

**Open Source SW Project**

**Final Report**

**Team5 : B1G4**

**좌석 혼잡도 기반 경로 추천 서비스**

Submitted By

20162406 이설희

20174470 전희수

20172030 김윤희

20163453 현도연

20161344 허정우

June 22 2019

목차

[**1.** **프로젝트 소개** 3](#_Toc12139877)

[**A.** **주제** 3](#_Toc12139878)

[**B.** **필요성** 3](#_Toc12139879)

[**2.** **관련 기술, 서비스, 오픈소스 소프트웨어 소개** 4](#_Toc12139880)

[**A.** **차별성** 4](#_Toc12139881)

[**B.** **오픈소스 커뮤니티에 미칠 효과** 6](#_Toc12139882)

[**3.** **구현물 설명** 7](#_Toc12139883)

[**A.** **아키텍쳐** 7](#_Toc12139884)

[**B.** **Workflow / Usage Scenario** 11](#_Toc12139885)

[**C.** **핵심 기술 설명** 11](#_Toc12139886)

[**1)** **버스 데이터 파싱** 11](#_Toc12139887)

[**2)** **혼잡도 데이터 분석** 12](#_Toc12139888)

[**3)** **버스 경로 추천 알고리즘** 15](#_Toc12139889)

[**4)** **안드로이드** 17](#_Toc12139890)

[**5)** **서버** 22](#_Toc12139891)

[**6)** **지하철 경로 추천 알고리즘** 23](#_Toc12139894)

[**D.** **Implementation 이후로 개선된 부분** 28](#_Toc12139895)

[**4.** **성능 평가 및 구현물 데모** 28](#_Toc12139896)

[**5.** **Github repository** 31](#_Toc12139898)

[**6.** **기타** 34](#_Toc12139899)

1. **프로젝트 소개**
   1. **주제**

이번 프로젝트의 주제는 ‘혼잡도 기반 경로 추천 서비스’ 이다. 기존의 여러 경로 추천 서비스는 보통 최단거리, 혹은 최단시간 소요의 길을 안내한다. 하지만 일부 사람들의 경우, 한 장소에서 다른 장소로 이동할 때에 최단거리 혹은 최단 시간이 우선순위가 아닌 경우도 있다. 몸이 불편하거나 이동이 불편한 교통약자, 혹은 짐이 많거나 피곤에 지친 사람들의 경우이다. 이들은 단순히 빠르게 이동하기보다는 대중교통에서 앉아갈 수 있기를 우선적으로 바라게 된다. 그런데 현재 혼잡도에 기반하여 최대한 앉아서 이동할 수 있도록 길을 추천해주는 서비스는 존재하지 않는다. (‘차별성’ 에서 기존의 서비스와의 비교를 통해 상세하게 다룰 예정이다) 따라서 이 주제로 해당 프로젝트를 진행하게 되었다. 혼잡도와 환승 횟수, 최대 시간 등을 고려하여 사람들이 최대한 편하게 길을 찾을 수 있도록 도와주고, 이 서비스를 최종적으로는 모듈화하여 다른 프로젝트나 서비스에도 쉽게 추가하고 실행할 수 있도록 구현할 예정이다

* 1. **필요성**

먼저, 신체적으로 활동에 제약을 받는 교통 약자는 대중교통을 이용한 이동이 쉽지 않다. 한국교통장애인 협회의 전국 교통약자 현황 통계자료에 따르면, 장애인 250만명, 65세 이상 노인 650만명, 9세 이하 유아 460만명을 포함하여 임산부, 보행 불편자, 신체 약자 등 전체 인구의 약 33%(1,700만명)에 해당하는 비율이 이동제약자인 것으로 확인되었다. 비록 이동에 어려움이 있는 교통약자라도 대중교통 이용은 불가피하다. 실제로 지역 내 외출 시 주로 이용하는 교통수단 실태 조사 결과 임산부(58.9%)는 버스, 고령자는 지하철(46.4%)의 빈도가 가장 높았고 장애인은 버스(24.6%), 지하철(22.4%)을 비슷한 수준으로 이용함을 확인할 수 있었다.

그리하여 교통약자의 이동권 문제를 해결하기 위해 다양한 서비스 및 지원정책들이 꾸준히 제안되고 있으며 각종 시스템이 등장하였다. 대표적인 몇 가지 예로는 교통약자 승차지원시스템과 버스 예약 어플리케이션이 있다. 교통약자 승차지원시스템은 버스 정류소에 설치된 단말기로부터 버스번호, 교통약자 유형 등의 대기 상황을 입력하면 운전기사가 출입문 개방 및 도착 알림 등을 제공하는 서비스로 현재 도입을 앞두고 있다. MyBus는 “한국교통약자버스이용협동조합”에서 제공하는 버스 예약 어플리케이션으로 장애인들을 대상으로 운영되고 있는 시스템이다. 그럼에도 대중교통 내부의 혼잡도와 관련하여, 최대한 앉아 이동할 수 있도록 경로를 제공하는 서비스는 아직 존재하지 않았다.

뿐만 아니라, 해당 서비스는 교통 약자가 아닌 직장인 또는 대학생의 현대인에게도 꼭 필요하다. 간단히 주변만 둘러보아도 피곤한 몸을 이끌고 혼잡한 대중교통에 몸을 실은 채 언제쯤 자리가 날까 눈치를 보는 수많은 현대인이 흔히 보인다. 뿐만 아니라 짐이 많은 사람, 몸 컨디션이 좋지 않은 사람, 피로에 지친 시민 등 조금이라도 여유 있는 대중교통에서 편안히 이동하고자 하는 모두가 필요로 하는 서비스임을 알 수 있다. 이와 관련한 사용자 니즈를 만족시키고자 2017년 5월부터 서울 시내 버스에서 실시간 버스 차내 혼잡도 안내서비스를 제공하고 있다. 버스 정류소에서, 그리고 어플리케이션에서 실시간 운행되는 버스의 좌석 현황을 확인할 수 있는데 이는 직관적인 혼잡도 이해를 돕지만 실제적으로 혼잡도 고려한 경로추천까지는 이어지지 않는다는 한계가 있었다.

따라서, 해당 프로젝트는 혼잡에 취약한 교통 약자와 지친 현대인에게 여유로운 대중교통에서 되도록 앉아갈 수 있도록 하는 서비스를 제공할 것이다. 이에 추가적으로, 해당 프로젝트가 성공적으로 진행되었을 때 사람들의 이동 경로를 기존의 최단 경로에서 혼잡도 기반 경로로 분산한다면 부분적인 혼잡도 해소까지도 기대할 수 있으므로 더욱 필요성이 부각된다.

1. **관련 기술, 서비스, 오픈소스 소프트웨어 소개**
   1. **차별성**

이미 출시되어 있는 여러 어플리케이션과의 비교를 통해 차별성을 알아보았다

* + 1. 지하철 종결자, 카카오지하철, 또타지하철, 지하철매니저

: 해당 앱들이 제공하는 기능은 대표적으로 실시간 지하철 도착 정보, 하차 알람 기능, 출발시간 기준 / 도착시간 기준 경로 검색, 최소 환승 또는 최소 시간으로 경로를 검색하는 기능이다. 이 경우 대중교통 중 지하철 이용만 고려했다는 점과, 혼잡도를 고려하지 않고 단순히 최소 시간만 생각했다는 점이 특징이다

* + 1. 지하철 혼잡도(바글바글), 지하철혼잡도알리미, R-Subway (<https://mplatform.seoul.go.kr/w/contest/award/2016/wnpz/selectWinner.do>)

: 지하철 혼잡도 통계 값을 제공하는 앱이다. 이 경우 대중교통 중 지하철 이용만 고려했다는 점과, 혼잡도는 제공하지만 구체적인 길 찾기 기능은 제공하지 않는다는 단점이 있다. 또한 앱 자체에 오류가 많다는 후기들이 있었다. R-Subway의 경우 역시 지하철 이용만 고려했지만 부가적으로 통계적인 혼잡도 정보와 역의 사고 여부를 제공한다는 특징이 있다.

* + 1. 전국 스마트 버스, 카카오버스, busconditionServer(<https://github.com/aerain/busconditionServer>)

: 실시간 버스 위치정보와 좌석 혼잡도 현황, 하차 알람 및 일반 경로 찾기를 지원하는 앱이다. 이 경우 대중교통 중 버스 이용만 고려했다는 점과 혼잡도 정보를 제공 만 하고 이를 고려한 경로 추천까지는 해주지 않는다는 한계가 있다.

* + 1. 서울교통포털, 실시간 교통정보, Jamco(<https://github.com/carolrizzi/Jamco>)

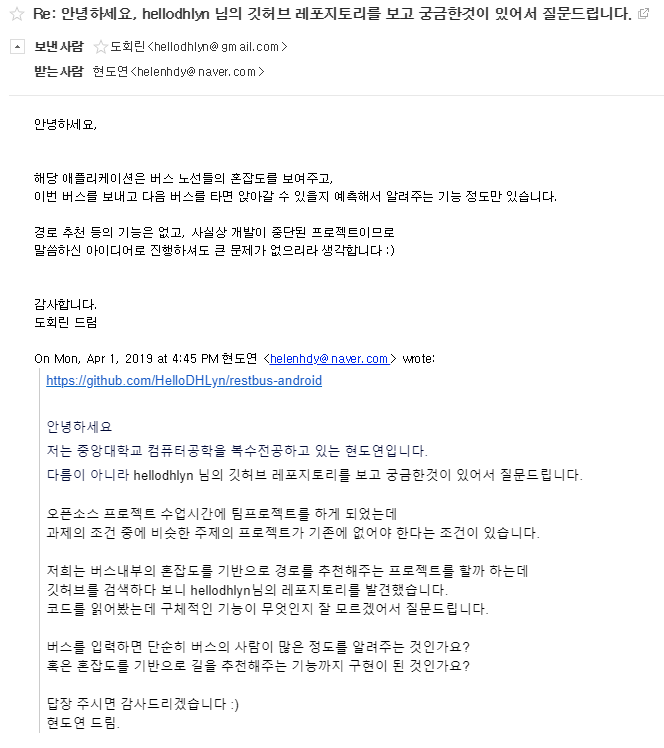
: 주변 지하철 및 정류소를 조회하고, 경로 검색 및 현재 일시 기준 도로통제정보를 확인, 통계 값에 기반한 교통예측정보 확인이 가능한 앱이다. 서울교통포털 앱은 버스와 지하철을 통합하여 길찾기를 최소시간으로 제공하지만 앱 자체가 너무 느리고 오류가 많다는 후기들이 있었다. 또한 실시간 교통정보 앱과 Jamco의 경우 대중교통 사용보다는 자가용 사용에 초점이 맞추어진 도로 별 혼잡도 제공 어플리케이션이었다.

* + 1. 카카오맵, 네이버 지도, 구글 맵 (일반 길찾기 앱)

: 주변 지하철, 정류소 조회 기능과 길찾기를 지하철, 버스, 택시, 도보 등 다양한 경로로 제공하는 앱이다. 버스와 지하철, 도보, 택시 등의 여러 수단을 제공하며 혼잡도 정보는 버스만 제공하고, 경로 찾기는 수단별로 최소 시간 경로들을 제공하였다. 버스 앱과 마찬가지로 혼잡도 정보가 단순히 제공되기만 하고 이를 고려한 경로 탐색은 이루어지지 않았다.

* + 1. Restbus(<https://github.com/HelloDHLyn/restbus-android>)

: github에 있는 해당 프로젝트의 경우, 버스의 혼잡도 정보를 이용한 길찾기 프로젝트였는데 정확히 어떤 기능을 구현하였는지 알 수 없어 해당 레퍼지토리의 소유자에게 이메일을 보내 우리 조의 프로젝트와의 차별성을 확인해 보았다. (자세한 내용은 아래 사진 첨부)



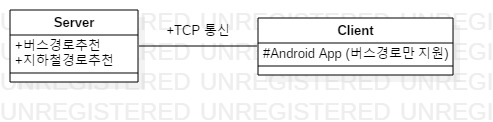
통합해 보았을 때 기존의 여러 앱 중 단순히 최단 시간이 아니라, 최대한 오래 앉아서 편안히 이동할 수 있도록 혼잡도 기반 길 찾기를 제공하는 앱은 없었다. 이 점에 차별성을 두어 본 프로젝트를 진행하고자 한다.

