Dijkstra Algorithm과 Bellman-Ford Algorithm을 구현하여 최단 거리 도출 성능 분석: 청주 시내 경로 탐색을 바탕으로

[목차]

1. 서론

이 주제는 12주차에 김태준 교수님의 '컴퓨터 네트워크' 수업의 내용을 바탕으로 결정되었습니다. 12주차 수업에서는 Network Layer의 Control Plane 원칙을 배웠는데 그 중 Routing Protocol을 배웠습니다. 각 노드 사이의 비용을 바탕으로 최단 경로를 도출하는 Link state algorithm 중 'Dijkstra algorithm' 과 각 노드가 이웃 노드들과 비용 정보를 교환하며 최단 경로를 도출하는 Distance vector algorithm 중 'Bellman-Ford Algorithm' 이 그 내용이었습니다.

수업을 들을 당시 저는 이 내용에 대해 완벽히 이해를 하지 못하였습니다. 하지만 네트워크 분야와 인공지능 분야에 관심을 크게 가지고 있는 저에게 이번 미니 프로젝트가 기회라 생각되어 Python을 이용하여 이 두 개의 알고리즘을 실제로 구현해보고 다각도로 분석해보는 시간을 가져보았습니다. 이 과정을 통해 두 개의 알고리즘을 완벽히 이해하고자 합니다.

네트워크 형식의 그래프를 랜덤으로 생성한 후 직접 구현한 두 알고리즘을 적용하고자 했지만, 더 직관적이고 면을 생각하여 실제 우리가 흔히 사용하는 '지도' 내에서의 최단 경로를 찾는데 적용하였습니다.

충북대학교가 위치한 청주시의 도로망 데이터를 OpenStreetMap으로부터 추출한 후, 직접 구현한 'Dijkstra algorithm'과 'Bellman-Ford Algorithm'을 이용하여 두 지점간 최단 경로탐색 결과의 정확도와 효율성을 분석하는 것을 이번 미니 프로젝트의 목표로 정했습니다. 특히, 효율성 부분에서 결과를 도출하는데 소요되는 시간, 메모리 사용량을 비교해보고 외부 서비스(카카오맵, 네이버지도)와의 경로 결과도 비교해보았습니다.

- **개발 환경**
  - OS 환경: Ubuntu 24.04 on Windows 11 WSL2

```
(routing_algorithm) gyuha_lee@KONGSUNILAPTOP:~/Opensource_dev/routing_algorithm$ cat /etc/*release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=24.04
DISTRIB_CODENAME=noble
DISTRIB_DESCRIPTION="Ubuntu 24.04 LTS"
PRETTY_NAME="Ubuntu 24.04 LTS"
NAME="Ubuntu"
VERSION_ID="24.04"
VERSION="24.04 LTS (Noble Numbat)"
```

  - Python 버전: 3.9.20 with Conda Virtual Environment

```
(routing_algorithm) gyuha_lee@KONGSUNILAPTOP:~/Opensource_dev/routing_algorithm$ python
Python 3.9.20 (main, Oct  3 2024, 07:27:41)
[GCC 11.2.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

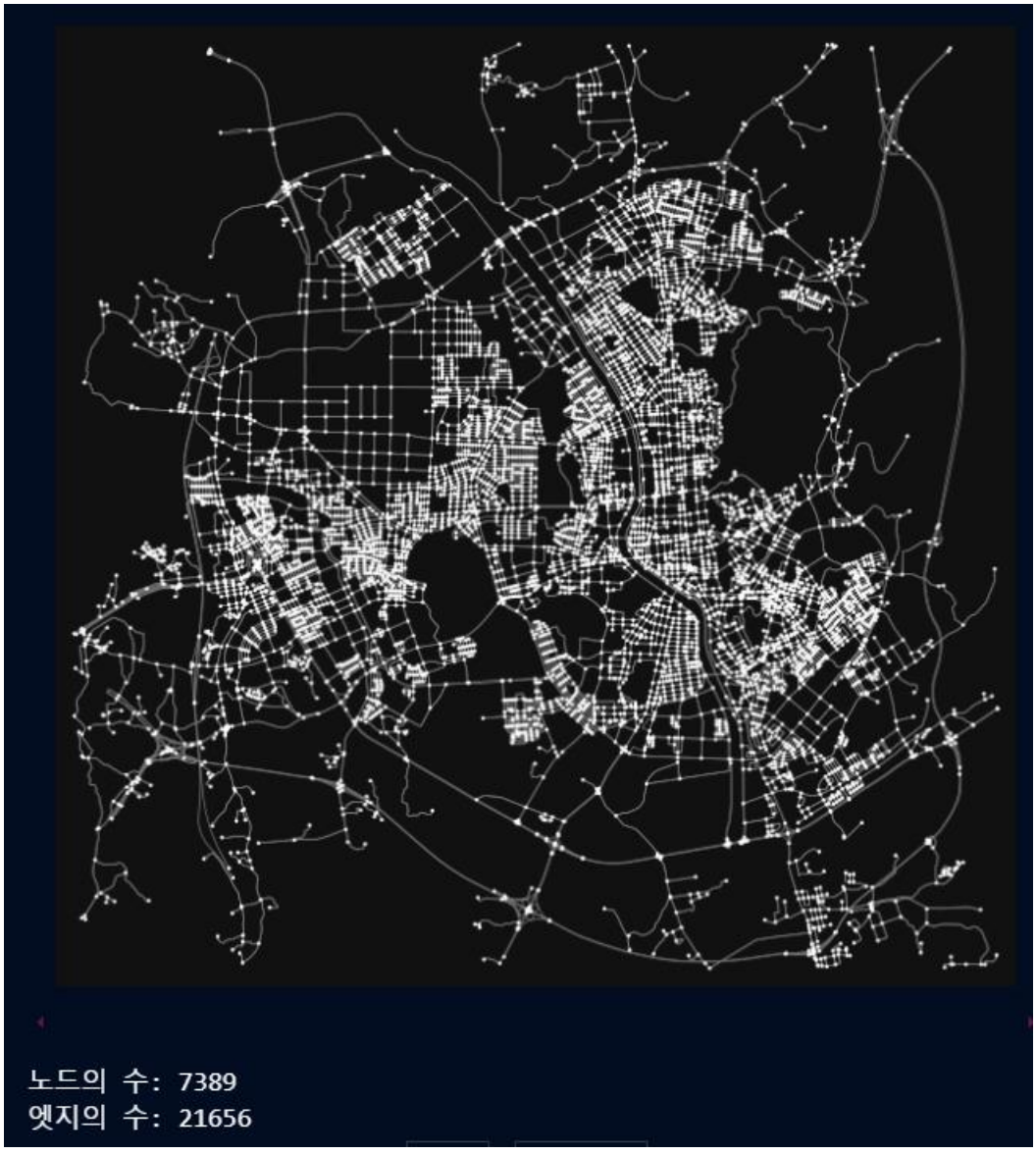
  - Conda env에서 사용된 패키지 목록: (따로 첨부한 spec-file.yaml에 기재됨)

2. 본론

[Step 1: 데이터 준비 및 네트워크 생성]

- **데이터 추출**: 충청북도 청주시의 시계탑오거리 좌표(위도 36.63547, 경도 127.46891)를 기준으로 반경 6000m 내의 도로망 데이터를 OpenStreetMap에서 추출했습니다. 운전 경로를 도출하는 가정하여 network_type="drive" 설정을 사용하여 운전용 네트워크를 구축하였습니다.
