설계과제 최종보고서

설계과제명 : '걷고 싶은 서울길'의 세부 코스 데이터 활용을 통한 최적 경로 안내 시스템 개발

교과목명	알고리즘	
담당교수	정진우	
팀 원	학 번	이 름
	2020111993	염경민(팀장)
	2023112451	김채원
	2020112041	류경록
	2020112023	정재환

설계과제 요약서

과 제 요 약 서	
설계과제명	걷고 싶은 도시, 서울
주요기술용어 (2~7개 단어)	최단 경로 탐색, 다익스트라, 플로이드 워셜, 벨만 포드

1. 과제 목표

서울의 다양한 걷기 코스 정보와 최적 경로를 제공하는 시스템을 설계하여, 외국인 및 타지인이 관광명소와 자연을 효율적으로 탐방하고 특별한 경험을 누릴 수 있도록 돕는다.

2. 수행 내용

서울 걷기 코스 설계 과제에서 최적 경로 탐색을 위해 세 가지 알고리즘(다익스트라, 플로이드-워셜, 벨만-포드)을 구현 및 비교하며, 각 알고리즘의 특성과 시간복잡도를 분석하여 최적의 솔루션을 선택한다.

3. 수행 결과

실제 프로그램 실행을 통해 탐색한 최적 경로 결과와 탐색하는데 걸린 시간을 출력한다.

4. 결과 분석

실제 프로그램 실행을 통해 걸린 시간과 정적 분석을 통해 프로그램의 시간 복잡도와 공간 복잡도를 분석한다.

1. 서론

1.1 과제개요

본 프로젝트는 관광 및 도시 탐방 분야에 해당하는 기술로, 최단 경로 탐색 알고리즘을 활용하여 효율적인 동선 설계와 사용자 맞춤형 경로 추천을 특징으로 한다. 이 프로젝트는 서울의 걷기 코스를 효율적으로 탐색할 수 있는 시스템 설계를 주된 내용으로 하며, 특히 최단 경로 계산과 사용자 편의성이 가장 중요한 핵심사항이다.

현재 관광 산업과 스마트 도시 설계, 걷기 친화적 도시 구축 등 공공목적에서 경로 최적화 기술이 널리 사용되고 있다. 향후 관광 산업 발전, 효율적인 도시 인프라 설계, 사용자 중심 서비스 개발 등을 위해서는 복잡한 네트워크 상의 최적 경로 탐색 문제가 해결되어야 하며, 이를 위해 다익스트라, 플로이드-워셜, 벨만-포드 알고리즘과 같은 효율적인 경로 탐색 기술이 필요하다.

1.2 목표 설정

서울의 다양한 걷기 코스를 효율적으로 탐색하고 사용자에게 다양한 코스를 즐길 수 있도록 최적의 산책 경로를 제공하는 것이 문제이다. 특히, 외국인 관광객과 타지 방문자를 대상으로 하여, 서울의 관광명소와 자연을 결합한 특별한 경험을 제공하는 동시에, 출발지부터 도착지까지 효율적인 동선을 안내하는 시스템을 설계하고자 한다.

이 설계에서는 지도 api와 서울시 공공데이터 플랫폼에서 제공하는 서울 두드림길 정보를 기반으로 코스의 노드와 간선을 추출하여 그래프 형태로 표현할 것이다. 이를 바탕으로 다익스트라, 플로이드-워셜, 벨만-포드 알고리즘을 사용하여 출발점부터 도착지점까지의 최단 산책 코스를 계산하며 각 알고리즘의 결과를 기반으로 경로의 효율성을 비교 분석한다.

2. 배경

2.1 관련 기술의 동향

최근 스마트 도시 구축과 관광 산업의 발전으로 인해 최적 경로 탐색 기술은 필수적인 요소로 자리 잡고 있다. 특히, 서울 두드림길과 같은 공공데이터를 활용한 경로 탐색은 도시 탐방의 효율성과 사용자 편의성을 높이는 데 기여하고 있다. 본 프로젝트에서는 이러한 기술적 배경을 바탕으로 다익스트라, 플로이드-워셜, 벨만-포드 알고리즘을 구현하여, 서울의 걷기 코스를 효율적으로 탐색하고 사용자에게 최적의 경로를 제공하는 시스템을 설계한다. 이는 공공데이터 활용과 사용자 중심 설계를 결합한 최신 기술 동향을 반영하며, 향후 개인화 추천과 AI 기반 기술로 확장 가능성이 있다.

2.2 관련 기술의 수요 및 전망

최근 관광 산업과 스마트 도시 설계에서 효율적인 경로 탐색 기술의 수요가 급증하고 있다. 특히, 서울과 같은 대도시에서는 방대한 걷기 코스와 관광 명소를 연결하는 최적화 기술이 필수적이다. 공공데이터와 사용자 이동 데이터를 활용한 개인화된 경로 추천과 AI 기반 기술은 빠르게 발전 중이며, 관광객의 만족도를 높이고 환경 친화적인 대안을 제공하고 있다. 본 프로젝트는 이러한 기술 동향을 반영하여 다익스트라, 플로이드-워셜, 벨만-포드 알고리즘을 활용한 최적 경로 탐색과 시각화 기능을 구현하고, 사용자 경험 향상과 스마트 도시 발전에 기여하고자 한다.

