자료구조 CSE3080 2020년 11월 26일

프로젝트 중간보고서

공공데이터를 이용한 광역버스 시간표 자동생성 프로젝트

팀 원

20171266 박 건

20190544 배 상 원

120200199 김 종 현

목차

프로젝트 개요

기능	버스의 실제 운행기록을 바탕으로 운행 시간표를 생성한다. 버스의 실제 운행기록은 경기 버스정보에서 제공하는 버스위치정보 조회 API 서비스를 이용하여 수집한다.
입력	API 호출로 얻은 XML형식 파일을 구문분석(parsing)한 텍스트파일
출력	각 버스 노선에 대한 요일 별 버스 운행 시간표

함수 설명

이름	입력	출력	기능
read()	File *f	headp_pointer headp	API 데이터가 저장된 파일을 읽어서 linked list를 활용해 데이터들을 저장
make_table()	Headp_pointer headp RouteNo_node	Time_table node	read() 함수의 결과와 버스 노선 번호를 통합하여 최종 버스 시간표를 출력

구조체 설명

구조체	멤버	설명
_head_p	headp_pointer down_p	하나의 버스 노선의 데이터에 접근할 수 있다.
	headp_pointer next_p	다른 버스 노선의 head로 연결된다.
	struct tm qdate	해당 데이터 기록 날짜이다.
	int totalBus	각 노선의 총 배차 수이다.
	int routeId	버스 노선의 routeId이다.
_head_c	headc_pointer next_c	해당하는 버스의 데이터를 가지는 node로 연결된다.
	headc_pointer down_c	해당 노선의 다른 버스로 연결된다. (버스 호차 구분)
	int busNo	해당 버스의 호차 번호이다.
	char plateNo[]	해당 버스의 차량 번호이다.
	int stationExp	다음 정류장의 Station Sequence이다.
bus_node	struct tm qtime	time.h의 tm struct로 버스의 정류소 도착시간을 저장한다.
	int routeId	버스 노선의 routeId이다.
	int stationId	가장 최근 정류장의 Station Id이다.
	int stationSeq	가장 최근 정류장의 Station Sequence이다.
	int endBus	막차 구분 정보이다.
	int lowPlate	저상버스 여부 구분 정보이다.
	char plateNo[]	해당 버스의 차량 번호이다.
	int plateType	차량 종류 구분 정보이다.
	int remainSeat	남은 좌석 정보이다.

구조체 코드

```
typedef struct _head_p* headp_pointer;
typedef struct _head_c* headc_pointer;
typedef struct bus_node* node_pointer;
//head head
typedef struct _head_p {
    headp_pointer down_p;
    headp_pointer next_p;
    struct tm qdate;
    int totalBus;
    int routeId;
} _head_p;
//head
typedef struct _head_c {
    headc_pointer next_c;
    headc_pointer down_c;
    int busNo;
    char* plateNo;
    int stationExp;
};
```

[사진 1 - 자료구조 code_1]

```
//main struct
typedef struct bus_node {
    //data
    struct tm qtime;
    int routeId;
    int  stationId;
    int stationSeq;
    int endBus;
    int lowPlate;
    char* plateNo;
    int plateType;
    int remainSeat;
    //link
    node_pointer next;
}bus_node;
```

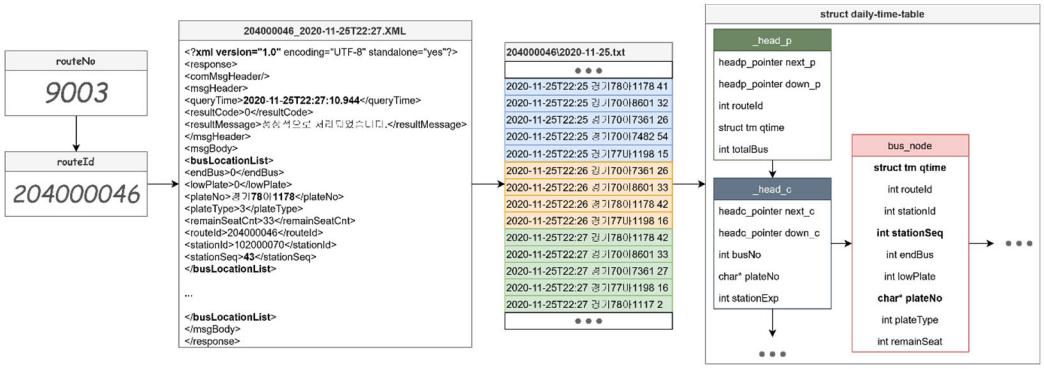
[사진 2 - 자료구조 code_2]