  청주 권역을 도보로만 움직이지 않을 것이라는 판단 하에 결정하였습니다.

- **네트워크 그래프 생성**: 추출한 데이터를 바탕으로 노드와 엣지로 구성된 도로 네트워크 그래프를 생성했습니다. 이 그래프는 각각 도로망의 교차로와 경로를 나타내며, 총 7389개의 노드와 21656개의 엣지를 포함하고 있습니다. 각 노드는 도로 네트워크의 지점을 나타내고, 엣지는 노드 간의 연결 정보(실제 도로의 정보)를 담고 있습니다.



- **데이터 구조 확인 및 검증**:
  - 구조: 생성된 데이터에는 '노드 데이터'와 '엣지 데이터'로 존재

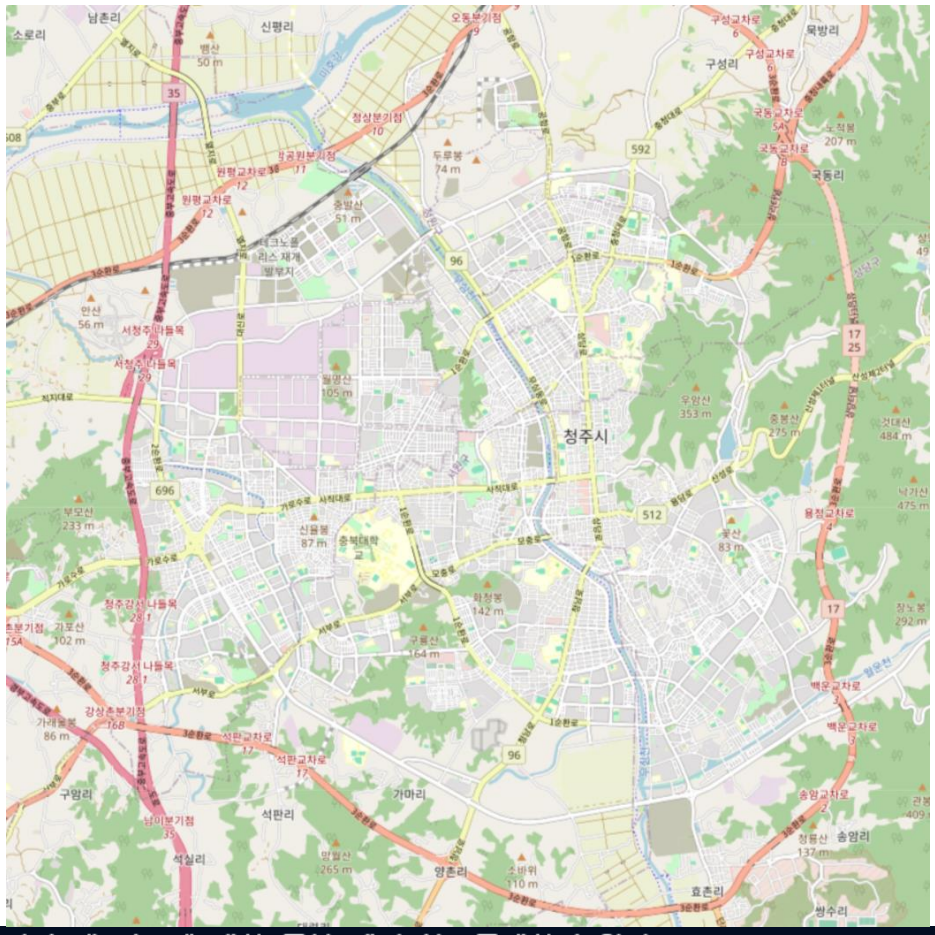
[노드 데이터]

osmid	OpenStreetMap에서 노드를 식별하기 위한 고유 ID
y	노드의 위도 좌표
x	노드의 경도 좌표
street_count	해당 노드에 연결된 도로(엣지)의 개수
geometry	노드의 지리적 정보

[엣지 데이터]

osmid	OpenStreetMap에서 도로를 식별하기 위한 고유 ID
length	해당 엣지(도로)의 길이, **경로 탐색 알고리즘에서 가중치로 사용**
highway	도로 유형을 나타내는 속성
geometry	도로의 실제 선형 도형 정보를 나타내는 정보
name	도로의 이름
oneway	도로가 일방통행인지 여부를 나타내는 정보
u	엣지의 시작 노드 ID
v	엣지의 끝 노드 ID
key	동일한 출발 노드(u)와 도착 노드(v)를 가지는 다중 엣지를 구분하기 위한 값.

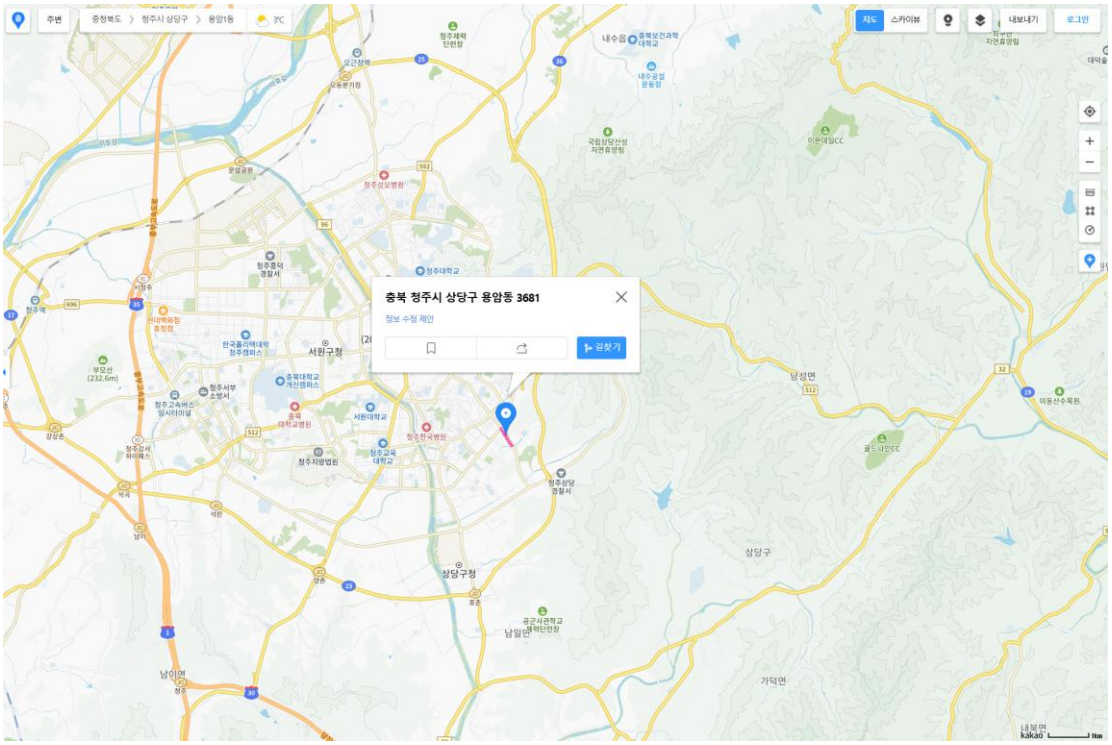
- 검증:
  도로망의 구조가 예상한 대로 잘 반영되었는지 시각화된 그래프와 OpenStreetMap에서 서비스 되고있는 지도의 시각적 비교를 통해 확인하였습니다. 혹여나 눈으로는 식별되지 않는 중복된 노드와 엣지가 있지 않을까 싶어 노드 데이터와 엣지 데이터 각각에 대해 중복 여부를 체크하였습니다.



