Master's degree in Data Science and New Digital Businesses (SDNUM)

Optimization Project Report

2022-2023

: Realized by

Ben Marzouk

Wissem

Professor

Mr. Soutil Eric

TABLE OF CONTENTS

FIGURES LIST

INTRODUCTION

- In this project, we will be studying an optimization problem related to the management of a self-service bicycle (SSB) fleet in a city. The company in charge of the fleet wants to balance the bike stations at night by sending a trailer to visit different stations and adjust the number of bikes at each station to its ideal level. The goal is to minimize the global imbalance, defined as the sum of the imbalances of each station, and, among all the possible tours that minimize the global imbalance, to minimize the distance traveled by the trailer. To solve this problem, we will propose a model and use the Julia programming language, with the JuMP library and a free integer linear programming (ILP) solver (Cbc), to solve small instances and show its limits, then implement a heuristic and a metaheurtic method to solve larger instances, and finally we will determine the lower bounds of the imbalances of all the instances by modifying the model written for the exact solution in order to compute the continuous relaxation of the problem. The lower bound will then be useful to evaluate the quality of our heuristic methods.

CHAPTER I

ILP MODEL WITH CBC AND JUMP

Introduction

In this chapter, we will be implementing a program (LBBF.jl) using the Julia programming language and the JuMP library to solve small instances of the self-service bicycle fleet management problem. We will be using an integer linear programming (ILP) solver, Cbc, to find optimal solutions to the problem. The goal of this chapter is to demonstrate the effectiveness of the proposed model and solution method for small instances of the problem. We will first describe the model and the necessary constraints, and then we will present some screen shots of the program running to solve small instances. Finally, we will evaluate the results and discuss any observations or insights gained from the solution process

1 ILP model with Cbc and JuMP

2 Modeling of the problem

2.1 Mathematical Model

2.1.1 Data definition

- n : Number of stations visited during a tour
- k : Truck capacity
- nbp_i: Number of bikes in the station
- cap_i: Capacity of the station i
- $ideal_i$: Ideal number of bikes to be present at the station i
- d_war . n-vector.d_war_i :the distance between the warehouse and the station i.Thes distances are computed by using the coordinates of each station and the ones of the warehouse

2.1.2 Definition of variables

- x_{ij} : binary variable equal to 1 if station i was visited at step j and 0 otherwise. Avec $i \in \{1...n\}$ *et* $j \in \{0...n\}$
- $load_j$: Non-negative integer that represents the number of bikes on the truck after step j. The number of bikes on the truck at the start of Warehouse is : $Load_0$
- $drop_{ij}$: Positive ,zero or negative integer that represents the number of bicycles left at the station i if the number of bicycles is missing or removed from the station i if some bicycles are in excess.

This can be expressed as follows :

$$(weight * (\sum_{i=1}^{n} imbalance_i) + \sum_{i=1}^{n} d\_war_i \; x_{ij} + \sum_{j=1}^{n-1} \sum_{i=1}^{n} \sum_{k=1}^{n} d_{ik} \; y_{ijk} + \sum_{i=1}^{n} d\_war_i x_{i,n}$$

Under the constraints :

- $\sum_{j=1}^{n} x_{ij} = 1 \;\; \forall i$: Each station i is visited exactly once.
- $\sum_{i=1}^{n} : x_{ij} = 1 \; \forall j$ at each step, only one station is visited.
- $load_j = load_{j-1} - \sum_{j=1}^{n} drop_{ij}$
- $imbalance_i \geq nbp_i + \sum_{j=1}^{n} drop_{ij} - ideal_i$
- $imbalance_i \geq -nbp_i - \sum_{j=1}^{n} drop_{ij} + ideal_i$
- $imbalance_i \geq 0$ and integer
- $y_{ijk} \geq x_{ij} + x_{k,j+1} - 1$
- $y_{ij} \in \{0, 1\}$
- $y_{ijk} \geq 0$

- $0 \leq load_j \leq K \ \forall j$: Ensure that the total number of bikes on the truck does not exceed its capacity K .
- $drop_{ij} \leq load_j \ \forall i, j$: Ensure that the total number of bikes dropped off at station i in step j does not exceed the number of bikes on the truck.
- $nbp_i + drop_{ij} \ \leq cap_i \ \forall i, j$: Ensure that the number of bikes at each stage j of the tour does not exceed the capacity of the station.
- $-cap_i \, x_{ij} \ \leq drop_{ij} \ \leq cap_i \, x_{ij} \ \forall i, j$: Ensure that $drop_{ij} \ \neq 0$ if and only if $x_{ij} \ \neq 0$
- $load_j = load_{j-1+} drop_{ij} \ \forall i \ et \ j \in \ \{1...n\}$: Ensure that the number of bikes on the truck at each step j must equal the number of bikes on the truck at step j-1 plus the number of bikes dropped off or picked up at the current station i.

3 Example of the program running with small instance

```
Result - Optimal solution found

Objective value:                189.00000000
Enumerated nodes:               23
Total iterations:               6331
Time (CPU seconds):             1.01
Time (Wallclock seconds):       1.01

Total time (CPU seconds):       1.03   (Wallclock seconds):       1.03

We took 1.0360000133514404 seconds to finish.
The optimization resulted in an overall imbalance of: 0
Total distance of: 189
We start our tour with load 0 = 6
At step 1 we go to station 4, we drop 2
At step 2 we go to station 2, we drop 4
At step 3 we go to station 1, we drop 0
At step 4 we go to station 3, we drop -3
At step 5 we go to station 5, we drop 0
The problem was solved to the optimum
```

Figure I.1 – Execution example using mini 6 instance

```
Result - Optimal solution found

Objective value:                1539.00000000
Enumerated nodes:               12
Total iterations:               4896
Time (CPU seconds):             1.21
Time (Wallclock seconds):       1.21

Total time (CPU seconds):       1.23   (Wallclock seconds):       1.22

We took 1.2330000400543213 seconds to finish.
The optimization resulted in an overall imbalance of: 3
Total distance of: 249
We start our tour with load 0 = 2
At step 1 we go to station 4, we drop 2
At step 2 we go to station 5, we drop 0
At step 3 we go to station 3, we drop -2
At step 4 we go to station 1, we drop 0
At step 5 we go to station 2, we drop 2
The problem was solved to the optimum
```

Figure I.2 – Execution example using mini 2 instance

```
Result - Optimal solution found

Objective value:                249.00000000
Enumerated nodes:               26
Total iterations:               6803
Time (CPU seconds):             1.27
Time (Wallclock seconds):       1.27

Total time (CPU seconds):       1.30   (Wallclock seconds):       1.29

We took 1.3020000457763672 seconds to finish.
The optimization resulted in an overall imbalance of: 0
Total distance of: 249
We start our tour with load 0 = 4
At step 1 we go to station 2, we drop 4
At step 2 we go to station 1, we drop 0
At step 3 we go to station 3, we drop -3
At step 4 we go to station 5, we drop 0
At step 5 we go to station 4, we drop 2
The problem was solved to the optimum
```

Figure I.3 – Execution example using mini 5 instance

4 Results obtained

The following tables show the results obtained for each instance.