주요변수 설명

변수명	항목명(국문)	예	항목설명
resultCode	결과코드	0 또는 에러코드(자연수)	API 호출 결과
plateNo	차량번호	경기 71 사 1189	차량번호
remainSeatCnt	차량 빈자리 수	41	빈자리수
routeId	노선아이디	233000031	노선 ID
queryTime	호출시간	2020-10-29 15:04:07.801	API 호출시간
routeName	노선번호	9003	노선번호
regionName	지역명	수원, 화성	노선 운행지역
upFirstTime	평일 기점 첫차시간	4:50	평일 기점 첫차 출발시간
upLastTime	평일 기점 막차시간	22:10	평일 기점 막차 출발시간
stationId	정류소아이디	200000165	정류소 ID
stationSeq	정류소 순번	32	정류소순번
stationName	정류소명	이목동차고지.이목동입구	정류소 명칭

프로세스 설명

프로세스 단위 설명



공공데이터를 이용한 광역버스 시간표 자동생성 프로젝트의 목표는 노선 번호(**routeNo**)를 입력하였을 때 운행시간표를 생성해주는 것이다. 사용자가 알고 있는 노선 번호를 넣으면 **노선정보 항목조회API**를 호출하여 노선ID(**routeId**)를 반환한다. **함수 [getSwitch]**를 호출하여 현재 버스가 운행중인 시간인지 판단하여 API호출 여부를 결정하고, 운행중이면 API를 호출한다. 노선ID를 입력하여 **버스위치정보 조회API**를 호출하면 현재 운행중인 차량의 차량번호(plateNo)와 정류소순번(stationSeq)을 **XML 파일**로 반환한다. **함수 [parser]**를 호출하여 XML형식에서 태그를 제거하고 값만 남긴 **TXT 파일**을 저장한다. 여기까지 과정을 파이썬으로 진행하였다.

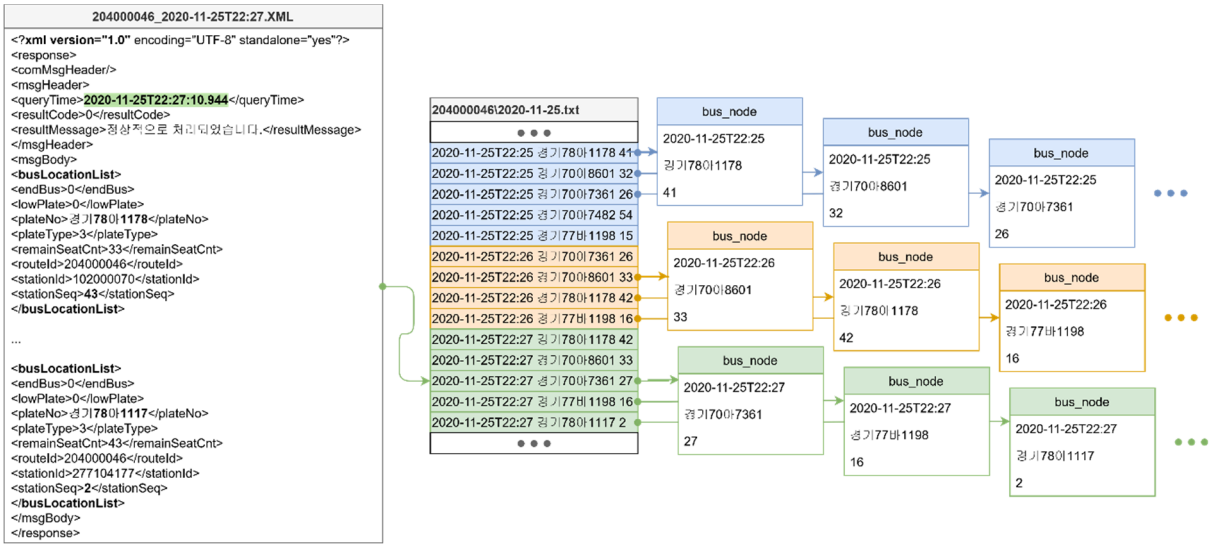
stn	66	67	68	69	70	71
Seq	70 아 7482	78 아 1178	70 아 8601	70 아 7361	77 바 1198	78 아 1117
1	NULL	NULL	NULL	NULL	NULL	NULL
2	20:40	21:00	NULL	21:40	22:00	NULL
3	20:42	NULL	21:22	21:41	22:01	22:27
4	20:43	21:03	21:23	21:42	22:02	22:28
15	20:57	21:22	21:40	21:59	22:23	22:42
16	20:58	21:24	21:42	22:00	22:25	22:44
17	21:00	21:26	21:45	22:02	22:27	22:46
18	21:02	21:28	21:47	22:04	22:28	22:48
	...					
26	21:22	21:47	22:08	22:23	22:47	23:06
27	21:28	21:52	22:12	22:27	22:51	23:09
28	NULL	NULL	NULL	22:31	22:54	23:11
	...					
32	21:39	22:00	22:23	22:37	23:00	23:17
33	21:42	22:04	22:26	22:39	23:03	23:20
34	21:46	22:05	22:28	22:41	NULL	NULL
	...					
40	22:03	22:23	22:38	22:54	23:15	23:34
41	22:05	22:25	22:40	22:56	23:16	23:36
42	22:07	22:26	22:42	22:58	23:18	23:37
43	22:09	22:29	22:45	23:00	23:20	23:40
53	NULL	NULL	NULL	23:14	23:31	23:52
54	22:25	NULL	23:00	NULL	NULL	NULL
55	NULL	NULL	NULL	NULL	NULL	NULL