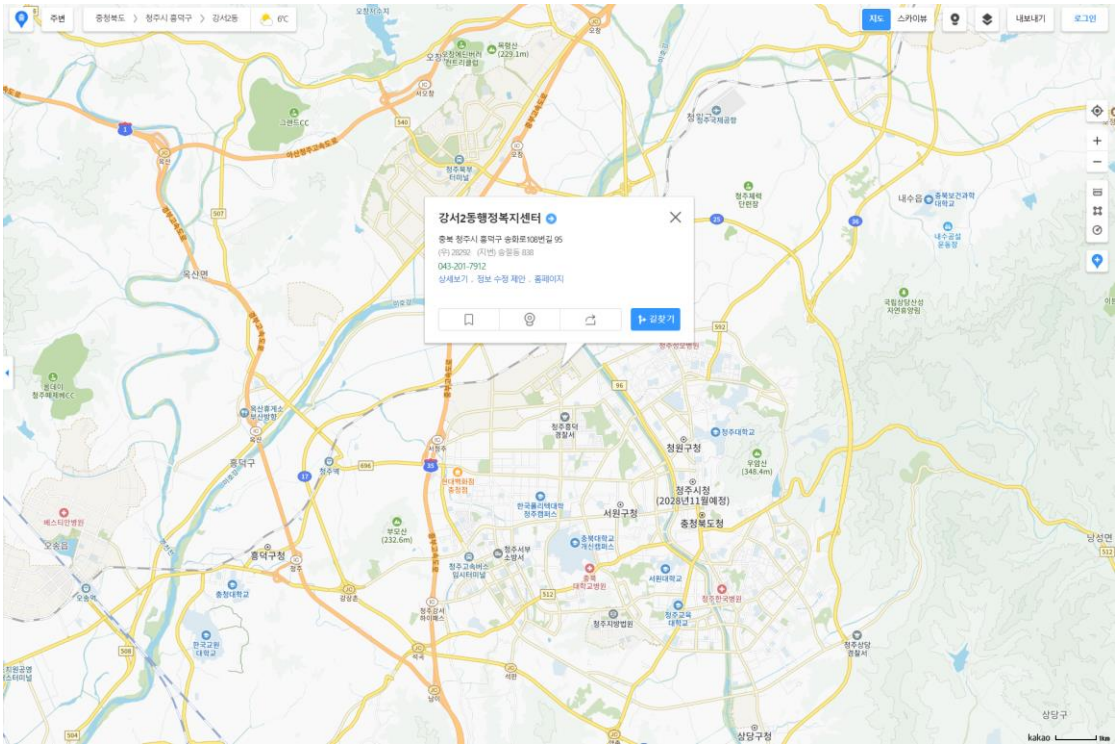

```
데이터 내 좌표에 대한 중복 데이터는 존재하지 않아요.
'geometry' and 'length' 파라미터에 따라 중복 엣지는 존재하지 않아요
```

[Step 2: Dijkstra algorithm 및 Bellman-Ford Algorithm 구현]

- **출발지와 도착지 지정**
  - 출발지: 36.617646, 127.517828
    (충북 청주시 상당구 용암동 3681, 올리브영 청주동남점 앞 사거리)



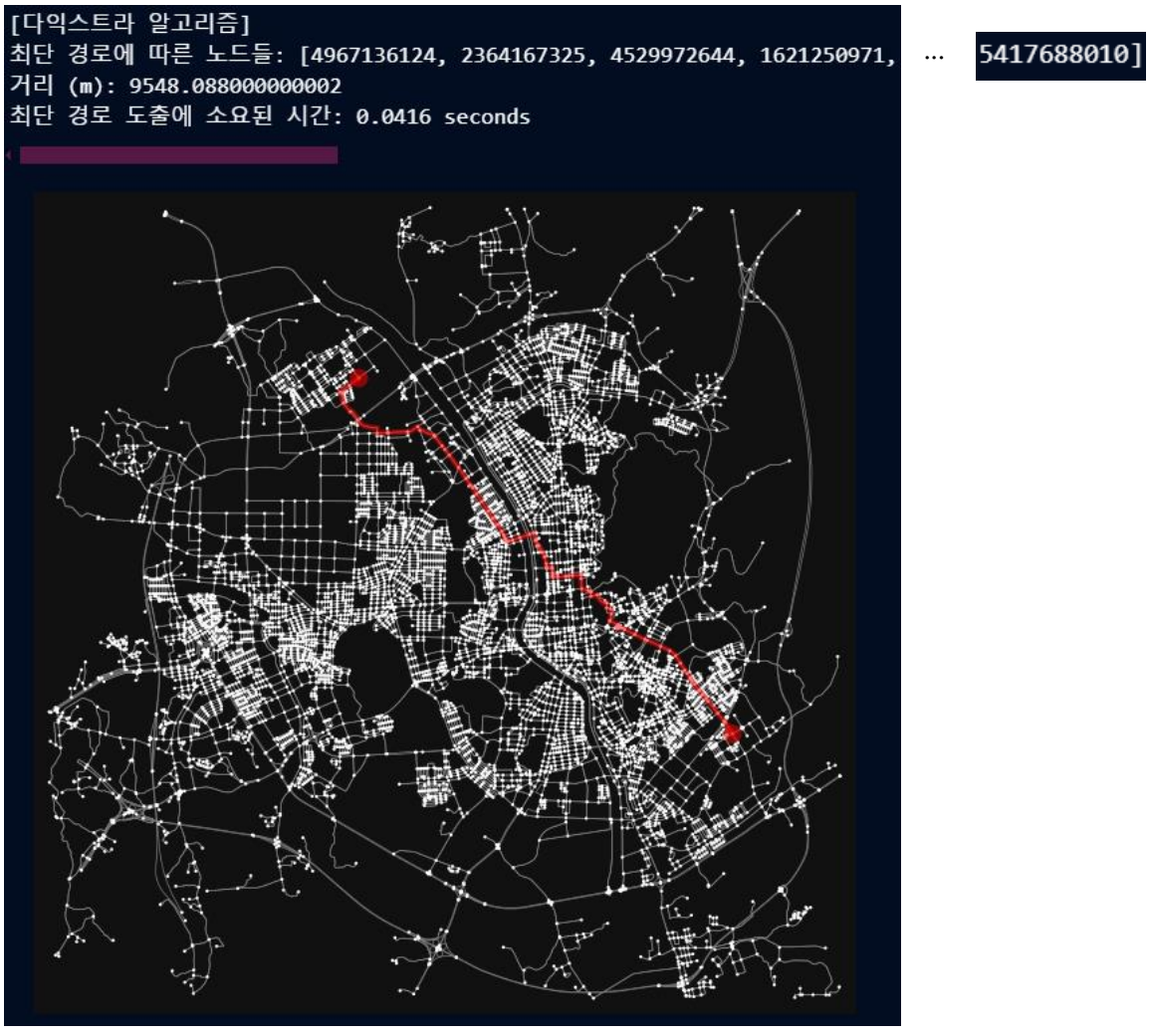
  - 도착지: 36.666138, 127.453691
    (충북 청주시 흥덕구 송화로108번길 95, 테크노폴리스 강서2동 행정복지센터)



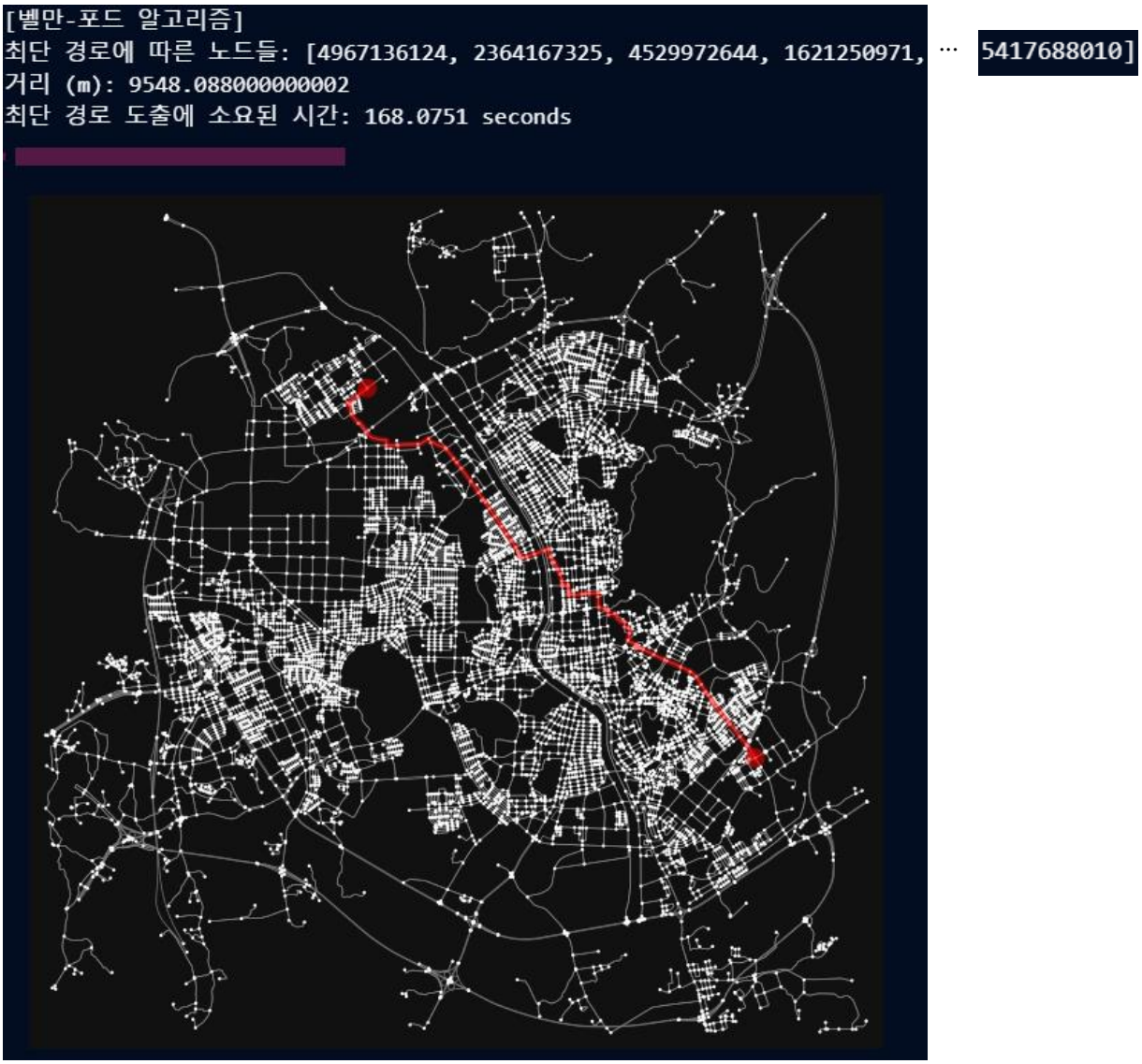
  - 출발지와 도착지에 해당되는 노드 ID 확인

```
출발지의 노드 ID: 4967136124, 도착지의 노드 ID: 5417688010
```

- **Dijkstra algorithm 구현 코드의 흐름**