Instance			Exact solution					Output file
Instance			Exact solution					Output file
Name	n	k	Optimum found within less than 5 min (yes/no)	CPU time (s) for the optimal solution	#nodes of B&B	Imbalance	Distance	
Mini_6	5	6	yes	0.99	23	0	189	Mini_6(cbc).sol
Mini_5	5	5	yes	1.21	26	0	249	Mini_5(cbc).sol
Mini_2	5	2	yes	1.12	12	3	249	Mini_2(cbc).sol
tsdp_1_s10_k6	10	6	yes	151.24	29168	1	447	tsdp_1_s10_k6(cbc).sol
tsdp_2_s11_k11	11	11	no	-	-	-	-	
tsdp_2_s12_k11	12	11	no	-	-	-	-	
tsdp_2_s15_k11	15	11	no	-	-	-	-	
tsdp_2_s20_k11	20	11	no	-	-	-	-	
tsdp_3_s20_k11	20	11	no	-	-	-	-	
tsdp_4_s50_k10	50	10	no	-	-	-	-	
tsdp_5_s100_k10	100	10	no	-	-	-	-	
tsdp_6_s200_k10	200	10	no	-	-	-	-	
tsdp_7_s200_k14	200	14	no	-	-	-	-	
tsdp_8_s200_k12	200	12	no	-	-	-	-	
tsdp_9_s500_k14	500	14	no	-	-	-	-	

Figure I.4 – Results cbc

While our model and implementation using Julia and Cbc are effective in solving small instances of the self-service bicycle fleet management problem, we have found that as the number of stations increases, the program struggles to find optimal solutions in a reasonable amount of time. In testing, we found that for instances with more than 10 stations, the program took more than 5 minutes to solve. This indicates that for larger instances of the problem, alternative solution methods may be necessary to achieve acceptable runtimes.

In the following chapter, we will discuss the implementation of tow heuristic methods that

can be used to quickly find good, though potentially suboptimal, solutions for larger instances of the problem.

CHAPTER II

RESOLUTION OF THE PROBLEM WITH HEURISTICS

Introduction

In this chapter, we will be discussing the use of heuristic and metaheuristic methods to solve larger instances of the problem. We will first introduce a self invented heuristic method which we named (Heuritic_RSS.jl (Random Swap Station)), which aims to quickly find good solutions to the problem by using randomness and iterations to make informed decisions based on the current state of the stations and the traveled distance,we will then describe the use of a metaheuristic using simulated annealing algorithm, which we named (Heuritic_SA.jl (simulated annealing)) to further improve the solutions found by the heuristic especially with large number of stations in which the Heuritic_RSS struggle to find good solutions. To evaluate the performance of these heuristic and metaheuristic methods, we will present the results of their execution on various test instances. In these results, we will also use a modified version of the program implemented in Chapter 1 and we named it (LBBF_Relaxed.jl), which determines the lower bound of the global imbalance by relaxing the problem, to provide a reference point for the quality of the solutions found by the heuristics. By comparing the global imbalance of the heuristic solutions to the lower bound, we will be able to assess the effectiveness of our heuristic and metaheuristic approaches. It's important to keep in mind the difficulty of the problem and the context in which the heuristic will be used. In some cases, a "good enough" solution that is produced quickly may be more valuable than a slower, more accurate solution that is unlikely to be found.

1 RSS Heuristic: Random Station Swap Explanation

The main function kailxyv2 attempts to find a good solution to the problem through iterations by exploring different potential solutions and try to find the one that best minimizes the cost function. The function starts by initializing the current state of the system, calculates the cost of this state, and initializes the "load" and "drop" vectors, which keep track of the number of bikes that are loaded onto and dropped off from the trailer at each station, then it iteratively searches for a tour that minimizes the global imbalance and total distance traveled.

It first generates a random permutation of the stations as the initial tour, and then repeatedly generates new tours by randomly shuffling the order of the stations in the current tour then randomly swapping two stations in that tour (this is done to make the search space even bigger and thus probably escaping local minima). For each new tour, it calculates the cost which is the objective function of our problem (with a weighting coefficient strictly superior to 1 applied to the imbalance) and if the new cost is better than the best cost (initialized before the loop), it updates the best tour, best cost, best drop and best load. This process continues for a fixed number of iterations (num_iter). This main function uses other secondary function through calls so here the explanation of the other functions:

- The **perform_tour function** function is responsible for performing a tour with the trailer for a given set of stations, and loading and unloading bikes at each station as needed in order to bring the station's number of bikes closer to the ideal number. We start by randomly selecting the $load_0$ of the trailer to further increase the randomness thus increasing the search space of the problem, then we iterate through each station in the tour input argument and calculating the imbalance at that station (the difference between the number of bikes at the station and the ideal number of bikes). If the station has too few bikes, bikes are loaded onto the trailer from the station until the station's imbalance is reduced to 0 or the trailer is full. If the station has too many bikes, bikes are unloaded from the trailer until the station's imbalance is reduced to 0 or the trailer is empty. The function returns the final state of the stations (nbp_cur) and the load on the trailer at each station (load) and the number of bikes unloaded at each station (drop).

- The **distances** function calculates the distances between each pair of stations, as well as the distance between each station and the warehouse (where the trailer starts and ends its tour). It does this by using the Euclidean distance formula, which calculates the distance between two points based on their x and y coordinates.

- The **calc_global_imbalance function** calculates the global imbalance, which is the sum of the absolute differences between the actual number of bikes at each station and the ideal number of bikes.

- The **random_swap** function performs a random permutation of the order of the stations to visit during the tour.

- Finally, the **tour_distance** function calculates the total distance of a given tour by summing the distances between each pair of consecutive stations in the tour, as well as the distances from the warehouse to the first and last stations in the tour.

1.1 Example of RSS Heuristcs running

The image below shows an example of RSS execution .