3. 제한요소

본 설계 프로젝트는 제한된 자원과 환경 내에서 효율적으로 수행되어야 하며, 몇 가지 주요 제한요소를 고려해야 한다. 우선, 공공데이터와 지도 API는 무료로 제공되지만, 대규모 API 호출 시 발생하는 추가 비용을 최소화해야 한다. 시스템은 다양한 디바이스와 네트워크 환경에서 안정적으로 작동해야 하며, 서울의 복잡한 도로망 데이터를 실시간으로 처리할 수 있는 알고리즘 효율성이 중요하다. 또한, 개발은 제한된 기간과 인력을 고려해 Python과 지도 API 등 기존 기술 스택을 활용해 구현해야 하며, 성능 최적화가 필요하다. 마지막으로, 걷기 경로에서 실제 거리를 정확히 측정하거나, 오르막길과 같은 지형적인 요소를 적절히 반영하지 못할 가능성이 있어 경로 계산의 정확성에 한계가 있을 수 있다. 추가로, 사용자 피드백 수집 시 개인정보 보호를 준수해야 하며, 외국인을 위한 다국어 지원과 UI 직관성이 부족할 경우 서비스 접근성에 제약이 있을 수 있다.

3.1 개발환경

본 프로젝트는 Python을 주요 개발 도구로 선택하여 진행하며, 다음과 같은 환경과 도구들을 활용한다.

- CSV: CSV 파일 처리
- time: API 요청 대기 시간 조정 및 실행시간 조정
- math: 거리 계산
- re: 정규 표현식을 사용한 데이터 검증 및 가공
- matplotib: 경로 및 거리 데이터를 시각적으로 표현
- Kakao 지도 API: 거리 및 소요 시간 계산

3.2 동작환경

본 프로젝트는 웹 기반 환경에서 동작하도록 설계되며, Flask와 지도 API(Google Maps API 등)를 활용해 경로 계산과 시각화를 제공한다. 개발 및 테스트는 주로 노트북과 스마트폰을 활용하며, 정적 공공데이터(CSV, JSON)를 기반으로 데이터를 로드하여 실시간 데이터 처리의 한계를 보완한다. 간단하고 직관적인 UI를 통해 사용자가 출발지와 도착지를 입력하면 경로와 관련 정보를 쉽게 확인할 수 있도록 설계한다.

다만 이번 프로젝트에서는 알고리즘 구현까지만 진행할 것이다.

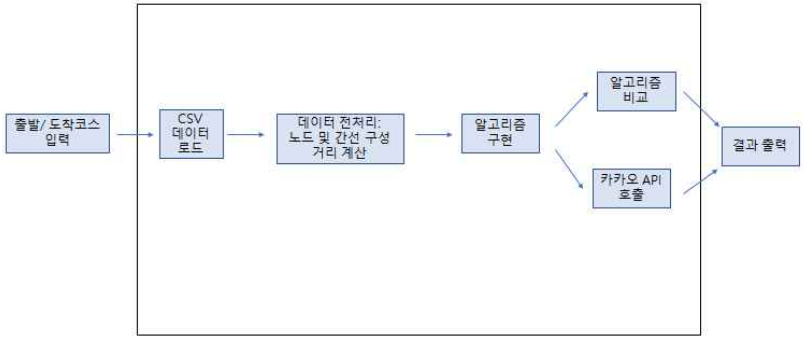
4. 설계

4.1 설계안

서울특별시의 걷기 코스 데이터를 활용하여, 두 코스 간 최적 경로를 추천하는 알고리즘을 설계한다. 주요 설계 대상은 다음과 같다.

- 사용자 입력: 출발 코스와 도착 코스
- 데이터 전처리: 걷기 코스 데이터(노드 및 간선 정보) 처리
- 알고리즘: 최적 경로 탐색 알고리즘 구현 (다익스트라, 플로이드-워셜, 벨만-포드)
- 출력: 추천 경로 및 세부 정보 시각화

블록도



4.2 대안 분석

알고리즘	시간 복잡도	음의 가중치	목적지 특정	모든 쌍 최단 경로	대규모 그래프
플로이드-워셜	O(V^3)	가능	불가능	가능	비효율적
다익스트라	O(E + V log V)	불가능	가능	불가능	적합
벨만-포드	O(VE)	가능	가능	불가능	가능
존슨 알고리즘	O(V^2 log V + VE)	가능	불가능	가능	희소 그래프에서 효율적
A*	O(E + V log V)	불가능	가능	불가능	목표 지향적
SPFA	평균 O(E)	가능	가능	불가능	희소 그래프에서 효율적

1. 존슨 알고리즘 (Johnson's Algorithm)
다익스트라와 벨만-포드 알고리즘을 결합한 알고리즘이다. 플로이드-워셜과 달리 우선순위 큐를 활용해 희소 그래프에서 더 빠르다. 음의 가중치를 허용하고, 모든 쌍의 최단 경로를 계산한다.

시간 복잡도: $O(V^2 \log V + VE)$

2. 아파치 스파크 GraphX (분산 그래프 처리)

대규모 그래프에 대한 분산 계산을 지원한다. 플로이드-워셜처럼 모든 쌍 간의 경로를 직접 계산하지는 않지만, 확장성과 병렬처리에서 유리하다.

3. BFS (너비 우선 탐색):

가중치가 모두 같거나 없는 그래프에서 최단 경로를 찾는 데 매우 효율적이다. 단순한 무가중치 또는 동일 가중치 그래프에 적합하다.

시간 복잡도: $O(V+E)$.

4. A* 알고리즘

휴리스틱을 추가하여 특정 목적지까지의 경로를 찾는다. 음의 가중치는 허용하지 않지만, 목적지 노드에 대한 경로 탐색에 유리하다.