  - **최단 거리 테이블 초기화**: 모든 노드에 대해 최단 거리 테이블을 초기화합니다. 시작 노드의 거리는 0으로 설정하고, 나머지 노드의 거리는 무한대로 설정합니다. 이로써 시작 노드부터 모든 노드에 대한 경로를 찾을 준비를 합니다.
  - **우선순위 큐 사용**: 경로 탐색의 효율성을 위해 heapq 라이브러리을 사용한 우선순위 큐(최소 힙)를 활용 하였습니다. 일반적인 리스트 구조를 사용하면 모든 노드를 일일이 탐색해야 하므로 시간복잡도가 높아지는 반면, 우선순위 큐를 사용하면 매번 최단 거리를 가진 노드를 효율적으로 추출할 수 있어 시간복잡도를 크게 줄일 수 있습니다. 시작 노드를 큐에 추가하여 알고리즘을 시작합니다.
  - **노드 추출 및 인접 노드 탐색**: 우선순위 큐에서 가장 짧은 거리의 노드를 추출하고, 해당 노드의 인접 노드들을 탐색합니다. 각 인접 노드에 대해 현재 노드를 경유하는 거리를 계산하고, 이 거리가 기존 기록된 거리보다 짧으면 거리와 경로를 갱신합니다. 갱신된 인접 노드는 다시 우선순위 큐에 추가하여 다음 탐색에 포함시킵니다.
  - **거리 갱신**: 인접 노드로의 더 짧은 경로를 발견할 때마다 최단 거리 테이블을 갱신하고, 해당 노드로의 이전 노드 정보를 업데이트합니다. 이러한 과정은 우선순위 큐가 비워질 때까지 반복됩니다.
