# PolyHACK 2020: ELCA Challenge - Traffic optimization with real-time simulation

**Team Name: 2020**
**Team Members**
**Amaris Chen, Chi-Ching Hsu, Ya-Chi Yu, Ting-Yu Chen**
chenam@student.ethz.ch, hsuch@student.ethz.ch, yuyachi@student.ethz.ch, tinchen@student.ethz.ch
**Date: 2020 November 7-8**

## 1 Introduction

The **Level 1** challenge is to minimize the maximum number of simultaneous buses and minimize the number of pedestrians not transported. Intuitively we want each bus to transport as many pedestrians as possible. Our algorithm could transport most of the pedestrians with much less buses compared with the given naive method.

The **Level 2** challenge has two more criteria, minimize the kilometers driven by all buses and the total wait time of all transported passengers. We did not have time with the **Level 3** challenge, but our methods should also be applicable to the **Level 3** challenge.

## 2 Methods

Our goal is to reach the criteria of both **Level 1** and **Level 2**, see below. Given this goal, several assumptions are made to simplify the problems.

**Level 1**

1. maximum number of simultaneous buses
2. number of passengers not transported at the end of the simulation

**Level 2**

1. maximum number of simultaneous buses
2. kilometers driven by all buses
3. total wait time of all transported passengers
4. number of passengers not transported at the end of the simulation
5. bonus for completing Level 2

### 2.1 Time Interval

First, we simulate the first 21600 seconds (6 hours) and generate **Figure 1**, which indicates *depart* versus number of passengers. A series of clusters can be found in **Figure 1**, where each cluster corresponds to a 30 minutes time interval. Based on **Figure 1** we decide to deploy a batch of buses every 30 minutes, so each batch of buses should fulfill the demand of each cluster in **Figure 1**.

In addition, it is recommended to use the same type of bus (type BUS_L, with capacity 8) only in both **Level 1** and **Level 2**, so we decide to assign at most $N_p$ passengers to each bus.

To decide which group of passengers should take the same bus, intuitively we want to pick up passengers that have starting points close to each other and similar departure time. The problem with similar departure is already solved by introducing a batch of buses every 30 minutes. To know if passengers are close to each other, we make an assumption according to the name of edge (street).

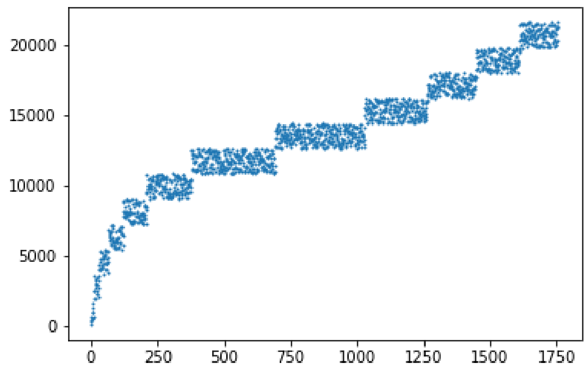

Figure 1. Depart versus number of passengers generated in the first 21600 seconds (6 hours)

We assume that edges with similar ID would be close to each other, and thus we sort the pedestrians by their *edge_from* attribute in each time interval. We introduced several sorting methods because the name of edges have different lengths, symbols, configurations. Some of them consist of digits only but some of them have alphabet and some even have special characters such as #. Sorting methods are discussed in the following section.

## 2.2 Sorting Methods (*edge_from*)

### 1) Baseline Sort

In **Baseline Sort**, we sort *edge_from* according to the default sorting algorithm in python, which negative sign goes before digits and the digits are sorted in ascending order.

### 2) Remove Negative Sort

By checking *osm.net.xml* in the *trafficmap* repository we found out that there are several streets with the same name but different signs (e.g. 142174400 and -142174400). We assume that they are the same streets but in different directions, since one of them is from junction A to B and the other is from B to A.

This implies that the same name but different signs edges should be close to each other, so we remove the negative sign and sort. We also tried to remove the # mark, but the sorting sequence is the same for most of the cases we tried, so in the end we did not remove the # mark.

3) **Remove Negative Sort + Length**

This method is built upon **2) Remove Negative Sort.** We made a further assumption that for example ‘123’ should be closer to ‘124’ than ‘1234’, so edge names are sorted by not only the digits but also the length after removing the negative sign.

| Sorting methods | 1) | 2) | 3) |
|---|---|---|---|
| Results | '-23205071#8', '-258487526#1', '107620891#0', '12905039', '142174392#2', '146070361#0', '390364165', '41216837#0', '51834234', '823646232#0' | '107620891#0', '12905039', '142174392#2', '146070361#0', '-23205071#8', '-258487526#1', '390364165', '41216837#0', '51834234', '823646232#0' | '-23205071#8', '-258487526#1', '41216837#0', '107620891#0', '12905039', '390364165', '146070361#0', '142174392#2', '51834234', '823646232#0' |

After we have sorted passenger list, a batch of buses are deployed accordingly. The number of passengers assigned to each bus $N_p$ is 8 in our case because bus type BUS_L with capacity 8 is used. First we take 8 passengers according to their departure time and then drop them off.

For each group we run the following algorithm. We divided all passengers with departure time in certain time interval by $N_p$ and each $N_p$ correspond to a group and for each group we assign a bus. The route would naively depends on the sorting order of pedestrian, the pseudo codes are shown below:

```
Naive Approach
----------------------------------------
Add a Bus
Bus departs from the deport
Sort pedestrians by edge_from
For p in sorted pedestrians
    Pick up p
For p in sorted pedestrians
    Drop off p
Bus back to the depot
```

## 2.3 Greedy Algorithm

Given the target pedestrians of each bus, we try to minimize kilometers driven by all buses by greedily searching for nearest stops. Assumption: we assume that the distance between two stops is directly proportional to the number of edges of the dijkstra route between them.