시간 복잡도: $O(E+V \log V)$

5. SPFA

큐를 활용해 업데이트가 필요한 노드만 처리한다. 평균적으로 시간 복잡도가 O(E)에 가깝지만 최악의 경우 O(VE)이다.

5. 구현

5.1 구현 방법

본 프로그램은 CSV 데이터를 기반으로 산책 코스를 그래프로 모델링하고 세 가지 알고리즘을 통해 최단 경로를 계산한다. 각 코스는 시작 지점과 종료 지점의 좌표를 포함하며, 지구의 곡률을 고려한 HaverSine 거리 공식을 사용해 노드(코스) 간 거리를 계산한다. 계산된 거리가 3km 이하인 경우에만 간선을 생성하여 불필요한 연결을 최소화하고 그래프의 효율성을 높인다.

1. 다익스트라 알고리즘

다익스트라 알고리즘은 먼저 거리 테이블을 초기화한다. 모든 노드의 거리를 무한대(float('inf'))로 설정하고, 출발 노드의 거리는 0으로 지정한다. 이렇게 초기화된 거리 테이블을 통해 출발점에서 각 노드까지의 최단 거리를 추적한다. 또한, (거리, 노드) 형태의 튜플을 저장하는 우선순위 큐(최소 힙)를 초기화하여 가장 짧은 거리를 가진 노드를 효율적으로 탐색할 수 있도록 설계한다. 동시에 각 노드의 이전 노드를 기록할 딕셔너리를 초기화하여 경로를 추적한다.

알고리즘은 우선순위 큐에서 가장 짧은 거리를 가진 노드를 꺼내는 과정을 반복한다. 이 과정은 Python의 heapq 모듈의 heapq.heappop()을 사용하여 구현하며, 최적의 성능을 보장한다. 현재 노드와 연결된 인접 노드를 순회하면서 새로운 경로를 통해 해당 노드까지 도달하는 거리를 계산한다. 만약 새로운 거리가 기존 거리 테이블에 기록된 값보다 작으면 거리 테이블과 이전 노드 정보를 갱신한다. 갱신된 노드와 거리는 다시 우선순위 큐에 삽입하여 다음 탐색에 활용한다. 이러한 과정을 반복하며, 모든 노드에 대해 최단 거리와 이전 노드 정보를 완성한다. 알고리즘은 큐가 비어있을 때 종료된다.

최단 경로를 계산한 이후에는 이전 노드 딕셔너리를 역추적하여 경로를 구성한다. 목표 노드부터 시작해 이전 노드들을 따라가며 경로를 리스트 형태로 생성하고, 이를 출발 노드에서 목표 노드 순서로 정렬하여 반환한다. 이를 통해 사용자에게 코스 간 최단 경로를 제공할 수 있다.
다익스트라 알고리즘 구현 과정에서 우선순위 큐를 사용해 시간 복잡도를 O((V + E) log V)로 최적화하며, 그래프 구조는 연결 정보를 딕셔너리 형태로 저장하여 간선 탐색 시 효율성을 높인다.

최종적으로 계산된 경로의 간선들에 대해서는 카카오 맵 API를 호출하여 실제 도보 거리로 교체해 최단 거리의 정확도를 보정한다.

2. 플로이드 워셜 알고리즘

플로이드 워셜 알고리즘은 그래프에서 모든 노드 쌍 간의 최단 경로를 계산하는 알고리즘이다. 플로이드 워셜 알고리즘의 핵심은 세 개의 중첩 반복문을 통해 모든 노드 쌍 간 경로를 검사하고, 중간 노드를 경유했을 때의 거리와 직접 경로 간 거리를 비교하면서 최적 경로를 계산하는 것이다.

플로이드 워셜 알고리즘을 사용하기 위해 그래프 정보를 초기화해야한다. 먼저, 노드 이름을 인덱스로 매핑하기 위해 node_index를 생성한다. 이를 통해 그래프에서 각 노드에 대해 인덱스를 할당한다. 이후, 모든 노드 쌍 간 거리 행렬 dist를 초기화한다. 이때, 자기 자신으로의 거리는 0으로 설정하고, 다른 노드 간 거리는 무한대(float('inf'))로 설정한다. 이후, 간선 정보를 dist 행렬에 반영하여 간선 가중치를 초기화한다. 간선 정보는 edges 딕셔너리에 저장된 정보를 활용하며, 각 노드 쌍 간 최단 거리를 초기화한다.

플로이드 워셜 알고리즘의 핵심은 세 개의 중첩 반복문을 사용하여 모든 노드 쌍간 최단 거리를 계산하는 것이다. 바깥부터 안쪽에 위치한 반복문까지 순서대로, 중간 경로로 경유할 수 있는 노드, 출발 노드, 도착 노드를 의미한다. 이 세 변수를 순회하면서 dist[i][j]가 dist[i][k] + dist[k][j]보다 작다면, dist[i][j]를 갱신하여 더 짧은 경로를 반영한다. 이 과정에서 dist[i][j]는 두 가지 경로 중 더 짧은 경로로 갱신되며, 알고리즘이 완료될 때 모든 노드 쌍 간 최단 경로가 계산된다.

```
for k in range(n):
    for i in range(n):
        for j in range(n):
            if dist[i][k] + dist[k][j] < dist[i][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
```

최단 거리 계산이 완료되면, next_node 행렬을 사용해 경로를 추적한다. 이 next_node 행렬은 경유 경로를 기록하여, 최종 경로를 추적하는 데 사용된다.