C에서 TXT 파일을 읽어서 구조체에 저장한다. 차량번호 한 개의 정보를 구조체 **[bus_node]**에 저장하고 같은 차량번호를 같는 노드를 모아서 구조체 **[_head_c]**에 저장하고 배차 순서를 정수형 멤버 'busNo'에 기록한다. 첫차부터 막차까지 기록된 [_head_c]를 구조체 **[_head_p]**에 저장한다. 구조체 [_head_p]는 특정일에 특정 노선에 배차정보를 가지고 있고 이 정보가 특정노선의 일일 운행시간표 정보다. 배차수가 같은 날의 일일 운행시간표 정보를 평균하여 최종 운행시간표를 작성한다.

노선번호를 입력하여 텍스트파일 생성

노선번호를 입력하면 함수 [makeInfoFile]을 이용하여 **노선번호 목록조회 API**를 호출하여 노선ID(routeId)를 포함하는 XML형식의 데이터를 얻고 데이터를 텍스트 파일로 저장한다. 함수 [parser]를 호출하면 함수 [getSwtich]를 호출하여 운행여부를 확인한다. 첫차시간과 막차시간 그리고 API의 출력결과를 기준으로 버스 위치정보 조회 API를 호출하고 XML형식의 데이터를 얻는다.

다이어그램



코드설명

함수 [makeInfoFile]

함수 [makeInfoFile]은 노선정보 항목조회 API를 호출하는 함수다. 노선번호를 입력하여 노선정보를 텍스트 파일로 저장한다. API를 호출하기 위해선 routeId(노선ID)가 필요한데 routeId는 노선번호에 부여된 고유숫자다. API를 호출하면 routeId와 평일 기점 첫차시간, 평일 기점 막차시간, 지역명 등을 알 수 있다. 첫차시간과 막차시간은 뒤에서 나오는 함수 [getSwitch]에서 사용되고, 지역명은 같은 노선번호를 같는 데이터를 구분할 때 사용한다.

```
def makeInfoFile(routeId):
    directory = getDirectory('info', routeId)
    infoFilePath = directory + str(routeId) + '.txt'
    try:
        infoFile = open(infoFilePath, 'w', encoding = 'utf-8', newline = 'Wn')
    except:
        createFolder(directory)
        infoFile = open(infoFilePath, 'w', encoding = 'utf-8', newline = 'Wn')

    # 노선정보항목조회 busrouteservice/info :
    # 해당 노선에 대한 노선번호, 기점/종점 정류소, 첫차/막차시간, 배차간격, 운행업체 등의
    # 운행계획 정보를 제공합니다.
    url = 'http://openapi.gbis.go.kr/ws/rest/busrouteservice/info'
    queryParams = '?' + urlencode({ quote_plus('serviceKey') : apiKey,
                                    quote_plus('routeId') : routeId })
    # print(url+queryParams)

    request = Request(url + queryParams)
    request.get_method = lambda: 'GET'
    oneLineXML = urlopen(request).read().decode('utf8')
    # print(oneLineXML)

    xtree = ET.fromstring(oneLineXML)
    resultCode = int(xtree[1].find("resultCode").text)
    msgBody = xtree[2]

    busRouteInfoItem = msgBody[0]
    i=0
    infoFile.write("INFOFILE ")
    infoFile.write(str(routeId))
    infoFile.write('Wn')

    for info in busRouteInfoItem:
        infoFile.write(info.tag)
        infoFile.write(' ')
        infoFile.write(info.text)
        infoFile.write('Wn')
        # print(f"{i} {info.tag} : {info.text}")
        i+=1
    infoFile.close()
```

