교차로 교통상황 분석을 통한 신호체계 최적화

졸업과제 최종 보고서

부산대학교 전기컴퓨터공학부 정보컴퓨터공학전공

팀명 : 훈의 아이들

201324428 김재현

201324445 박민하

201424496 오세현

지도교수 : 이도훈 (인)

목차

1. 서론

1.1. 개발 목표

교차로의 CCTV영상은 많은 정보를 내포한다. 이를 분석하여, 추출한 데이터를 활용해 교차로의 신호주기를 최적화 하는 것을 목표로 한다.

- 특정 교차로의 CCTV 영상을 분석하여 해당 교차로의 교통 효율을 향상시킬 수 있는 최적화된 신호체계를 설계한다.

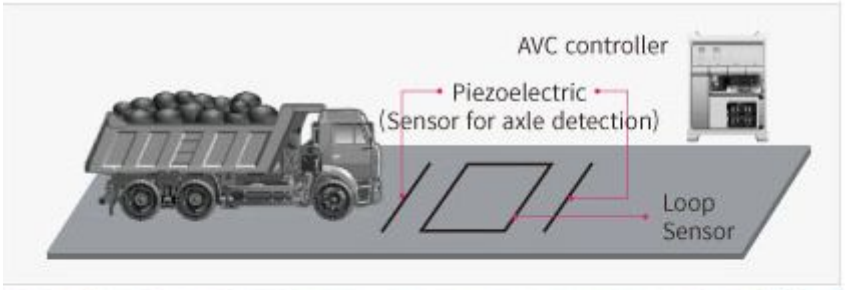
- 시뮬레이터를 제작하여 설계한 최적 교통신호를 적용하여 결과값의 향상을 확인한다.

1.2. 개발배경

1가구 1차량 시대에 도로 혼잡도가 점점 증가하고 있는 상황에서, 도로 교통을 효율적으로 관리할 수 있는 시스템 개발에 관심을 갖게 되었다. 현존하고 있는 오픈소스 기술 motion-path-extraction는 차량, 사람 등 객체를 감지하여 그 움직임을 추적하는데 초점을 맞추고 있다. 이를 활용하여 시간당 차량의 개수나 움직임을 파악, 신호체계의 최적화를 통해 교통 효율성의 향상을 기대할 수 있다고 보고 개발을 시작하였다.

1.3. 유사 시스템 분석

1.3.1) AVC 시스템




[2]

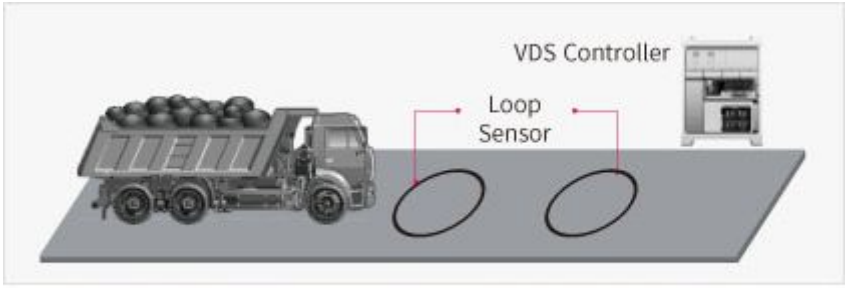
도로의 교통량, 점유율, 속도 등을 산출하고 차량의 차종 (국토해양부 지정 12종)을 분류 하는 교통량 조사장비

기능

- 차종 (12종) 분류
- 교통량 측정
- 차량 역주행 판별
- 차종별/시간별 통계
- 자체 진단 기능

1.3.2) VDS 시스템

도로의 교통정보(교통량, 속도, 점유율 등)를 수집하여 센터에 전송하는 교통량 조사장비




[2]

기능

- 교통량 측정
- 차량길이 측정
- 차량 역주행 판별
- 자체 진단 기능

1.3.3) VISSIM

VISSIM은 1992년도에 독일 PTV사에서 개발된 교통 시뮬레이션 소프트웨어이다. 도시교통과 대중교통 운영 등을 모형 화하기 위해 개발된 프로그램으로 차로조합, 교통운영, 차량제어 등과 같은 기능이 탑재되어 교통 류와 대중교통을 포함한 교통운영 분석이 가능하며, 교통공학과 계획의 효과 측정 등 다양한 대안 평가에 유용하다. 이용자 중심의 그래픽 인터페이스(GUI)를 제공하여 편의성이 높고 분석결과의 시각적 표현도 우수하다.[1]

[1]

구분	세부항목
기능	현실적 도로 기하구조 표현(3D 가능)
	대규모 도로 네트워크표현 가능
	GUI 기능 및 교차로 기하구조 및 신호, 운영방식 코딩 가능
	실제상황과 똑같은 운전자 행태 표현
	시간대별 OD 조정 가능 및 다양한 종류의 Output Data 제공
	실제 교통량을 반영할 수 있어 보다 더 현실적인 표현이 가능
	고정노선 및 고정 노선을 이용하는 차량 정의 가능
분석	교통 분석, 교통계획, 교통 관리
	혼합정보, 유고 상황에 대한 시뮬레이션 기능
	교차/합류 부분에서의 분석 가능

자료: 이주건, 버스 우선 신호기법 및 전략에 관한 연구. 서울시립대학교. 2005.

1.3.4) 차별점

우리가 만들고자 하는 시스템은 기존에 설치되어있는 교차로의 CCTV를 활용하여, CCTV를 통해 촬영된 영상을 통해 교통량을 분석하였다.

또한, VISSIM 등 실제로 활용되는 기존에 있는 교통상황 시뮬레이터를 활용하지 않고, 우리가 직접 시뮬레이터를 구현하고자 하였고, 이를 통해 교통신호를 최적화하고자 하였다.

2. 시스템 요구사항 및 제약사항

2.1. 시스템 요구사항

- 시스템은 도로 CCTV 영상을 통하여 교통량 데이터를 추출할 수 있어야한다.
- 시스템은 교통량 데이터를 통하여, 실제 도로를 가상으로 구현하여 시뮬레이팅하도록 해야한다.
- 시스템은 가상의 도로에서 최적의 신호주기를 찾아낼 수 있어야한다.
- 시뮬레이터 상에서 최적의 신호를 적용하고 시스템 사용 전 대비 교통 효율이 향상되는 것을 확인할 수 있어야 한다.

2.2. 시스템 제약사항

- 국내 영상이 아닌 해외 도로 CCTV 영상을 사용하는데 충분한 교통량 데이터를 뽑아낼 수 있을만큼 장시간의 영상을 확보하기 어려움.
- 결과적으로 최적의 교통 신호체계를 설계했다 하더라도 실제 현장에 적용시켜볼 수 없기 때문에 결과를 확인할 수 있는 방안 모색의 필요

3. 시스템 설계

3.1. 프로그램

3.1.1) Motion Paths Extraction

영상에서 차량 및 보행자 등의 객체의 모션 경로를 추출하기 위해 다중 객체를 감지하고 이를 트래킹하는 모듈로, 이 방법은 다음 알고리즘을 사용한다.