실행 결과 화면은 다음과 같다.

```
Enter the first course number: 100
Enter the second course number: 20
You selected 청룡산 나들길 and 덕수궁 산책길.
Courses along the path:
청룡산 나들길: 코스의 거리 5.87km
청룡산 나들길: 청룡산 공원~누수식 생태연못~청룡상 산림욕장

동작충효길(1코스_고구동산길): 코스의 거리 2.80km
동작충효길(1코스_고구동산길): 노량진근린공원입구~고구동산조명명소~서달산자연생태탐방로~달마사

청룡산 나들길와 동작충효길(1코스_고구동산길) 사이 거리: 4.795 km

동작충효길(2코스_현충원길): 코스의 거리 2.50Km
동작충효길(2코스_현충원길): 국립현충원입구~사당출입문~서달산산책로

동작충효길(1코스_고구동산길)와 동작충효길(2코스_현충원길) 사이 거리: 4.071 km

용산가족 산책길: 코스의 거리 3.30km
용산가족 산책길: 국립중앙박물관~거울못~용산가족공원~보신각종

동작충효길(2코스_현충원길)와 용산가족 산책길 사이 거리: 7.603 km

남산구간: 코스의 거리 4.2km
남산구간: 장충체육관~국립극장~N서울타워~남산 팔각정~백범광장

용산가족 산책길와 남산구간 사이 거리: 7.381 km

덕수궁 산책길: 코스의 거리 6.02km
덕수궁 산책길: 덕수궁 돌담길~대한문~경운궁 양이재~홍화문~정동공원

남산구간와 덕수궁 산책길 사이 거리: 2.699 km

당신이 산책할 총 거리 : 51.239 km
플로이드 워셜 알고리즘 실행 시간: 0.1473 seconds

종료 코드 0(으)로 완료된 프로세스
```

다익스트라 알고리즘과 플로이드 워셜 알고리즘 모두 최단 경로 문제를 해결하기 위한 방법으로,

간선의 가중치를 기반으로 최단 경로를 계산하며, 최단 경로를 추적하기 위한 데이터를 저장한다는 공통점이 있다.

다익스트라 알고리즘이 특정 노드에서 다른 모든 노드까지의 최단 경로를 계산하는 반면, 플로이드 워셜 알고리즘은 모든 노드 쌍 간 최단 경로를 계산한다는 차이점이 존재한다. 이 때문에 플로이드 워셜의 시간복잡도(O(V^3)가 다익스트라의 시간복잡도(O((V+E)logV))에 비해 높다.

플로이드 워셜 알고리즘은 모든 노드 쌍 간 최단 경로를 한 번에 계산하고 저장하여 사용자 요청 시 빠르게 경로를 제공할 수 있다. 즉, 모든 경로 정보를 한 번에 계산하기 때문에 응답 속도가 빠르지만, 계산량이 많아 노드 수가 많으면 성능이 저하될 수 있다. 반면, 다익스트라 알고리즘은 한 출발점에서 특정 목표 노드까지의 경로를 실시간으로 계산하는 데 초점을 맞춘다. 즉, 출발점에서 하나의 경로만 필요할 때 효율적이지만, 전체 경로를 반복적으로 계산할 경우 비효율적이다.

이와 같이 두 알고리즘은 서로 다른 상황에서 각각의 강점을 발휘하며, 문제 요구사항에 맞게 선택적으로 활용해야 한다.

3. 벨만-포드 알고리즘
벨만-포드 알고리즘은 음수 가중치 간선을 허용하고 음수 사이클을 탐지할 수 있는 특징이 있다. 우선 그래프를 인접 리스트 형태로 표현한 후, 각 노드를 이웃 노드와 해당 노드에 대한 가중치로 구성한다. 시작 노드를 지정한 뒤, 다른 모든 노드까지의 최단 경로를 계산한다. 모든 노드까지의 거리를 무한대로 초기화하고, 시작 노드의 거리는 0으로 설정한다. 또한, 각 노드의 이전 노드를 추적하기 위한 딕셔너리를 초기화하여 최단 경로를 복원하는 데 필요한 정보를 저장한다.

거리 갱신 단계에서는 노드 개수 - 1만큼 반복하여 각 노드의 이웃 노드에 대해 거리를 갱신한다. 매 반복마다 현재 노드에서 이웃 노드로 가는 경로의 거리를 계산한 후, 기존 거리보다 짧을 경우 거리와 이전 노드 정보를 갱신한다.
음수 사이클 탐지 단계에서는 모든 간선을 한 번 더 확인하여 음수 사이클이 있는지 검사한다. 만약 거리 갱신이 발생하면 음수 사이클이 존재한다고 판단하고, 이를 처리하기 위해 해당 간선을 그래프에서 제거한다. 음수 사이클이 발견되면 알고리즘을 재귀적으로 다시 실행하여 최단 경로를 다시 계산한다.
알고리즘의 마지막 단계에서는 최단 거리와 각 노드의 이전 노드를 포함하는 딕셔너리를 반환한다. 이 정보는 최단 경로를 추적하는 데 유용하며, 사용자가 원하는 최단 경로를 쉽게 복원할 수 있도록 돕는다.
그래프 생성 시에는 HaverSine 거리 기반 간선 생성으로 초기 연산을 최적화하고, 필요 시 API를 활용해 정확도를 높이는 방식으로 설계되었다. 이를 통해 산책 경로 추천 및 최단 거리 계산이 효율적이고 확장 가능한 형태로 구현되었다.