* 1. **오픈소스 커뮤니티에 미칠 효과**

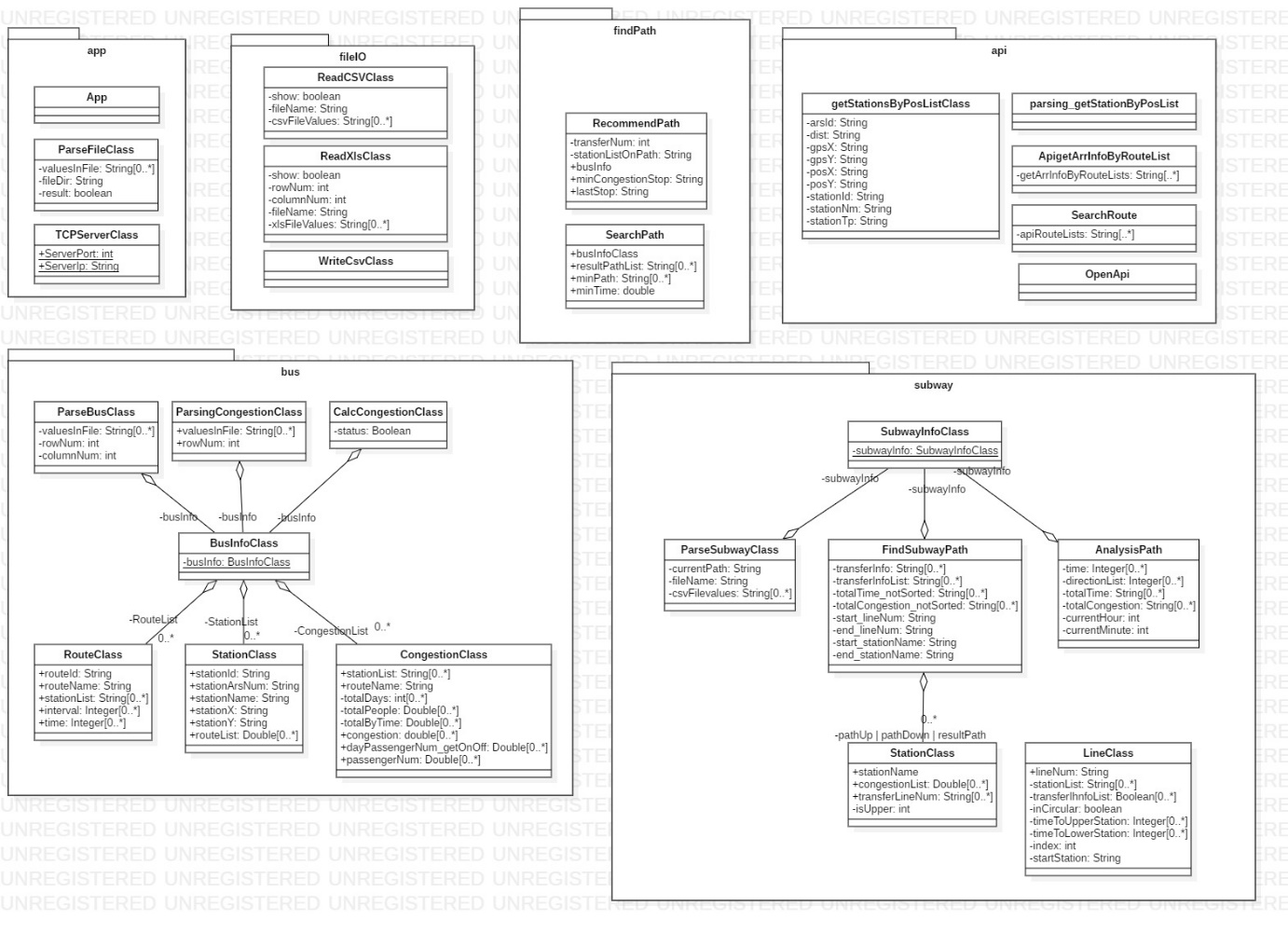
기존의 길찾기 (최단경로 기반)의 경우 다른 여러 프로젝트에 활용할 수 있도록 api가 오픈되어 있다. 마찬가지로 해당 프로젝트 역시 모듈화시켜 이식할 수 있게 제작하고 오픈소스 커뮤니티에 공개한다면, 다른 프로젝트 진행 시에 사람들이 사용할 수 있을 것이다. 단순히 시간을 우선순위로 한 경로들이 아니라, 사용자에 맞게 혼잡도를 우선적인 필터로 고려할 수 있으므로 많은 교통 약자와 피곤에 지친 현대인을 생각해 보았을 때 많은 수요가 예상된다.

1. **구현물 설명** 
   1. **아키텍쳐**
      1. Server와 Client

Server와 Client가 TCP통신을 하도록 했다. Android App의 형태로 사용자에게 UI를 제공하며 Server에게 통신을 하면 Server는 경로추천을 하여 다시 Client에게 보낸다.

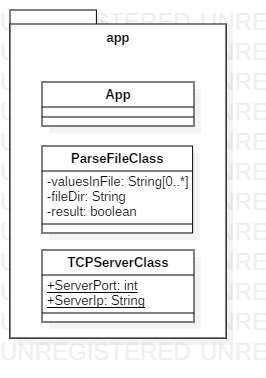
* + 1. 전체 Class Diagram

다음은 Server에 해당하는 class Diagram이다. 경로 추천 기능이 본 프로젝트의 핵심이기 때문에 Server에 들어가는 코드를 중심적으로 작성하였다.



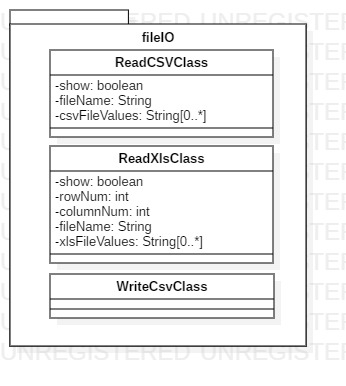
* + 1. App

main함수는 App 클래스 내부에 있으며 처음 Server를 실행할 경우 파일 읽기와 TCP 소켓을 여는 작업을 한다.



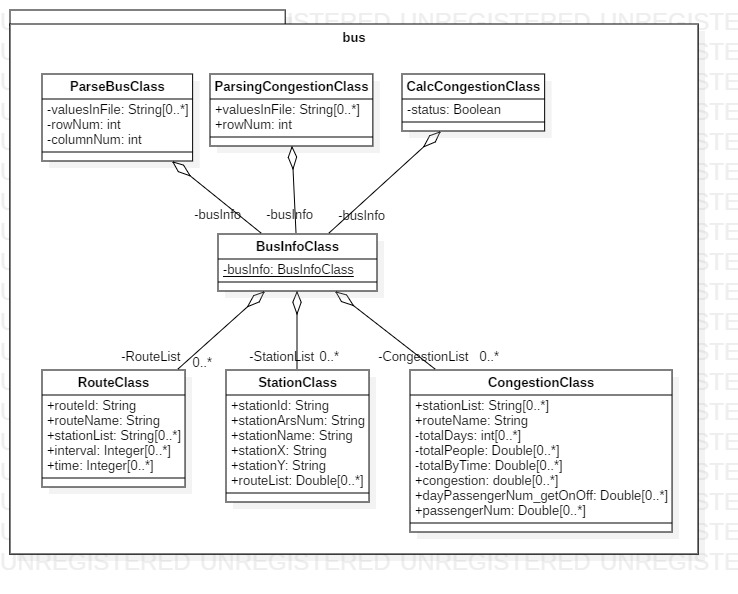
* + 1. fileIO

csv파일과 xls파일을 읽고 쓰는 기능을 담당한다.



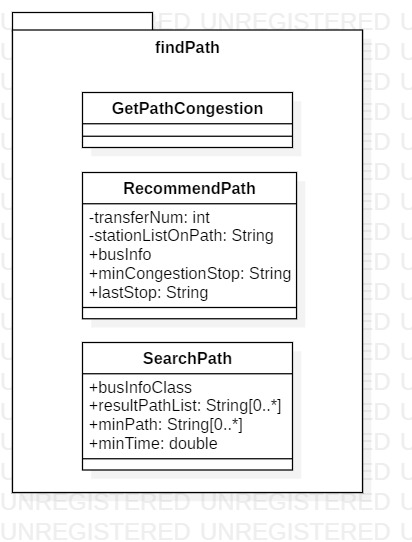
* + 1. bus

혼잡도 기반 버스 경로를 추천하기 위한 모든 버스노선, 버스 정류장, 혼잡도 등의 정보를 저장, 관리한다.



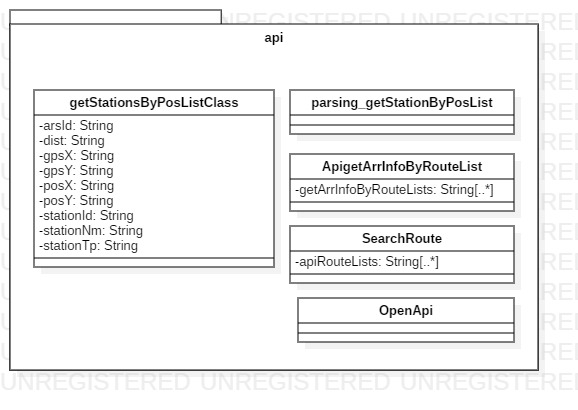
* + 1. findPath

혼잡도 기반 버스 경로 추천 알고리즘에 관련된 기능이다.



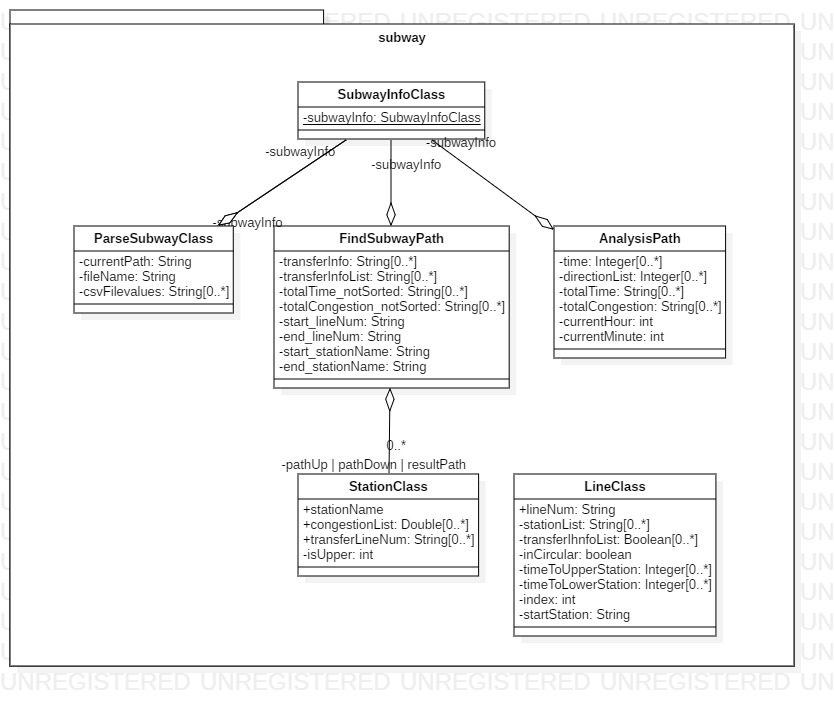
* + 1. api

혼잡도 기반 버스 경로 추천 시 필요한 api 호출을 할 때 사용되는 기능들이다.

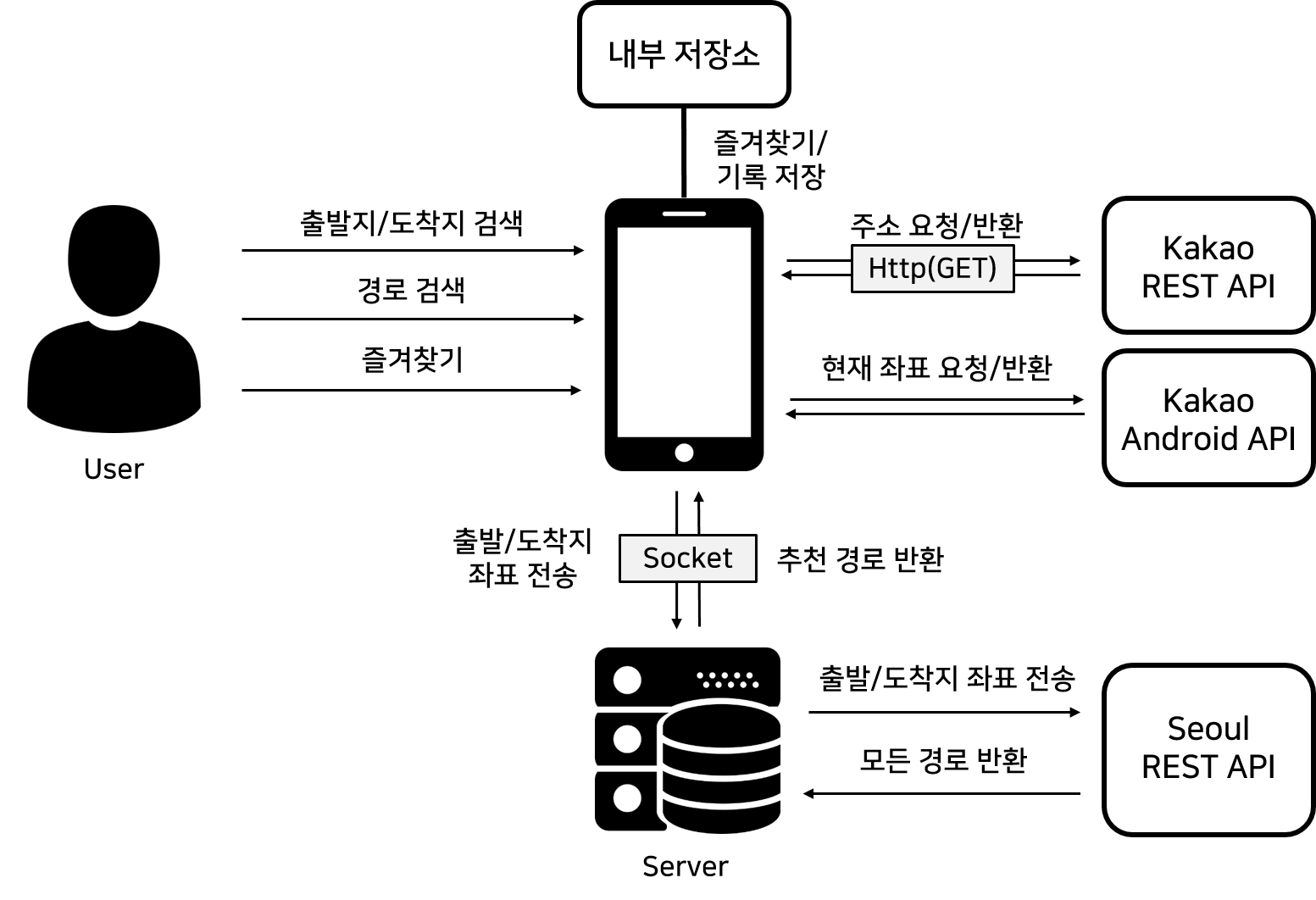


* + 1. subway

혼잡도 기반 지하철 경로를 계산하기 위한 기능이다. 이는 안드로이드와는 연동이 되지 않았다.