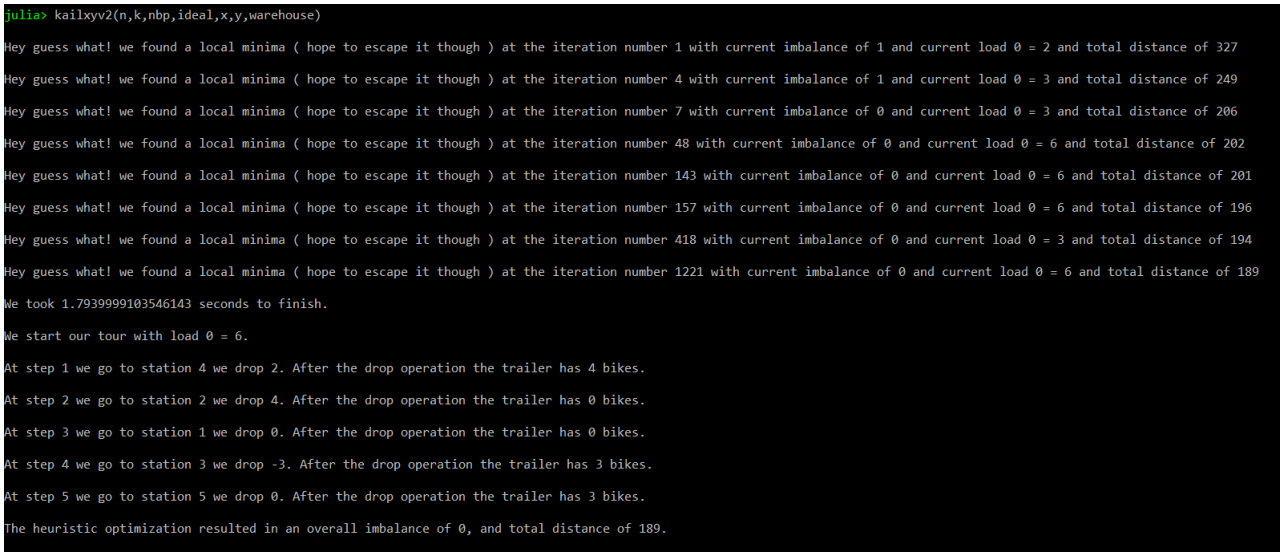

Figure II.1 – RSS execution results

1.2 Results Obtained

The following table showing us the results after the RSS execution

Instance			Exact solution					Heuristic_RSS				Gap (heu -LB)/heu	Lower bound of the imbalance(LB)	Output file
Name	n	k	Optimum found within less than 5 min (yes/no)	CPU time (s) for the optimal solution	#nodes of B&B	Imbalance	Distance	Imbalance	Distance	CPU time (s)	Iterations number			
Mini_6	5	6	yes	0.99	23	0	189	0	189	1.83	1000000	0	0	Mini_6_RSS.sol Mini_6(cbc).sol
Mini_5	5	5	yes	1.21	26	0	249	0	249	1.83	1000000	0	0	Mini_5_RSS.sol Mini_5(cbc).sol
Mini_2	5	2	yes	1.12	12	3	249	3	249	1.79	1000000	66.67%	1	Mini_2_RSS.sol Mini_2(cbc).sol
tsdp_1_s10_k6	10	6	yes	151.24	29168	1	447	1	447	3.21	1000000	-	0	tsdp_1_s10_k6_RSS.sol tsdp_1_s10_k6(cbc).sol
tsdp_2_s11_k11	11	11	no	-	-	-	-	2	411	3.5	1000000	0	2	tsdp_2_s11_k11_RSS.sol
tsdp_2_s12_k11	12	11	no	-	-	-	-	5	447	3.83	1000000	0	5	tsdp_2_s12_k11_RSS.sol
tsdp_2_s15_k11	15	11	no	-	-	-	-	4	619	4.74	1000000	0	4	tsdp_2_s15_k11_RSS.sol
tsdp_2_s20_k11	20	11	no	-	-	-	-	6	796	6.23	1000000	0	6	tsdp_2_s20_k11_RSS.sol
tsdp_3_s20_k11	20	11	no	-	-	-	-	15	752	6.15	1000000	0	15	tsdp_3_s20_k11_RSS.sol
tsdp_4_s50_k10	50	10	no	-	-	-	-	7	2753	16.94	1000000	14.29%	6	tsdp_4_s50_k10_RSS.sol
tsdp_5_s100_k10	100	10	no	-	-	-	-	28	5210	30.13	1000000	71.43%	8	tsdp_5_s100_k10_RSS.sol
tsdp_6_s200_k10	200	10	no	-	-	-	-	88	10444	56.58	1000000	-	0	tsdp_6_s200_k10_RSS.sol
tsdp_7_s200_k14	200	14	no	-	-	-	-	75	10275	57.34	1000000	90.67%	7	tsdp_7_s200_k14_RSS.sol
tsdp_8_s200_k12	200	12	no	-	-	-	-	106	10773	57.56	1000000	34.91%	69	tsdp_8_s200_k12_RSS.sol
tsdp_9_s500_k14	500	14	no	-	-	-	-	301	26568	139.22	1000000	92.36%	23	tsdp_9_s500_k14_RSS.sol

Figure II.2 – RSS execution results

The results of the heuristic method show that it is able to find good solutions to the problem in significantly less time compared to the ILP solver. For instances with a small number of stations, the gap between the global imbalance of the heuristic solutions and the lower bound is very small, indicating that the heuristic is able to find solutions that are close to optimal. However, as the number of stations increases, we see that the gap between the heuristic solutions and the lower bound also increases. This suggests that the heuristic may struggle to find optimal solutions for larger instances of the problem (superior to 70 stations in our problem). Nonetheless, the runtime advantage of the heuristic method makes it a valuable tool for quickly finding good solutions to the problem.
- In the following section, we will discuss the use of the simulated annealing algorithm as a metaheuristic to further improve the solutions found by the heuristic.

