Bike Rebalancing Problem - Project I

COMP 6651, Fall 2019

October 9th 2019

Presented to:
Dr. Brigitte Jaumard
PhD, Professor
Computer Science & Software Engineering
Concordia University

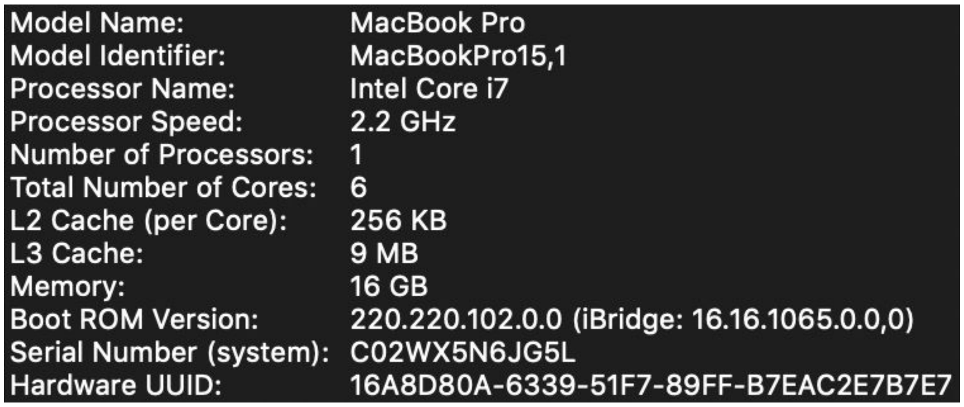
Presented by:
Tian Nan Liu
Computer Science, Honours
ID: 26580017
Concordia University

Table of Content

Abstract

The purpose of this report is to provide full details regarding part I of the BRP project, which aims at minimizing the number of vehicles needed to ensure all bike stations are half full. The first section contains details about the dataset generation methodology, followed by the description of the Greedy Algorithm of the BRP with details regarding its time complexity. Lastly, the experimental analysis of the code performance, dataset generation, and results of the BRP will be analyzed in detail.

Machine details:



```
Model Name:                MacBook Pro
Model Identifier:          MacBookPro15,1
Processor Name:            Intel Core i7
Processor Speed:           2.2 GHz
Number of Processors:      1
Total Number of Cores:     6
L2 Cache (per Core):       256 KB
L3 Cache:                  9 MB
Memory:                    16 GB
Boot ROM Version:          220.220.102.0.0 (iBridge: 16.16.1065.0.0,0)
Serial Number (system):    C02WX5N6JG5L
Hardware UUID:             16A8D80A-6339-51F7-89FF-B7EAC2E7B7E7
```

Dataset Generation

In order to simulate the BRP problem under the guidelines of the project, it is quintessential to build a code structure in order to scale multiple experiments to test the Greedy Algorithm. I am using a Generator object to generate data used for the bike and topology datasets and to parse those inputs from text files into dictionaries. The bike dataset is generated by first randomly generating the number of deficit stations under the given conditions of at least 3 with an upper bound determined by the number of available bikes and capacity of each bike station. The number of bikes for each deficit station is generated randomly of integers from the range of [min_deficit_bikes, max_deficit_bikes], which is [0, 4] for the case of part I. The remaining number of available bikes are then randomly distributed to the non-deficit bike stations with each being in the range of [min_deficit_bikes, min(max_station_bikes, num_avail_bikes)]. This allows the number of non-deficit and deficit stations to be even out in the long-run and will ensure that there will be enough bike station visits to satisfy the lower bound condition of 4 bike stations. The bike stations and the intersections are shuffled before generating the text file for the bike dataset, which contains the number of available bikes with the number of bikes in each bike station. The time complexity is roughly linear times because the bike dataset generation is separated by first generating the deficit bike stations followed by the non-deficit stations, which is bounded by the number of bike stations.

The bike dataset generation provides the vertices of the graph, which will then be used for the topology dataset generation to generate random edges to form a connected and undirected graph. Until the number of edges has been reached, each iteration will randomly two vertices without replacement and will be added to the graph if the pair nor the inverted pair exists in the graph. Each iteration will also generate a random weight for the edge pair, which is bounded by [30, 40] minutes. Once completed, the edges will be passed onto a Graph object, which will use a dictionary to store the list of vertices followed by their accessible vertices and weights. This entire process ensures that the graph is connected, else it will restart the entire graph generation before generating the text file for the topology dataset. The time complexity is roughly linear times for the generation of random edges before passing into the Graph object to store its edges. However, the time complexity can increase to polynomial of degree 2 if the graph yields non-connected graph for many iterations, which is not very likely for this project.