5.2 구현 도구
본 프로그램은 데이터 처리와 그래프 기반 경로 탐색을 위해 파이썬 언어를 사용하여 구현하였다. 파이썬은 CSV 데이터와 같은 다양한 형식의 데이터를 읽고 처리하는 데 있어 효율적이며, 직관적인 문법과 풍부한 라이브러리 지원으로 개발 생산성을 높일 수 있다. 특히, 그래프 구조를 다룰 때 데이터 조작이 용이하여 본 프로젝트의 요구사항에 적합하였다.

개발 환경으로는 파이참 커뮤니티를 사용하였다. 파이참은 코드 작성, 디버깅, 실행 등 모든 개발 과정을 지원하며, CSV 파일 관리 및 API 통신 작업에서 발생할 수 있는 오류를 빠르게 파악할 수 있도록 돕는다. 또한, 커뮤니티 버전은 무료로 제공되며, 프로젝트의 기능 구현에 필요한 플러그인과 도구들을 충분히 활용할 수 있는 장점이 있다.

외부 거리 계산 및 도보 경로 데이터를 확보하기 위해 카카오 지도 API를 사용하였다. 이 API는 출발 지점과 도착 지점 간의 실제 도보 거리 및 소요 시간을 반환하며, 간단한 HTTP 요청으로 데이터를 제공받을 수 있어 연동이 용이하다. 또한, 무료 제공량이 있어 초기 테스트 및 개발 과정에서 비용 효율성을 확보할 수 있었다. 이를 통해 추천 경로의 거리 데이터를 유클리드 거리에서 실제 도보 거리로 보정하여 정확도를 높이는 데 활용하였다.

6. 결과
6.1. 결과물 설명
코스 노드 구조체

```
# 구조체
class CourseNode:
    def __init__(self, course_name, distance, start_point_name, last_point, course_level, course_points=None, start_coordinates=None, end_coordinates=None):
        self.course_name = course_name
        self.distance = distance
        self.start_point_name = start_point_name
        self.last_point = last_point
        self.course_level = course_level
        self.course_points = course_points # 코스 세부 코스
        self.start_coordinates = self.convert_to_tuple(start_coordinates)
        self.end_coordinates = self.convert_to_tuple(end_coordinates)

    def __repr__(self):
        return (f"CourseNode(course_name='{self.course_name}', "
                f"distance='{self.distance}', "
                f"start_point_name='{self.start_point_name}', "
                f"last_point='{self.last_point}', "
                f"course_level={self.course_level}', "
                f"course_points='{self.course_points}', "
                f"start_coordinates={self.start_coordinates}', "
                f"end_coordinates='{self.end_coordinates}')")

    def convert_to_tuple(self, coord_str):
        # 문자열 좌표를 튜플로 변환
        coord_str = coord_str.strip("()")  # 괄호 제거
        coord_list = coord_str.split(",")  # 쉼표로 분리
        return float(coord_list[0]), float(coord_list[1])  # 튜플로 변환
```

노드 간 거리를 구하는 haversine 함수

```
# 지구의 곡률을 고려한 거리 계산 (단위: km)
def haversine_distance(lat1, lon1, lat2, lon2):
    if None in [lat1, lon1, lat2, lon2]:
        return None  # 좌표가 None이면 거리 계산을 하지 않음

    R = 6371  # 지구 반지름 (단위: km)
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])  # 각도를 라디안으로 변환

    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = math.sin(dlat / 2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    return R * c
```

1. 다익스트라 알고리즘

엣지를 생성하는 메소드

```python
# 두 객체 간의 거리 비교 (간선)
def create_edges(course_nodes):
    edges = {}  # 딕셔너리 형태로 수정
    for i in range(len(course_nodes)):
        for j in range(i + 1, len(course_nodes)):
            node1 = course_nodes[i]
            node2 = course_nodes[j]

            # 좌표가 None인 경우 건너뛰기
            if None in [node1.end_coordinates, node2.start_coordinates]:
                continue

            # 출발 코스의 마지막 지점과 도착 코스의 시작 지점 사이 거리 계산
            end_lat1, end_lon1 = node1.end_coordinates
            start_lat2, start_lon2 = node2.start_coordinates

            distance = haversine_distance(end_lat1, end_lon1, start_lat2, start_lon2)

            # 거리 기준에 맞는 간선 추가 (distance가 None이 아니고 3km 이하일 경우)
            if distance is not None and distance <= 3:  # 3km 이하일 때 간선을 그린다.
                if node1.course_name not in edges:
                    edges[node1.course_name] = []
                if node2.course_name not in edges:
                    edges[node2.course_name] = []
                edges[node1.course_name].append((node2.course_name, distance))
                edges[node2.course_name].append((node1.course_name, distance))
    return edges
```

다익스트라 알고리즘

```python
# 다익스트라 알고리즘 구현 (경로 추적 추가)
def dijkstra(edges, start_node):
    # 최단 거리 테이블 초기화 (모든 거리 초기값은 무한대, 출발 노드는 0)
    distances = {node: float('inf') for node in edges}
    distances[start_node] = 0
    # 우선순위 큐 (최소 힙) 초기화
    priority_queue = [(0, start_node)]  # (거리, 노드)

    # 이전 노드 추적을 딕셔너리
    previous_nodes = {node: None for node in edges}

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        # 현재 노드에 대해 이미 처리된 경우 건너뛰기
        if current_distance > distances[current_node]:
            continue

        # 인접한 노드들 탐색
        for neighbor, weight in edges.get(current_node, []):
            distance = current_distance + weight

            # 더 짧은 경로가 발견되면 갱신
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_nodes[neighbor] = current_node  # 이전 노드 기록
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances, previous_nodes
```