- Facebook의 Detectron Mask / Faster R-CNN
- SORT- 비디오 시퀀스에서 2D 다중 객체 추적을 위한 실시간 추적 알고리즘
- deep SORT- 메트릭을 사용한 실시간 추적 알고리즘

3.1.2) data_analyzer.ipynb

Motion Paths Extraction을 통해 추출한 트래킹 데이터를 분석하는 프로그램.

intersection_simul.py의 실질적 구현을 위해 데이터를 분석, 시각화 하는 단계.

❖ **library**

- pandas : 데이터 분석 라이브러리
- matplotlib : 데이터 시각화 라이브러리

❖ **class**

- Ojbect.py : 영상 속 객체들의 객체화를 위한 class
- Car.py : 객체들 중에서도 차량 객체를 위한 class

❖ **input**

- cutting_tracks.csv : Motion Paths Extraction의 트래킹 결과 데이터

❖ **output**

- object_data.csv : 영상 속 객체들의 정보
- cars_data.csv : data_analyzer의 차량 데이터 분석 결과 데이터

3.1.3) intersection_simul.py

data_analyzer.ipynb를 통해 추출한 cars_data.csv를 활용해

CCTV 영상 속 교차로의 최적 신호주기를 실험하기 위한 가상의 교차로를 구현하고 실험하는 프로그램.

교차로의 최적 신호주기를 도출하는 단계.

❖ **library**

- pyOpenGL :가상 교차로 실험
- pandas : 데이터 분석 라이브러리
- matplotlib : 실험 결과 시각화

❖ **class**

- CGGame.py : OpenGL 실험 화면 구성을 위한 Class
  - Camera.py : OpenGL 실험 화면의 시점을 위한 Class
  - Lighting.py : OpenGL 실험 화면의 광원을 위한 Class
  - Timer.py : OpenGL 실험 화면의 시각 확인을 위한 Class
  - Scene.py : OpenGL 실험 화면의 도형 렌더링을 위한 Class
  - Background.py : OpenGL 실험 화면의 배경 렌더링을 위한 Class
- Car.py : 차량 객체 Class
- Traffic.py : 신호 체계 Class

❖ **input**

- cars_data.csv : data_analyzer의 데이터 분석 결과 데이터

❖ **output**

- whole_dur_list : 실험 신호주기 리스트
- avg_latency_list : 실험 신호주기 별 차량 평균 대기시간 리스트
- traffic_time_list : 실험 신호주기 별 신호주기 회전율
- most_efficiet_duration : 실험 결과, 교차로의 최적 신호주기

3.2. 시나리오

1. Motion Path Extracion

input : CCTV교차로 영상.

process : 영상 속 객체들을 tracking

ouput : 객체들의 트래킹 정보, cutting_tracks.csv

2. data_analyzer.ipynb

input : 객체들의 트래킹 정보, cutting_tracks.csv

process : 객체들의 트래킹 정보를 analyze

output : 차량 객체 정보, cars_data.csv

3. intersection_simul.py

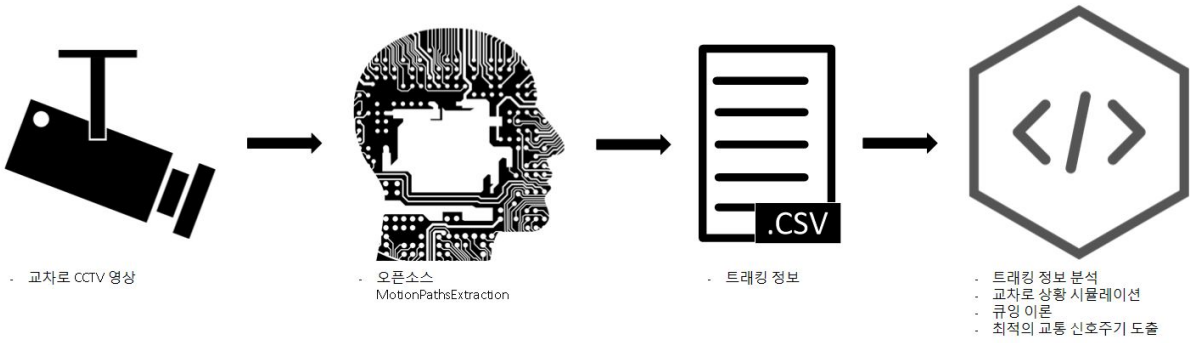
input : 차량 객체 정보, cars_data.csv

process : 가상의 교차로 구현 및 신호주기 실험

output : 해당 교차로의 최적 신호주기 실험값, **most_efficiet_duration**

4. 시스템 구현

4.1. 전체 시스템 구조



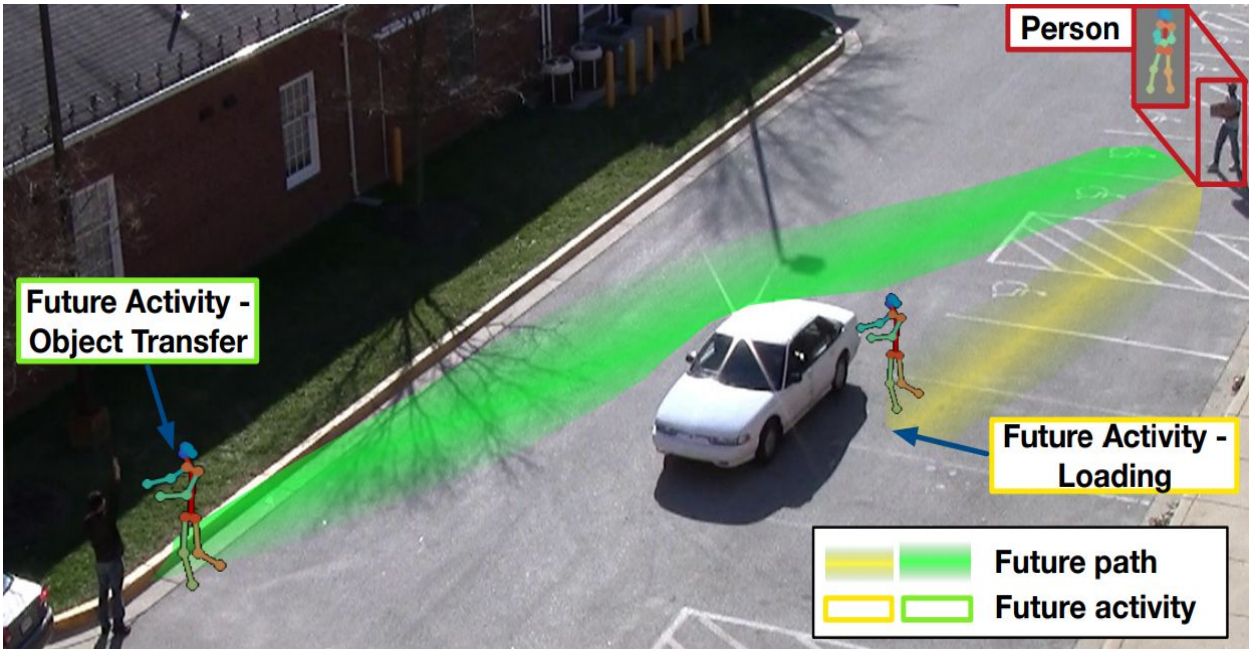
- 도로의 CCTV 영상에서 객체들의 움직임을 감지하고 추적하여 교통상황과 교통량을 판단할 수 있는 의미있는 데이터를 추출한다.

