경로가 있으면 갱신함. 이 과정에서 거리와 경로 둘 다 함께 업데이트함.

10. 사용자 입력(출발역, 도착역)

```
[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):
```

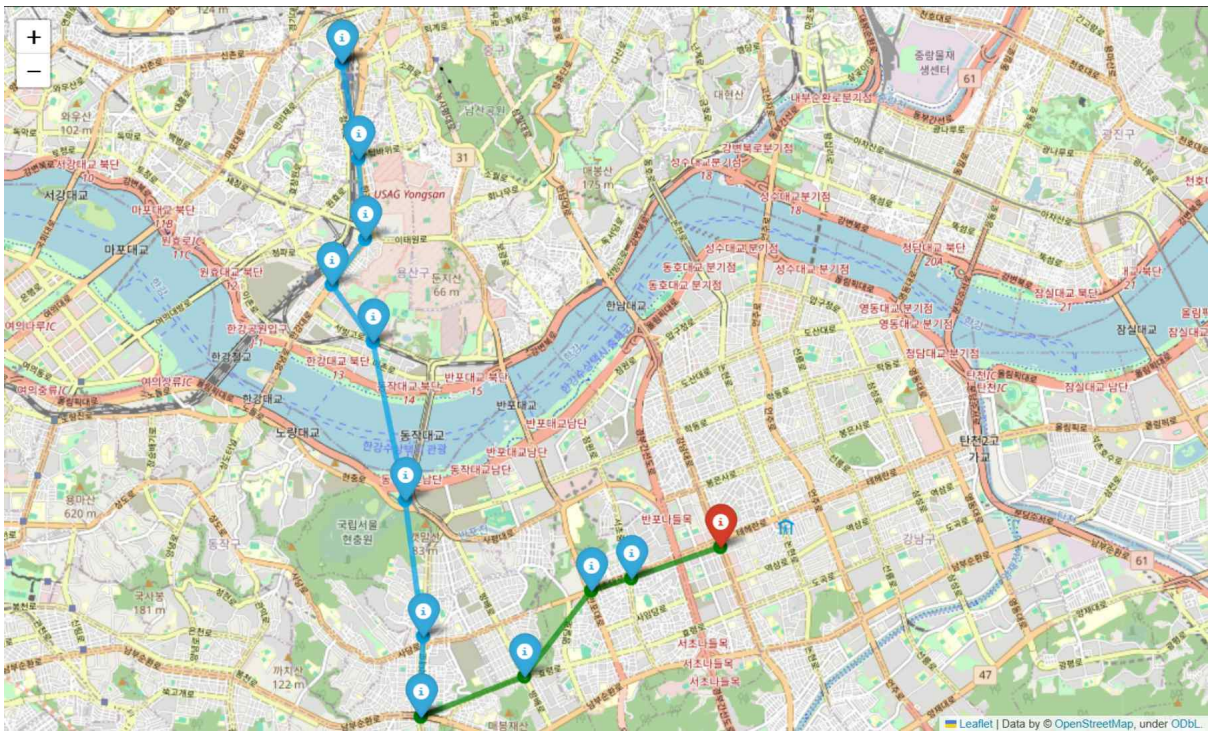
```
출발 역과 호선을 입력해주세요 (예: 서울역 1):
서울역 1
도착 역과 호선을 입력해주세요 (예: 강남역 2):
강남역2
입력 형식이 잘못되었습니다. 예시처럼 '서울역 1'과 같이 입력해주세요.
도착 역과 호선을 입력해주세요 (예: 강남역 2):
강남역 2
```

출발 역과 도착 역의 호선을 잘못 입력했을 시 입력 형식이 잘못되었다는 문구와 함께 다시 사용자 입력으로 돌아감.