* 1. **Workflow / Usage Scenario**



* 1. **핵심 기술 설명** 
     1. **버스 데이터 파싱**
        1. 사용 데이터

공공데이터포탈(data.go.kr)에서 제공하는 데이터들을 사용하였다. 서울시 노선 현황 정보(2019년 1월 24일 기준), 서울시 버스 정류장 정보(2015-2018년), 서울시 버스 노선 정보(2019년 1월 24일 기준)을 이용해 파싱을 진행하였다. 해당 파일들은 모두 xls파일로 DataParsing/OSSWProj 에 위치해있다. ([https://github.com/b1g4/DataParsing/tree/master/ OSSWProj](https://github.com/b1g4/DataParsing/tree/master/%20OSSWProj))

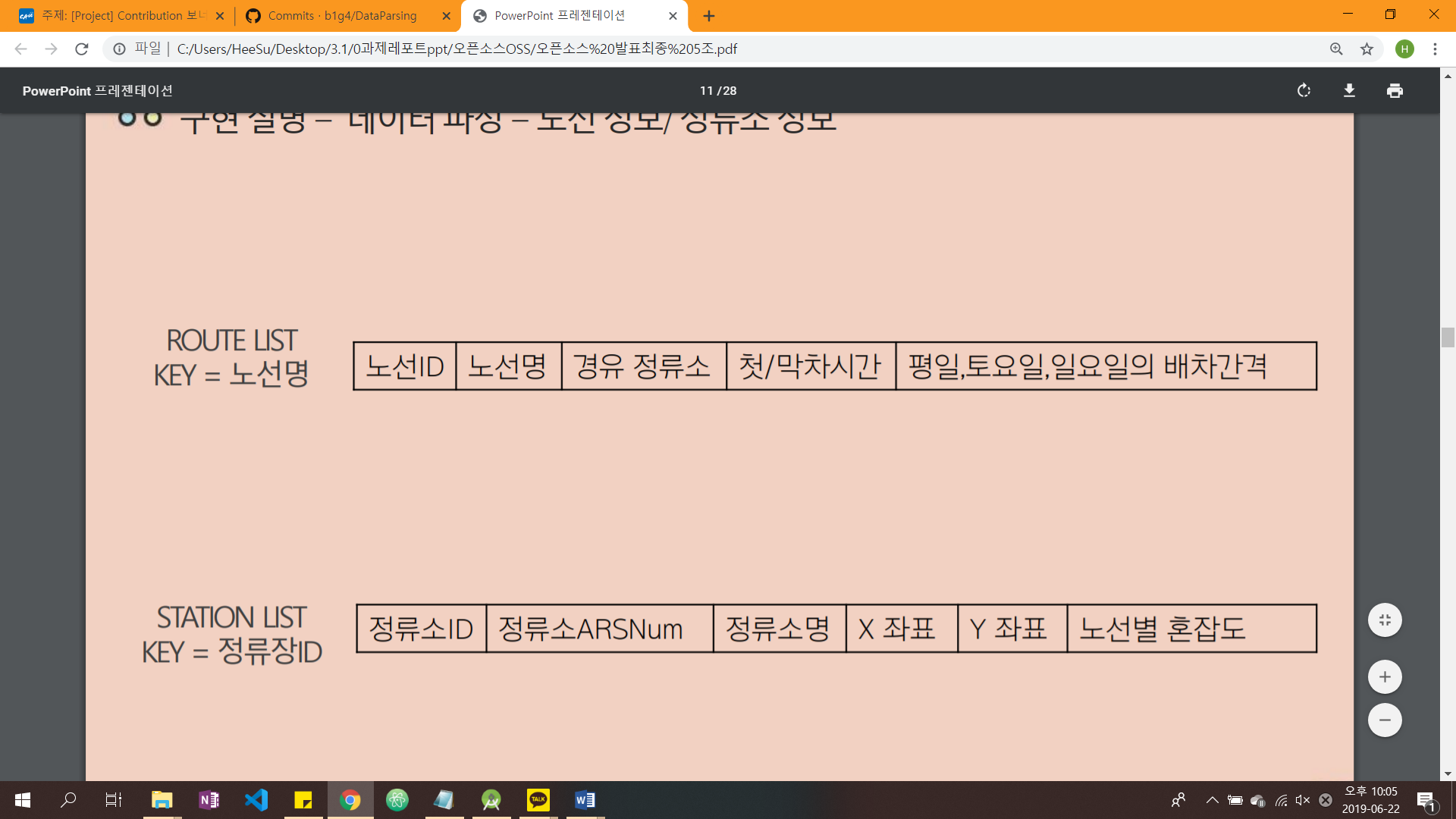
* + - 1. 데이터 파싱

먼저 데이터 파싱의 경우 따로 레포지토리를 두고 진행했다. ([https://github.com/ b1g4/DataParsing](https://github.com/%20b1g4/DataParsing)에서 진행하였다) 해당 레포지토리의 README 파일에 사용데이터 및 사용 API를 링크와 함께 명시해두었다.

데이터 파싱의 경우 먼저 얻은 데이터에서 필요로 하는 데이터를 얻어 가공하는 것이 목적이었다. 크게 3가지, 모든 노선 정보(RouteList), 모든 정류장 정보(StationList), 모든 정류장별 혼잡도 정보(CongestionList)를 각각 hashmap형태로 접근할 수 있게 하기 위하여 먼저 기존 데이터들을(xls파일) csv파일로 가공하여 만들었다. 노선 정보 / 정류소 정보 파일의 경우 아래 그림과 같은 구조를 가지고 있었다.



이 중 필요한 정보는 RouteList의 경우, 노선ID, 노선명, 경유 정류소, 첫/막차시간, 평일, 토요일, 일요일의 배차간격이다. StationList의 경우 정류소ID, 정류소ARSNum, 정류소명, X좌표, Y좌표, 노선별 혼잡도이다. 해시맵의 키값 및 구조를 표현하면 아래 그림과 같다. 결론적으로는 노선으로 검색했을 때 모든 정류장의 정보를(경유 정류소에 담겨있다), 정류장으로 검색했을 때는 해당 정류장의 정보를 알 수 있게 한 것이다.



버스데이터 파싱을 마치면 routelist를 담고 있는 routecsv.csv.파일과 staionlist를 담고 있는 stationcsv.csv파일이 만들어지고, 해당 파일들은 저장되어 필요할 때마다 이용되어진다.

* + 1. **혼잡도 데이터 분석**

혼잡도를 구하는 과정은 다음과 같다.

* + - 1. **승객의 인원수 데이터 파일을 읽는다. (ParsingCongestionClass.java)**

원래 2015년부터 2018년까지의 정보를 공공데이터 포털에서 얻을 수 있지만 버스노선의 경우 매년 매달 변경된 것이 많기 때문에 가장 최근인 2018년 정보를 이용해서 혼잡도를 계산했다. 싱글톤인 BusInfoClass에서 노선명을 key로 하고 CongestionClass 객체를 value로 하는 HashMap 변수를 만들어 모든 정보를 저장했다.

* + “BUS\_STATION\_BOARDING\_MONTH\_201801.csv” - “~201809.csv” 파일 파싱

201801부터 201809파일까지 perMonth파일을 파싱한다. 예를 들어 201801.csv파일은 1월에 1일부터 31일까지 특정노선, 특정정류장에 하루동안 탑승하고 하차한 총 인원이 저장되어 있다. 가장 친숙한 동작01번 버스가 10월 이후로는 정보가 누락되어 있어 01월부터 09월까지만 분석을 했다.

* “2018년\_버스노선별\_정류장별\_시간대별\_승하차\_인원\_정보.csv” 파일 파싱

2018년 한 해 특정노선, 특정정류장에 시간별로 탑승하고 하차한 총 인원이 저장되어 있다. 예를 들어 2018년 1년 동안 3-4시에 동작01번 중앙대 정문 정류장에 탑승한 총인원과 하차한 총인원이 저장되어 있다.

* + - 1. **읽은 데이터를 바탕으로 혼잡도를 계산한다. CalcCongestionClass.java**

특정요일 특정시간 특정노선, 특정정류장별로 승하차 인원이 저장되어있는 정보가 없기에 요일 정보를 갖고있는 “BUS\_STATION\_BOARDING\_MONTH\_201801-09.csv” 파일과 시간별 정보를 갖고있는 “2018년\_버스노선별\_정류장별\_시간대별\_승하차\_인원\_정보.csv”파일을 혼합하여 승하차 인원수를 우선적으로 구해야 한다.

* clac\_getOnOff()

평일, 토요일, 일요일 별 시간당 승차, 하차 인원 수를 다음과 같은 식으로 구한다.

*평일, X시의 승차 인원수 = X시의 승차인원 \* ( 평일의 총승차 인원 / 평일의 날짜수 ) / 평일+토요일+일요일 총승차인원*

* calc\_Passenger()

노선별로 재차인원을 구한다. 우리의 주된 관심사는 승차, 하차 인원이 아니라 결과적으로 ‘버스에 몇 명의 사람이 타고 있는가’ 였다. 즉 재차인원을 구해야 했다.

*재차인원(N 정류장) = 승차인원(N 정류장) – 하차인원(N 정류장) + 재차인원(N-1 정류장)*

이때 이전 정류장의 재차인원을 구하는 과정에서 반복이 일어난다. 재차인원을 구하기 위해 재차인원을 알아야 하는 오류가 생긴 것이다. 그래서 다음과 같은 식을 이용했다.

|  |
| --- |
| *Int Time = 구하고자 하는 시간대;*  *Int 최종재차인원=0;*  *While( N정류장의 첫차시간 혹은 아무도 탑승,하차 하지 않았을 때 까지){*  *최종재차인원 += 승차인원(N 정류장, Time) – 하차인원(N 정류장, Time);*  *N--;*  *Time -= 해당노선의 정류장별 시간간격;*  *}* |

예를 들어서 동작01번 버스가 ‘중앙대후문’에서 3시의 재차인원을 구하고 싶다면 3시의 승하차 인원을 우선 더한다. 그리고 동작01번 버스는 정류장 사이의 시간간격이 10분이라고 한다면 ‘중앙대후문’의 이전정류장인 ‘중앙대중문’에서 2시 50분의 승하차 인원을 누적하여 더해준다. 이와 같은 과정을 첫차시간에 도달하기까지 반복한다.

* calc\_congestion()

계산한 재차인원과 노선별 버스의 좌석, 손잡이 개수와 비교한다. 만약 재차인원이 좌석수보다 적다면 승객이 좌석에 앉을 수 있으므로 -1, 재차인원이 좌석에 앉을 수 없고 손잡이는 잡을 수 있다면 0, 재차인원이 손잡이도 못 잡을 정도로 많다면 1의 값을 부여한다.

* + - 1. **혼잡도 정보를 파일로 출력한다.**

다음과 같은 형식으로 congestioncsv.csv로 출력한다.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 노선명 | 정류장ID | 0-23시  평일  재차인원 | 0-23시  토요일  재차인원 | 0-23시  일요일  재차인원 | 0-23시  평일  혼잡도 값 | 0-23시  토요일  혼잡도 값 | 0-23시  일요일  혼잡도 값 |

* + - 1. **문제점**

데이터를 계산하기 이전에 잘못된 데이터가 많았다. 승하차 인원이 저장되어 있는 2가지 파일은 노선별로 정류장별로 없는 데이터가 있었다. 한 노선이 일관적으로 모든 날짜 모든 시간에 데이터가 없을 때도 있고 동작01번 버스처럼 1월부터 9월파일까지는 데이터가 있는데 10월부터는 데이터가 없을 때도 있었다. 또한 정류장사이의 시간간격에서도 문제가 있다. 원래는 api를 사용하여 정류장별 첫차시간을 이용해서 정류장 사이의 시간간격을 알아내려고 했으나 막상 api를 사용하려고 했을 때는 정보가 제공되지 않는 정류장들도 있어서 api사용은 불가능 했다. 그래서 시간간격은 버스 종류별로 대략적인 값을 설정할 수밖에 없었다. 좌석수와 손잡이 개수 또한 문제가 있었다. 기본적으로 공식적으로 정보가 제공되는 사이트가 없었다. 그래서 공항버스, 마을버스 등 버스 종류별로 1대씩 직접 타서 좌석수와 손잡이 수를 세어보려 했으나 어떤 마을버스는 같은 노선임에도 불구하고 버스가 소형, 중형, 대형 모두 다녔으며 심지어 성북05 버스는 차종이 스타렉스였다. 재차인원을 구하는 과정에서 노선별로 첫차시간이 필요했다. 총 6백여개에 달하는 모든 버스 노선을 팀원5명이서 나누어서 네이버에 일일이 검색하여 첫차시간, 막차시간, 배차간격을 검색했다. 하지만 이중에서도 예외는 있었다. 공항버스 중에 하나는 같은 노선임에도 불구하고 공항에서 서울로 오는 경로와 서울에서 공항으로 가는 경로에 대하여 각각 첫차시간과 막차시간이 존재했다. 어떤 버스는 주말에만 운영하는 버스도 있었고 어떤 버스는 평일 출근시간에만 잠깐 운행하는 버스도 있었다. 이로 인해 첫차 막차시간에도 부정확한 데이터가 들어갈 수 밖에 없었다. 이 모든 데이터의 문제점들이 해결될수만 있다면 이를 바탕으로 좀더 향상된 알고리즘을 짜고싶은 욕심이 든다.