- 추출한 데이터를 기반으로 교통 효율을 향상시킬 수 있는 최적의 교통 신호를 구상 및 설계한다.

- 설계한 최적 교통신호를 가상 시뮬레이터를 통해 적용해 봄으로써 적용 전 대비 향상된 결과값을 확인한다.

4.2. 과제 수행 내용 및 결과

4.2.1) 초기 과제 목표



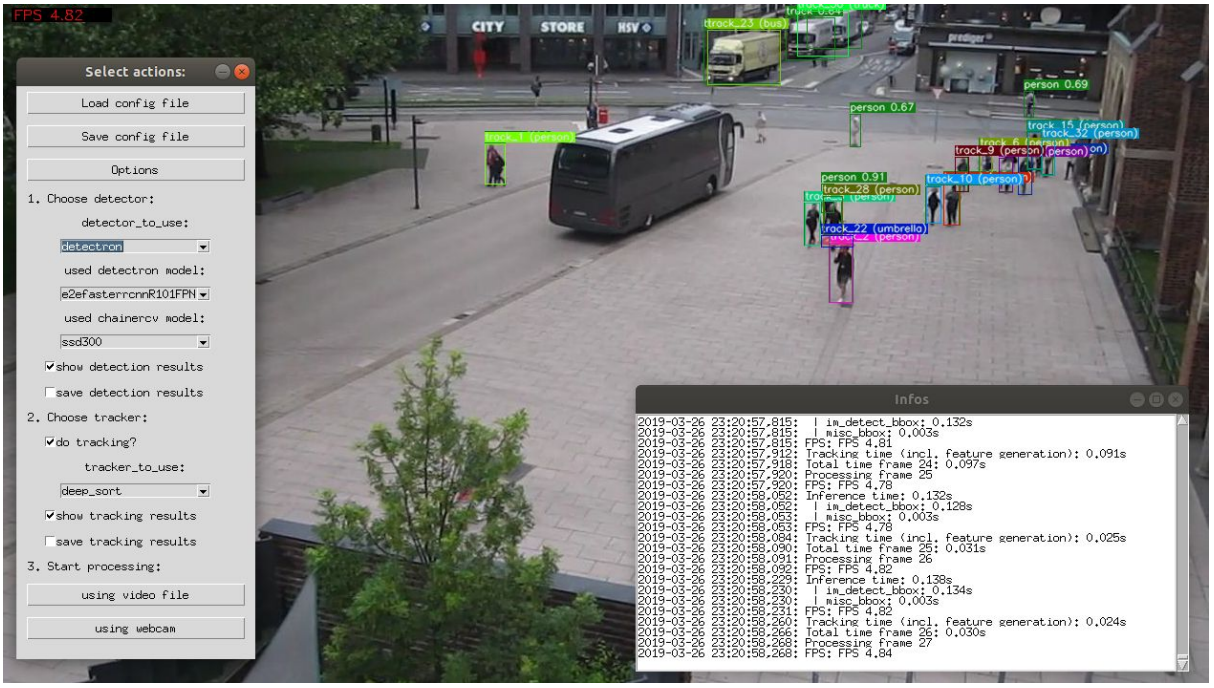
오른쪽 위 모서리에있는 사람이 의도에 따라 다른 경로를 택할 수 있다. 예를 들어 녹색 경로를 따라 물건을 옮기거나, 노란색 경로를 따라 물건을 차에 실을 수 있다.

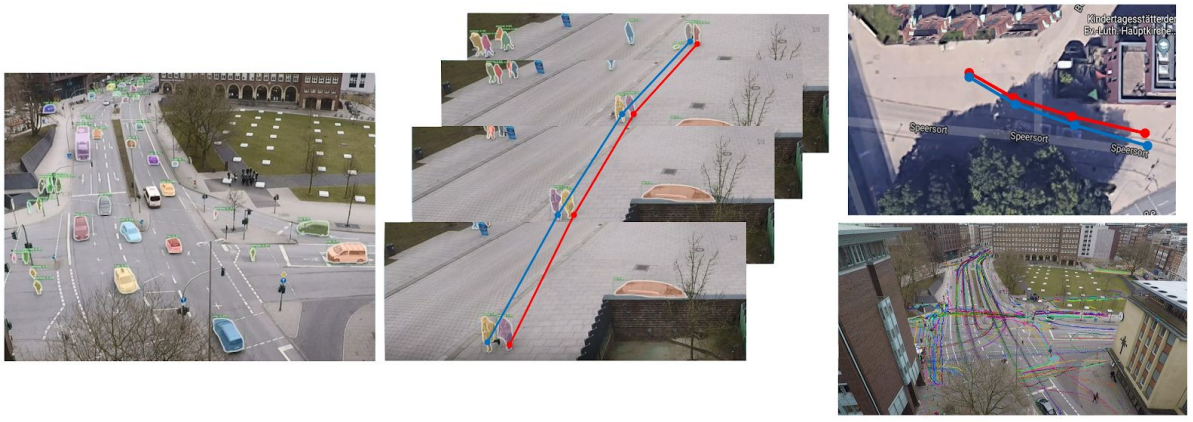
우리는 예시와 같이 '행동과 미래 경로를 공동으로 모델링하는 인간 행동예측 모델'인 Next 모델을 개선해 CCTV영상에서 차량-사람 움직임 예측 모델을 구현함을 목표로 진행을 했다.

하지만 우리가 개선하고자 했던 Next 모델의 한계점으로 인하여 과제 목표를 변경하게 되었다.

제약 사항	설명	대책
Next 모델의 한계	인간 행동과 경로를 예측하여 예측과 실제 video가 일치하는 일치율만을 보여주어 실제로 활용하기 어려움	비디오에서 차량-사람 객체의 움직임 경로를 추출하는 Motion Paths Extraction 모델을 활용한다.

4.2.2) 변경사항





우리는 '차량-사람 미래 행동 예측 모델' Next 모델의 개선을 목표로 하였지만, 진행 과정에서 이와 관련된 모듈을 활용하는데 어려움을 겪어, 영상에서 차량-사람 객체 트래킹 경로의 tracking line을 추출하여 이를 활용한 프로그램을 만들기로 목표를 변경하였다. 그 과정에서 제안된 주제는 다음과 같다.

주제 제안	내용
교통량 자동 분석 시스템	영상을 통해 거리의 일반차, 버스, 사람의 통행량을 자동으로 체크해서 도표로 보여주는 응용프로그램
교통 신호 최적화 시스템	차량, 사람 경로에서 산출한 시간별 개체 이동수를 통해 교통신호를 최적화 하는 응용프로그램
도로 종류	특정 도로에 대해 차량과 사람의 이동경향을 통해 그 도로가 어떤 종류인지

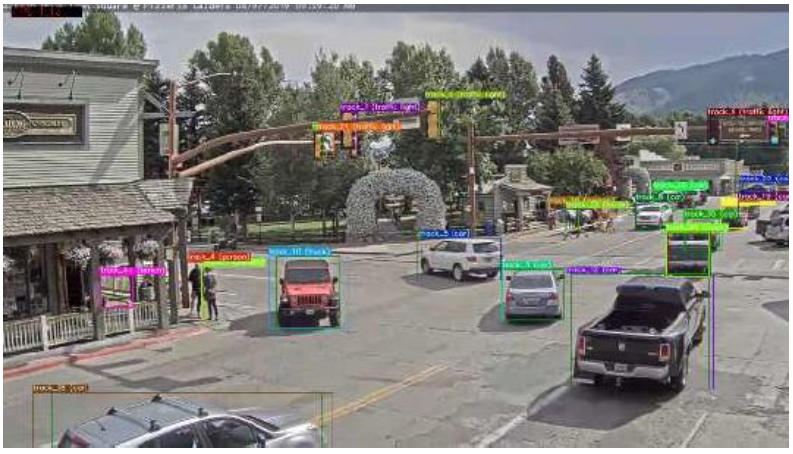
예측 시스템	유추하는 응용 프로그램 예) — 번화가 교차로: 차량들이 서로 가로질러 통행하고 매우 많은 사람들이 도로를 가로질러 보행한다 — 고속도로 인터체인지: 차량이 직진하다 우회하여 둥글게 돌아 나가는게 반복된다. — 왕복대로: 사람의 통행은 거의 없으며 중앙선을 기준으로 차량들이 반대방향으로 이동한다.