11.1. 다익스트라 실행 결과 출력

```python
# time.time()은 초 단위(소수점 6자리 정도)로 측정되기 때문에 수 ms 이하는 부정확
# time.perf_counter()는 보다 정밀하기 ns까지 측정 가능

start_time = time.perf_counter()                                    # 시간 측정 시작
total_time, prev_route = dijkstra(graph, start_node, end_node)
end_time = time.perf_counter()                                      # 시간 측정 종료
path = reconstruct_path(start_node, end_node, prev_route)

optimized_path1, saved_time = y_clear(path, graph)
print("-----------------최단경로-----------------")
print("총 소요 시간:", format_time(total_time - saved_time))
print("-------------------출발-------------------")
path_print(optimized_path1, graph)
print()


min_transfers, total_time, transfer_path = min_transfer_and_time(start_node, end_node, graph, nodes)
optimized_path2, saved_time = y_clear(transfer_path, graph)
print("-----------------최소환승-----------------")
print(f"최소 환승 경로 소요 시간: {format_time(total_time-saved_time)}")
print("-------------------출발-------------------")
path_print(optimized_path2, graph)
print()

m = plot_path_folium(optimized_path1, station_coords)
m
```

```
------------------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
```



지나는 역 정보, 최단 경로와 총 소요시간, 환승 구역 및 환승 정보 및 환승 소요 시간, 최소환승 경로와 총 소요시간, 환승 횟수 전부 다 잘 출력되고 있음.

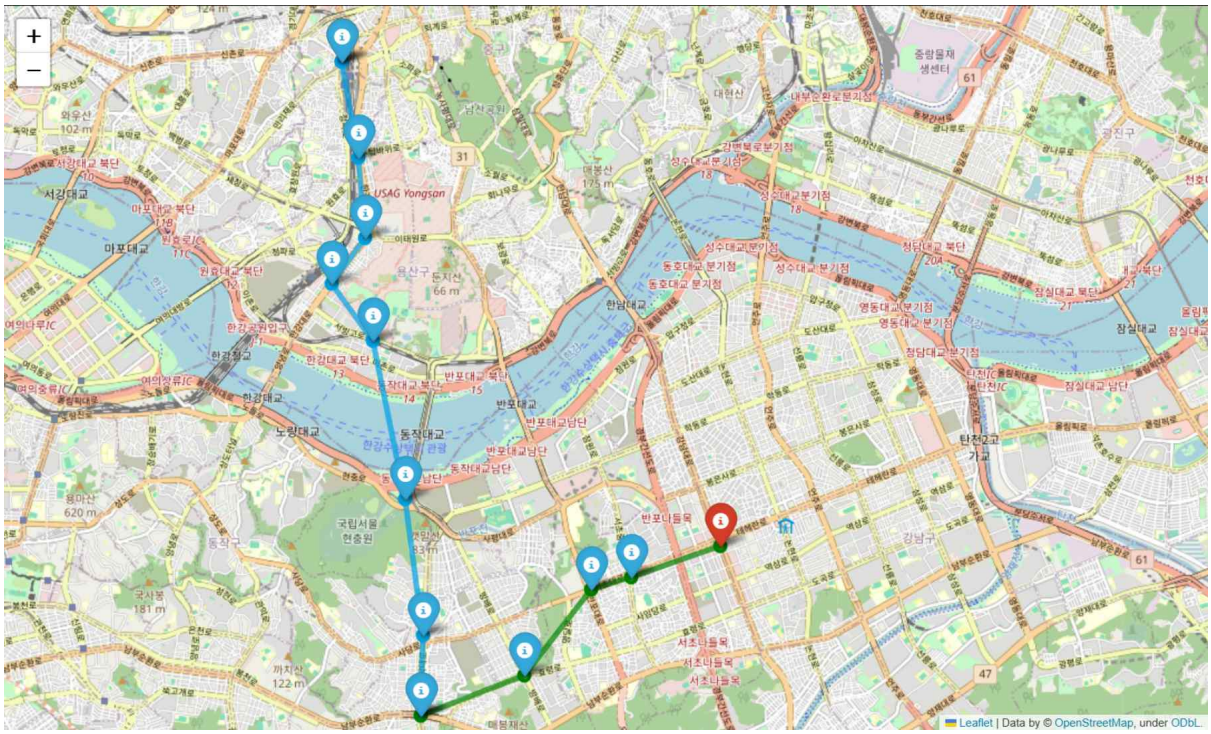
또한 지도를 보면, 각 호선 별로 다른 색을 갖고 있고, 어디에서 환승해서 어디로 가는지 쉽게 확인할 수 있음. 지도는 최단 경로 기준임.

```
[38]  # 다익스트라 실행 소요 시간

      dijkstra_time = end_time - start_time
      print(f"다익스트라 실행 시간: {dijkstra_time:.10f}초")

      다익스트라 실행 시간: 0.0011224310초
```