함수 [getSwitch]

함수 [getSwitch]는 노선 운행시간을 기준으로 API를 호출한다. 함수 [CallAPI]에서 특정 routeId에 대해 호출 여부를 결정한다. 날짜가 바뀌었을 때 값이 쓰여지는 파일을 새로 생성하고 API호출 수를 줄이는데 목적이 있다. 반환값은 함수 [parser]에서 함수 실행여부를 결정한다. 모든 routeId는 첫차 시간과 막차 시간이 정해져있다. 현재 시간에 따라서 스위치(함수를 호출하는...)값을 변화하여 API 호출여부를 결정한다. 조건은 다음과 같다.

1. [현재시간]이 [첫차시간]과 [막차시간] 사이에 존재하면 스위치는 켜져있어야 한다.

2. 스위치가 켜져있는 상태에서 [현재시간]이 [막차시간]을 넘어가면(현재시간>막차시간) 막차시간보다 늦게 차고지에 도착하는 버스가 있으므로 API호출 결과 알 수 있는 [resultCode]의 값에 따라 진행한다. 만약 API가 제대로 호출되었다면([resultCode] == 0) 변경 사항 없이 계속 진행한다. 만약 API가 제대로 호출 되지 않았다면 ([resultCode] != 0) 스위치를 끄고 [첫차시간]과 [막차시간]을 현재를 기준으로 업데이트(+1Day)한다 ([첫차시간]과 [막차시간]이 모두 현재시간보다 뒤에 있으므로 현재시간보다 항상 작음).

3. 스위치가 꺼져있는 상태에서 [현재시간]이 [첫차시간]을 넘어가면(현재시간>첫차시간) 스위치를 킨다. 다만 첫차가 조금 일찍 출발할 수 있으므로 첫차시간을 조정한다.

state	switch	첫차 운행	막차 종점	Result Code	상황	작업
0	0	FALSE	FALSE	4	API 호출 X, 현재시간이 첫차 운행 시작 전	NONE
1-1	0	**TRUE**	**FALSE**	4	API 호출 X, 첫차 운행 시작 시간이 지났지만 배차 X	switch = 1
1-2	0	**TRUE**	**FALSE**	0	API 호출 X, 첫차 운행 시작 시간이 지났고 배차 O	switch = 1
ERR	0	FALSE	FALSE	0	API 호출 X, 첫차 운행 시작 시간 전에 배차 O	발생하지 않음
2	1	**TRUE**	**FALSE**	**0**	API 호출 O, 첫차 운행 시작후 일반적인 상황	NONE
ERR	1	TRUE	FALSE	4	API 호출 O, 첫차 운행 시작 후 API 호출 오류	로그에 기록
2-1	1	**TRUE**	**TRUE**	**0**	API 호출 O, 막차 종점 도착 시간 후 종료 X	NONE
2-2	1	**TRUE**	**TRUE**	**4**	API 호출 O, 막차 종점 도착 시간 후 종료 O	switch = 0 시간 업데이트
0	0	FALSE	FALSE	4	API 호출 X, 현재시간이 첫차 운행 시작 전	NONE

```
def getSwitch(routeId):
    global scheduleDict, switchDict, dttFileDict, logFileDict
    nowTime = getNowTime()
    if(not switchDict[routeId]): #BOOL 1 - switchDict[routeId] : FALSE
        if(nowTime > scheduleDict[routeId]['start']): #BOOL 2 - 첫차 : TRUE
            switchDict[routeId] = True
            return False # FTF_ > FALSE
        else: #BOOL 2 - 첫차 : FALSE
            return False # FFF_ > FALSE
    if(switchDict[routeId]): #BOOL 1 - switchDict[routeId] : TRUE
                             #BOOL 2 - 첫차 : TRUE
        xtree = openAPICall(routeId) # 함수 [ openAPICall ] 호출... xtree를 생성하였습니다.
        try:
            resultCode = int(xtree.find('msgHeader').find('resultCode').text)
        except:
            return None
        if(nowTime > scheduleDict[routeId]['end']): #BOOL 3 - 막차 : TRUE
                if(resultCode): #BOOL 4 - result code : TRUE
                    print('State #3', end='Wt') # State : 3...
                    print(routeId)
                    switchDict[routeId] = False
                    scheduleDict[routeId]['start'] += datetime.timedelta(days=1)
                    scheduleDict[routeId]['end'] += datetime.timedelta(days=1)
                    scheduleDict[routeId]['today'] += datetime.timedelta(days=1)
                    dttFileDict = makeTextFile('dtt', routeId, dttFileDict)
                    logFileDict = makeTextFile('dtt-log', routeId, logFileDict)
                    return False # TTTT > FALSE
                #BOOL 4 - result code : FALSE
                return xtree # TTTF > xtree
        else: #BOOL 3 - 막차 : FALSE
            if(resultCode): #BOOL 4 - result code : TRUE
                return False # TTFT
            else: #BOOL 4 - result code : FALSE
                return xtree # TTFF > xtree
```

함수 [getSwtich]의 알고리즘 순서도가 부록에 있습니다.