Greedy Algorithm

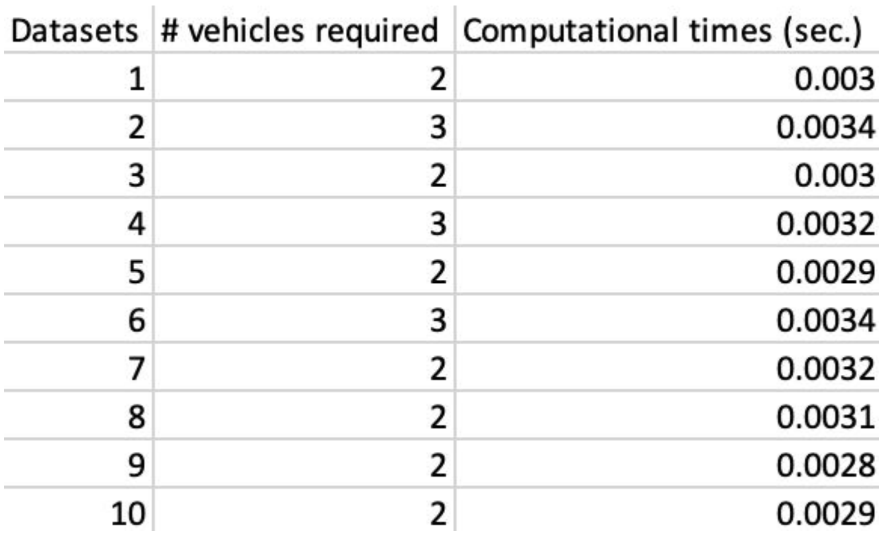
This algorithm depends on the bike dataset and graph generated from the Dataset Generation. The idea of this algorithm first starts with maximizing the capacity of each vehicle from the non-deficit bike stations by visiting to the closest non-deficit bike station from the depot, and then moving to the closest one from the previous non-deficit bike station. This process will continue until the vehicle reaches its maximum capacity or when it reaches the maximum number of bikes needed in the deficit bike stations to fully satisfy the demands. Each station's bike numbers are determined by min(station's bikes - min_station_bike, max_station_bikes - vehicle's bikes). The max_station_bikes is determined by min(max_station_capacity, bikes needed to fulfill all demands). The algorithm will now visit the closest deficit station from the previous non-deficit station that demands bikes. The algorithm will check the condition of weight(u, v) and weight(v, depot) as it will choose path(v, depot) if there is not enough time to process both paths. The remaining bikes are initialized for the next vehicle and the algorithm will continue once all demands have been met for the deficit bike stations.

The time complexity starts with the generation of shortest paths from depot to each bike station, which can take O(#bike_stations*(#vertices + #edges)*log(#vertices)), followed by O(#bike_stations*log(#bike_stations)) to sort the paths by closest to the depot. The shortest path costs O((#vertices + #edges)*log(#vertices)) because of using a linear time for extract_min() and a set to store for the vertex Q. The next step is bounded by the number of deficit stations that demand bikes, which then visits each non-deficit stations and compute the shortest path between the stations. The same goes for the deficit stations, but they need to compute one extra shortest path between the next potential station and depot to ensure there is enough time to get back to depot before the 480 minutes limit. This again is bounded by O(#bike_stations*(#vertices + #edges)*llog(#vertices)) because of having to repeat the generation of shortest path between the bike stations. More specifically, the non-deficit stations can take O(#non_deficit_stations*(#vertices + #edges)*log(#vertices)) and the deficit stations can take O(#deficit_bike_stations*2*(#vertices + #edges)*log(#vertices)) because of having to do 2 shortest paths for each deficit bike station to ensure having the time to come back to depot. So the total overall time needs to consider traveling to the all of the bike stations, and the worse case scenario is that there exist a deficit station such that it cannot be visited back and forth from the depot within the 480 minute limit, then the demand condition will never be met. Other than this case, the algorithm can fulfill the demand at worse by O(#deficit_stations*inner time computed above), which gives O(2*#deficit_stations^2*(#vertices + #edges)*log(#vertices). Overall, the time complexity is polynomial of degree 3 times the logarithm time, which is roughly equivalent to **O(n^3log(n)).**