11.2. 플로이드-워셜 실행 결과 출력

```
start_time = time.perf_counter()
dist, next_node, node_indices = floyd_warshall(nodes, edges)
end_time = time.perf_counter()
total_time = dist[node_indices[start_node]][node_indices[end_node]]
path = reconstruct_path(start_node, end_node, next_node, nodes, node_indices)

optimized_path1, saved_time = y_clear(path, graph)
print("-----------------최단경로-----------------")
print("총 소요 시간:", format_time(total_time - saved_time))
print("-------------------출발-------------------")
path_print(optimized_path1, graph)
print()

min_transfers, total_time, transfer_path = min_transfer_and_time(start_node, end_node, graph, nodes)
optimized_path2, saved_time = y_clear(transfer_path, graph)
print("-----------------최소환승-----------------")
print(f"최소 환승 경로 소요 시간: {format_time(total_time-saved_time)}")
print("-------------------출발-------------------")
path_print(optimized_path2, graph)
print()

m = plot_path_folium(optimized_path1, station_coords)
m
```

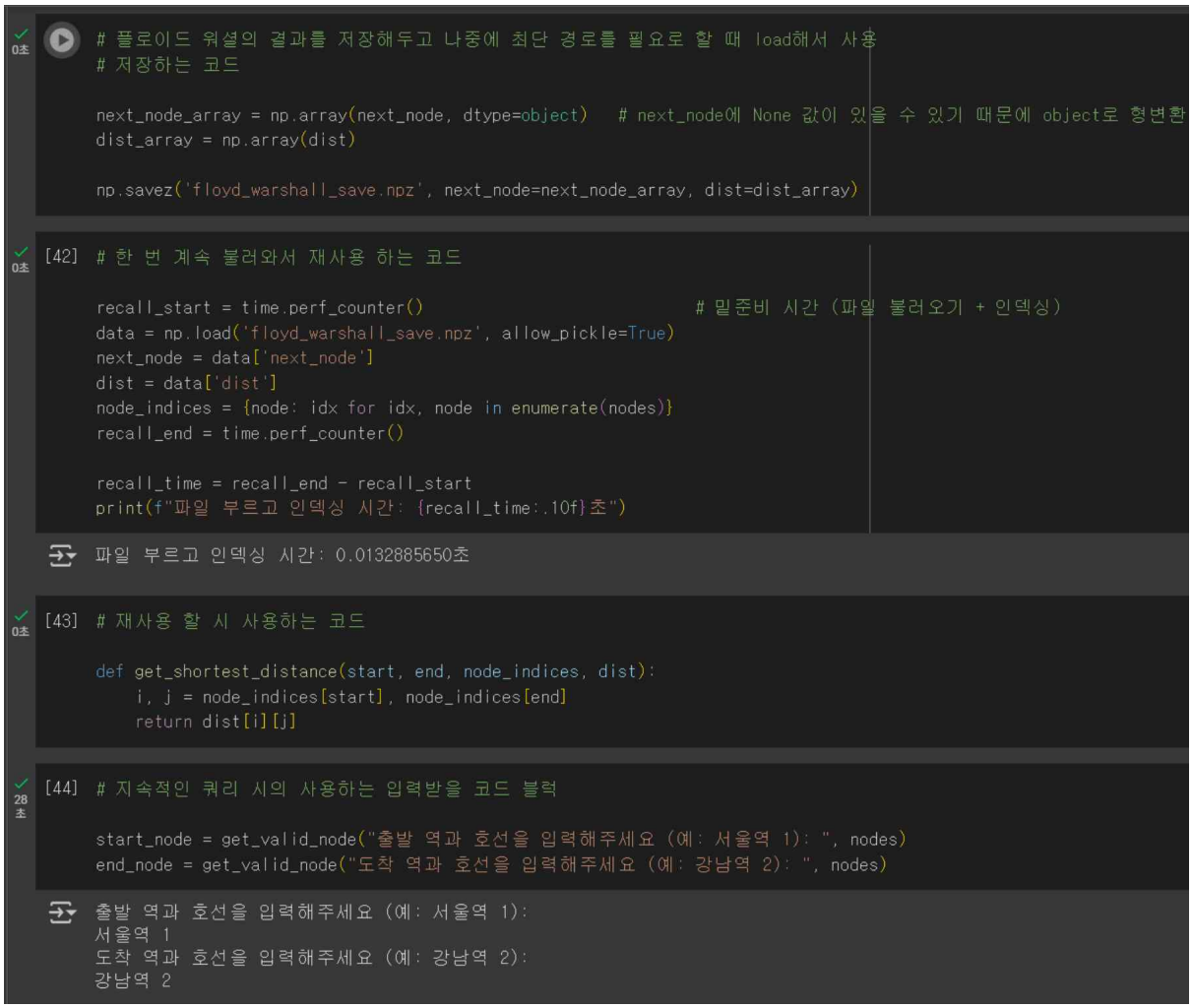
```
-----------------최단경로-----------------
총 소요 시간: 25분
-------------------출발-------------------
0초
서울역(1호선)
[1호선에서 4호선으로 환승] : 2분 12초
------------------------------------------
13분
서울역(4호선) → 숙대입구역(4호선) → 삼각지역(4호선) → 신용산역(4호선) → 이촌역(4호선) → 동작역(4호선) → 총신대입구역(4호선) → 사당역(4호선)
[4호선에서 2호선으로 환승] : 1분
------------------------------------------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




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