* + 1. **버스 경로 추천 알고리즘**

1,2방법대로 데이터를 파싱 하여 우리는 경로 추천 서비스를 위해 필요한 데이터를 모두 얻을 수 있었다. 그렇다면 이제 어플리케이션의 동작 순서에 따라, 어떤 알고리즘을 사용하여 경로를 추천하는지 설명한다.

|  |
| --- |
| [어플리케이션 순서]  1. 출발지와 도착지의 X,Y좌표를 통해 걸어갈 수 있는 거리의 주변 정류소를 API를 통해 찾는다.  2. 출발 정류소와 도착 정류소 사이의 경로를 API를 통해 탐색한다.  3. 나온 모든 경로들 중 최단경로를 우선적으로 저장한다.  4. 실시간 API를 사용한 알고리즘을 통해 앉아갈 수 있는 경로를 저장한다.  5. 최단경로와 앉아갈 수 있는 경로를 모두 사용자에게 보여준다. (3번 + 4번) |

**1. 출발지와 도착지의 X,Y좌표를 통해 걸어갈 수 있는 거리의 주변 정류소를 API를 통해 찾는다.**

어플리케이션을 처음 실행하면, 사용자에게 출발지 및 도착지를 키워드 기반으로 입력 받는다. 그리고 이 입력 받은 정보를 ‘mapAPI’ package안의 searchByAddress 클래스와 SearchByKeyword 클래스를 통해 API를 사용하여 각각 WGS84기준의 좌표로 변환한다. 그 뒤에 사용자가 ‘길찾기’ 버튼을 누르면, 내부적으로 알고리즘이 진행된다.

**2. 출발 정류소와 도착 정류소 사이의 경로를 API를 통해 탐색한다.**

사용자가 위치한 좌표를 기준으로, 걸어갈 수 있는 거리의 주변 정류소를 찾는다. 700m 를 기준으로 찾으며, 제공되는 API를 사용하였다. 또한, 현재 위치에서 걸어가는 거리를 최소화하기 위해 가장 가까운 정류소를 출발지를 기준으로 최대 3개, 도착지를 기준으로 최대 2개를 찾아 저장한다. 그 이후, 그 정류소들 간의 경로를 모두 찾아낸다. 이 부분은 SearchRoute 클래스를 통해 api를 호출하여 구현했으며, 나온 경로들을 모두 ArrayList<ArrayList<String>> 의 형태로 저장한다. 예를 들어, 출발 정류소 주변에 2개의 정류소가 있고, 도착 정류소 주변에 1개의 정류소가 있을 때, 우리는 2\*1번의 API 호출을 통해 2\* N개보다 더 많은 경로를 모두 찾아낸다. 이 모든 과정들은 searchPath 클래스를 통해 구현한다.

**3. 나온 모든 경로들 중 최단경로를 우선적으로 저장한다.**

하지만 이 경로들은 모두 최단시간을 기반으로 추천된 경로이며, 우리가 원하는 혼잡도 기반의 경로들이 아니다. 그러므로 이 많은 경로 중 최단경로 하나만을 저장한다. 그리고 나머지 경로중 실시간 API 사용하여 찾아낸 앉아갈 수 있는 경로를 저장한다.

**4. 실시간 API를 사용한 알고리즘을 통해 앉아갈 수 있는 경로를 저장한다.**

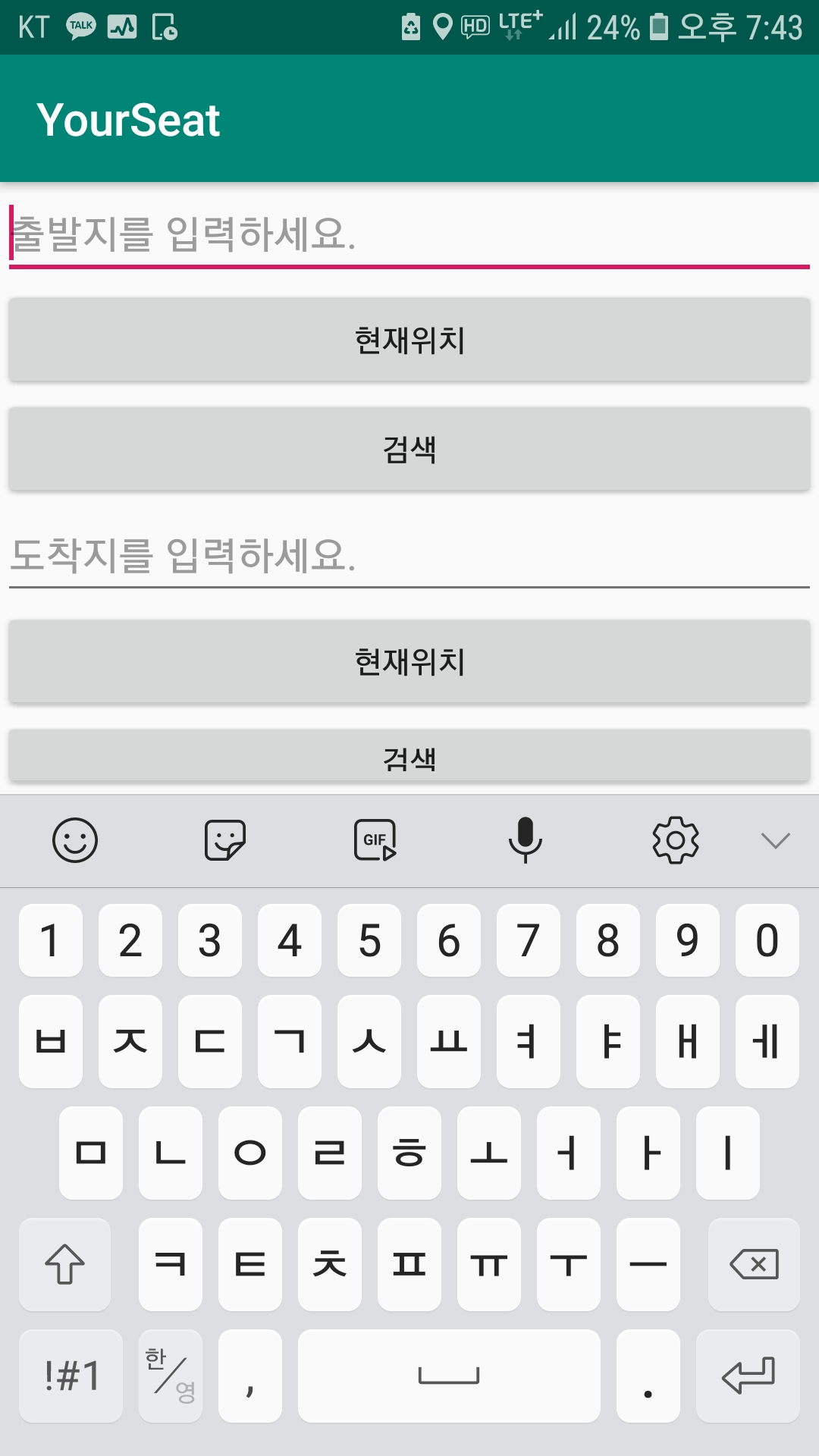
실시간 API를 통해 경로를 얻어내는 방법은 다음과 같다. 나에게 다가오는 버스가 혼잡한지를 판단할 수 있는 기준은 그 버스의 재차인원이다. 즉, 재차인원이 적으면 적을수록, 특히 그 버스의 의자 개수보다 적다면 나는 100%의 확률로 앉을 수 있다. 하지만 현재 내가 찾은 재차인원이, 실제로 버스를 타는 지점까지 도착했을 때 까지 유지된다는 보장이 없다. 즉, 지금 내가 타려는 버스가 나한테 오기까지 몇 명을 더 태우고 몇 명이 하차할지는 지금 알 수 없는 미래의 일이다. 따라서 이 어플리케이션에서는 CurrentLocationXY 클래스를 통해 실시간으로 나의 좌표를 계속 확인한다. RecommendPath클래스의 isNearestStop 메소드를 통해 시간이 흐르다가 내가 환승할 지점 전 정류소에 도착했을 때를 찾는다. 그 때 실시간 api를 사용하여 나에게 다가오는 버스들의 재차인원을 확인한다. 그리고 그 버스가 내가 있는 위치에 도착할 때까지의 재차인원 변화 추이를 앞에서 계산한 혼잡도 통계치를 사용하여 예측한다. 이 과정을 통해 사용자가 앉을 수 있는 가능성이 가장 큰 버스를 추천한다.

**5. 최단경로와 앉아갈 수 있는 경로를 모두 사용자에게 보여준다. (3번 + 4번)**

처음에 추천된 경로를 통해 사용자가 앉았고, 앞으로도 환승이 없는 경우에는 어플리케이션의 목적을 달성한 샘이므로, 종료한다. 하지만, 앉음 유무와 상관없이 환승이 있는 경우에는, 원래의 환승 지점보다 더 혼잡도가 낮은 환승 지점을 추천하여 사용자가 앉을 확률을 높여주어야 한다. 이 때, 우리는 RecommendPath클래스의 calcTotalCongestionInPath 메소드를 통해 기존의 환승 지점을 포함하여 사용자가 이동할 전 정류장 중 통계 혼잡도가 가장 낮은 환승 지점을 탐색한다. 통계 혼잡도는 앞서 설명한 혼잡도 데이터를 이용한다. 그렇게 혼잡도가 가장 낮은 지점을 탐색하면 앞서 사용한(4번) 실시간 API 알고리즘을 통해 경로를 찾고 그 경로를 실시간으로 사용자에게 추천하여 더 나은 환승을 유도한다.

* + 1. **안드로이드**

**출발지/도착지 검색**

****

[검색어 입력] [주소 검색] [키워드 검색]

**1. XY 좌표 추출**

카카오 지도 REST API를 사용하여 주소의 좌표 정보를 받아온다. 검색어를 URL 헤더에 담아 GET으로 요청하면 JSON 객체로 응답을 받는다.

안드로이드에서 통신을 수행하기위해서는 해당 과정을 쓰레드로 처리해야한다. 따라서 HTTP(GET) 방식으로 데이터를 받기 위해 AsyncTask를 사용하여 쓰레드로 동작시킨다.

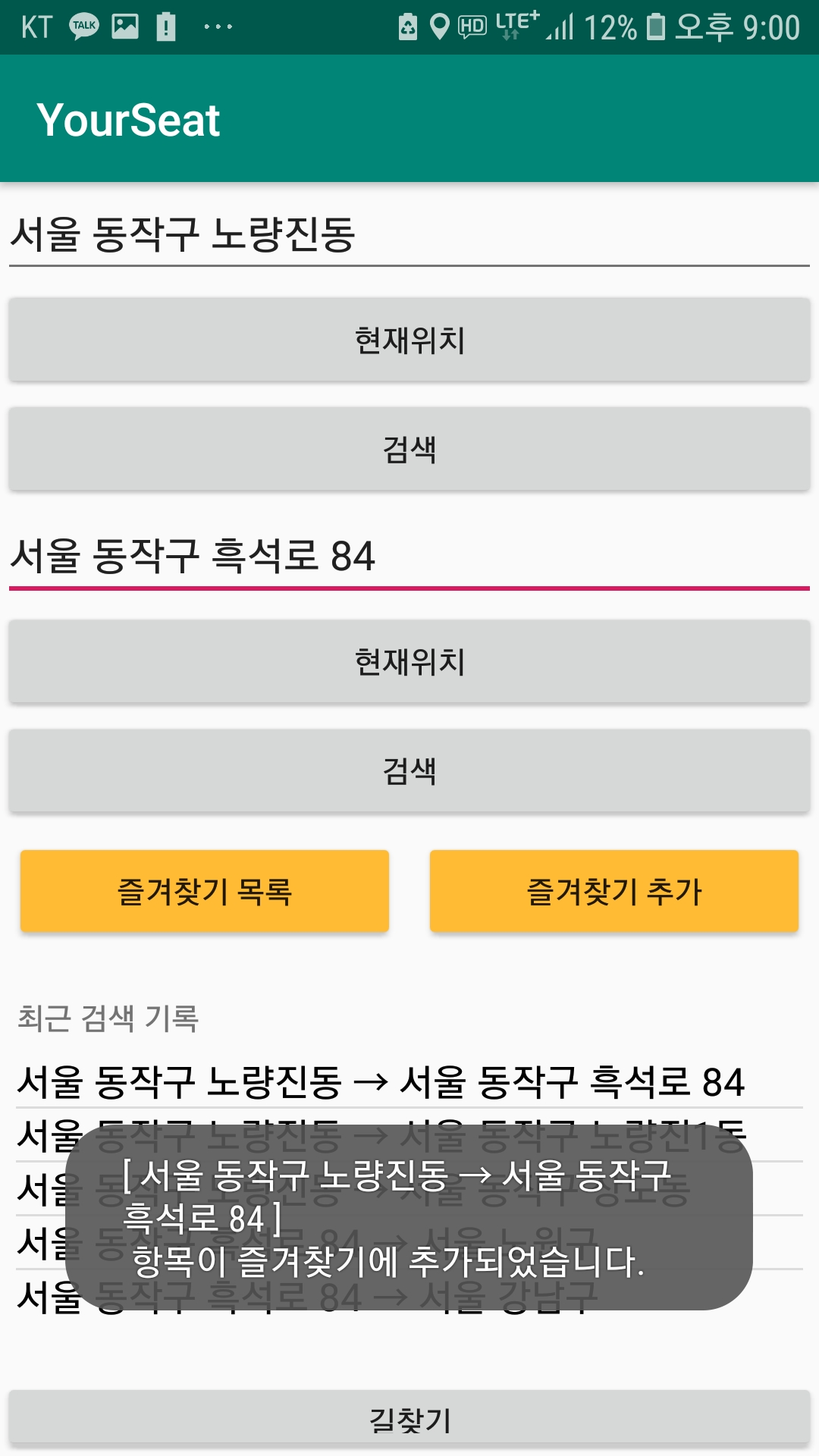
검색어가 주소인지 키워드에 따라 다른 URL을 호출하게 된다. 응답 JSON의 구조 또한 다르기 때문에 요청은 mapAPI 디렉토리의 SearchByAddress, SearchByKeyword 클래스에서, JSON 데이터 파싱은 SearchByAddressData, SearchByKeywordData 각각 처리한다.