교차로 영상에서 추출한 트래킹 데이터를 통해 교통 상황을 분석하고,

교통 효율 향상을 위한 최적의 교통신호를 실험적으로 도출함을 주제로 선정.

4.2.3) 수행 과정

(1). Motion Path Extraction, Detection : 영상 속 객체를 감지하고 정보를 나타내 준다.



(2). Motion Path Extraction, Tracking :객체를 tracking해 이동경로를 나타내어 준다.

(3) data_analyzer.ipynb : Data Analyze

(3 - 1). raw data 준비

- data read : cutting_tracks.csv

```
frame,id,x1,y1,x2,y2,type,-1,-1,-1
3,1,305.4,395.41,22.16,78.4,1,-1,-1,-1
3,2,1151.93,179.22,12.47,32.54,10,-1,-1,-1
3,3,692.34,428.74,157.52,100.69,3,-1,-1,-1
3,4,281.56,387.64,25.64,83.19,1,-1,-1,-1
3,5,708.58,361.18,113.66,68.13,3,-1,-1,-1
3,6,638.51,146.58,20.36,57.84,10,-1,-1,-1
3,7,511.35,164.68,23.93,63.36,10,-1,-1,-1
3,8,981.58,295.6,50.51,33.81,3,-1,-1,-1
3,9,1061.44,171.42,14.4,41.54,10,-1,-1,-1
```

- data transformation : 필요없는 cols 제거, 객체 type 치환 결과

```
[48956 rows x 7 columns]
      frame   id      x1      y1      x2      y2    type
1         3    1   305.4  395.41   22.16    78.4  person
4         3    4  281.56  387.64   25.64   83.19  person
15        3   16  820.67  303.76   10.69   34.92  person
23        4    1  305.34  395.83   22.05   78.08  person
26        4    4  281.57  387.71    25.6   83.07  person
...     ...  ...     ...     ...     ...     ...     ...
65417  2872  775  426.45  480.72  311.87  160.56   truck
65443  2873  775  425.28  480.66  313.45  160.74   truck
65469  2874  775  424.66  480.64  314.62  160.79   truck
65495  2875  775  421.92  480.39  317.32  161.58   truck
65522  2876  775  420.75  480.31   318.8   161.8   truck
```

- objects_data.csv 작성 : 객체들의 데이터 저장.

```
# write objects_data.csv
objects_data.to_csv('./objects_data.csv', header = True, index = False)
```

```
frame,id,x1,y1,x2,y2,type
3,1,305.4,395.41,22.16,78.4,person
3,4,281.56,387.64,25.64,83.19,person
3,16,820.67,303.76,10.69,34.92,person
4,1,305.34,395.83,22.05,78.08,person
4,4,281.57,387.71,25.6,83.07,person
4,16,820.65,303.75,10.7,34.95,person
5,1,305.31,396,22,78.04,person
5,4,281.51,387.8,25.55,82.89,person
5,16,820.64,303.78,10.69,34.93,person
```

- 등장 객체 개수 도출 : 서로 다른 트래킹 id 개수에서 전체 등장 객체 개수 도출

```
objects_id_set = set(objects_data['id'])
print('obj id set : ')
print(objects_id_set)

objects_id_list = list(objects_id_set)
print('obj id list : ')
print(objects_id_list)

object_counts = len(objects_id_set)
print("object counts : ", object_counts)
```

```
object counts :  377
```

(3 - 2). Object class 객체 생성

- **Object 객체 구성**

```
class Object:
    def __init__(self):
        self.id = -1
        self.type = -1
        self.frame_list = [] # 등장 프레임 기록
        self.point_list =  [] # 각 프레임에서의 위치 기록
```

- **objects_data.csv의 정보에서 Object 객체 생성**
  - **id : tracking id**
  - **type : detection type**
  - **frame_list : 객체의 tracking id가 등장하는 frame**
  - **point_list : 객체의 ground truth 중심점**

```
# init objects
init_start_time = time.time()
# init
for i, row in objects_data.iterrows():
    current_id = objects_data.ix[i]['id']
    current_idx = objects_id_list.index(current_id) # 해당 id를 가지는 obj의 idx
    print('current id ', current_id, ' in current idx ', current_idx)
    current_type = objects_data.ix[i]['type']
    print('current type : ', current_type)
    current_frame = objects_data.ix[i]['frame']
    print('current frame : ', current_frame)
    current_point = center_of_rect(float(objects_data.ix[i]['x1']), float(objects_dat
                                   float(objects_data.ix[i]['x2']), float(objects_dat
    print('current point : ', current_point)

    objects[current_idx].set_object(current_id, current_type, current_frame, current_

    print('init : {}/{} done'.format(i, len(objects_data)))
init_finish_time = time.time()
```

- **전체 init time :** `init_time :  89.76281881332397` **(대략 90초 소요)**

(3 - 3). Object data 분석

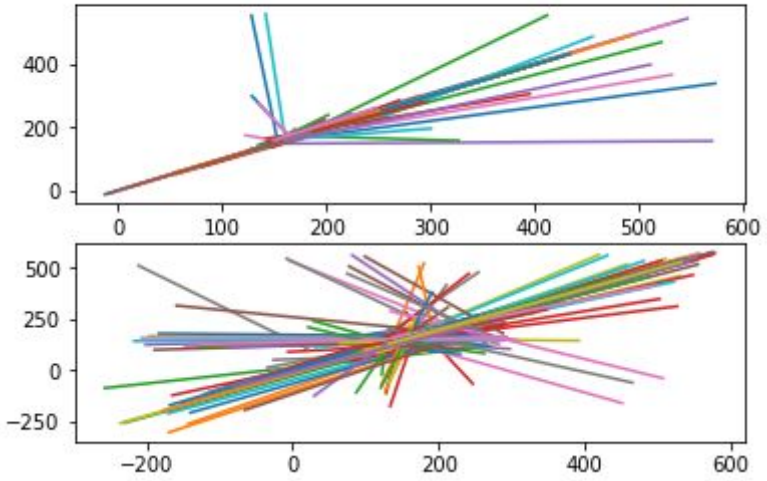
- **사람 객체와 차량 객체로 분류**

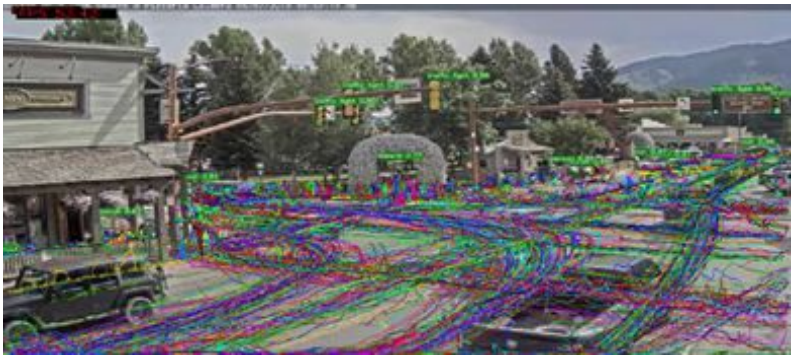
```
# obj 종류 별 데이터 분석
persons = []
vehicles = []
for obj in objects:
    if obj.get_type() == 'person':
        persons.append(obj)
    else:
        vehicles.append(obj)
```