2. 플로이드 워셜 알고리즘

간선을 생성하는 함수

```python
# 두 객체 간의 거리 비교 (간선)
def create_edges(course_nodes):
    edges = {}  # 딕셔너리 형태로 수정
    for i in range(len(course_nodes)):
        for j in range(i + 1, len(course_nodes)):
            node1 = course_nodes[i]
            node2 = course_nodes[j]

            # 좌표가 None인 경우 건너뛰기
            if None in [node1.end_coordinates, node2.start_coordinates]:
                continue

            # 출발 코스의 마지막 지점과 도착 코스의 시작 지점 사이 거리 계산
            end_lat1, end_lon1 = node1.end_coordinates
            start_lat2, start_lon2 = node2.start_coordinates

            distance = haversine_distance(end_lat1, end_lon1, start_lat2, start_lon2)

            # 거리 기준에 맞는 간선 추가 (distance가 None이 아니고 3km 이하일 경우)
            if distance is not None and distance <= 3:  # 3km 이하일 때 간선을 그린다.
                if node1.course_name not in edges:
                    edges[node1.course_name] = []
                if node2.course_name not in edges:
                    edges[node2.course_name] = []
                edges[node1.course_name].append((node2.course_name, distance))
                edges[node2.course_name].append((node1.course_name, distance))
    return edges
```

플로이드 워셜 알고리즘

```python
# 플로이드 워셜 알고리즘 구현 (경로 추적 포함)
def floyd_warshall(course_nodes, edges):
    # 노드 이름을 인덱스로 변환
    node_index = {node.course_name: idx for idx, node in enumerate(course_nodes)}
    n = len(course_nodes)

    # 거리 행렬 및 경로 추적 행렬 초기화
    dist = [[float('inf')] * n for _ in range(n)]
    next_node = [[None] * n for _ in range(n)]

    # 자기 자신으로의 거리는 0
    for i in range(n):
        dist[i][i] = 0

    # 간선으로 초기 거리 설정
    for node, neighbors in edges.items():
        for neighbor, weight in neighbors:
            u, v = node_index[node], node_index[neighbor]
            dist[u][v] = weight
            next_node[u][v] = v

    # 플로이드 워셜 알고리즘
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    next_node[i][j] = next_node[i][k]

    return dist, next_node, node_index
```

3. 벨만 포드 알고리즘

간선을 생성하는 함수

```python
# 두 객체 간의 거리 비교 (간선)
def create_edges(course_nodes):
    edges = {}  # 딕셔너리 형태로 수정
    for i in range(len(course_nodes)):
        for j in range(i + 1, len(course_nodes)):
            node1 = course_nodes[i]
            node2 = course_nodes[j]

            # 좌표가 None인 경우 건너뛰기
            if None in [node1.start_coordinates, node2.end_coordinates]:
                continue

            # 두 코스의 시작점과 끝점 좌표
            start_lat1, start_lon1 = node1.end_coordinates[0], node1.end_coordinates[1]
            end_lat2, end_lon2 = node2.start_coordinates[0], node2.start_coordinates[1]

            # 두 좌표 간 haversine 거리 계산
            distance = haversine_distance(start_lat1, start_lon1, end_lat2, end_lon2)

            # 코스 레벨에 따라 음수 가중치 설정
            level_weight = -0.1e-17 * int(node1.course_level)  # 레벨에 따라 음수 가중치 적용
            distance += level_weight

            # 거리 기준에 맞는 간선 추가 (distance가 None이 아니고 3km 이하일 경우)
            if distance is not None and distance <= 3:  # 3km 이하일 때 간선을 그린다.
                if node1.course_name not in edges:
                    edges[node1.course_name] = []
                if node2.course_name not in edges:
                    edges[node2.course_name] = []
                edges[node1.course_name].append((node2.course_name, distance))
                edges[node2.course_name].append((node1.course_name, distance))
    return edges
```

벨만 포드 알고리즘

```python
# 벨만포드 알고리즘 구현
def bellman_ford(edges, start_node):
    # 음수 가중치 적용 및 음수 사이클 검출.
    # 노드 초기화
    distances = {node: float('inf') for node in edges}
    distances[start_node] = 0
    previous_nodes = {node: None for node in edges}

    # 벨만포드 거리 갱신 (노드 개수 - 1만큼 반복)
    for _ in range(len(edges) - 1):
        for node, neighbors in edges.items(): # 각 노드에 대해
            for neighbor, weight in neighbors: # 각 이웃에 대해
                if distances[node] + weight < distances[neighbor]: # 거리 갱신이 발생하면
                    distances[neighbor] = distances[node] + weight # 거리 갱신
                    previous_nodes[neighbor] = node

    # 음수 사이클 확인
    for node, neighbors in edges.items(): # edges.items한 노드와 이웃들을 반환
        for neighbor, weight in neighbors:
            if distances[node] + weight < distances[neighbor]: # 음수 사이클이 있으면(거리 갱신이 발생하면)
                print(f"Negative-weight cycle detected: {node} -> {neighbor}")
                # 음수 사이클이 있는 간선을 제거
                edges[node].remove((neighbor, weight))
                if not edges[node]:
                    del edges[node]
                return bellman_ford(edges, start_node)  # 다시 계산

    return distances, previous_nodes
```