```
플로이드-워셜 실행 시간: 4.0921214510초
```

12. 플로이드-워셜 결과 저장 & 재사용



```
# 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)
```

```
[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
```

```
[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]
```

```
[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
```

플로이드-워셜 알고리즘의 계산 결과(dist, next_node)를 파일로 저장하고, 이후 동일한 쿼리를 빠르게 처리하기 위해 메모리로 불러오는 구조임. 이 구조는 계산 시간과 resource를 절약하는 데 유용함. 플로이드-워셜 알고리즘의 연산 결과를 .npz포맷으로 저장하고, 이후 반복적인 질의 처리 시 재사용할 수 있도록 구성됨. 초기 계산 시 생성한 거리 행렬(dist)과 경유지 정보(next_node)는 메모리 부담을 줄이기 위해 numpy 배열로 저장되며, 이후에는 np.load()를 통해 즉시 복원되어 빠른 질의 처리에 활용됨. 특히 쿼리의 수가 많고 실시간 응답이 요구되는 시스템에서 매우 효율적인 접근 방식임. 저장-복원 후 인덱싱 처리 시간은 약 0.013초 수준으로 측정되었으며, 이는 실용적인 응답 속도를 보장한다.

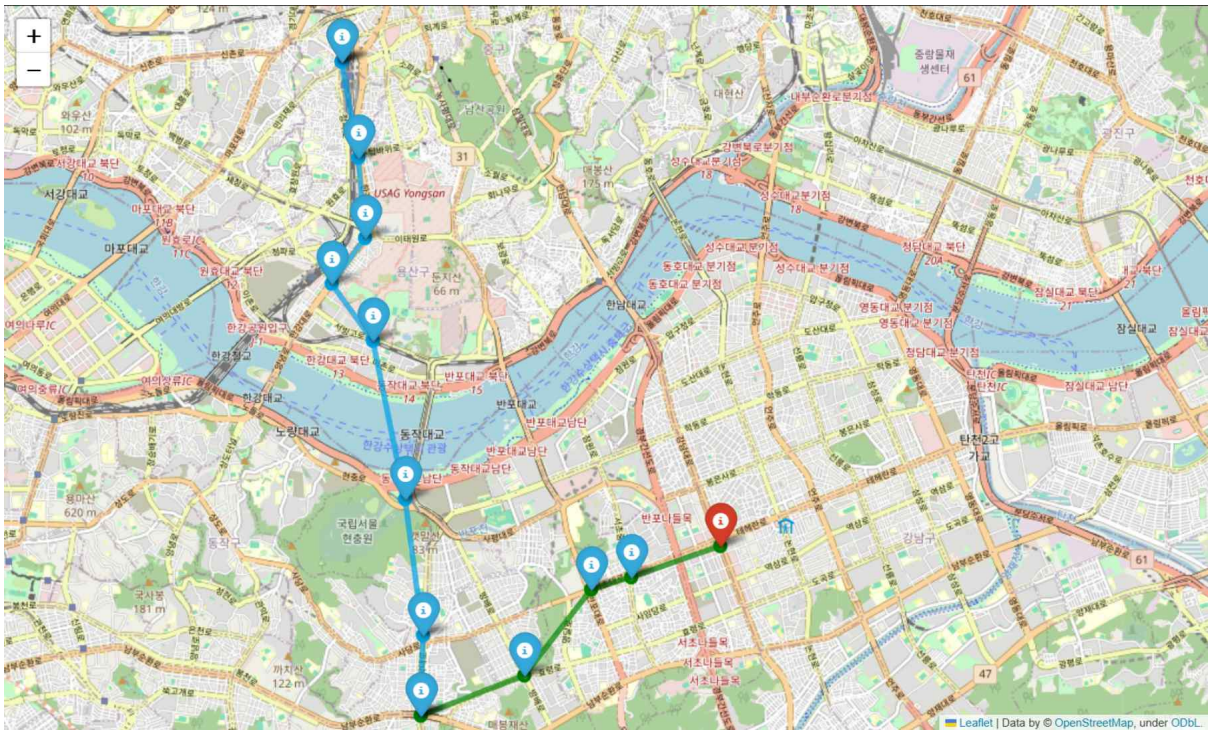
get_shortest_distance에서 return dist[i][j]를 함으로써 미리 계산된 거리 행렬에서 O(1)로 결과를 반환함.

```
[48]  # 비교를 위한 시간 처리
      # 최단 경로와 시간 계산

      reuse_start_time = time.perf_counter()                                    # 시간 측정 시작
      shortest_distance = get_shortest_distance(start_node, end_node, node_indices, dist)
      reuse_end_time = time.perf_counter()                                      # 시간 측정 종료

      path = reconstruct_path(start_node, end_node, next_node, nodes, node_indices)
      total_time = dist[node_indices[start_node]][node_indices[end_node]]
```