2 Metaheuristic SA(Simulated Annealing)

2.1 Heuristic_SA explanation

In this section we will be discussing the use of the simulated annealing algorithm as a metaheuristic to further improve the solutions found by the heuristic method. The simulated annealing algorithm is a probabilistic technique used for finding an approximate global optimum

of a given function. We will describe the details of the application of the simulated annealing algorithm. We will then present an example of the program running with a test instance and finally present the results of the execution of the metaheuristic on the provided test instances, along with the gap between the solutions found and the lower bound of the relaxed problem. By comparing the results of the heuristic and simulated annealing methods, we will be able to assess the effectiveness of the simulated annealing algorithm compared to the solutions found by the heuristic. The function simulated_annealing uses simulated annealing optimization algorithm that attempts to find the global minimum of the objective function by repeatedly perturbing the current solution and accepting or rejecting the new solution based on a probability function. The function begins by calculating the distances between the stations using the distances function and initializing the temperature T to a high value. It then generates a random permutation of the stations to use as the current tour, and initializes the load, nbp, and drop vectors and the current state to the initial values. It also calculates the imbalance and distance of the current tour. The function then enters a loop in which it repeatedly generates new candidate tours by randomly swapping two stations in the current tour, calculates the imbalance (which is empircally weighted by 10) and distance of the new tour, and accepts the new tour with a probability "p" that depends on the imbalance and distance of the new tour and the current temperature T. In other words, "p" is a function of the difference in imbalance and distance between the current tour and the new candidate tour, and the current temperature T. If the difference in imbalance or distance is large, "p" will be relatively high, meaning that the new candidate tour is more likely to be accepted. If the difference in imbalance or distance is small, "p" will be relatively low, meaning that the new candidate tour is less likely to be accepted. If the temperature T is high, "p" will be relatively high, meaning that the new candidate tour is more likely to be accepted even if the difference in imbalance or distance is small. If the temperature T is low, "p" will be relatively low, meaning that the new candidate tour is less likely to be accepted even if the difference in imbalance or distance is large. The "r" in the simulated_annealing function is a random number between 0 and 1 that is generated at each iteration of the loop. If "r" is less than "p", the new candidate tour is accepted. If "r" is greater than or equal to "p", the new candidate tour is not accepted and the current tour remains unchanged. By using the probability "p" and the random number "r" in this way, the function is able to explore different solutions and escape local minima (suboptimal solutions that are difficult to escape from) while still finding good solutions overall. As the temperature T decreases, the probability "p" becomes smaller, and the algorithm becomes more selective in accepting new candidate tours, eventually converging on a good solution. If the new tour is accepted, it becomes the current tour and the imbalance and distance of the current tour are updated. If the new tour is better than the best tour found so far (either because it has a lower

imbalance or because it has the same imbalance but a shorter distance), it is saved as the best tour. The temperature T is reduced by a small amount at the end of each iteration. The loop continues until the temperature T falls below a certain threshold (1e-3). This function calls the same secondary functions explained above.

2.2 Example of the heuristic_SA running

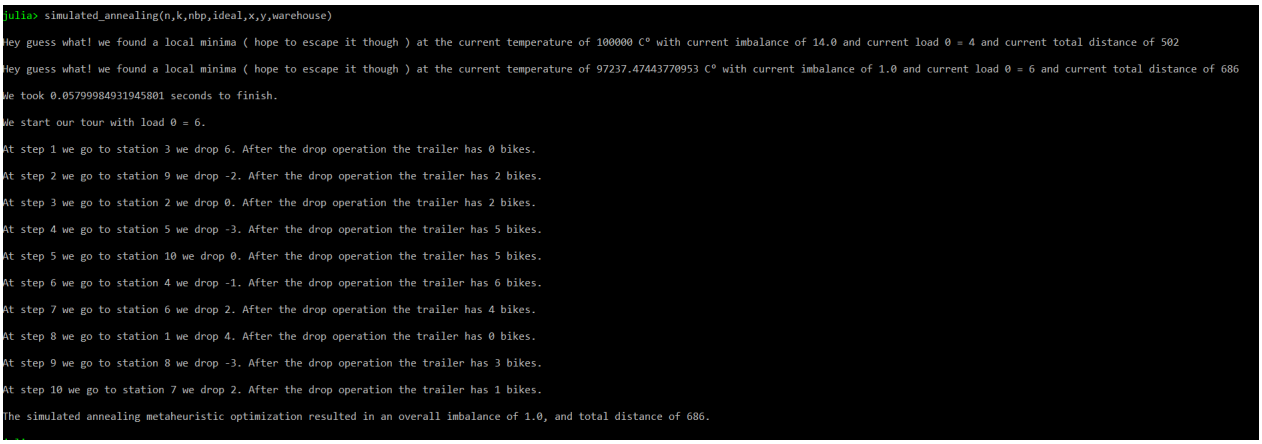

Figure II.3 – SA execution example results

2.3 Results obtained

Instance			Exact solution					Heuristic_(SA)			temperature T/cooling rate	Gap (heu –LB)/heu	Lower bound of the imbalance(LB)	Output file
Name	n	k	Optimum found within less than 5 min (yes/no)	CPU time (s) for the optimal solution	#nodes of B&B	Imbalance	Distance	Imbalance	Distance	CPU time (s)				
Mini_6	5	6	yes	0.99	23	0	189	0	189	0.02	10000/0.999	0 %	0	Mini_6_SA.sol Mini_6(cbc).sol
Mini_5	5	5	yes	1.21	26	0	249	0	249	0.03	10000/0.999	0 %	0	Mini_5_SA.sol Mini_5(cbc).sol
Mini_2	5	2	yes	1.12	12	3	249	3	258	0.035	10000/0.999	66.67%	1	Mini_2_SA.sol Mini_2(cbc).sol
tsdp_1_s10_k6	10	6	yes	151.24	29168	1	447	1	649	0.058	10000/0.999	-	0	tsdp_1_s10_k6_SA.sol tsdp_1_s10_k6(cbc).sol
tsdp_2_s11_k11	11	11	no	-	-	-	-	2	740	0.062	10000/0.999	0%	2	tsdp_2_s11_k11_SA.sol
tsdp_2_s12_k11	12	11	no	-	-	-	-	7	706	0.07	10000/0.999	28.57%	5	tsdp_2_s12_k11_SA.sol
tsdp_2_s15_k11	15	11	no	-	-	-	-	5	821	0.07	10000/0.999	20%	4	tsdp_2_s15_k11_SA.sol
tsdp_2_s20_k11	20	11	no	-	-	-	-	8	1242	0.006	10000/0.999	25%	6	tsdp_2_s20_k11_SA.sol
tsdp_3_s20_k11	20	11	no	-	-	-	-	19	972	0.1	10000/0.999	21.05%	15	tsdp_3_s20_k11_SA.sol
tsdp_4_s50_k10	50	10	no	-	-	-	-	7	1072	0.22	10000/0.999	14.29%	6	tsdp_4_s50_k10_SA.sol
tsdp_5_s100_k10	100	10	no	-	-	-	-	9	2636	0.48	10000/0.999	11.11%	8	tsdp_5_s100_k10_SA.sol
tsdp_6_s200_k10	200	10	no	-	-	-	-	13	5268	0.96	10000/0.999	-	0	tsdp_6_s200_k10_SA.sol
tsdp_7_s200_k14	200	14	no	-	-	-	-	15	5502	0.85	10000/0.999	53.33%	7	tsdp_7_s200_k14_SA.sol
tsdp_8_s200_k12	200	12	no	-	-	-	-	91	3488	1.00	10000/0.999	24.18%	69	tsdp_8_s200_k12_SA.sol
tsdp_9_s500_k14	500	14	no	-	-	-	-	39	15500	2.85	10000/0.999	41.03%	23	tsdp_9_s500_k14_SA.sol