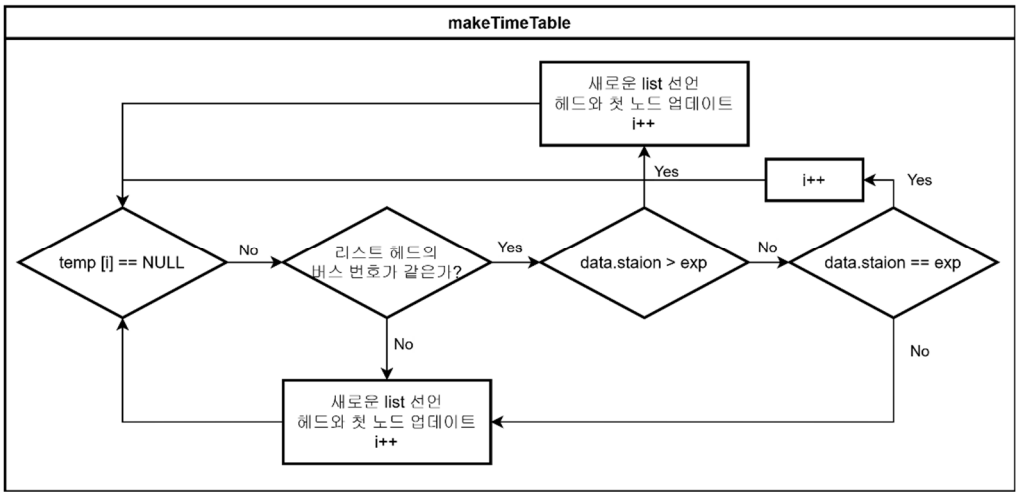
함수 [parser]

함수 [parser]는 python 내장 라이브러리 <xml.etree.elementTree>를 이용하여 XML형식의 텍스트 파일을 파싱한다. 함수 [getSwtich]의 결과 값이 XML데이터면 파싱을 하고 False면 함수를 종료한다. 결과는 routeId와 기록 날짜에 따라 루트폴더 아래 daily-time-table₩(routeId)₩(datetime).txt로 생성된다.

```
def parser(routeId):
    global scheduleDict, switchDict, dttFileDict, logFileDict
    dttFile = open(dttFileDict[routeId], "a", encoding = 'utf-8', newline = 'Wn')
    xtree = getSwitch(routeId)
    if(xtree):
        msgHeader = xtree[1]
        queryTime = msgHeader[0]
        msgBody = xtree[2]
        for busLocationList in msgBody:
            busLocationValueList = []
            dttFile.write(getNowTime().isoformat())
            dttFile.write(' ')
            for busLocation in busLocationList:
                dttFile.write(busLocation.text)
                dttFile.write(' ')
            dttFile.write("Wn")
    dttFile.close()
```

일일시간표 제작

알고리즘 흐름



1. Txt파일의 한 줄이 각 temp의 데이터에 해당한다.
2. 각 list들이 각 호차에 따른 시간당 정류소에 해당하며, 각 헤드들을 통해 접근할 수 있다.
3. 각 헤드들 역시 list로 연결되어 있고 head of heads로 접근할 수 있다.
4. API에서 받아온 정보에 특정 버스가 몇호차인지에 대한 정보가 없기에 이를 확인하는 작업이 필요하다.
5. 새로운 temp[i]가 들어오면 기존 list의 헤드와 plateNo를 비교하여 현재 운행중인 버스의 새로운 정보인지, 새로운 호차가 시작된 건지 확인한다.
6. 해당하는 값이 없으면 새로운 호차이므로 새 list를 선언한다.