주소에 따른 검색의 경우 주소 형식에 관계없이 지번 주소, 도로명 주소, 우편명 등을 모두 지원하며 키워드에 따른 검색의 경우 역명, 학교명, 가게명 등의 검색어를 지원한다. JSON 응답 바디는 x, y좌표 뿐만 아니라 주소에 대한 다양한 정보를 제공하기 때문에 확장성을 염두에 두어 좌표 이외에도 모든 데이터를 클래스에 저장하였다.

**2. 현재위치**

카카오 지도 앱 API를 활용하여 트래킹모드를 설정하면 현재 사용자의 위치가 업데이트됨에 따라 변화하는 좌표 값을 받아올 수 있다. 이렇게 받아오는 좌표 값은 여러 액티비티에서 호출되어 사용되므로 CurrentLocationXY 클래스를 생성하여 싱글톤 패턴으로 관리한다. MainActivity에서 현재위치 버튼을 클릭할 시 CurrentLocationXY에 저장되어있는 좌표데이터를 받아와 주소로 변환한 후 출발지/도착지를 설정한다. 이때, 앱 API에서 제공하는 GeoCoding 기능을 사용하면 x, y 좌표를 주소로 변환할 수 있다.

**즐겨찾기**

[즐겨찾기 추가] [즐겨찾기 목록]

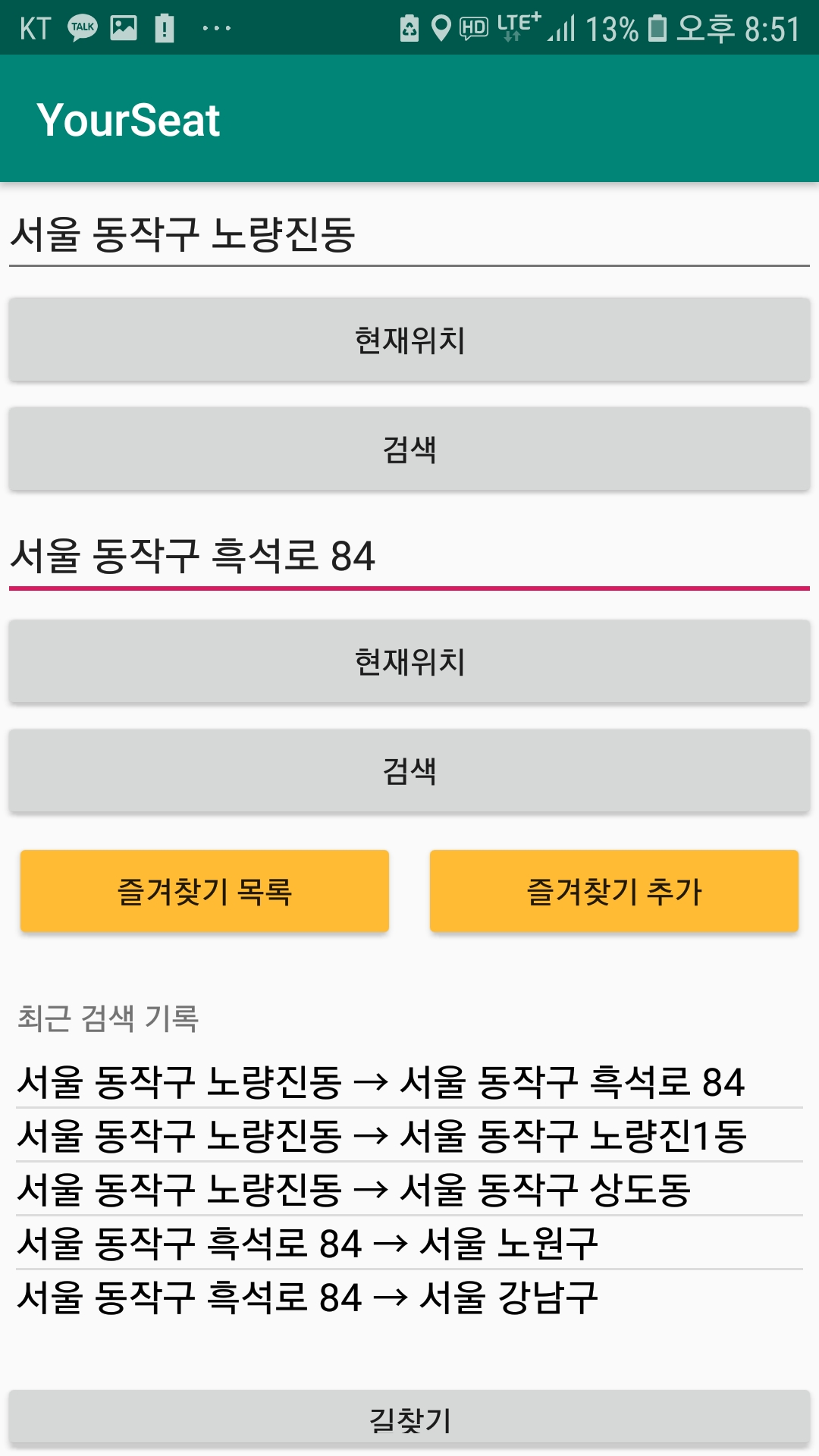
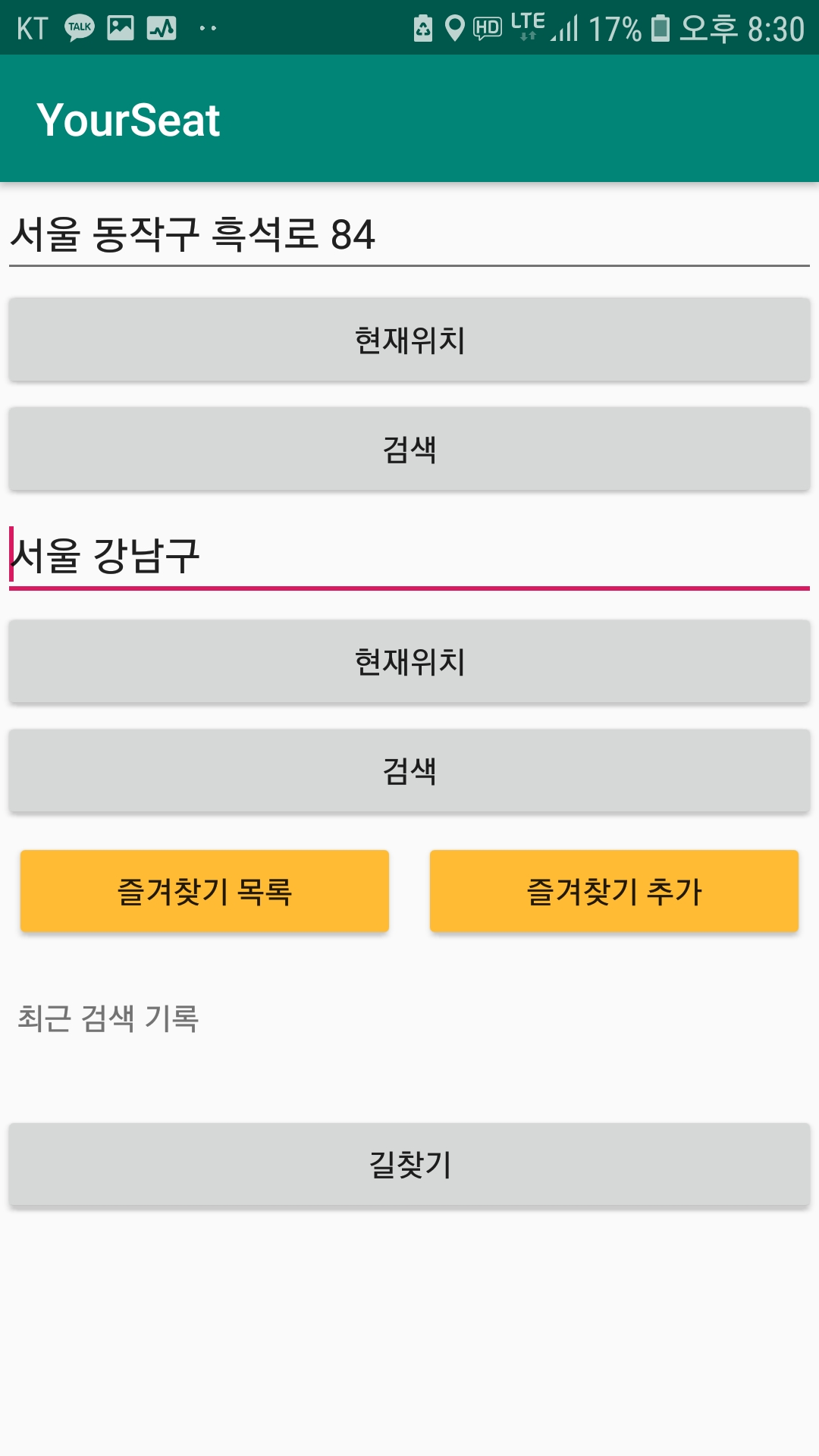
**1. 즐겨찾기 추가**

길찾기를 수행하고 검색했던 출발지/도착지 정보를 즐겨찾기에 추가할 수 있다. 즐겨찾기 추가 버튼을 누르면 SavedSharedPreference 클래스의 setStarListAddress 함수가 호출되며 파라미터로 출발/도착지 정보를 ArrayList<String>으로 전달한다. 데이터를 저장하기 위해 SharedPreferences를 사용하여 어플리케이션에 파일 형태로 데이터를 저장한다. 해당 데이터는 어플이 삭제되기 전까지 보존된다. 주소 정보 String 데이터를 JSON 객체로 변환하여 저장하였다.

**2. 즐겨찾기 목록**

즐겨찾기 목록 버튼을 누르면 SavedSharedreference 클래스의 getStarListAddress 함수가 호출되며 JSON으로 저장된 객체를 받아와 ArrayList<String> 형식으로 반환받는다. 해당 데이터는 GetStarListActivity 클래스로 전달되어 리스트뷰로 출력된다. 리스트의 항목을 길게 누르면 해당 경로가 즐겨찾기에서 해제되며, 항목을 선택하면 액티비티가 종료되면서 MainActivity의 출발지/도착지에 해당 경로를 가져온다.