  - **최단 경로 추적**: 탐색이 끝난 후, 도착지부터 시작지까지 이전 노드를 역추적하여 최단 경로를 추적합니다. 추적된 경로는 거꾸로 되어 있으므로, 이를 뒤집어 올바른 경로를 반환합니다.



- **Bellman-Ford Algorithm 구현 코드의 흐름**
  - **최단 거리 테이블 초기화**: 그래프 내의 모든 노드에 대해 최단 거리를 무한대로 초기화하고, 시작 노드의 거리를 0으로 설정합니다. 이를 통해 시작 노드에서 다른 모든 노드까지의 거리를 추적할 준비를 마칩니다. 또한, 각 노드의 이전 노드를 저장할 딕셔너리를 생성하여 초기화합니다.
  - **엣지 반복 과정**: 그래프의 모든 엣지을 (노드 개수(V) - 1)번 반복하여 확인합니다. 각 반복에서 엣지의 출발 노드 u와 도착 노드 v간의 최단 거리를 비교하고, 더 짧은 경로가 발견되면 해당 경로로 갱신합니다. 노드 개수가 V개일 때, 최단 경로를 완전히 계산하기 위해 최대 V – 1번의 반복이 필요합니다. 이는 그래프에서 최단 경로를 구하기 위해 모든 가능한 경로를 탐색하고 갱신하는 과정입니다.
  - **최단 경로 추적**: 도착 노드부터 출발 노드까지의 경로를 역추적하기 위해, 최단 거리 테이블 초기화 과정에서 초기화된 딕셔너리를 사용합니다. 도착 노드에서 시작하여 이전 노드를 계속 추적하면서 경로를 구성한 후, 이를 뒤집어 올바른 순서로 반환합니다. 최종적으로 최단 경로와 해당 경로의 총 거리를 함께 반환합니다.



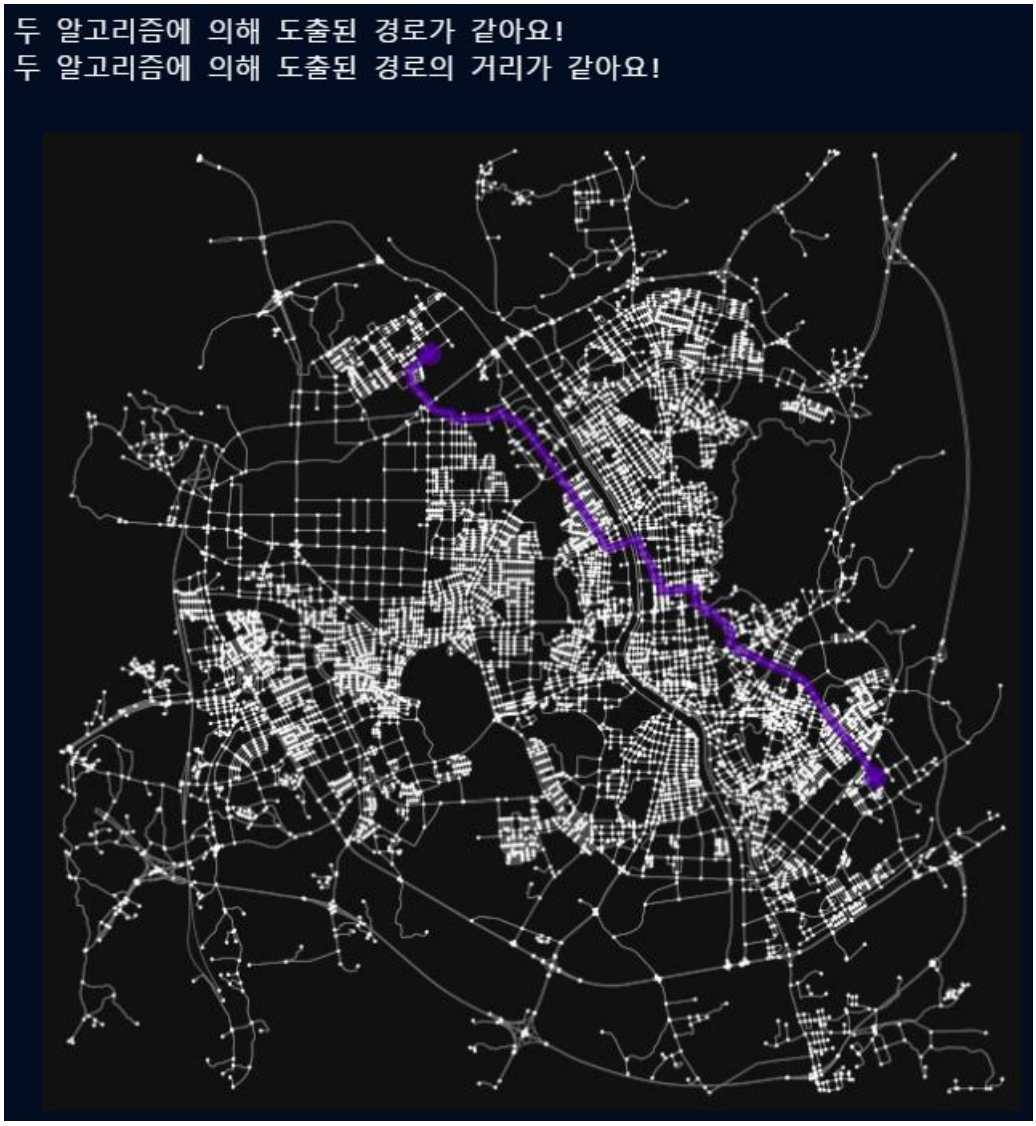
[Step 3: Python에서 구현한 Dijkstra algorithm 및 Bellman-Ford Algorithm의 경로 및 성능 비교]

- **두 알고리즘으로부터 도출된 최단 경로와 거리 비교**

  빨간색: Dijkstra algorithm에 의해 도출된 경로, 9548m

  파란색: Bellman-Ford algorithm에 의해 도출된 경로, 9548m

  → 두 경로가 완전히 일치하기에 보라색으로 보이게 됩니다.



- **두 알고리즘으로부터 최단 경로를 도출하기까지 경과된 시간**
  - Python의 time 모듈을 이용하여 각각의 알고리즘이 구현된 함수가 시작할 때부터 끝날때까지의 타이머를 걸어 최단 경로를 도출하기까지 걸린 시간을 측정해보았습니다.

    Dijkstra algorithm 구현 함수 실행 시간: 0.0416초

    Bellman-Ford algorithm 구현 함수 실행 시간: 168.0751초

    (이 정보에 대한 스크린 샷은 Step 2의 각 알고리즘 별 경로 생성 결과 스크린 샷에서 확인할 수 있습니다.)