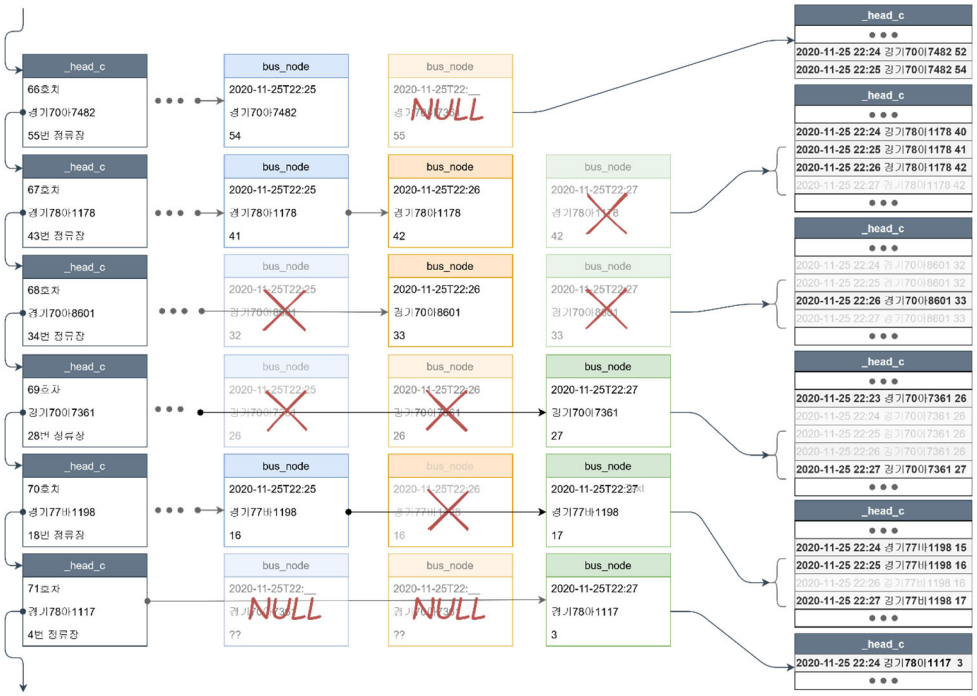
7 해당하는 값이 있으면 head의 exp 값과 temp.stationSeq 값을 비교한다. Exp값에 대한 설명은 후에 다루도록 한다.

8 temp,stationSeq이 exp보다 크면 운행중인 버스가 다음 정류장에 도착했다는 뜻이므로 list에 새로운bus_node를 추가한다.

9 temp,stationSeq이 exp와 같다면 필요없는 정보이므로 버린다.

10 temp,stationSeq이 exp보다 작다면 해당 plateNo를 갖는 버스가 전에 운행을 끝내고 새로운 운행을 시작했다는 의미이므로 새로운 list를 선언한다.

결과물 도식화



프로그램을 통해 특정 routeId(9003번 버스)의 하루 시간표를 구한 예시이다. Head of heads를 통해 원하는 노선에 접근할 수 있고, head를 통해 원하는 호차에 접근할 수 있다. 또 list들을 통해 각 시간별 정류소를 알 수 있다.

초기 모델과 차이 및 앞으로 개선할 점

초기 모델과 차이

초기 모델의 문제점

1. API를 통해 1분마다 새로운 정보를 호출하므로 아직 버스가 다음 정류장으로 도착하지 않았는데도 호출되어 필요 없는 정보가 생긴다.
2. 초기 모델은 모든 데이터를 우선 저장하고 stationSeq가 달라지는 시점의 time을 추출하는 과정이었으므로 필요 없는 정보까지 저장할 memory가 필요하다.

대안

1. 이를 보완하기 위해 exp를 고안했다. 예를 들어 버스가 5번 정류장을 지났다면 exp를 5로 업데이트한다.
2. 1분 후 temp.stationSeq이 exp보다 크다면 다음 정류소에 도착한 것이므로 해당 list에 추가한다. 이는 1분동안 2개 이상의 정류장을 지났을 경우 오류가 생기는 문제점 또한 보완할 수 있다.
3. temp.stationSeq이 exp와 같다면 아직 6번 정류장을 향해 이동중이란 뜻이므로 이 정보를 저장하지 않는다.
4. temp.stationSeq이 exp보다 작은 경우는 다음과 같다. 한 버스가 운행을 마치고 차고지에 들어간 경우 exp는 마지막 정류장의 번호가 된다. 하지만 시간이 흐른 뒤 이 버스는 새로운 운행을 시작하게 되고 temp.stationSeq가 1이 될 것이다. 따라서 exp보다 값이 작으면 새로운 호차가 시작된 것으로 인식하여 새로운 list를 선언한다.

이는 필요 없는 정보들을 따로 저장하지 않으므로 memory를 아낄 수 있다. List 구조를 사용하여 2차원 배열보다 원하는 정보에 접근하기 수월하다.

대안 2

1. 또 다른 대안으로 모든 정보를 routeId별로 묶어 시간순으로 정렬한다.
2. StationSeq가 차이가 생기는 지점의 인덱스를 추출한다.