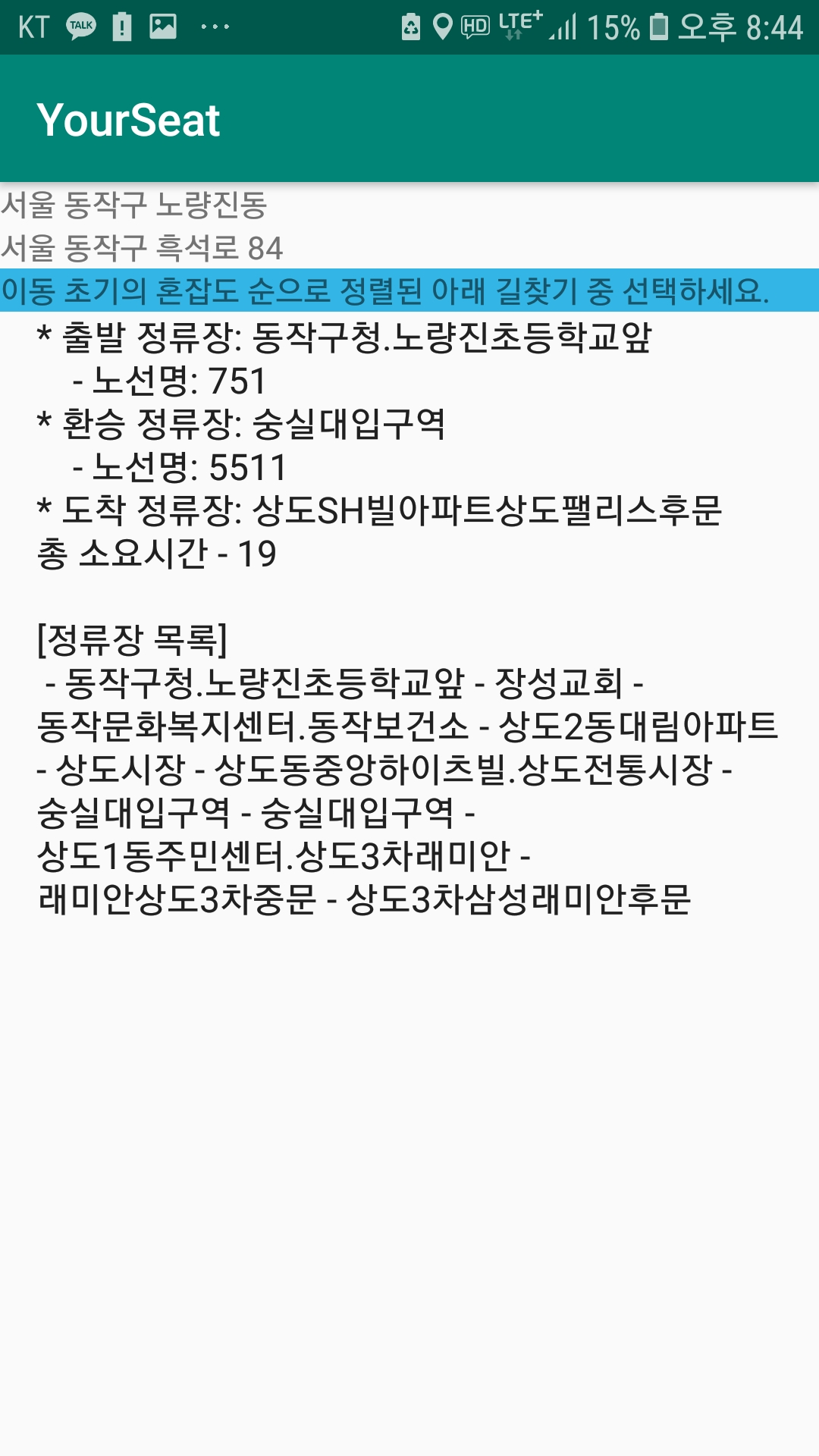
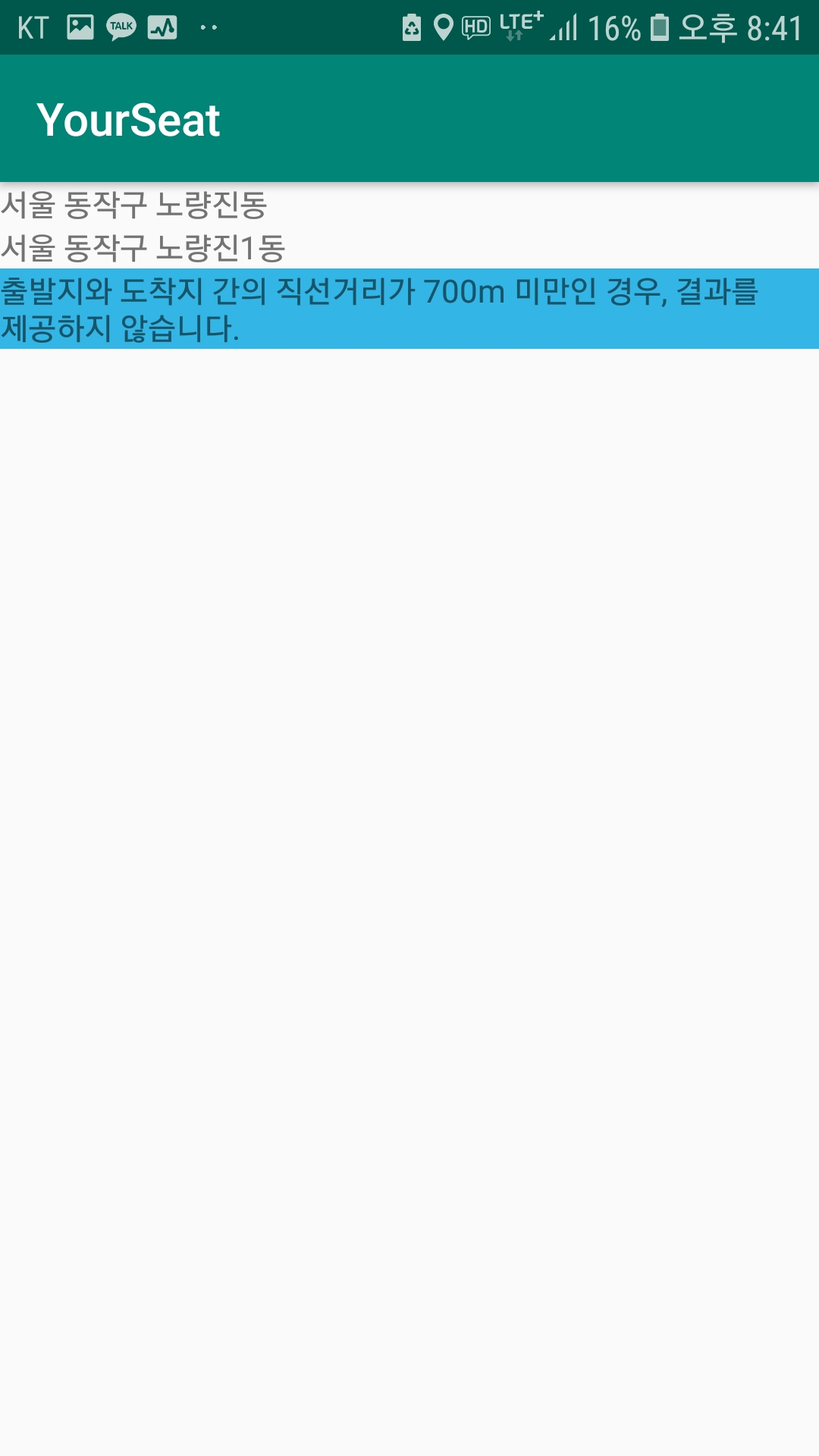
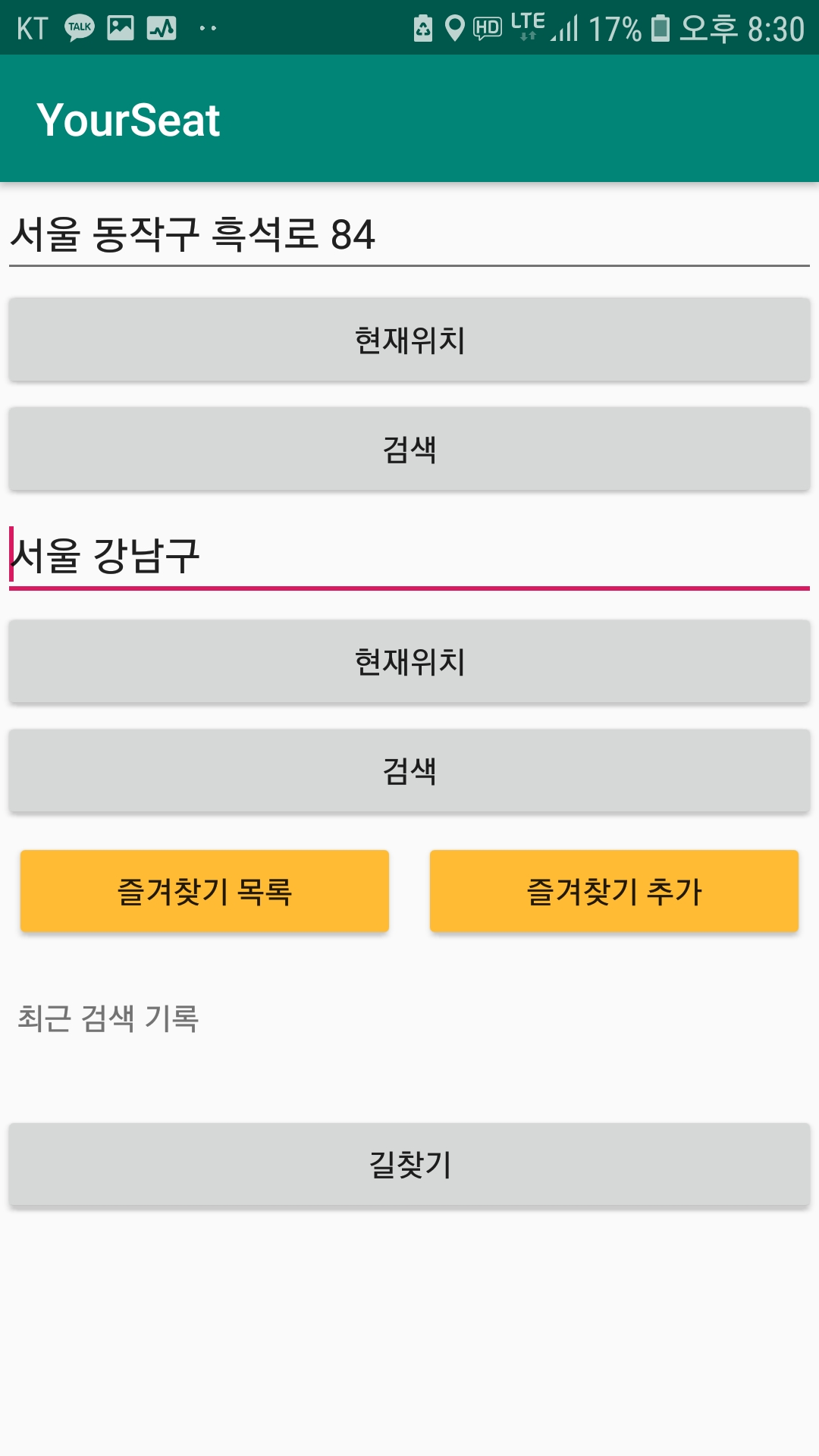
**최근검색기록**



**1. 데이터 저장**

길찾기 버튼이 클릭되면 SavedSharedPreference 클래스의 setAddressList 함수가 호출되어 검색 기록을 저장한다. 데이터는 SharedPreferences를 사용하여 어플리케이션에 파일 형태로 저장된다. 검색 기록은 최근 10개까지로 제한하여 추가로 더해지는 데이터는 이전 기록부터 삭제하여 10개의 데이터를 유지하였다.

**경로검색**



[길찾기] [거리 제한] [검색결과]

**1. 거리 제한**

출발지 – 도착지 간 거리가 700m 이하일 경우 길찾기를 수행하지 않는다. 이를 판단하기 위해 출발/도착지의 좌표 정보를 이용하여 거리를 계산한다. xy 좌표의 경우 WGS84좌표계를 따르는데 CoordinatesDistance 클래스의 calDistance함수에서 해당 좌표계를 미터 단위로 변환 시킨 후 거리를 계산하여 반환한다. 거리가 700m 이하일 경우 다음 액티비티로 isSearched 변수에 false값을 전달하여 “출발지와 도착지 간의 직선거리가 700m 이하”라는 안내문을 표시하도록 한다.

**2. 추천 경로 목록**

거리 제한을 통과할 경우 출발/도착지 좌표를 서버로 전송하여 경로 탐색 알고리즘을 수행하도록 한다. 서버로부터 전송된 추천 경로 결과는 GetSearchedRouteActivity 클래스에서 처리된다. 우선 출발/도착지 주소명을 MainActivity에서 전달받아서 상단에 출력한다. 추천 경로 리스트는 서버로부터 ArrayList<ArrayList<String>> 형식으로 반환 받는다. 경로 결과는 각각의 경로를 파싱하여 리스트뷰로 출력하며 다음과 같은 두가지 형태로 보여준다.

* + 1. 출발/환승/도착 정류장

출발/환승/도착 각각의 정류장 명과 노선명, 그리고 총 소요시간을 보여준다.

* + 1. 상세 정류장 목록

각각의 노선 별 거쳐가는 정류장의 목록을 ‘-’로 연결하여 보여준다.

* + 1. **서버**

기존의 방식은 정제한 데이터를 저장해둔 csv파일 3개를 사용자가 앱을 실행할 때 읽고 그 이후에 경로를 검색하게 했다. 하지만 이는 파일용량이 너무 커서 앱을 실행하는 데만 1분도 넘는 시간이 걸렸고 심각한 성능저하로 이어졌다. 그래서 이를 보완하기 위해 서버를 따로 두었다.

**서버구축** : 현재 구축된 서버는 i5-6600/16GB RAM/100Mbps광랜/Ubuntu 18.04로 구성되어 있다. 사양은 굉장히 낮지만 현재 서버로 사용 가능한 컴퓨터가 없어 위 컴퓨터로 진행하였다.

* + - 1. 데이터 통신

처음으로 서버 구축을 시작했을 때 모든 데이터를 웹으로 주고받으려 하였다. 마침 현재 프로그램도 Java를 사용하고 있기 때문에 JSP를 이용해 구축하여 통신을 진행했다. 하지만 여기서 서버를 담당한 팀원 전체가 JSP를 사용하는데 익숙하지 않고 시간이 부족한 문제가 발생하여 결국 단순 Socket을 이용한 통신을 수행하기로 결정하였다.

* + - 1. 코드 수정

**서버** : 안드로이드에 삽입했던 혼잡도기반 경로 추천 알고리즘 코드를 서버컴퓨터로 옮겼다. 서버가 처음 실행될 때 데이터 분석을 한 후 생성되는 3가지 파일을 routecsv.csv, stationcsv.csv congestioncsv.csv 읽고 컴퓨터의 메모리상에 정보를 갖고 있는다. 이후에 TCPServer 클래스를 실행한다. 이 클래스는 TCP를 바탕으로 Client인 안드로이드와 통신을 한다. 소켓을 열어 대기하고 있다가 client의 접속이 감지되면 스레드를 생성하여 수행을 한다. Client는 사용자의 출발, 도착지에 해당하는 x, y 좌표를 String으로 보낸다. 그러면 서버는 이를 잘라서 혼잡도 기반 버스 경로 알고리즘을 이용해 경로를 찾는다. 다시 안드로이드로 보낼 때는 경로에 관한 정보와, 경로에 포함된 모든 정류장의 이름을 String으로 변환하여 전송한다. String의 형태는 다음과 아래와 같다.

“총경로개수 {첫번째경로의단어수 (출발정류장ID 출발정류장명 노선명 도착정류장ID 도착정류장명)\* 소요시간} {첫번째경로의버스정류장개수 (버스정류장명)\* } {두번째경로 …}”

**안드로이드** : 안드로이드에서는 TCPClient라는 클래스를 통해 서버와 통신한다. 처음에는 메인 쓰레드에서 바로 소켓을 사용하여 구현하려고 하였으나, 안드로이드 버전이 상승하면서 이렇게 진행하면, 오류가 생긴다는 것을 알았고 asynctask를 통한 다른 방식으로 구현을 시도했다. 먼저, 사용자에게 입력 받은 좌표를 서버에게 전달하고, 그에 상응하는 결과값을 받는다. 결과값이 하나의 긴 string값인 것을 고려하여 TCPClient클래스 안에 convertMsg 라는 메소드를 구현하였다. 이 메소드는 서버로부터 전달받은 결과값을 사용이 용이하게 가공하는 역할을 맡는다. 변환한 결과값은 ResultOfServer라는 클래스 안에 저장한다. 이 클래스안에는 경로에 대한 결과값이 두가지로 나누어서 저장되는데, 각각 환승 지점, 소요시간, 출발 및 도착지를 저장한 arraylist와 경유 정류장 전부를 저장한 arraylist이다. 이 클래스를 view와 연결하여 사용자에게 보여준다.

* + - 1. 문제점
      2. 현재 서버에서 가장 큰 문제점으로 어느 정도까지 버틸 수 있는지 검증이 되지 않았다. 나중에 실제 테스트를 수행하면서 확인 및 수정이 가능하지만 데모에서는 크게 문제가 되지 않기 때문에 일단은 그대로 둔 상태이다.