```
0초
      optimized_path, saved_time = y_clear(path, graph)
      print("-----------------최단경로-----------------")
      print("총 소요 시간:", format_time(total_time - saved_time))
      print("-------------------출발-------------------")
      path_print(optimized_path, graph)
      print()

      min_transfers, total_time, transfer_path = min_transfer_and_time(start_node, end_node, graph, nodes)
      optimized_path, saved_time = y_clear(transfer_path, graph)
      print("-----------------최소환승-----------------")
      print(f"최소 환승 경로 소요 시간: {format_time(total_time-saved_time)}")
      print("-------------------출발-------------------")
      path_print(optimized_path, graph)
      print()

      m = plot_path_folium(optimized_path, station_coords)
      m
```

```
-----------------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
```

```
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
```



다익스트라 1번 = 0.001s

플로이드-워셜 1번 = 4.0s

파일 불러오는 시간 = 0.01s

불러온 파일을 기반으로 최단거리를 계산하는 시간 = 0.0002s

3. Conclusion & Differentiator

- Conclusion

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 효율적이며,
Floyd-Warshall은 다중 쿼리 시 유리하지만 초기 계산량과 공간 부담이 큼.
Y환승 최적화 및 시각화는 전체 흐름상 시간에 큰 영향을 주지 않음.

- 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 참

고

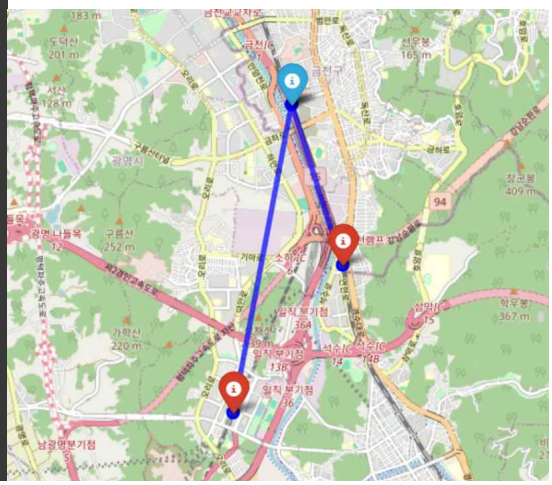
2. Floyd-Warshall 기반 이중 모드 지원
대부분의 경로 탐색기는 실시간 Dijkstra 또는 A* 알고리즘 기반이지만, 본 시스템은 선계산 후반복 쿼리 대응을 위한 Floyd-Warshall 기반도 지원함. 이를 통해 수십~수백건의 경로 탐색이 반복될 때 $O(1)$ 수준의 쿼리 응답이 가능함.
단건 실시간 탐색에는 Dijkstra 기반,
대규모 반복 탐색에는 Floyd-Warshall 기반 사전 계산 방식을 선택 할 수 있도록 함.
다시 말하자면, 원래는 실행 할 때마다 다익스트라 알고리즘을 계속 수행해야 하는데, 플로이드-워셜 알고리즘을 통해 미리 최단거리와 최단 경로를 파일로 저장한 후 새롭게 사용할때마다 파일을 불러와서 사용하는 방식으로 구현함. 처음에만 O(V^3)이지만 이후의 쿼리에는 배열 인덱싱만 하면 되므로 O(1)으로 매우 빨라짐. 또한 메모리 사용량은 증가하지만 속도가 빨라짐.(쿼리가 많을수록 유리함)

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 자료형 visited의 key를 (노드, 호선)으로 구성하고 value를 (환승 횟수, 소요 시간)으로 구성하여 불필요한 경로 재탐색을 방지를 통해 효율성을 높임.

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

++ Y 환승 최적화 로직 구현 방법
금천구청역을 예시로 들자면, 역을 2개로 나눠서 기존의 금천구청역, 그리고 새로운 임의의 역인 금천구청광명역을 생성함. 광명역-석수역인 경우 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
```



- 역할 분담

곽용진: 플로이드 워셜 알고리즘, y 환승 구역 최적화, 맵 시각화

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

크게 나누면 역할 분담은 이렇게 진행됐고, 서로 계속해서 피드백 하는 방식으로 진행함. 기존 틀을 각자 잡고, 같이 피드백하고 의견을 주고 받으며 구체화 시킴.