의 방법이 제시되었다. 현재의 방법과 complexity는 같지만 초기 모델과 마찬가지로 memory의 효율성이 좋지 못해 현재의 방법을 선택하게 되었다.

코드설명

함수 [makeTimeTable]

함수 [makeTimeTable]는 운행기록(raw data)를 정제하고 일일 시간표 연결리스트 구조로 작성하여 텍스트 파일에 저장한다. 같은 차량번호를 같는 데이터를 따로 분류한 다음 데이터 기록 시간을 기준으로 정렬한다. 정류소번호(stationSeq)가 역전되는 인덱스를 기준으로 하나의 호차로 구분한다. 같은 정류소번호를 갖는 데이터 중에서 가장 작은 인덱스만 남긴다.

```
def makeTimeTable():
    # 각 차량 번호를 기준으로 반복문을 수행합니다.
    for plateNo in plateNoList:

        # 특정 차량번호와 같은 운행기록을 "DateTime을 기준으로 오름차순으로 정렬합니다."
        _sameBusList = _df[_df['plateNo'] == plateNo]
        sameBusList = _sameBusList.sort_values(['DateTime']).reset_index(drop=True)

        # 버스는 뒤로가지 않으므로 "stationSeq"가 작아지는 인덱스 i의 리스트를 만듭니다.
        # API 호출 오류로 stationSeq가 작아지는 경우가 있습니다.
        # 이 문제를 해결하기 위해 보정계수 diffK를 둡니다.
        indexList = []
        diffK = 10 if ( True ) else 0
        for i in range(1,len(sameBusList)):
            if(sameBusList.iloc[i-1]['stationSeq'] > sameBusList.iloc[i]['stationSeq'] + diffK):
                indexList.append(i)
        indexList.append(len(sameBusList))

        # 첫번째 인덱스, 인덱스의 리스트, 마지막 인덱스를 조합하여 호차별로 운행기록을 구분합니다.
        index = 0
        tupleList = []
        for i in indexList:
            tupleList.append((index, i))
            small_df = sameBusList.iloc[index:i].groupby(by=['stationSeq']).min()

            # 정류소 데이터프레임에 합칩니다.
            merge_df = pd.merge(station_df, small_df, how='outer',
                                left_index=True, right_index=True)
            _oneDay_df[str(plateNo)+'_'+str(index)] = merge_df['DateTime']
            index=i

        return indexList
```

함수 [makeTimeTable]의 알고리즘 순서도가 부록에 있습니다.

최종시간표

stn	1	2	3		72	73	stationName
Seq	78 아 1147	77 바 1097	70 아 8789		70 아 6926	70 아 7474	
1	05:00	05:20	05:42		22:50	23:10	운중동먹거리촌
2	05:01	05:21	05:42		22:50	23:10	운중동푸르지오하임
3	05:02	05:21	05:42		22:51	23:11	한빛교회
4	05:03	05:22	05:43		22:52	23:12	운중초등학교
5	05:04	05:23	05:44		22:53	23:13	운중중학교
6	05:05	05:24	05:45		22:54	23:14	운중동행정복지센터
7	05:07	05:25	05:46		22:57	23:16	뫼루니육교
8	05:08	05:27	05:48		22:58	23:17	판교원마을 1.2 단지
9	05:09	05:28	05:49		22:58	23:17	판교청소년수련관
10	05:09	05:29	05:50		22:59	23:18	한림아파트
52	06:22	06:54	07:13		00:18	00:33	한빛교회
53	06:23	06:55	07:14		00:19	00:33	운중동푸르지오하임
54	06:24	06:56	07:15		00:20	00:33	운중동먹거리촌
55	06:25	06:57	07:16		00:21	00:34	한국학중앙연구원

요약

진행 상황 요약

API 데이터 호출과 프로토타입 제작은 편의를 위해 파이썬으로 진행하였다.

1. API로 버스 정보를 호출하고 C언어에서 작업할 수 있도록 적절히 Parsing하는 데 성공하였다.
2. Raw 데이터 중 필요한 정보만 추출할 수 있는 알고리즘과 적절한 구조체를 설계하였다.
3. C코딩은 하나의 노선, 하루의 데이터만 처리할 수 있는 단계까지 완성하였다.

추후 작업 예정 요약

1. 하루의 데이터만 처리할 수 있는 C 코딩을 여러 날과 여러 노선을 한 번에 처리할 수 있도록 확장하는 과정이 필요하다.
2. C를 통해 모은 유의미한 데이터들을 평균 내어 최종 결과물인 시간표를 만드는 과정이 필요하다.