6.2. 시험 및 평가

평가항목	평가기준	근거 (보통기준)	평가점수				
			아주미흡 (1)	미흡 (2)	보통 (3)	우수 (4)	아주우수 (5)
실행 시간	sec	20	5	10	20	30	40
시간복잡도	Big O	100	20	60	100	140	180

시간 복잡도 분석

다익스트라 알고리즘: $O((V+E)\log V)$: 180점
플로이드 워셜 알고리즘:$O(V^3)$: 60점
벨만 포드 알고리즘: $O(V+E)$: 140점

*V는 노드의 개수, E는 간선의 개수

실제 실행시간 분석

다익스트라 알고리즘 < 벨만 포드 알고리즘 < 플로이드 워셜 알고리즘

다익스트라 알고리즘: 40점
플로이드 워셜 알고리즘: 10점
벨만 포드 알고리즘: 30점

순위/총점
1. 다익스트라 알고리즘: 220점
2. 벨만 포드 알고리즘: 170점
3. 플로이드 워셜 알고리즘: 70점

6.3 기대성과
1. 학문적 / 기술적 성과
다양한 알고리즘을 활용하여 서울의 다양한 걷기 코스 정보를 최적화하는 연구에 기여한다. 다익스트라, 플로이드-워셜, 벨만-포드 알고리즘의 비교 분석을 통해 최적 경로 탐색 알고리즘의 실제 응용 가능성을 평가하며, 데이터 전처리 및 그래프 모델링 기술의 발전에 기여할 수 있다. 이러한 성과는 스마트 시티, 관광 데이터 분석 등의 학문적 영역에서 응용할 수 있다.

2. 경제적 효과
효율적인 걷기 코스 추천은 서울을 방문하는 외국인 및 타 지역 방문자의 관광 경험을 향상시켜 관광 산업 활성화에 기여할 수 있다. 또한, 관광객의 이동 효율성이 증대되면서 지역 경제의 활성화와 관광지 연계성 강화 등 간접적인 경제적 효과를 기대할 수 있다.

3. 환경적 영향
최적 경로 추천은 불필요한 이동을 줄여 걷기 기반의 친환경적 여행 문화를 조성한다. 이는 대중교통 및 도보 여행의 선호도를 높이고, 이산화탄소 배출 감소 등 지속 가능한 환경 관리에 긍정적인 영향을 미칠 수 있다.

4. 윤리적 영향
걷기 코스를 제안하는 알고리즘은 공정성과 투명성을 기반으로 설계되어야 하며, 특정 관광지나 지역에 과도한 집중이 발생하지 않도록 신중하게 고려되었다. 이는 서울을 찾는 관광객들에게 공평한 기회를 제공하며, 지역 간 균형 발전을 지원하는 윤리적 책임을 준수한다.

7. 작업진행 방법
7.1 설계 일정 및 역할 분담

	작업이름	11. 2024				12. 2024					담당자
		3-9	10-9	17-9	24-9	1-10	8-10	15-10	22-10	29-10	
1	프로젝트 시작										염경민, 김채원, 류경록, 정재환
2	프로젝스 설계										염경민, 김채원, 류경록, 정재환
3	프로젝트 구현										염경민, 김채원, 류경록, 정재환
4	프로젝트 연동										염경민, 김채원, 류경록, 정재환
5	시험 실시										염경민, 김채원, 류경록, 정재환
6	시험 결과 보고										염경민, 김채원, 류경록, 정재환
7	설계 프로세서 문서화										염경민, 김채원, 류경록, 정재환
8	프로젝트 종료										염경민, 김채원, 류경록, 정재환

8. 결론
8.1 결론
본 프로젝트는 서울의 걷기 코스 데이터를 활용하여 최적 경로를 탐색하는 시스템을 개발하는 데 중점을 두었다. 이 과정에서 다음과 같은 단계가 진행되었다.

1. 동기 유발
도시 환경에서 걷기 코스는 건강과 환경을 고려한 중요한 요소로, 사용자에게 효율적이고 안전한 경로를 제공하는 필요성이 제기되었다. 특히, 서울과 같은 대도시에서 다양한 걷기 코스를 효과적으로 연결하고 정보 제공을 통해 관광객과 시민의 경험을 향상시키고자 하였다.

2. 문제 도출
기존의 경로 탐색 시스템은 사용자의 요구를 충분히 반영하지 못하거나, 실시간 거리 및 소요 시간 정보를 제공하지 못하는 한계가 있었다. 따라서 사용자 맞춤형 경로 추천 시스템이 필요하다는 문제를 도출하였다.

3. 진행

프로젝트는 Python과 Kakao API를 활용하여 구현되었으며, 사용자로부터 출발지와 도착지를 입력받아 최적의 경로를 탐색하고, 각 경로에 대한 상세 정보를 제공하는 방향으로 진행되었다. 데이터 수집과 전처리를 통해 걷기 코스 정보를 정리하고, 알고리즘을 설계하여 경로 탐색 기능을 구현하였다.