- **사람 객체와 차량 객체들의 이동 모습 : 객체의 등장, 시작 point를 이은 그래프**

  차례대로 사람 객체들의 이동, 차량 객체들의 이동

```
print_2plots(persons, vehicles)
```





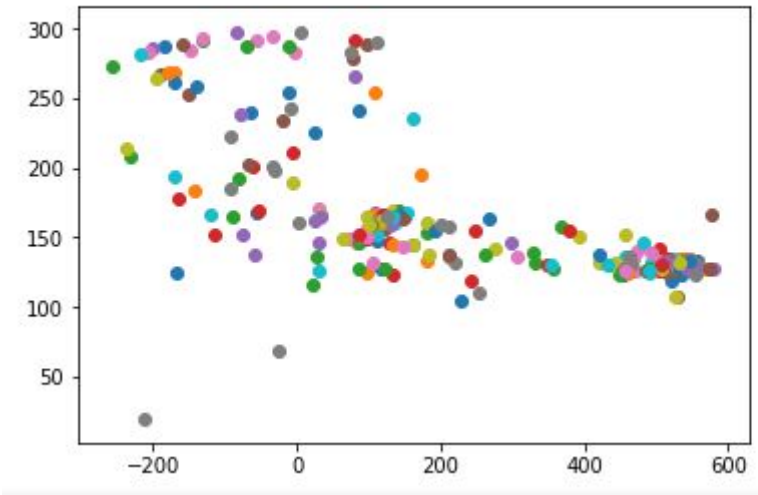
- **영상 속 도로 모습이 반영된 것을 확인할 수 있다.**

- **도로 좌표 근사**

  가정 : 차량 객체들의 시작점에서 도로의 끝점을 근사 할 수 있을 것이다.

```
# 도로 근사를 위한 데이터 분석 : 시작 위치에 최대한 근사 시키자
#print(vehicles[0].get_start_point())
#print(vehicles[0].get_start_point()[0])
#print(vehicles[0].get_start_point()[1])

for vehicle in vehicles:
    start_point = vehicle.get_start_point()
    print('start point : ', start_point)
    finish_point = vehicle.get_finish_point()
    print('finish point : ', finish_point)

    plt.scatter(vehicle.get_start_point()[0], vehicle.get_start_point()[1])
plt.show()
```

```
start point :  [553.755 123.815]
finish point :  [553.565 123.315]
start point :  [532.52  131.695]
finish point :  [530.025 130.18 ]
```

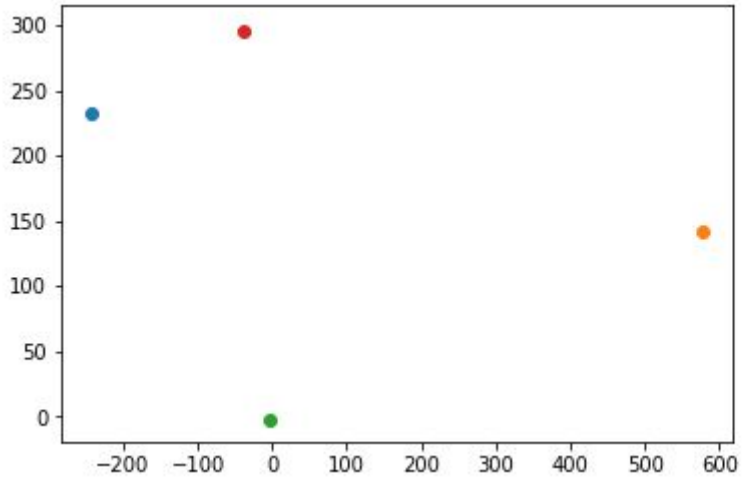


  - **차량의 시작 좌표 (x, y)만을 그려본 그래프**

도출 : 적절히 선택한 차량 객체들의 시작점을 평균하여 도출할 수 있었다.

```
# plt
plt.scatter(horizontal_small_point[0], horizontal_small_point[1])
plt.scatter(horizontal_big_point[0], horizontal_big_point[1])
plt.scatter(vertical_small_point[0], vertical_small_point[0])
plt.scatter(vertical_big_point[0], vertical_big_point[1])
plt.show
```

```
horizontal small :  (-241.49333333333334, 231.875)
horizontal big :  (576.5799999999999, 141.09833333333333)
vertical small :  (-2.7600000000001, 64.25833333333334)
vertical big :  (-37.53333333333334, 296.03833333333336)

<function matplotlib.pyplot.show(*args, **kw)>
```

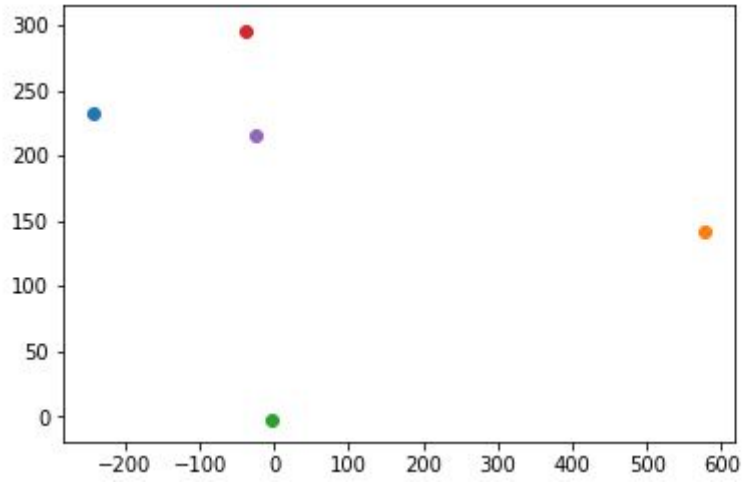


- **교차로 좌표 근사**

  도출 : 두 도로직선의 교점으로 교차로 좌표를 근사할 수 있었다.

```
# plt
plt.scatter(horizontal_small_point[0], horizontal_small_point[1])
plt.scatter(horizontal_big_point[0], horizontal_big_point[1])
plt.scatter(vertical_small_point[0], vertical_small_point[0])
plt.scatter(vertical_big_point[0], vertical_big_point[1])
plt.scatter(road_center[0], road_center[1])
plt.show
```

```
(-24.291179447877177, 215.54678904237588)

<function matplotlib.pyplot.show(*args, **kw)>
```