  - 이 결과에서 확인할 수 있듯이, Dijkstra algorithm이 Bellman-Ford algorithm에 비해 훨씬 빠르게 최단 경로를 도출한다는 점을 알 수 있습니다. 이러한 차이는 알고리즘의 **시간 복잡도와 관련**이 있었습니다.

    Dijkstra algorithm은 우선순위 큐를 사용한 경우 시간 복잡도가 **O((V + E) log V)** 입니다.

이는 네트워크 그래프의 노드 수 V와 엣지 E의 수에 비례하되, 로그가 붙어서 최단 경로를 효율적으로 찾게 됩니다. 특히 양의 가중치만 있는 이번 상황에서는 더 빠른 경로 탐색을 하기 위해 우선순위 큐를 적용하였습니다.
만약, 우선순위 큐를 사용하지 않은 경우에는 시간 복잡도가 $O(V^2)$ 입니다. 단순히 정점마다 최단 거리를 선형 탐색으로 선택하는 방식이라 정점의 수가 적을 때는 효율적일 수 있지만, 정점이 많아질수록 모든 정점을 매번 선형 탐색해야 하므로 성능이 떨어지게 됩니다.

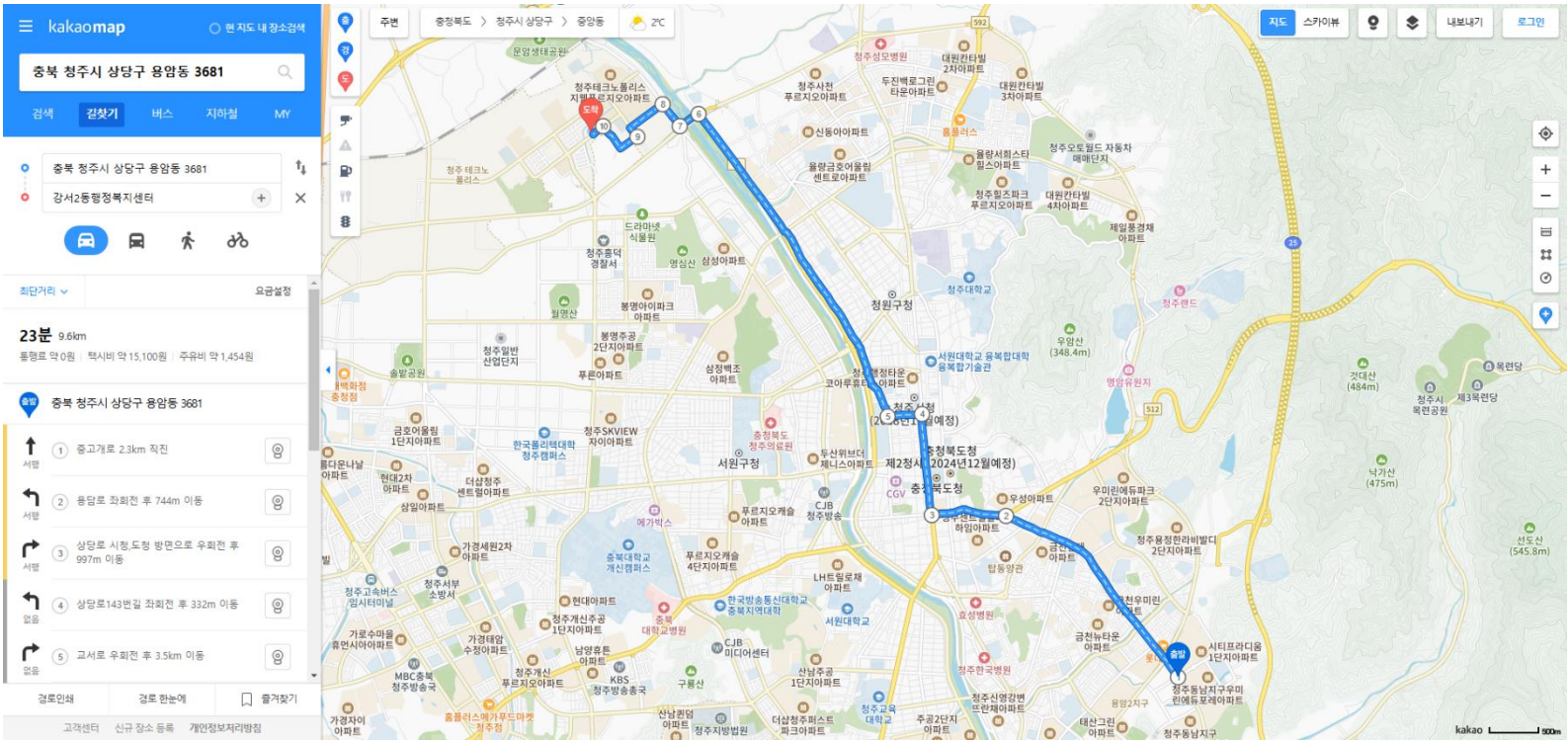
- **Bellman-Ford algorithm**의 시간 복잡도는 **O(V * E)** 입니다. 이 알고리즘은 모든 엣지를 여러 번 반복적으로 검사하여 최단 경로를 업데이트하는 방식으로 동작합니다. 엣지를 여러 번 검사하는 특성 때문에 정점과 간선의 수가 증가할수록 시간이 기하급수적으로 증가합니다. 따라서 복잡한 그래프에서는 시간적인 성능이 매우 떨어지게 됩니다.

- **두 알고리즘을 구현한 함수를 실행시킬 때의 메모리 사용량**
  - Python의 tracemalloc 모듈을 이용하여 두 알고리즘이 사용하는 메모리의 최대 사용량을 측정하였습니다. 메모리 사용량을 측정한 이유는 두 알고리즘이 동일한 그래프를 처리할 때 자원을 얼마나 사용하는지 궁금했었습니다. 코딩테스트 문제를 풀다보면 파이썬은 타 언어에 비해 메모리 관리에 대해 소홀하기 때문에 이 부분에 대해서 신경을 쓰지 않을 수 없었습니다.
    Dijkstra algorithm 구현 함수 최대 점유 메모리: 0.8724 MB
    Bellman-Ford algorithm 구현 함수 최대 점유 메모리: 0.8711 MB
    ```
    다익스트라 알고리즘의 메모리 사용량: 0.8724 MB
    벨만-포드 알고리즘의 메모리 사용량: 0.8711 MB
    ```
  - 메모리 사용량에서 두 알고리즘 간의 큰 차이는 거의 없었습니다. 두 알고리즘 모두 그래프의 기본적인 구조를 표현하는 방식은 동일하고, 노드와 엣지을 저장하는 데 필요한 자료구조 자체 또한 비슷하게 사용되기 때문이라 생각합니다.