Figure II.4 – SA results

- The results of the simulated annealing algorithm as a metaheuristic show that it is able to significantly improve the solutions found by the heuristic method for larger instances of the problem. For instances with a large number of stations, the gap between the global imbalance of the simulated annealing solutions and the lower bound is very small, indicating that the algorithm is able to find solutions that are close to optimal. In terms of runtime, the simulated annealing algorithm outperforms both the ILP solver and the heuristic method, making it a practical and effective solution method for larger instances of the problem. However, for smaller instances with a lower number of stations, we see that the gap between the simulated annealing solutions and the lower bound increases. This suggests that the algorithm may not be as effective for smaller instances of the problem.

CONCLUSION

In conclusion, our proposed optimization techniques provide a range of effective solution methods for the self-service bicycle fleet management problem, depending on the size and complexity of the instance. For small instances of the problem, the integer linear programming (ILP) model and solution method using Julia and Cbc provide optimal solutions with reasonable runtime. The heuristic_RSS algorithm is a relatively simple heuristic optimization algorithm that relies on randomness to explore the search space. While it was able to give optimal results for a limited number of stations, with larger numbers of stations, the number of local minima increases, making it difficult for the algorithm to escape them and leading to suboptimal solutions.

On the other hand, the heuristic_SA (simulated annealing) algorithm is able to find very close solutions to the lower bound of the problem even with a large number of stations.

This is likely due to the fact that the simulated annealing algorithm may accept worse solutions as the current working solution, with a decreasing likelihood of acceptance as the search progresses. This gives the algorithm the opportunity to first locate the region of the global optima, escape local minima, and then try to locate the optima itself. Overall, the combination of these optimization techniques allows us to effectively solve the self-service bicycle fleet management problem for a wide range of instances.

APPENDIX

3 LBBF code

```
#Author: Wissem Ben Marzouk
using JuMP
using Cbc

function parse_file(filename::String)
    # Open the file and read all lines
    if isfile(filename)
     file = open(filename)
     lines = readlines(file)

     # Initialize variables to store information
     n=0
     k = 0
     x = []
     y = []
     nbp = []
     capa = []
     ideal = []
     warehouse = (0, 0)

     # Parse each line
     for line in lines

        # Check if the line starts with "K"
        if startswith(line, "K")
            # Parse the number after "K" as the capacity of the trailer
            k = parse(Int64, split(line)[2])
        elseif startswith(line, "stations")
        # do nothing and break
        "if" so that this line doesn't get included in "else" condition
        elseif startswith(line, "name")
# do nothing and break "if" so that this line doesn't get included in "else" condition
        elseif startswith(line, "#")
# do nothing and break "if" so that this line doesn't get included in "else" condition
        elseif startswith(line, "warehouse")
```

```
        # Parse the line as the warehouse coordinates
            coords = split(line)[2:3]
            warehouse = (parse(Int64, coords[1]), parse(Int64, coords[2]))
        else

            tab = split(line, " ")
            push!(x, parse(Int64, tab[2]))
            push!(y, parse(Int64, tab[3]))
            push!(nbp, parse(Int64, tab[4]))
            push!(capa, parse(Int64, tab[5]))
            push!(ideal, parse(Int64, tab[6]))

        end
     end
     n=size(x, 1)

     # Close the file
     close(file)

     # Return the parsed information

    end
     return n, k, warehouse, x, y, nbp, capa, ideal
end

function distances(x::Vector{Any}, y::Vector{Any},w::Tuple{Int64, Int64})
    n=size(x, 1)

    d = zeros(Int64, n, n)
    d_war= zeros(Int64, n)
    for i in 1:n
        d_war[i]=round(sqrt((w[1] - x[i])^2 + (w[2] - y[i])^2))
        for j in 1:n
            d[i, j] = round(sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2))
        end
    end
```

```
    return d, d_war
end

function LBBF(n::Int64,k::Int64, nbp::Vector{Any}, capa::Vector{Any},
    ideal::Vector{Any},x::Vector{Any}, y::Vector{Any},w::Tuple{Int64, Int64})

    d ,d_war= distances(x,y,w)
    wight = n * maximum(d)

m = Model(Cbc.Optimizer)

# Varibales definition
    @variable(m, x[i in 1:n, j in 1:n], Bin)
    @variable(m, y[i=1:n, j=1:n-1, k=1:n], Bin)
    @variable(m, load[j in 0:n], Int)
    @variable(m, drop[i in 1:n, j in 1:n], Int)
    @variable(m, imbalance[i in 1:n], Int)

 # Constraints definition
 @constraint(m, c1[i in 1:n], sum(x[i,j] for j in 1:n) == 1)
 @constraint(m, c2[j in 1:n], sum(x[i,j] for i in 1:n) == 1)
 @constraint(m, c3[j in 0:n], load[j] <= k )
 @constraint(m, c4[j in 1:n], load[j] == load[j-1]-sum(drop[i,j] for i in 1:n))
 @constraint(m, c5[i in 1:n,j in 1:n], drop[i,j] <= (capa[i] - nbp[i]) * x[i,j])
 @constraint(m, c6[i in 1:n,j in 1:n], -nbp[i] * x[i,j]  <= drop[i,j])
 @constraint(m, c7[i in 1:n], nbp[i] + sum(drop[i,j] for j in 1:n) - ideal[i]
 <= imbalance[i])
 @constraint(m, c8[i in 1:n], -nbp[i] - sum(drop[i,j] for j in 1:n) + ideal[i]
 <= imbalance[i])
 @constraint(m, c9[i in 1:n, j in 1:n-1, k in 1:n], y[i,j,k] >= x[i,j] + x[k,j+1] - 1)
 @constraint(m, c10[j in 0:n], load[j] >= 0)
 @constraint(m, c11[i in 1:n], imbalance[i] >= 0)
 @constraint(m, c12[i in 1:n, j in 1:n-1, k in 1:n], y[i,j,k] >= 0)
 # Objective function
 @objective(m, Min, wight * sum(imbalance[i]
```

```
    for i in 1:n) + sum(d_war[i] * x[i,1] for i in 1:n) +
    sum(d_war[i] * x[i,n] for i in 1:n) + sum(sum(sum(d[i,k] * y[i,j,k]
    for k in 1:n) for i in 1:n) for j in 1:n-1))

    # Start chronometer
    start = time()

    # Less talking, more doing
set_silent(m)

    # Solve the model
    optimize!(m)

    # Finish chronometer
    finish = time()

    #OBJ is the minimized value
    OBJ = objective_value(m)
sum_imbalance = 0
   for i in 1:n
      sum_imbalance += JuMP.value(imbalance[i])
   end

    sum_imbalance=Int(trunc(sum_imbalance))
    sum_distance= Int(trunc((OBJ - wight * sum_imbalance)))
    load0=JuMP.value(load[0])
    load0=Int(trunc(load0))
    println("We took ",finish-start," seconds to finish.")
    println("The optimization resulted in an overall imbalance of: ", sum_imbalance)
    println("Total distance of: ", sum_distance)
    println("We start our tour with load 0 = ", load0)

for j in 1:n
for i in 1:n
xij = JuMP.value(x[i,j])
if (xij >= 0.99 && xij <= 1.01)
```

```
                dropij=round(JuMP.value(drop[i,j]))
                dropij=Int(trunc(dropij))
                println("At step $j we go to station $i, we drop $dropij")
end
end
end


#println(m)

    # Open the file and write the header
    file = open("mini_2.sol", "w")
    write(file, "name tsdp_2_s12_k11\n")
    write(file, "imbalance $sum_imbalance\n")
    write(file, "distance $sum_distance\n")
    write(file, "init_load $load0\n")

    # Write the station header
    write(file, "stations\n")

    #
    for j in 1:n
for i in 1:n
xij = JuMP.value(x[i,j])
if (xij >= 0.99 && xij <= 1.01)
                dropij=round(JuMP.value(drop[i,j]))
                dropij=Int(trunc(dropij))
                write(file, "$i  $dropij\n")


end
end
end


    # Write the end marker
```

```
    write(file, "End\n")

    # Close the file
    close(file)

# Getting the status of the solution
status = termination_status(m)
isOptimal = status == MOI.OPTIMAL # true if the problem has been optimally solved

if isOptimal println("The problem was solved to the optimum")
else println("The problem wasn't solved to the optimum")
end

end
```