(3 - 4). Object class 객체 생성

- **Car 객체 구성 :**

```
class Car:
    def __init__(self):
        self.appear_frame = -1
        self.start_road = -1
        self.finish_road = -1
        self.velocity1 = -1
        self.velocity2 = -1

        self.loc = np.array([0.0, 0.0, 0.0])
        self.vel = np.array([0.0, 0.0, 0.0])
        self.radius = 0.3
        self.mass = 1.0

        self.force = np.array([0., 0., 0.])
        self.gravity = np.array([0., 0., 0.])
        self.colPlane = np.array([0., 0.5, 0., 0.])

        self.latency = 0.0
```

- **차량 객체들의 집합 vehicles에서 Car 객체 생성**
  - **appear_frame : 차량 객체의 등장 frame**
  - **start_road : 차량 객체가 입장하는 도로**
  - **finish_road : 차량 객체가 퇴장하는 도로**
  - **velocity1 : 입장 도로에서 교차로 까지의 평균 속도(dist/frame len)**
  - **velocity2 : 교차로에서 퇴장 도로까지의 평균 속도(dist/frame len)**

    *** 교차로 중심 좌표와 가장 가까운 차량 좌표로 계산**

  - **loc : 차량 객체가 그려질 위치([x, y, z]의 3차원 벡터)**
  - **vel : 차량 객체의 속도([x, y, z]의 3차원 벡터)**
  - **radius : 구로 그려질 차량 객체의 반지름**
  - **mass : 차량 객체의 질량**
  - **force : 차량 객체에 가해지는 힘**
  - **gravity : 차량 객체에 가해지는 중력**
  - **colPlane : 차량 객체가 부딪힐 평면, 디디고 있는 지면**
  - **latency : 차량 객체의 vel가 0이 된 시간, 신호에 걸린 대기시간**

    *** 최적의 신호주기를 찾기 위한 기준, 최소의 평균 신호 대기시간**

- **cars_data.csv 작성 : 차량 객체들의 데이터 저장.**
  - **data_analyzer.py 실행 최종 결과.**

```
# cars_data.csv 저장
appear_frame_col = []
start_road_col = []
finish_road_col = []
velocity1_col = []
velocity2_col = []
for car in cars:
    appear_frame_col.append(car.get_appear_frame())
    start_road_col.append(car.get_start_road())
    finish_road_col.append(car.get_finish_road())
    velocity1_col.append(car.get_velocity1())
    velocity2_col.append(car.get_velocity2())

cars_df = pd.DataFrame(data = {
    'appear_frame' : appear_frame_col,
    'start_road' : start_road_col,
    'finish_road' : finish_road_col,
    'velocity1' : velocity1_col,
    'velocity2' : velocity2_col,
})
print(cars_df)
cars_df.to_csv('./cars_data.csv', header = True, index = False)
```

```
appear_frame,start_road,finish_road,velocity1,velocity2
3.0,0,3,0.0,0.0
2465.0,1,0,5.862921283520026,228.43136161694 76
3.0,0,1,2.0421029905120704,4.628680559241736
3.0,3,0,1.5419013287814 47,8.7371410614 39338
1346.0,2,3,5.987429465617678,0.2421598139391402
222.0,2,3,4.4928702089138875,0.2746826091681411
1619.0,0,3,0.0,0.0
1787.0,0,2,4.950304404821589,0.0
1810.0,3,0,1.7370513353283612,246663.0266679626
1277.0,3,1,1.8566679605918408,51.981690394328 17
1263.0,0,3,0.0,0.0
1582.0,0,2,0.526018940791808,0.0
382.0,2,0,10.200382363004723,0.44391907940384007
292.0,1,0,4.688111902426992,25.77407248305814
95.0,2,3,6.271597828028752,0.07840787532942753
1751.0,0,3,9.811731060806434,0.48954259563343416
2558.0,1,0,5.800932754700337,0.7910590568850361
474.0,2,0,4.550634993564184,0.6501703146862893
1560.0,1,3,2.7097831678184225,1.5413552736939111
```

(4) intersection_simul.py : Intersection Simulate

- **도로 좌표 근사**

  실제 도로들의 좌표 값과 차량들의 이동 좌표들은 값이 너무나 크고 중구난방임.

  교차로 좌표에 신호체계를 구현한다 하더라도 건널목의 크기를 상정하기가 어려움.

  그래서 차량 객체는 입장도로와 차량도로만을 가지고 그 값은 [0, 1, 2, 3] 사이의 값임.

  [0, 1, 2, 3]의 값은 [from_bottom, from_right, from_up, from_right]를 의미한다.

  차량의 위치를 ground truth의 중심으로 근사하였고

  그것을 촬영하는 CCTV의 각도가 있기에 차선 정보를 도출할 수가 없었다.

  그래서 교차로의 각 도로를 2차선으로 단순화했다.

  이렇게 함으로써 교차로를 (0.0)에 고정할 수 있었다.

- **차량 속도, 위치 연산 : Car.simulate(et)**

  프로그램의 run time, et에 60을 곱하여(60 fps의 영상) run frame을 계산.

  run frame이 차량의 appear frame을 넘어서면 속도와 위치를 연산함.

```
def simulate(self, et, dt):
    run_frame = et * 60
    if run_frame >= self.appear_frame:
        acc = self.gravity + self.force / self.mass
        self.vel = self.vel + acc*dt
        self.loc = self.loc + self.vel*dt
```

- **차량 회전 : Car.corner()**

  차량의 start road와 finish road에 따라 vel 값을 초기화

- **차량간 상호 작용 : Car.colHandlePair(traffic_light)**

  두 객체(구) 사이의 거리, dist와 반지름 합, R을 비교한다.

  dist < R : 두 차량이 충돌

  충돌에 의한 반작용을 계산한다.

  (dist >= R) and (dist <= R + safety_dist) : 뒤 차량이 안전거리 내 존재

  뒤 차량의 속도에 앞 차량의 속도를 대입하여 충돌을 방지한다.

- **차량, 신호 상호 작용 : Car.colHandle(traffic_light)**

  1. 바닥과의 충돌을 계산

  2. 신호와의 충돌을 계산

  차량의 위치가 교차로 내에 있고 빨간불일 때 : 차량 속도를 0으로 초기화.

- **차량의 대기시간 계산 : Car.cal_latency(dt):**

```
def cal_latency(self, dt):
    if self.start_road == 0 and self.start_road == 2:
        if self.vel[2] == 0.0:
            self.latency += dt
    else: # start road == 1 or 3
        if self.vel[0] == 0.0:
            self.latency += dt
```

  - 차량의 진행 방향 속도가 0일 때, 대기시간에 더 해줌.

- **Traffic class 객체 구성**
  - **light_on_order**

    입장 차량이 많은 도로의 순서를 신호 순서로 정함.