지속적으로 Baidu쪽에서 데이터를 쏘는 작업을 하여 프로그램이 종료되는 경우가 있어 서버에서 프로그램을 24시간 실행시키지 못하는 상황이다. 시간상 보안 관련 설정은 전부 제외하고 환경설정을 수행한 결과 랜덤하게 뿌리는 패킷에 걸려 예상치 못한 서버 다운을 겪고 있다. 이 부분은 보안관련 설정을 하여 보완할 수 있지만 빈도가 높지 않고 데모때만 실행하기 때문에 일단은 놔둔 상태이다.

* + 1. **지하철 경로 추천 알고리즘**

버스와 마찬가지로 지하철 경로 또한 시간대별 승/하차 인원을 분석해 나온 혼잡도 수치와 총 소요시간을 계산하여 추천을 하도록 하였다.

* + - 1. **혼잡도**

지하철은 특성상 상/하선 인원 구분이 되지 않는다. 또, 혼잡한 시간대의 열차라도 어느 차량에 타는지에 따라 혼잡도가 달라지게된다. 예를 들어, 7호선 부평구청 방향의 경우 통학하면서 직접 타고 다녔을 때 3,4 번 차량은 사람이 너무 많아서 탈 수가 없었지만 1번차량의 경우 널널해서 탈 수 있었고 심지어 가끔은 앉아서 갈 수 있었다. 때문에 지하철은 혼잡도가 버스에 비해 부정확할 수 있다.

혼잡도 기준은 현재 차량의 혼잡도를 종합적으로 고려하여 전체 수치가 **33.33 미만일 경우: 여유, 차량에 탑승했을 때 앉을 수 있는 좌석이 존재**

**66.66 미만일 경우: 보통, 차량에 탑승했을 때 손잡이는 잡을 수 있음.**

**66.66 이상일 경우: 혼잡, 손잡이를 잡지 못하고 서있어야 함**.

위 수치들은 실제로 서울시에서 통합 데이터를 제공하면서 혼잡도 관련 사항에서 여유, 보통, 혼잡을 나누는 방법이다.

지하철을 얘기하고 혼잡도를 얘기했을 때 2호선이 현재 혼잡도 분석 및 안내 서비스를 하고 있다고 하는 피드백을 많이 받았었다. 하지만 실제로 이용해본 결과 2호선 혼잡도 서비스에 문제가 많음을 알 수 있었다.



그림 1 2호선 혼잡도 안내 서비스

그림 1을 보면 모든 차량에 여유라고 기록되어 있다. 실제로 해당 차량은 공식적인 기준에 따르면 혼잡으로 나타나야 할 정도로 사람이 많았고 이는 5,6,7,9번 차량도 마찬가지였다.

위와 같은 문제를 가진 채 혼잡도를 계산하기 위해 우선 사용할 수 있는 모든 지하철 승/하차 인원 통계자료를 가져와서 계산을 수행하였다.

첫번째로 현재 정거장에서 상/하선 별 승/하차 인원의 구분 비율은 현재 정거장에서 다음 정거장으로 넘어갔을 때 어느 방향의 인원이 더 많은지를 판단하였습니다. 부정확한 방법이고 정확도 또한 계산이 불가능하지만, 기본적으로

**1)기본적으로 많은 사람이 이용하는 노선이 존재**

**2)1의 노선에서도 특정 시간대별로 많은 사람이 이용하는 방향이 존재**

한다는 내용을 바탕으로 모든 시간대가 아니더라도 대부분의 시간대에서는 어느 정도의 정확도를 얻을 수 있었다.

두번째로 현재 정거장에서 환승이 가능할 때 구분이다. 이 자료가 환승 게이트에서 사람들이 카드를 찍었을 때 해당 내용도 통계가 이루어지는지 알 수 없기 때문에 어려웠다. 계산을 위해 현재 정거장에서 한 노선의 누적 승차 인원과 환승 노선의 누적 승차인원을 계산하여 각각 다음 정거장에서 누적 승차인원을 비교해 어느정도 수준의 인원이 보통 환승을 진행하는지 알아보았다. 예를 들어, 3호선 고속터미널 역에서 현재 데이터로는 7호선으로만 환승이 가능하다. 이 때, 3호선의 누적 승차인원이 100이고 7호선이 50이라 했을 때 각각 잠원/반포에서 누적 승차 인원이 110/80 이고 해당 정류장에서 각각 +5/+10 정도 승/하차 인원이 존재했다고 하면 3호선에서 7호선 방향으로 많은 사람들이 환승했음을 의미하고 이를 적절하게 나눠 혼잡도를 계산하였다.

여러 기준을 바탕으로 많은 데이터를 이용해 혼잡도를 계산하고 실제로 경로 탐색까지 수행하였지만 여전히 불확실한 값에 의존하는 점이 높아 실제로 적용을 위해서는 실제 카드기에 카드를 찍는 사람을 실시간으로 구하면서 CCTV를 이용해 실제 탑승한 인원의 수를 비교하면서 하는 방법이 아닌 이상 정확한 인원이 구해질 수 없다는 점을 느끼게 되었다.

* + - 1. **경로 탐색**

우선 모든 노선과 정류장 정보는 싱글톤 객체인 SubwayInfo라는 Class에 모두 저장되도록 구현되었다. 각 노선 정보는 LineInfo라는 Class를 이용해 저장하고, 각 정류장 정보는 StationInfo라는 클래스를 이용하여 저장하였다.



그림 2 지하철 관련 데이터 저장

모든 정거장 연결 관련 정보는 Line Class의 Station List를 순서대로 저장하여 한번에 파악할 수 있도록 짜였다.

위 정보를 이용하여 경로를 탐색하는 알고리즘은 FindSubwayPath Class의 search\_all이라는 함수를 이용해 Recursive로 작성되었다.

기본적으로 현재 프로그램은 가능한 모든 경로를 탐색한 다음 혼잡도를 기반으로 최단경로가 아닌 다른 경로를 추천해 주어야 한다. 여기서 가장 큰 문제가 생기게 된다.

1. 모든 경로의 혼잡도는 종점/기점까지 진행하고 올 경우 낮아진다.
2. 2호선의 경우 혼잡도가 떨어지는 시간까지 지하철에서 내리지 않고 계속 타고 있을 경우 혼잡도가 낮아진다. 즉 순환하는 문제가 발생한다.
3. 환승 계속 진행하면서 혼잡도가 떨어지는 노선을 선택하면 전체 노선의 혼잡도가 떨어진다.
4. 환승을 계속 진행 가능하게 할 경우 프로그램이 끝나지 않는다.

등 위와 같은 여러 문제들이 생기는데 그 중 2,4의 경우 프로그램이 끝나질 않거나 call Stack이 포화상태가 되어 프로그램이 강제로 종료되는 경우가 생겨 매우 치명적이었습니다. 이를 해결하기 위해 몇 가지 기준을 세워 경로 탐색을 수행하였다.

1. 어디서 출발하더라도 종점에서 하행, 기점에서 상행을 탐색할 경우탐색을 멈춘다.
2. 순환 노선의 경우 처음으로 탑승한 지점을 저장하고 해당 지점을 다음 정류장으로 탐색하려는 경우 탐색을 멈춘다.
3. 환승은 최대 2번으로 제한한다.

위 기준 중 2는 쉽게 납득할 수 있다. 하지만 1,3의 경우 문제가 조금 있는데

1. 1의 경우 탑승 정류장이 종점/기점의 직전 정류장일 경우 종점/기점을 들려서 오는 것이 더 효율적이라고 할 수 있을 수 있다. 하지만 이 경우, 합리적인 판단 기준이 서지 않기 때문에 현재 경로 탐색 알고리즘에서는 생각하지 않기로 하였다.
2. 3의 경우 환승 제한이 너무 적다고 할 수 있다. 하지만 서울 지하철을 전체적으로 분석해보면 3개의 노선을 거치면 거의 모든 지역을 갈 수 있음을 알 수 있다. 또, 아무리 혼잡도를 기반으로 추천한다 하여도 30분이면 갈 수 있는 거리를 1시간 30분 이상 가야 하는 거리를 추천하는 것은 굉장히 비효율적이고 합리적이지 않기 때문에 이런 경우를 줄이기 위해 환승 횟수를 최대 2번으로 제한하였다.

이러한 기준을 바탕으로 search\_all에 경로 탐색 알고리즘을 작성하였고 실제 수행 시간은 Debug모드/서버 에서 1초 미만으로 확인되었다.

* + - 1. **경로 추천**

기본적으로 모든 경로 탐색을 마치면 FindSubwayPath에서는 탐색된 모든 경로의 소요시간, 혼잡도를 받을 수 있다. 이를 이용해서 적절한 기준을 세워 경로를 추천해야 한다. 이 부분은 아직 작성이 되지 않았다.

* 1. **Implementation 이후로 개선된 부분**

1) 서버 구축

위에서 설명 완료

2) 안드로이드

**현재 위치 & 지도 표시**

카카오 지도 api를 활용해 트래킹모드를 설정하여 현재 위치를 계속 업데이트 받을 수 있도록 하고, MainActivity에 지도와 현재 위치를 커서로 표시하여 직관적으로 위치를 확인할 수 있도록 하였다. 또한 출발지/도착지 입력창 하단에 현재 위치를 받아오는 버튼을 추가하여 해당 버튼을 클릭하면 현재 좌표를 GeoCoding 작업을 거쳐 주소로 변환시켜 출발지/도착지를 업데이트 하도록 하였다.

**지하철**

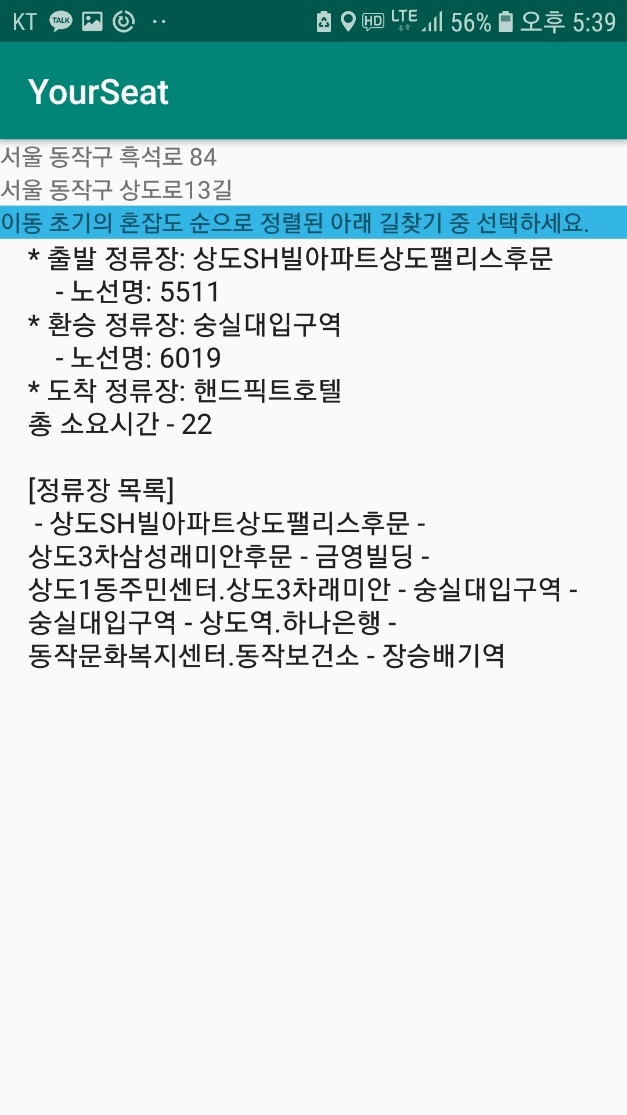
버스와 지하철에 대한 기능 분리를 위해 길찾기 버튼을 버스와 지하철 버튼으로 구분하였다. 두 경우에 대해 서버로 요청하는 요청값을 다르게 하여 처리하고 현재 지하철의 데이터를 받아오는 파트는 비워둔 채로 작업하였다.

1. **성능 평가 및 구현물 데모**

성능 평가를 가장 직관적으로 할 수 있는 방법은 출력되는 추천경로로 데모를 진행하는 것이었다. 하지만 서버구축 및 마무리 작업이 오래걸려서 테스트를 진행할 수 있는 시간적 여유가 많지 않았다. 따라서 적은 수의 테스트 케이스더라도 진행하기로 하였고, 방법은 다음과 같다. 같은 장소를 가는데, 한 팀은 카카오맵의 추천 경로로, 한 팀은 우리 팀이 진행한 프로젝트의 앱의 추천 경로로 이동하는 방식이다. 단, 평일 출퇴근시간과 같이 항상 붐빌 수 밖에 없는 경로와 시간은 제외하였다.

출발지는 중앙대학교(흑석로84)이고, 도착지는 신대방삼거리역(상도로길12)이다.

팀원 A와 팀원B는 각각 카카오맵 추천 경로, 프로젝트 앱의 추천 경로를 확인한다. 카카오맵의 추천 경로는 중앙대후문에서 동작 01을 타고 이동하다가, 상도노빌리티아파트에서 753번 버스를 타고 성대시장 정류장에서 하차하는 것이었다. 프로젝트 앱의 추천 경로는 상도시장까지 걸어서 이동한 이후, 5517번 버스를 타고, 동작문화복지센터에서 6019번 버스를 타서 핸드픽트호텔에 도착하는 경로였다. 아래 사진을 첨부하였다.

** **

**<왼쪽이 카카오맵 경로(첫번쨰 추천경로 선택함), 오른쪽이 YourSeat 경로>**

해당 경로로 이동하면서 5조의 출발 시각과 사진을 찍었고 그 이후에 정류장마다 재차인원을 세어 비교를 하기로 하였다. 조작 없이 진행할 수 있도록 탄 버스, 그리고 버스 내에서도 5조라고 쓴 손과 함께 사진을 찍어 진행했다. 아래 사진을 첨부하였다.