RSS Code

```
#Author: Wissem Ben Marzouk
using Random

function parse_file(filename::String)

    # Open the file and read all lines
    if isfile(filename)
     file = open(filename)
     lines = readlines(file)

     # Initialize variables to store information
     n=0
     k = 0
     x = []
     y = []
     nbp = []
     capa = []
     ideal = []
     warehouse = (0, 0)
```

```
    # Parse each line
    for line in lines

        # Check if the line starts with "K"
        if startswith(line, "K")
            # Parse the number after "K" as the capacity of the trailer
            k = parse(Int64, split(line)[2])
        elseif startswith(line, "stations")
# do nothing and break "if" so that this line doesn't get included in "else" condition
        elseif startswith(line, "name")
# do nothing and break "if" so that this line doesn't get included in "else" condition
        elseif startswith(line, "#")
# do nothing and break "if" so that this line doesn't get included in "else" condition
        elseif startswith(line, "warehouse")
            # Parse the line as the warehouse coordinates
            coords = split(line)[2:3]
            warehouse = (parse(Int64, coords[1]), parse(Int64, coords[2]))
        else

            tab = split(line, " ")
            push!(x, parse(Int64, tab[2]))
            push!(y, parse(Int64, tab[3]))
            push!(nbp, parse(Int64, tab[4]))
            push!(capa, parse(Int64, tab[5]))
            push!(ideal, parse(Int64, tab[6]))

        end
    end
    n=size(x, 1)

    # Close the file
    close(file)

    # Return the parsed information

   end
```

```
     return n, k, warehouse, x, y, nbp, capa, ideal
end

function random_swap(vec::Vector{Int64})
    n = length(vec)
    i, j = rand(1:n), rand(1:n)
    vec[i], vec[j] = vec[j], vec[i]
    return vec
end

function tour_distance(vec::Vector{Int64},d::Array{Int64,2},d_war::Vector{Int64})

    # distances from warehouse to first and last stations
    distance = d_war[vec[1]] + d_war[vec[end]]

    for i in 1:length(vec)-1

        distance += d[vec[i], vec[i+1]]

    end
    return distance
end

function distances(x::Vector{Any}, y::Vector{Any},w::Tuple{Int64, Int64})
    n=size(x, 1)

    d = zeros(Int64, n, n)
    d_war= zeros(Int64, n)
    for i in 1:n
        d_war[i]=round(sqrt((w[1] - x[i])^2 + (w[2] - y[i])^2))
        for j in 1:n
            d[i, j] = round(sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2))
        end
    end

    return d, d_war
```

```
end

# Calculate the global imbalance given the current state of the stations
function calc_global_imbalance(nbp_work::Vector{Any},ideal::Vector{Any})

    #Initialize the total imbalance
    imbalance = 0

    for i in 1:length(nbp_work)

        imbalance += abs(nbp_work[i] - ideal[i])

    end

    return imbalance
end

function perform_tour(tour::Vector{Int64},k::Int64,
nbp_work::Vector{Any},ideal::Vector{Any})

    # Copy the nbp variable to another variable
    # so when we ran the code more than once, we don't
    change the original nbp variable taken from the instance.
    nbp_cur=copy(nbp_work)

    #Initialize variables
    load = zeros(Int, length(nbp_work)+1)
    drop = zeros(Int, length(nbp_work)+1)

    # Randomise the load on the trailer at the warehouse
    load[1] = rand(0:k)


    # Iterate through each station on the tour
    for j in 2:length(nbp_work)+1
```

```
# Calculate the imbalance at the current station
imbalance = nbp_cur[tour[j-1]] - ideal[tour[j-1]]

# If the station is too empty, unload bikes from
the trailer then update load and nbp
if imbalance < 0

    drop[j] = min(-imbalance, load[j-1])

    load[j] = load[j-1] - drop[j]

    nbp_cur[tour[j-1]] += drop[j]

# If the station is too full, load bikes on the trailer
then update load and nbp

elseif imbalance > 0

    drop[j] = min(imbalance, k - load[j-1])

    load[j] = load[j-1] + drop[j]

    nbp_cur[tour[j-1]] -= drop[j]
    drop[j]=-drop[j]

# If the station is balanced, do nothing
else

    drop[j] = 0

    load[j] = load[j-1]

end

# We will not set conditions utilizing capa[i] because ideal[i] <= capa[i]
```

```
    end

    return nbp_cur,drop,load
end

# Heuristic kailxyv2 function
function kailxyv2(n::Int64,k::Int64, nbp_work::Vector{Any},
ideal::Vector{Any},x::Vector{Any}, y::Vector{Any},w::Tuple{Int64, Int64})

   # calculate distances
   d ,d_war= distances(x,y,w)
   # Set the weighting coefficient
   wight = n * maximum(d)

   # Initialize the current state
   curr_state = copy(nbp_work)
   curr_tour = shuffle(1:n)
   curr_distance=tour_distance(curr_tour,d,d_war)
   curr_cost = wight * calc_global_imbalance(curr_state, ideal) + curr_distance

   # Initialize the load and drop vectors
   load = zeros(Int, length(nbp_work)+1)
   drop = zeros(Int, length(nbp_work)+1)

   # Set the iteration number for the kailxyv2 loop
   num_iter = 1000000

   # Initialize the best state ,cost, load and drop
   best_state = copy(curr_state)
   best_cost = curr_cost
   best_drop=copy(drop)
   best_load=copy(load)
   best_tour=curr_tour
   # Start chronometer
   start = time()
   # Perform for a number of iterations
```

```
for i in 1:num_iter

    # Only for the first ietration fill up next_tour and next_distance
    variables and get them ready for the search (I was lazy to initialize them)
    if i<2
     next_tour=random_swap(curr_tour)
     next_distance=tour_distance(next_tour,d,d_war)
     # Perform a random tour and update the current state
    else
     curr_tour = shuffle(1:n)
     next_tour=random_swap(curr_tour)
     next_distance=tour_distance(next_tour,d,d_war)
    end
    next_sate,next_drop,next_load=perform_tour(next_tour,k,curr_state,ideal)

    #get the next cost
    next_cost = wight * calc_global_imbalance(next_sate, ideal) + next_distance

        # If the new state is the best so far, update the best state and cost
         if  next_cost < best_cost
            best_state = copy(next_sate)
            best_cost = next_cost
            best_drop=copy(next_drop)
            best_load=copy(next_load)
            best_imba=calc_global_imbalance(best_state, ideal)
            d_best=best_cost-wight*best_imba
            println(" ")
            println("Hey guess what! we found a local minima ( hope to
            escape it though ) at the iteration number ",i," with current
            imbalance of ",best_imba
            ," and current load 0 = ",best_load[1]," and total distance of ",
            d_best)
            best_tour=next_tour
        end

 end
```

```
        imb_final=calc_global_imbalance(best_state, ideal)
        d_final=best_cost-wight*imb_final

        finish = time()
        println(" ")
        println("We took ",finish-start," seconds to finish.")
        println(" ")
        println("We start our tour with load 0 = ",best_load[1],".")
        println(" ")

        for i in 1:n

         println("At step ",i," we go to station ",best_tour[i]," we drop ",
         best_drop[i+1], ".
         #After the drop operation the trailer has ",best_load[i+1]," bikes.")
         println(" ")
        end
        println("The heuristic optimization resulted in an overall imbalance
        of ",imb_final,", and total distance of ",d_final,".")



    # Open the file and write the header
    file = open("Mini_6_RSS.sol", "w")
    write(file, "name Mini_6\n")
    write(file, "imbalance $imb_final\n")
    write(file, "distance $d_final\n")
    best_loadv=best_load[1]
    write(file, "init_load $best_loadv\n")

    # Write the station header
    write(file, "stations\n")

    j=2
for i in best_tour
```

```
best_dropv=best_drop[j]
        write(file,"$i $best_dropv\n")
        j+=1
end

    # Write the end marker
    write(file, "End\n")

    # Close the file
    close(file)

end
```