  - **light_on**

    현재 파란불이 들어온 도로

  - **loc0 ~ 3 : 각 도로 별 신호등 위치**
  - **whole_duration : 전체 신호가 한번도는 주기(시간)**
  - **durations : 각 신호 별 파란불 시간**
  - **whole_duration * (해당 도로의 차량 개수/전체 차량개수) : 백분율로 초기화함.**
  - **start_time, finish_time : light_on 신호의 변경을 위한 시간 측정**

```
class Traffic:
    def __init__(self):
        self.light_on_order = []
        self.light_on = 0

        self.loc0 = np.array([3.0, 0.5, 10.0])
        self.loc1 = np.array([-10.0, 0.5, 3.0])
        self.loc2 = np.array([-3.0, 0.5, -10.0])
        self.loc3 = np.array([10.0, 0.5, -3.0])

        self.whole_duration = 30
        self.durations = np.array([0.0, 0.0, 0.0, 0.0])

        self.start_time = 0.0
        self.finish_time = 0.0
```

- **Traffic.simulate(dt)**

  현재 light_on 신호의 주기를 넘어서면 다음 신호를 light_on으로 초기화.

```
def simulate(self, dt):
    current_dur = self.durations[self.light_on]

    self.finish_time += dt
    if self.finish_time - self.start_time >= current_dur:
        self.start_time = self.finish_time
        self.light_on = self.light_on_order[(self.light_on_order.index(self.light_on) + 1) % len(self.light_on_order)]
```

- **Traffic.draw()**

  각 신호를 위치에 맞게 그려 냄.

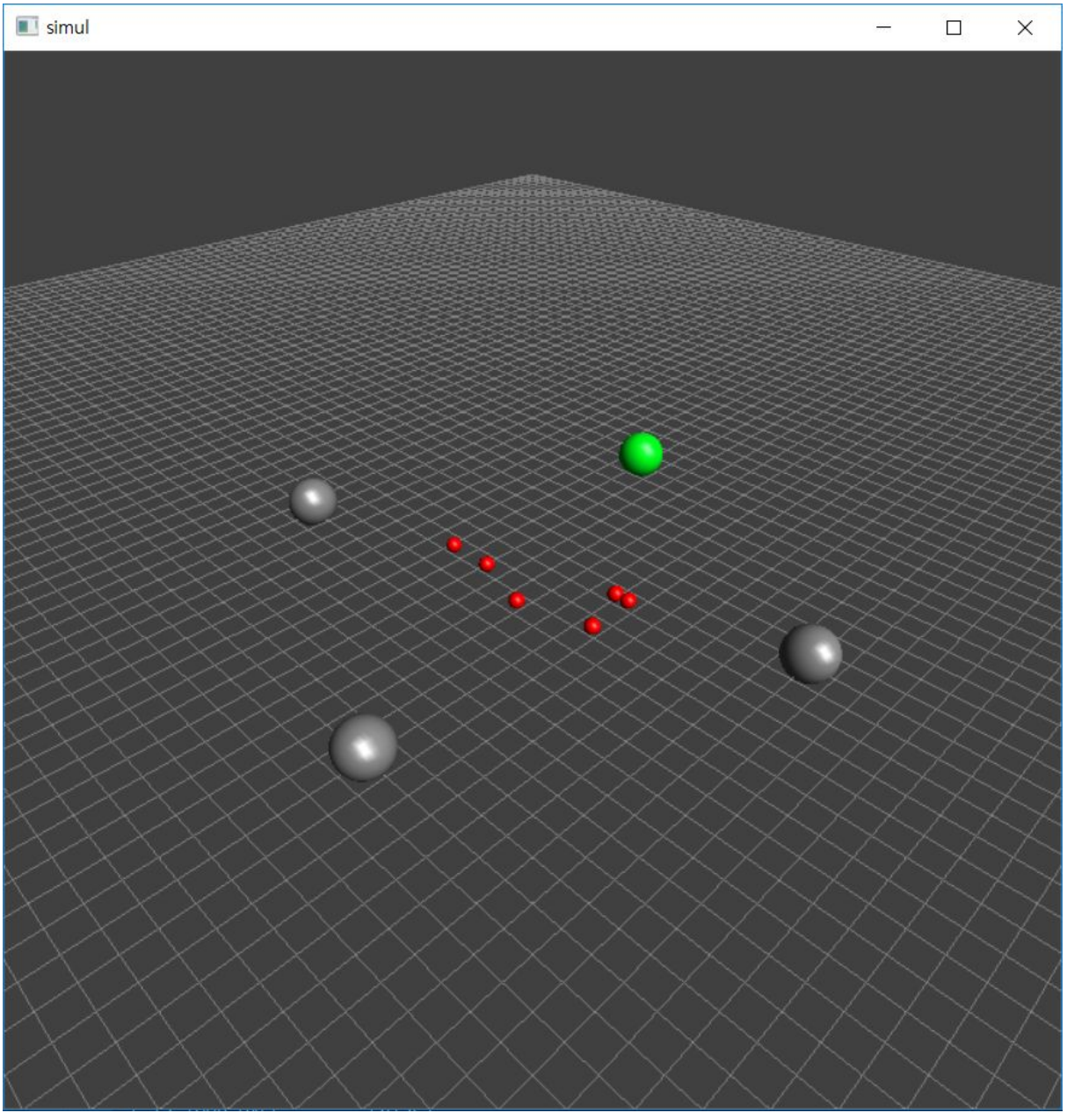
  단 light_on한 신호만 파란불로 그려냄.

- **실험 시나리오 : myGame.frame()**
  - **차량 회전 확인 : car.corner()**
  - **차량 속도, 거리 계산 : car.simulate(et, dt)**
  - **차량, 신호 상호작용 : car.colHandle(traffic_light)**
  - **차량의 대기시간 계산 : car.cal_latency(dt)**
  - **차량 간 상호작용 : cars[i].colHandlePair(cars[j])**
  - **프레임 별 차량 그리기 : car.draw(et)**
  - **신호 변경 : traffics.simulate(dt)**
  - **시간 별 신호 그리기 : traffics.draw()**

```
def frame(self):
    dt = self.getDt()
    et = self.getEt()

    super(myGame, self).frame()

    # cars
    traffic_light = self.traffics.get_light_on()

    for car in self.cars:
        car.corner()
        car.simulate(et, dt)
        car.colHandle(traffic_light)
        car.cal_latency(dt)

     # 충돌을 세팅, 탄성계수 0.1
    for i in range(len(self.cars) - 1):
        for j in range(i+1, len(self.cars)):
            self.cars[i].colHandlePair(self.cars[j])

    for car in self.cars:
        car.draw(et)

    # traffics
    self.traffics.simulate(dt)
    self.traffics.draw()
```

- **교차로 실험 및 실험 결과**
  - ❖ **종료 조건**
    - ➢ 신호 체계의 전체 duration, whole_duration을 dur_delta 만큼 줄여간다.
    - ➢ whole_duration <= 0이 되는 loop에 진입 시 실험을 종료한다.
    - ➢ 각 whole_duration의 avg_latency와 traffic_times를 그래프로 그려낸다.
    - ➢ avg_latency 중 최소 값을 most_efficient_duration을 출력한다.
    - ➢ 프로그램을 종료한다.