4) **Greedy Algorithm I - Greedily search for dropping off stop**

First we pick up the pedestrians in the order of their departure time. After picking up all the pedestrians, the bus would be at the picking up position of the last pedestrian. We greedily search for the next pedestrian to drop off by the distance between their destination and the current location of the bus. The pseudo code is shown below:

```
Greedy Algorithm I
----------------------------------------
Add a  Bus. Bus.location = depot
Sort pedestrians by departure time
For p in sorted pedestrians
    Pick up p
    Bus.location = p.srcLocation
For p in people on the bus
    Drop off p with destination nearest to Bus.location
    Bus.location = p.destLocation
Bus back to the depot
```

5) **Greedy Algorithm II - Greedily search for next picking up or dropping off stop**

The bus would first go to pick up the pedestrian with the earliest departure time, now the bus decides to either pick up the next pedestrian or drop off the pedestrians on the bus  by the distance between possible stops and current location. The pseudo code is shown below:

```
Greedy Algorithm II
----------------------------------------
Add a  Bus. Bus.location = depot
Sort pedestrians by departure time
Pick up the first person with earliest departure time
nextStops = {pick up the next person waiting on street with earliest departure
time, drop off the pedestrians on the bus}
For stop in nextStops
    Find the distance of stop to Bus.location
Pick the stop nearest to Bus.location from the nextStop set
Remove the picked stop from nextStop set
Update Bus.location
Bus back to the depot
```

## 2.4 Hyperparameter Tuning

The list of hyperparameters are listed as follows. We tried with different combinations of hyperparameters and the results are shown in the following section. The third hyperparameter *departure time of buses* means when the bus leaves the depot. The options are the mean, median, min, max of the group of $N_p$ passengers departure time. If it is set to min, the bus leaves the depot once the first passenger demand appears. If it is set to max,

**Hyperparameters**

1. Number of passengers assigned to each bus, $N_p$ : 4, 8
2. Simulation time (s): 10800, 86400
3. Departure time of buses: mean, median, min, max
4. Stop duration of buses: 20, 50
5. Length of time interval (mins): 20, 30
6. Pedestrians_seed: 30
7. Ordered by: **1) Baseline Sort, 2) Remove Negative Sort, 3) Remove Negative Sort + Length**
8. Route plan within group: **1) Naive approach, 2) Greedy Algorithm I, 3) Greedy Algorithm II**
9. Preprocessing of edge from: remove negative sign, remove #

In summary, we first assigned passengers into different groups based on their departure time. Then a batch of buses are deployed for each group with maximal $N_p$ passengers onboard. These $N_p$ passengers are selected by **1) Baseline Sort, 2) Remove Negative Sort, 3) Remove Negative Sort + Length**. The route of the bus is based on **1) Naive approach 2) Greedy Algorithm I, 3) Greedy Algorithm II.**

# 3 Results

Only two out of four machines in our team can run the simulation successfully without crashing and one of the two machines can only simulate up until 10800 seconds. Most of the following results are simulated only with 10800 seconds due to this reason.

## Level 1

Even though the maximum capacity of BUS_L is 8, we found that assigning 4 passengers to each bus would maximize the number of rides in simulations with 10800 seconds, see **Table 1**. When $N_p = 8$ people, the buses spend too much time picking up all 8 passengers and thus many of them are still on the bus at the end of simulation. The results in **Table 1** are simulated with **1) Baseline Sort**, time interval of 30 minutes, and for 10800 seconds.

| | maximum number of simultaneous buses | number of passengers not transported at the end of the simulation |
|---|---|---|
| 4 | 95 | 217 |
| 8 | 48 | 282 |

Table 1. Results of **Level 1** metrics with different $N_p$

### Level 2

The number of passengers not transported at the end of the simulated reduces from 193 to 173 when using **4) Greedy Algorithm I** and the kilometers driven by all buses is reduced from around 12k to 10k as shown in **Table 2**, which indicates that **4) Greedy Algorithm I** not only transports more passengers but also reduce the travel distance. In addition, the average waiting time of transported passengers for both methods are both around 80 seconds, but **4) Greedy Algorithm I** transports more passengers so the total waiting time is longer. The results in **Table 2** are simulated with a time interval of 20 minutes and for 10800 seconds.

| | number of passengers not transported at the end of the simulation | kilometers driven by all buses | total wait time of all transported passengers (average) |
|---|---|---|---|
| **3) Remove Negative Sort + Length** | 193 | 12469.97 | 85031.55 (79.76) |
| **4) Greedy Algorithm I** | 173 | 10728.86 | 108814 (79.58) |

Table 2. Results of **Level 2** metrics with different algorithms

## 4 Conclusions

A new bus dispatching approach is presented in this report to fulfill the requirements in both **Level 1** and **Level 2** challenges. The **5) Greedy Algorithm II** has the best performance over all methods we explored and is able to fulfill around 97% of the demand in a whole day simulation. Other methods and hyperparameters we tried can also reach around 94% of the demand.

Although we do not have time for the **Level 3** challenge, our method is still applicable to it by adjusting the parameter $N_p$. Instead of having a fixed $N_p$ we can assign different $N_p$ to different groups.

In the future studies, a more extensive search in hyperparameters could be done for the better model. In addition, different types of buses could also be implemented to cope with more complicated scenarios.

Last but not least, we would like to express our gratitude to ELCA for arranging this fascinating challenge. Hope to cooperate with you again soon. You can find our contraction details at the beginning of this report.