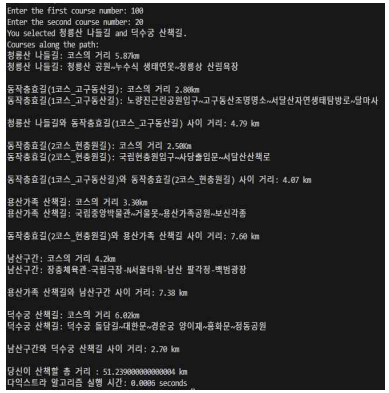
4. 설계

알고리즘 설계 단계에서는 다음의 세 가지 알고리즘을 비교 분석하여 최적의 경로 탐색 방식을 결정하였다.

1) 다익스트라 알고리즘 (Dijkstra Algorithm)
- 입력: 출발지와 도착지 노드를 입력한다.

- 처리
  * 출발지에서 다른 모든 노드로의 최단 경로를 탐색한다.
  * 우선순위 큐를 사용하여 효율적으로 거리를 갱신한다.

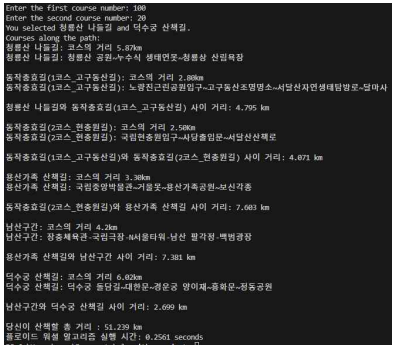
- 출력 결과
 * 경로 탐색 속도가 빠르며 시간 복잡도는 O(E+VlogV)이다.
 * 각 노드와의 최단 경로만 계산하므로 메모리 사용량이 효율적이다.
 * 사용자가 선택한 경로의 주요 지점과 연결된 세부 경로가 명확히 출력된다.



2) 플로이드-워셜 알고리즘 (Floyd-Warshall Algorithm)
- 입력: 출발지와 도착지 노드를 입력한다.

- 처리
  * 모든 노드 쌍 간 최단 경로를 계산한다.
  * 중간 노드를 경유하여 경로를 갱신한다.

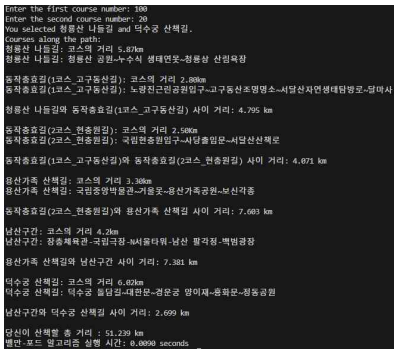
- 출력 결과
  * 시간 복잡도 $O(V^3)$로 데이터 크기가 작을 경우에도 실행 시간이 길다.
  * 모든 노드 간 최단 경로를 계산하므로 다중 쿼리에 적합하다.
  * 전체 경로 탐색 결과가 더 복잡하며, 다익스트라보다 출력 데이터가 방대하다.



3) 벨만-포드 알고리즘 (Bellman-Ford Algorithm)
- 입력: 출발지와 도착지 노드를 입력한다.

- 처리
  * 출발지에서 다른 모든 노드로의 최단 경로를 탐색한다.
  * 각 노드에 대해 반복하여 거리 갱신을 수행한다.

- 출력:
  * 음수 가중치 간선을 허용하며, 음수 사이클을 탐지할 수 있다.
  * 최단 경로가 보장되지 않는 경우도 있으므로 주의가 필요하다.
  * 메모리 사용량은 다익스트라 알고리즘보다 상대적으로 더 크다.



5. 결론

본 프로젝트의 결과물은 제안한 문제를 효과적으로 해결하고 있으며, 사용자 맞춤형 경로 추천 기능을 통해 걷기 코스의 정보 제공 및 효율적인 경로 탐색을 실현하였다. 이를 통해 관광객과 시민의 경험을 향상시키는 데 기여할 수 있다.

– 프로젝트 수행 과정을 통해 배운 점은 다음과 같다:
  * 기술적 측면: 다양한 알고리즘을 비교하고 적용함으로써 경로 탐색의 효율성을 이해하게 되었다. 또한 Kakao API를 활용한 실시간 데이터 처리의 중요성을 체감하였다.
  * 문제 해결 능력: 사용자의 요구를 충족시키기 위한 시스템 설계 과정에서 문제 해결 능력을 키웠다. 이러한 경험은 프로젝트 목표인 사용자 맞춤형 경로 추천 시스템을 성공적으로 구현하는 데 기여하였다.
  * 협업과 소통: 프로젝트 진행 과정에서 팀원 간의 협업과 소통의 중요성을 깊이 이해하게 되었다. 다양한 의견을 수렴하고 조율하는 과정에서 팀워크의 가치를 실감하였으며, 이를 통해 최적의 결과물을 도출할 수 있었다. 이러한 협업 경험은 프로젝트 목표 달성을 위한 필수적인 요소임을 인식하게 해주었다.

8.2 향후 계획

현재 해당 프로젝트는 알고리즘 구현 단계까지 완료했으며, 향후 계획은 사용자 친화적인 웹 또는 앱을 개발하여 실제 사용자가 효율적으로 이용할 수 있도록 시스템을 확장하는 데 중점을 둔다. 특히, 현재 서울시를 중심으로 개발된 시스템을 전국 단위로 확장하는 방안을 고려하고 있다. 이를 통해 다양한 지역의 사용자들이 보다 넓은 범위에서 최적 경로 탐색 서비스를 이용할 수 있도록 할 예정이다. 전국 단위로의 확장에는 다양한 도시와 지역의 데이터 통합이 포함되어, 사용자 맞춤형 경로 제공이 가능하도록 할 계획이다.