- **실험 화면**

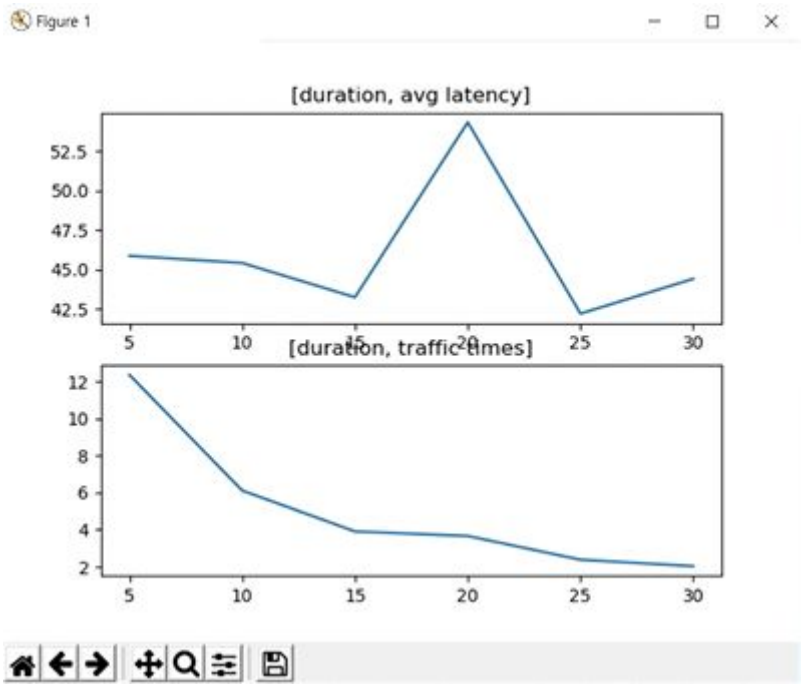


- **시연 영상 : https://youtu.be/eq0tEKQBlrM**

  신호체계 주기, whole_duration : 30

  신호체계 주기 변화량, dur_diff : 5

- **결과 분석**



  - **most_efficient_duration : 25, 약 2.4회의 신호주기로 모든 차량을 소화함.**

4.3. 산업체 멘토링 내용 및 반영

4.3.1) 프로젝트 진행 수준에 대한 의견

- **실제 서비스를 목표로 하는지, 딥러닝 프로세스를 체계적으로 진행하는 것이 목표인지를 정할 것**

❖ 실제 서비스를 목표로 그에 가까운 시스템을 완성하도록 하는 것으로 진행 방향을 결정했다.

4.3.2) 프로젝트 발전 방향에 대한 의견

- 교통신호 최적화를 하기에는 하나의 신호만을 조정하는 것으로는 한계가 있다.
- 교통신호 주기만을 조정하기보다 전체적인 도로교통 시스템을 고려할 필요가 있다.

❖ 다음과 같은 내용에 따라 CCTV영상에서 데이터를 추출하고 이를 통해 가상의 도로환경을 만들고, 시뮬레이팅 하는 과정을 일련의 시스템으로 구성하였다. 시스템을 하나의 교차로에서 전체 도로로 확장할 경우, 전체 도로의 교통신호를 최적화할 수 있을 것으로 보고, 우선 하나의 교차로에서의 교통량을 늘릴 수 있는 최적의 신호 주기를 조정하는 것을 과제로 삼았다.

5. 결론

5.1. 기대효과

- 충분한 컴퓨터 성능이 받쳐진다면, 긴 길이의 CCTV 영상을 분석하여 이를 기반으로 시뮬레이션 할 수 있을 것이다.
- 특정 교차로의 교통량을 CCTV 분석을 통해 그 정보를 데이터화하여 축적할 수 있다.
- 분석된 데이터를 바탕으로 교통 효율 증대를 위한 신호주기 최적화를 기대할 수 있다.
- 특정 교차로의 교통량 분석 및 신호주기 최적화를 하는 시스템을 기반으로하여, 도로망 전체에 대한 교통량 분석 및 시뮬레이팅 환경을 구성하면, 도로망 전체의 신호주기를 최적화 할 수 있을 것으로 기대한다.

5.2. 한계점

- 단일 교차로에 대해서만 신호주기를 최적화할 수 있다. 즉, 단일 교차로의 교통량 증가가 결과적으로 전체 도로의 교통량 증가를 불러일으킬 수 있는지에 대한 검증이 필요하다.
- 분석한 CCTV영상의 길이가 짧아, 이를 바탕으로 축적된 데이터를 기반으로한 시뮬레이션이 유효한 결과를 불러일으킬 수 있는지에 대한 검증이 필요하다.

5.1. 시스템 개선사항

- 실제 도로의 신호체계는 대도로에서 시작하여 소도로로, 전체를 복합적으로 고려한다. 이때 가장 큰 결정요인은 막힘없이 최대한 직진할 수 있도록 함이다. 본 시스템은 이 것을 최대한 반영하기 위해 전체 차량의 평균 신호 대기시간으로 최적의 신호주기를 찾아내지만 그럼에도 충분하지 못할 것이다. 이에대해 복수의 교차로에서 각각 결정된 최적의 신호가 전체적으로 어떤 결과를 내는지에 대한 실험이 추가적으로 필요하다.
- 차선의 정보를 반영하지 못한점이 가장 아쉽다. 이는 ground truth의 중심좌표로 둔 차량의 위치와 CCTV의 촬영 각도에 기인한다. 만일 시점을 교차로의 수직으로 둘 수만 있다면 차량의 위치 정보에서 충분히 차선 정보를 도출해낼 수 있을 것이다. 차선 정보까지 더해진다면 보다 실제에 가깝게 실험할 수 있을 것이고 보다 효과적인 솔루션을 제공할 수 있을 것이다.

6. 구성원별 역할 및 개발 일정

6.1. 구성원별 역할

구성원	역할
오세현	추출 데이터 분석 및 활용 opencv를 활용한 시뮬레이터 제작
박민하	오픈소스 분석 및 활용 교통량 데이터 추출
김재현	오픈소스 분석 및 활용 도로 CCTV 영상 데이터 확보
공통	최적화 신호체계 구상, 구현 모듈, 모델 및 아키텍처 분석, 개선, 구현 및 취약점 보완

6.2. 개발 일정

월	5				6				7				8					9			
주차	2	3	4	5	1	2	3	4	1	2	3	4	1	2	3	4	5	1	2	3	4
주제 선정	■	■	■	■	■																
주제 변경									■	■	■	■	■	■							
관련지식 학습				■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	
개발환경 구축					■	■	■	■	■	■	■	■	■								
설계							■	■	■	■	■	■	■	■							
구현													■	■	■	■					
통합 및 테스트													■	■	■	■	■	■	■		
디버깅															■	■	■	■	■	■	■
결과 보고서 작성																		■	■	■	■

7. 참고문헌

[1] 김원철. “VISSIM을 활용한 교통연구 사례” 『충남연구원 논단』, 2018

[2]SAT “AVC, VDS, VMS 시스템”, http://www.satech.co.kr/products-traffic_system04.html

[3] K. He, G. Gkioxari, P. Doll´ar, and R. Girshick. Mask r-cnn. In ICCV, 2017. 3

[4] H.-S. Fang, S. Xie, Y.-W. Tai, and C. Lu. RMPE: Regional multi-person pose estimation. In ICCV, 2017.

[5] J. Liang, L. Jiang, L. Cao, L.-J. Li, and A. Hauptmann. Focal visual-text attention for visual question answering. In CVPR, 2018.

[6] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In ECCV, 2018.

[7]H. Hu, J. Gu, Z. Zhang, J. Dai, and Y. Wei. Relation networks for object detection. In CVPR, 2018.

[8]Marc-André Vollstedt, “Motion Paths Extraction”, https://github.com/mavoll/MotionPathsExtraction