5 SA code

```
#Author: Wissem Ben Marzouk

using Random

function parse_file(filename::String)

    # Open the file and read all lines
    if isfile(filename)
     file = open(filename)
     lines = readlines(file)

     # Initialize variables to store information
     n=0
     k = 0
     x = []
     y = []
     nbp = []
     capa = []
     ideal = []
     warehouse = (0, 0)
```

```
    # Parse each line
    for line in lines

       # Check if the line starts with "K"
       if startswith(line, "K")
           # Parse the number after "K" as the capacity of the trailer
           k = parse(Int64, split(line)[2])
       elseif startswith(line, "stations")
# do nothing and break "if" so that this line doesn't get included in "else" condition
       elseif startswith(line, "name")
 # do nothing and break "if" so that this line doesn't get included in "else" condition
       elseif startswith(line, "#")
 # do nothing and break "if" so that this line doesn't get included in "else" condition
       elseif startswith(line, "warehouse")
           # Parse the line as the warehouse coordinates
           coords = split(line)[2:3]
           warehouse = (parse(Int64, coords[1]), parse(Int64, coords[2]))
       else

           tab = split(line, " ")
           push!(x, parse(Int64, tab[2]))
           push!(y, parse(Int64, tab[3]))
           push!(nbp, parse(Int64, tab[4]))
           push!(capa, parse(Int64, tab[5]))
           push!(ideal, parse(Int64, tab[6]))

       end
    end
    n=size(x, 1)

    # Close the file
    close(file)

    # Return the parsed information
```

```
    end
     return n, k, warehouse, x, y, nbp, capa, ideal
end


# Calculate the global imbalance given the current state of the stations
function calc_global_imbalance(nbp_work::Vector{Any},ideal::Vector{Any})

    #Initialize the total imbalance
    imbalance = 0

    for i in 1:length(nbp_work)

        imbalance += abs(nbp_work[i] - ideal[i])

    end

    return imbalance
end

function perform_tour(tour::Vector{Int64},k::Int64,
nbp_work::Vector{Any},ideal::Vector{Any})

    #Copy the nbp variable to another variable
    #so when we ran the code more than once, we don't change
    the original nbp variable taken from the instance.

    nbp_cur=copy(nbp_work)

    #Initialize variables
    load = zeros(Int, length(nbp_work)+1)
    drop = zeros(Int, length(nbp_work)+1)

    # Randomise the load on the trailer at the warehouse
    load[1] = rand(0:k)
```

```
# Iterate through each station on the tour
for j in 2:length(nbp_work)+1

    # Calculate the imbalance at the current station
    imbalance = nbp_cur[tour[j-1]] - ideal[tour[j-1]]

    # If the station is too empty,
    unload bikes from the trailer then update
    load and nbp
    if imbalance < 0

        drop[j] = min(-imbalance, load[j-1])

        load[j] = load[j-1] - drop[j]

        nbp_cur[tour[j-1]] += drop[j]

    # If the station is too full, load bikes on the trailer then update
    load and nbp

    elseif imbalance > 0

        drop[j] = min(imbalance, k - load[j-1])

        load[j] = load[j-1] + drop[j]

        nbp_cur[tour[j-1]] -= drop[j]
        drop[j]=-drop[j]

    # If the station is balanced, do nothing
    else

        drop[j] = 0

        load[j] = load[j-1]
```

```
        end

        # We will not set conditions utilizing capa[i] because ideal[i] <= capa[i]

    end

    return nbp_cur,drop,load
end

function distances(x::Vector{Any}, y::Vector{Any},w::Tuple{Int64, Int64})
    n=size(x, 1)

    d = zeros(Int64, n, n)
    d_war= zeros(Int64, n)
    for i in 1:n
        d_war[i]=round(sqrt((w[1] - x[i])^2 + (w[2] - y[i])^2))
        for j in 1:n
            d[i, j] = round(sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2))
        end
    end

    return d, d_war
end
function tour_distance(vec::Vector{Int64},d::Array{Int64,2},d_war::Vector{Int64})

    # distances from warehouse to first and last stations
    distance = d_war[vec[1]] + d_war[vec[end]]

    for i in 1:length(vec)-1

        distance += d[vec[i], vec[i+1]]

    end
    return distance
end
```

```
# main function

function simulated_annealing(n::Int64,k::Int64, nbp_work::Vector{Any}
,ideal::Vector{Any},x::Vector{Any}, y::Vector{Any},w::Tuple{Int64, Int64})

        # calculate distances
        d ,d_war= distances(x,y,w)

        # Initialize the temperature T to a high value
        T = 100000

        # Initialize the current tour to a random permutation of the stations

        tour = shuffle(1:n)

        # Initialize the load, nbp and drop vectors
        load = zeros(Int, length(n)+1)
        drop = zeros(Int, length(n)+1)
        curr_state = copy(nbp_work)

        # Initialize the current tour imbalance to the total imbalance of the tour
        imbalance_curr =10*calc_global_imbalance(nbp_work, ideal)

        # Initialize the current tour distance to the total distance of the tour
        d_curr=tour_distance(tour,d,d_war)

        # Initialize the best tour found
        so far and the corresponding imbalance,
        nbp, drop, load and distance
        best_state = copy(curr_state)
        tour_best = tour
        best_drop=copy(drop)
        best_load=copy(load)
        imbalance_best = imbalance_curr
        d_best = d_curr
```

```
# Start a chronometer
start = time()

# While the temperature T is above a certain threshold
while T > 1e-3

    # Generate a new candidate tour by randomly swapping two stations in tour
    i, j = rand(1:n, 2)
    tour_new = copy(tour)
    tour_new[i], tour_new[j] = tour[j], tour[i]
    new_sate,new_drop,new_load=perform_tour(tour_new,k,curr_state,ideal)

    # Calculate the imbalance and distance of the new candidate tour
    imbalance_new = 10*calc_global_imbalance(new_sate, ideal)
    d_new = tour_distance(tour_new,d,d_war)

    # Calculate the acceptance probability
    p = exp((imbalance_curr - imbalance_new) / T) * exp((d_curr - d_new) / T)

    # Generate a random number between 0 and 1
    r = rand()

    # If r < p, accept the new candidate tour
    if r < p

        tour = tour_new
        imbalance_curr = imbalance_new
        d_curr = d_new

        # Update the best tour, imbalance, and distance if the new
        candidate tour is better
        if imbalance_new < imbalance_best ||(imbalance_new == imbalance_best
        && d_new < d_best)
            best_drop=copy(new_drop)
            best_load=copy(new_load)
```

```
                tour_best = tour_new
                imbalance_best = imbalance_new/10
                d_best = d_new
                println(" ")
                println("Hey guess what! we found a local minima
                ( hope to escape it though ) at the current temperature of ",T,
                " C° with current imbalance of ",imbalance_best
                ," and current load 0 = ",best_load[1]," and current
                total distance of ",d_best)
            end
        end

        # Reduce the temperature T by a small amount
        T *= 0.999
    end

    finish = time()
println(" ")
println("We took ",finish-start," seconds to finish.")
println(" ")
println("We start our tour with load 0 = ",best_load[1],".")
println(" ")

for i in 1:n

    println("At step ",i," we go to station ",tour_best[i]," we drop "
    ,best_drop[i+1],
    ". After the drop operation the trailer has ",best_load[i+1]," bikes.")
    println(" ")
end


println("The simulated annealing metaheuristic optimization resulted in
an overall imbalance of ",imbalance_best,
", and total distance of ",d_best,".")
```

```
    # Open the file and write the header
    file = open("mini_5_SA.sol", "w")
    write(file, "name mini_5\n")
    write(file, "imbalance $imbalance_best\n")
    write(file, "distance $d_best\n")
    best_loadv=best_load[1]
    write(file, "init_load $best_loadv\n")

    # Write the station header
    write(file, "stations\n")

    j=2
for i in tour_best
best_dropv=best_drop[j]
        write(file,"$i $best_dropv\n")
        j+=1
end

    # Write the end marker
    write(file, "End\n")

    # Close the file
    close(file)

end
```