One of the key drawbacks is that the core part of the greedy algorithm is separated into two sets of bike stations, which forces some code to not be reusable with very small differences. This is usually not a very efficient way of writing code and can certainly be optimized with smaller helper functions and use of alternative data structures. The main issue is the computation of shortest paths, which is the main source of computation in this greedy algorithm. These paths are calculated between each traversal from non_deficit_station[i] to [j], last non_deficit_station before passing to the closest deficit_station, then followed by deficit_station[i] to [j] and to [depot] to ensure the vehicle can come back within 480 minutes. The graph traversal did not implement the most efficient way using fibonacci heaps, which could have reduced to O(#edges*log(#vertices)) compared to the dictionary data structure. Also, I realized that I could have generated paths with yielding Python Generators to boost some performances. This greedy algorithm works under the conditions given by the project, however it will not scale well with the current setting as the number of edges become significantly larger than the number of bike stations.

Appendix

Table 1: Performance of the Greedy Heuristic



Datasets	# vehicles required	Computational times (sec.)
1	2	0.003
2	3	0.0034
3	2	0.003
4	3	0.0032
5	2	0.0029
6	3	0.0034
7	2	0.0032
8	2	0.0031
9	2	0.0028
10	2	0.0029

Figure 1: Demand Decrease and Number of Vehicles

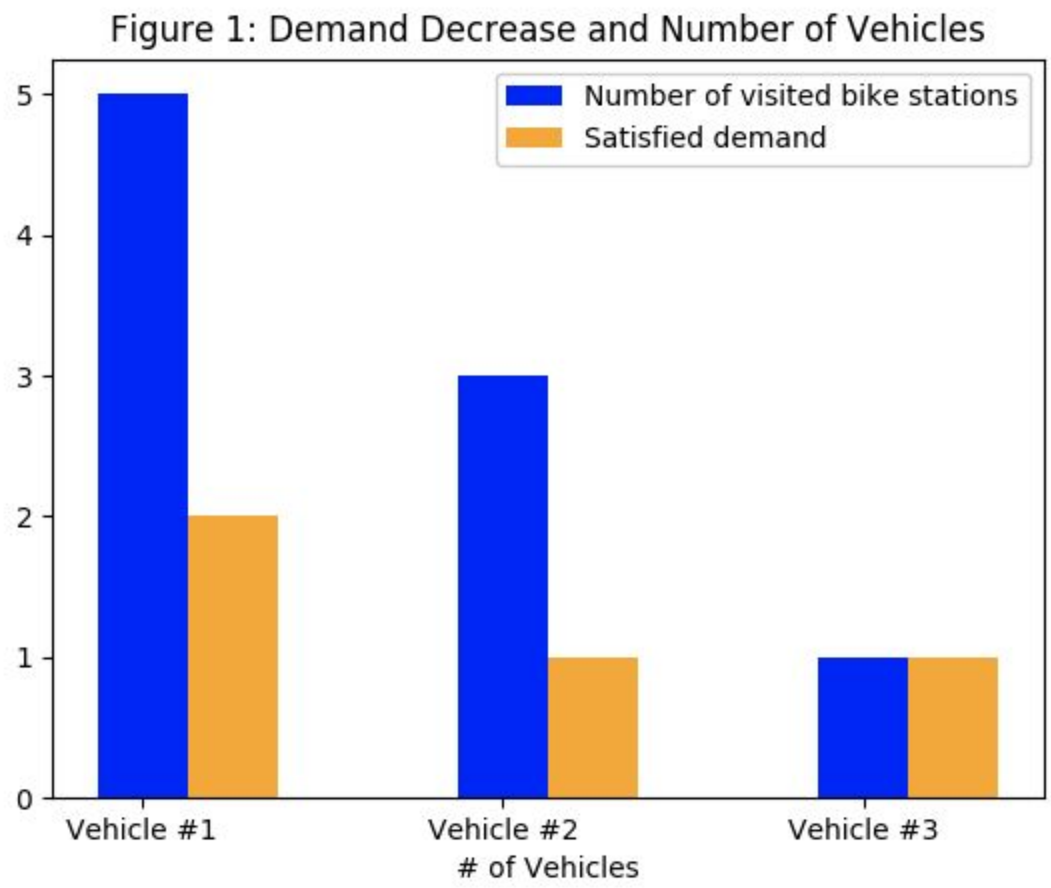


Figure 1: Demand Decrease and Number of Vehicles
Number of visited bike stations
Satisfied demand
Vehicle #1
Vehicle #2
Vehicle #3
# of Vehicles