먼저 카카오맵을 통해 이동한 결과는 다음과 같다.

동작 01 에서 재차인원의 변화는 17 – 17 – 16 – 9 – 내린 정류장 의 순서였다. 아무래도 내리기 직전 정류장이 지하철역과 가깝다 보니 재차인원이 급히 줄었다. 모든 경우에 빈 좌석이 1-2개 있어 앉을 수 있었다. 다음으로 753으로 환승한 이후에는 사람들이 좀 더 많았다. 재차인원은 19 – 20 – 21 – 18 이었다. 역시 앉을 수는 있었고 예정시간 즈음에 도착하였다.

다음으로 YourSeat을 통해 이동한 결과는 다음과 같다.

먼저 상도시장까지 20분 정도 걸어가야 했다. 그 이후 5517을 탑승했다. 재차인원은 25 – 22 – 21 이었다. 여기서 첫 번째 문제점을 알 수 있었다. 시간이 20분 정도 흐르면서 저녁시간에 가까워졌고 따라서 첫 번째 배제 요인을 제대로 갖출 수 없었다는 점이다. 그 후 내려서 환승해야 했는데, 여기서 매우 큰 결점이 있었다. 다음 탈 버스가 6019로 공항버스라는 점이다. 공항버스는 배차간격이 40분으로 40분을 기다려야 했다. 이렇게 되면 실제로 카카오맵의 경로를 통해 30분 정도면 이동할 경로를 걷는 시간을 포함하여 90~100분을 이동해야 한다. 신체적 약자의 큰 수요를 기대하고 만든 어플인데 20분 걸어가고 40분 기다리라는 점에서 문제가 많다는 것을 깨달았다. 결국 데모를 접고 돌아올 수 밖에 없었다. 또한 추가적으로, 20분을 걸어간 이후 경로 탐색을 하면 다시 또 새로운 경로가 나오게 되었다. 저녁시간에 가까워지면서 재탐색을 한 결과가 다르게 나온 것인데 이 역시 고려되어야 할 점이었다.

성공적으로 데모를 진행하지는 못했지만 몇 가지의 개선점을 알 수 있었다.

* + - * 1. 걸어가야 하는 거리에 제한을 두어야 할 것 같다.
        2. 공항버스를 제외시키는 방법이나, 실시간 배차를 고려해 가까울 경우에만 추천 경로에 포함시켜야 한다.

위의 테스트 결과를 보았을 때 카카오맵에서 수행한 길찾기를 통해서 앉아갈 수 있었다. 따라서 기존의 YourSeat에서는 혼잡도가 가장 낮은 경로를 추천해주는데, 그럴 것이 아니라 어느정도 이하의 혼잡도를 보인다면 사용자를 멀리 보내서 혼잡도가 최하인 경로를 추천해 줄 것이 아니라 임계값 이하의 혼잡도를 보이는 경로 중 최단시간경로(일반 길찾기 맵에서 나오는 경로)를 추천해 주는 것이 사용자 입장에서 현실적으로 이용할 수 있는 최적의 경로라는 생각을 하였다.

1. **Github repository**
   * **최종 commit 수, contributor 수, issue / pull request 수**

Data parsing repository

126 commits, 5 contributors (team member), no issue and pull request

Application repository

175 commits, 8 contributors, 5 issues, 11 pull requests

* + **다른 학생들이 등록한 issue, pull request에 대한 평가 (부적절한/불필요한 contribution은 아닌지) + 이들을 어떻게 처리했는지**

|  |
| --- |
| Issu2 1번과 3번을 등록한 학우가 2번 Pull Request를 해주었다.  <Issue>위치 권한 미설정 시 앱 튕김 현상 #1([**https://github.com/b1g4/Application/issues/1**](https://github.com/b1g4/Application/issues/1) **)**  <Issue> 출발지 미 기입시 튕김 현상 #3 (<https://github.com/b1g4/Application/issues/3> )  <Pull Request> 위치 권한 부여, 출발지 검색 오류 처리 #2 (<https://github.com/b1g4/Application/pull/2> )  위치 권한 설정을 하지 않아 사용자가 위치 권한을 거부하면 앱 튕김 현상이 일어나고 출발지를 기입하지 않고 길 찾기 버튼을 누르면 앱이 종료되는 버그가 있었다. 이를 학우분께서 몇 줄의 코드를 추가해 해결해 주었고 이를 merge하여 Android를 개선했다. |
| <Issue> 최근 검색 목록 기능 추가 #4 (<https://github.com/b1g4/Application/issues/4> )  아래 pull request와 같이 설명 |
| <Issue> Add the keyword function #7 (<https://github.com/b1g4/Application/issues/7> )  <Pull Request> Find the Best path by using keyword. #8 (https://github.com/b1g4/Application/pull/8 ) <Pull Request> find thd best spot by using keyword #10 (<https://github.com/b1g4/Application/pull/10> )  <Pull Request> Revert "find thd best spot by using keyword" #11 (<https://github.com/b1g4/Application/pull/11> )  출발지/도착지 검색할 때 확장하고자 했던 기능이었는데 이에 대한 이슈 날려 주셔서 기존 이 파트를 개발하던 getLocation 브랜치에 pull request를 올릴 것을 요청하였다. Pull request를 병합하는 과정에서 merge전 close를 해버리는 실수를 해서 같은 내용에 대한 pull request를 여러 개 등록하게 되었다. |
| <Issue> 주소 검색 시 지역 범위에 대한 범위 설정 #9 (<https://github.com/b1g4/Application/issues/9> )  기존 Proposal에는 길 찾기 개발 범위를 서울 시내로 한정했다. 하지만 구현하는 과정에서 미처 개발을 하지 못했다. 이는 추후에 수정할 계획이다. |
| <Pull Request> 최근 검색 목록 기능 추가 #5 ( https://github.com/b1g4/Application/pull/5 )  안드로이드의 클라이언트 부분의 기능 중, 최근 검색 목록 기능을 추가하고자 하는 pull request를 확인하였다. 이는, 우리 어플리케이션의 특성상, 사용자가 검색했던 경로를 또 검색하는 경우가 적지 않게 있을 수 있고, 자주 가는 장소에 대해 시각화 해주는 이로운 기능이라고 판단하여 코드 리뷰 및 merge를 진행하였다. 코드를 직접 clone하여 확인했을 때 별다른 이상이 없었으나, 마스터 브랜치로 바로 merge하기 전에 ‘pull requests’ 라는 새로운 브랜치로 안전하게 merge하고자 pull request 수정을 요청하였고, 이후 merge를 완료했다. |
| <Pull Request> 즐겨찾기 기능 추가 #6 (<https://github.com/b1g4/Application/pull/6> )  위의 이슈였던 최근 검색 목록 기능에서 발전하여 즐겨찾기 기능을 추가하고자 하는 pull request였다. 기존의 다른 여러 지도 맵에서 지원하는 기능인만큼 유용한 기능이라 판단하였다. code review를 진행하였을 때 기존의 다른 부분과 비슷하거나 같이 엮이는 내용이 없어 충돌나는 부분이 없었기에 pullrequests 브랜치로 merge를 진행할 수 있도록 pullrequest 수정을 요청한 후 해당 브랜치로 merge하였다. |

* + **이외에 얼마나 Github repository를 잘 활용했는지 보여줄 것**

기본적으로 팀원 전부가 Git, Github에 익숙하지 않아서 실수도 잦고 전체적인 활용도가 숙련자에 비해 떨어질 수 밖에 없었다.

처음으로 깃허브 사용 수칙을 정했을 때, Branch를 따서 활용하는 것이 익숙하지 않고, merge도중 여러 문제가 발생하여 Github에는 master branch만 활용하고, Local Git에서 Branch를 따서 작업하도록 하였다.

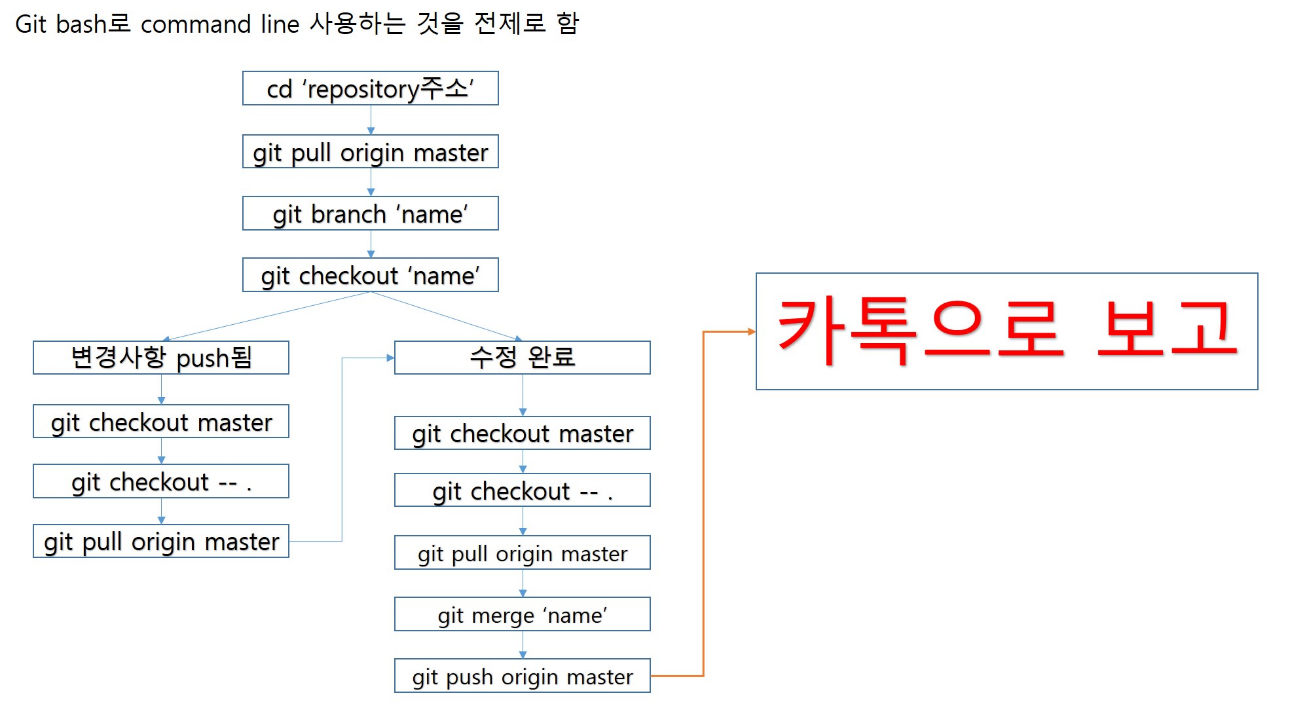


그림 3 초기 협업 방식

하지만 마스터 브랜치에 코드가 업로드 되면서 되던 코드가 되지 않을 때도 있고, 마스터 브랜치에 릴리즈 불가능한 버전이 자꾸 덮어써지면서 코드가 지저분해지는 문제가 생겼다. 때문에 방식을 바꿔 모두 Github 브랜치에서 작업한 뒤 모든 작업 내용이 안정적이고 검증이 가능할 때 마스터 브랜치에 머지를 하도록 진행하였다.

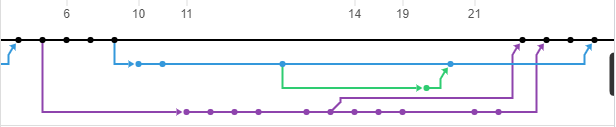


그림 4 바뀐 작업 내역 2

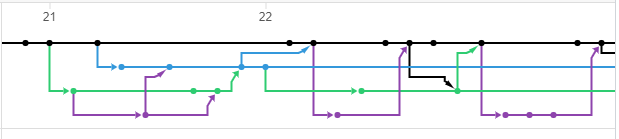


그림 5 바뀐 작업 내역 1

이렇게 프로젝트를 진행하면서 점점 프로젝트 진행이 안정적으로 바뀌었다.

DataParsing Repository를 활용하면서 겪은 시행착오를 바탕으로 Application 개발을 진행하면서는 시작부터 안정적인 개발이 가능하게 되었다.

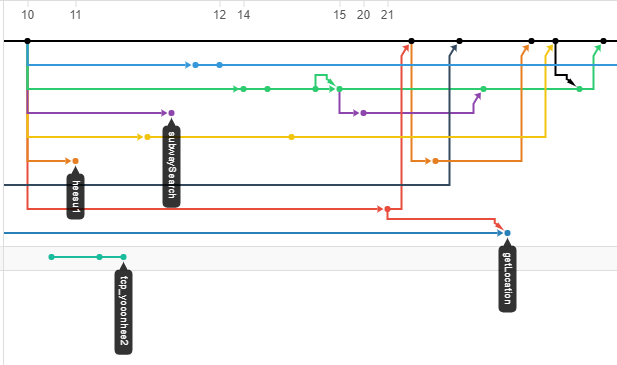


그림 6 Application Repo 작업 내역

1. **기타**
   * **업무분담**

기본적으로 모든 업무는 팀원이 함께 진행하며, 진행할 때 메인으로 맡는 사람을 정해 분담하였다.

|  |  |
| --- | --- |
| **역할** | **팀원** |
| 팀장 | 이설희 |
| 개발자 | 데이터 파싱 - 김윤희, 전희수  데이터 분석 - 이설희, 현도연  버스경로 구현- 김윤희, 허정우, 현도연  지하철 구현 - 허정우  안드로이드 구현 - 전희수, 이설희  서버구축 - 김윤희, 현도연, 허정우 |
| 문서작업 | Proposal report- 전희수(메인), 김윤희, 이설희, 허정우, 현도연  Final report - 김윤희, 이설희. 전희수, 허정우, 현도연 |
| 발표자 | Proposal - 전희수  Final - 김윤희 |