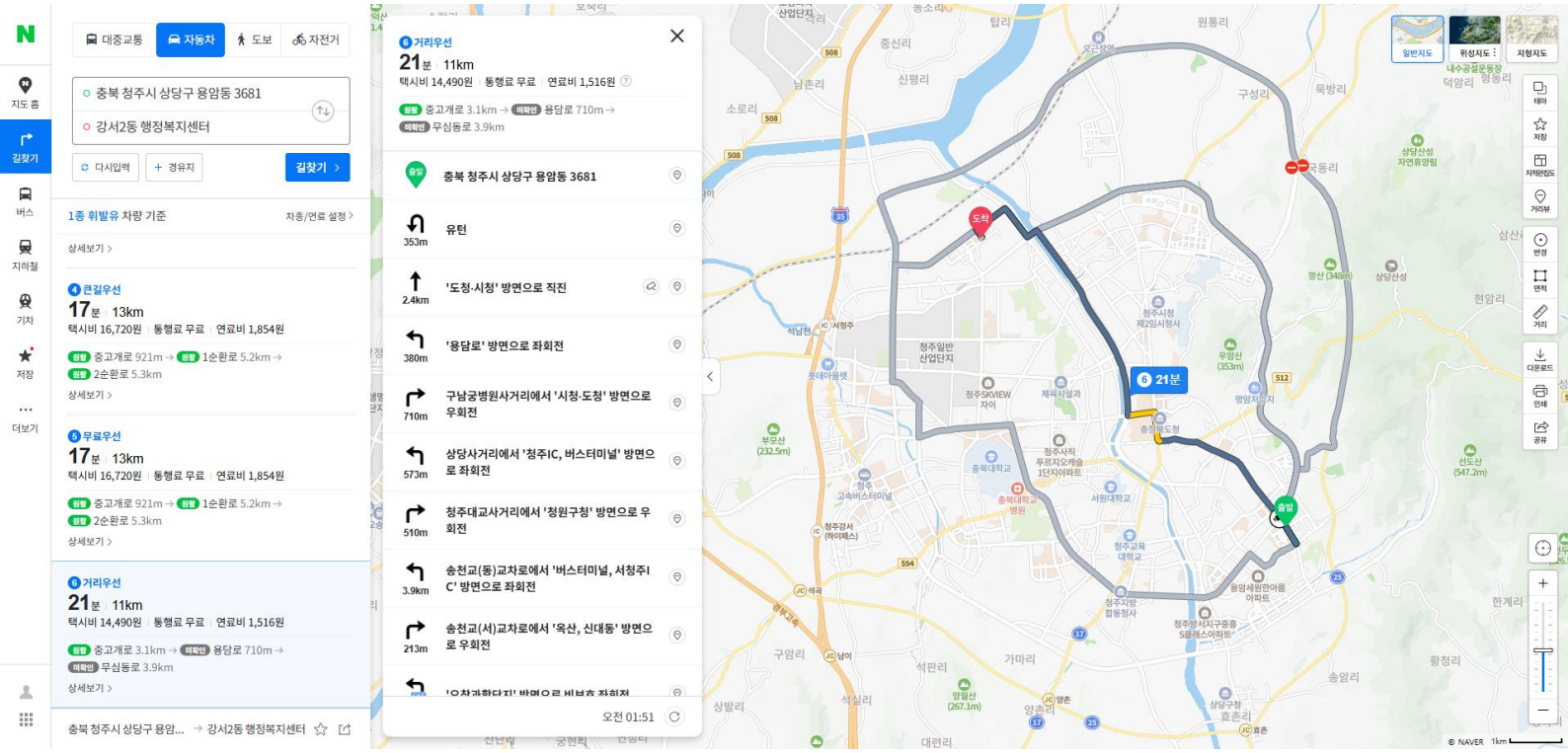
(매번 실행시킬 때 마다, 그리고 환경이 달라지면 두 알고리즘으로부터 최단 경로를 도출하기까지 경과된 시간과 두 알고리즘을 구현한 함수를 실행시킬 때의 메모리 사용량은 본 보고서에 기재된 수치와 미세한 수치로 다르게 출력됩니다.)

[Step 4(번외): 외부 지도 서비스와 결과 비교]

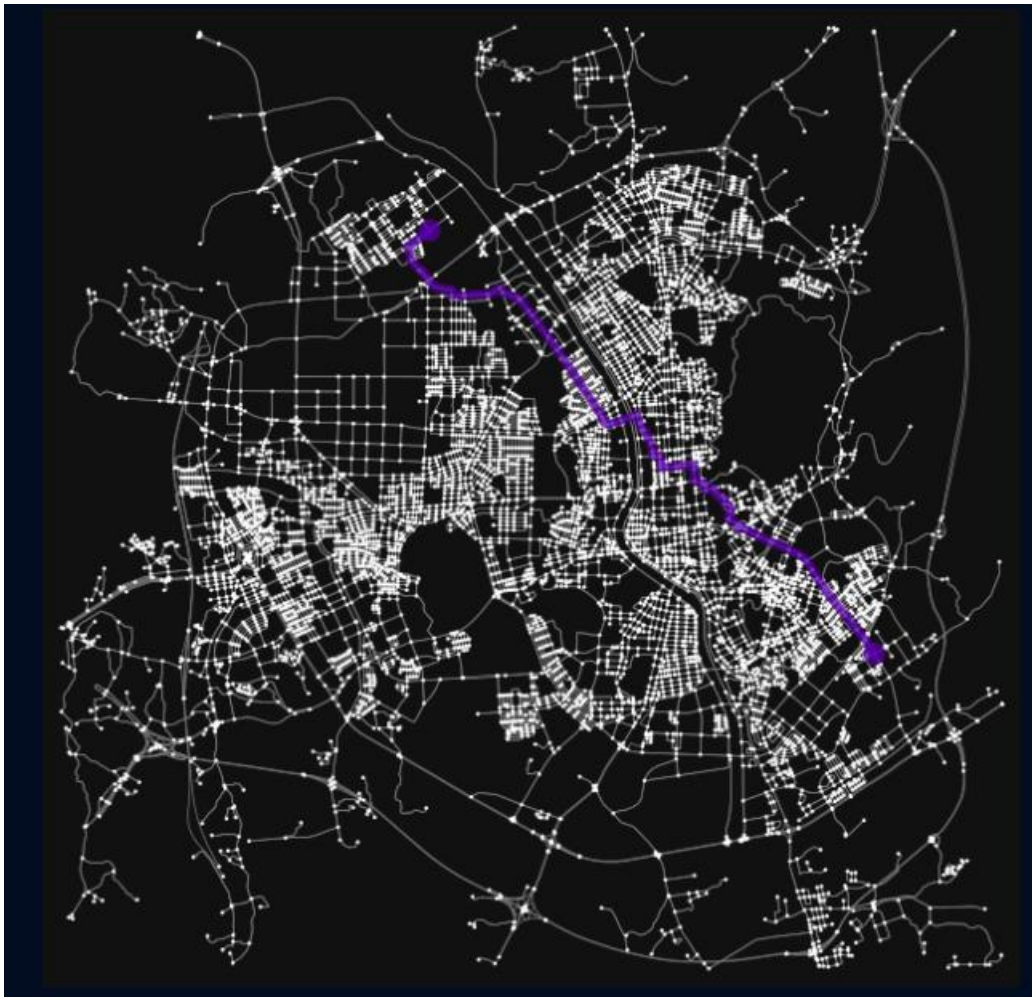
- **카카오맵, 네이버지도의 경로와 Dijkstra algorithm 및 Bellman-Ford algorithm의 경로 비교**
  - 카카오맵, 네이버지도와 Dijkstra algorithm 및 Bellman-Ford algorithm의 결과를 비교하려고 합니다. 이를 통해 실제 상용 지도 서비스가 제공하는 경로와 우리가 구현한 알고리즘의 차이를 분석하고 경로 최적화의 관점에서 어떻게 다른지 파악하고자 합니다.