개선점

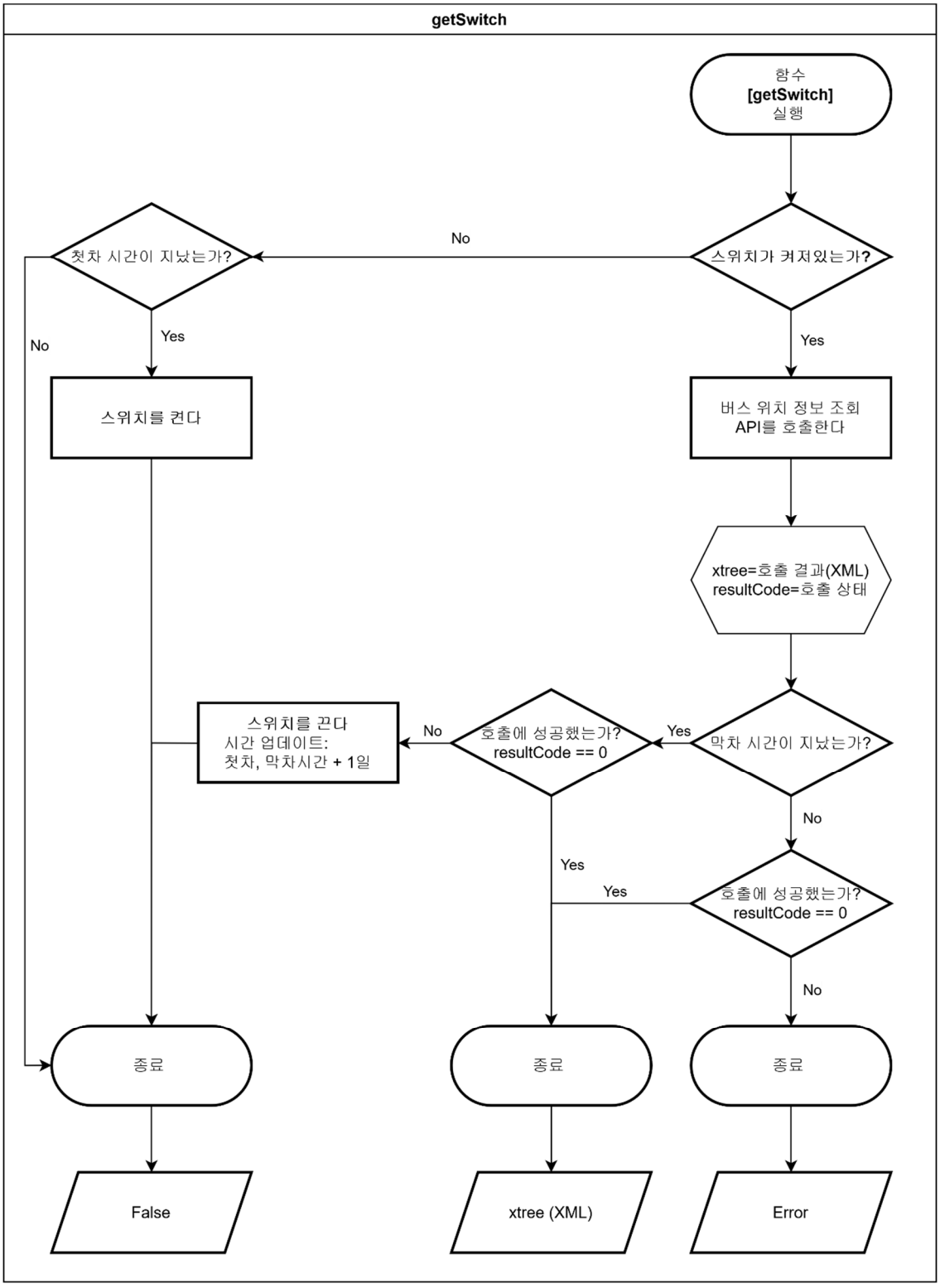
개선할 점

1. API상의 오류로 stationSeq가 갑자기 크게 늘었다 줄었다 하는 현상이 발견되었다.
2. 1분 동안 2개 이상의 정류소를 통과하는 경우로, 마지막 정류장을 list에 저장하지 못한 경우 다음 운행의 마지막 정류장이 앞의 list에 저장된다.

개선방안: 현재는 위치 정보만 비교하고 있는데 변수에 최대한 대응할 수 있도록 시간을 함께 비교하는 방안을 구상하고 있다.

부록

함수 [getSwitch] 순서도



함수 [makeTimeTable] 순서도