  - **카카오맵 – 최단 거리**: 경로 다름 (9.6Km)



- **네이버 지도 – 거리 우선**: 경로 다름 (11Km)



- Step 2에서 도출된 Dijkstra algorithm 및 Bellman-Ford algorithm의 경로(도청 이후 무심천 기준 좌측, 운천동 -> 봉명동 경로)와 두 서비스(도청 이후 무심천 기준 동측, 우암동 -> 사천동 경로)의 경로를 각각 비교했을 때 경로가 일부 다른 것을 확인 할 수 있었습니다.
  그리고 카카오맵(도청 주변을 지나가는 경로)과 네이버 지도(테크노폴리스쪽에 다다를 때 거쳐가는 경로)에서 도출된 경로끼리도 일부 다른 것을 확인 할 수 있었습니다.



- 이러한 차이는 각 지도 서비스가 반영하는 변수나 최적화 기준이 따로 마련되있을 것이라 생각합니다. 그리고 이런 서비스들은 도로 폐쇄, 제한 구역, 교통량 등을 반영하여 경로를 제시했기 때문에, 이번에 구현해본 Dijkstra algorithm 및 Bellman-Ford algorithm의 순수 경로와는 분명 다른점이 있어야 할 것입니다.

3. 결론

- 이번 프로젝트에서는 Dijkstra algorithm과 Bellman-Ford algorithm을 구현하여 청주 지역 도로 네트워크에서 최단 경로를 탐색하였으며, 거리, 소요 시간, 메모리 사용량 측면에서 두 알고리즘의 성능을 비교하였습니다. Dijkstra algorithm은 우선순위 큐를 사용해 빠르고 효율적인 경로 탐색을 수행했으며, Bellman-Ford 알고리즘은 모든 엣지을 (노드 개수(V) - 1)번 만큼 반복하여 확인하기 때문에 Dijkstra algorithm보다 확실하게 느리다는 것을 알 수 있었습니다.
- 라우팅 알고리즘에 대한 수업을 들을 때 구조와 원리에 대한 이해가 부족했지만, 프로젝트를 통해 Python으로 두 알고리즘을 구현하며 데이터 구조 설계, 우선순위 큐 활용, 반복문을 통한 최적화 과정을 직접 경험했습니다. 이 과정을 통해 알고리즘의 이론적 원리를 이해하고 최단 경로를 도출하는 방법을 명확히 이해할 수 있었습니다.
- 또한, 상용 지도 서비스(카카오맵, 네이버 지도)와 비교한 결과, 상용 서비스는 교통량, 도로 상황 등 실시간 데이터를 반영해 경로를 제안한다는 점에서 차별성을 확인하였습니다. 이를 통해 이번 프로젝트에서 구현한 알고리즘의 한계를 인식하고, 다음에 기회가 된다면 실시간 정보를 제공하는 API를 이용한 경로 찾기의 추가 개선 방향을 도출해보고 싶습니다.
- 그리고, 평소 머신러닝과 딥러닝의 인공지능 분야에도 큰 관심을 가지고 있어, 시간적 여유가 된다면 그래프 뉴럴 네트워크(GNN)나 강화 학습 기반의 알고리즘을 도입하여 이번에 다루게 되었던 전통적인 두 알고리즘과 비교해 얼마나 더 최적의 경로가 나오는지 분석해보고 싶고, 사용자의 평소 운전 습관이나 도로 선호도 등을 파악하고 인공지능 모델을 바탕으로 분석하여 사용자 맞춤형 경로 제안 기능을 구현해보고 싶습니다.

4. 참고문서

[Step 1: 데이터 준비 및 네트워크 그래프 생성]

OpenStreetMap 및 GIS 데이터 활용

OpenStreetMap Wiki, https://wiki.openstreetmap.org/

PyOSMnx 공식 문서, https://osmnx.readthedocs.io/

[Step 2: Dijkstra Algorithm 및 Bellman-Ford Algorithm 구현]

알고리즘 이론 및 구현 관련

Velog. 최단 경로 알고리즘: 다익스트라와 벨만-포드. https://velog.io/@panghyuk/%EC%B5%9C%EB%8B%A8-%EA%B2%BD%EB%A1%9C-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98

Tistory, "다익스트라 알고리즘 쉽게 이해하기", https://10000cow.tistory.com/entry/%EB%8B%A4%EC%9D%B5%EC%8A%A4%ED%8A%B8%EB%9D%BC-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%ED%95%9C-%EC%82%B4%EB%8F%84-%EC%9D%B4%ED%95%B4%ED%95%98%EB%8A%94-%EB%8B%A4%EC%9D%B5%EC%8A%A4%ED%8A%B8%EB%9D%BC-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98

나무위키. 다익스트라 알고리즘. https://namu.wiki/w/%EB%8B%A4%EC%9D%B5%EC%8A%A4%ED%8A%B8%EB%9D%BC%20%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98

GeeksforGeeks. (n.d.). Bellman-Ford algorithm. https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

Tistory. (n.d.). 벨만-포드 알고리즘. https://jeonyeohun.tistory.com/97

Crocus. (n.d.). 벨만 포드 알고리즘 개념. https://www.crocus.co.kr/534

Python 공식 문서, "heapq — Heap queue algorithm", https://docs.python.org/3/library/heapq.html

[Step 3: Python에서 구현한 Dijkstra algorithm 및 Bellman-Ford Algorithm의 경로 및 성능 비교]

성능 및 시간 복잡도 분석 관련

GeeksforGeeks. Dijkstra's shortest path algorithm using priority queue. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-priority_queue-stl/

GeeksforGeeks. Bellman-Ford algorithm. https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

Python 공식 문서. time — Time access and conversions. https://docs.python.org/3/library/time.html

[Step 4(번외): 외부 서비스와 결과 비교]

상용 지도 서비스 관련

Kakao Tech. (2022, March 10). 카카오맵 개발 이야기. https://tech.kakao.com/posts/436

Ahn, J., & Kim, T. (2018). 스마트폰 네비게이션의 경로 탐색 알고리즘과 효율성 분석. Journal of Smart Technology, 12(3), 45-56.

Lee, H., & Park, S. (2020). 한국 내 지도 서비스의 경로 최적화 알고리즘 비교: 네이버 지도와 카카오맵 중심으로. Proceedings of the Korea GIS Conference, 15(2), 23-30.