6 LBBF Relaxed Code

```
#Author: Wissem Ben Marzouk


using JuMP
using Cbc

function parse_file(filename::String)
```

```
    # Open the file and read all lines
    if isfile(filename)
     file = open(filename)
     lines = readlines(file)

     # Initialize variables to store information
     n=0
     k = 0
     x = []
     y = []
     nbp = []
     capa = []
     ideal = []
     warehouse = (0, 0)

     # Parse each line
     for line in lines

        # Check if the line starts with "K"
        if startswith(line, "K")
            # Parse the number after "K" as the capacity of the trailer
            k = parse(Int64, split(line)[2])
        elseif startswith(line, "stations")
# do nothing and break "if" so that this line doesn't get included in "else" condition
        elseif startswith(line, "name")
# do nothing and break "if" so that this line doesn't get included in "else" condition
        elseif startswith(line, "#")
# do nothing and break "if" so that this line doesn't get included in "else" condition
        elseif startswith(line, "warehouse")
            # Parse the line as the warehouse coordinates
            coords = split(line)[2:3]
            warehouse = (parse(Int64, coords[1]), parse(Int64, coords[2]))
        else

            tab = split(line, " ")
            push!(x, parse(Int64, tab[2]))
```

```
            push!(y, parse(Int64, tab[3]))
            push!(nbp, parse(Int64, tab[4]))
            push!(capa, parse(Int64, tab[5]))
            push!(ideal, parse(Int64, tab[6]))

        end
     end
     n=size(x, 1)

     # Close the file
     close(file)

     # Return the parsed information

    end
     return n, k, warehouse, x, y, nbp, capa, ideal
end


function LBBF_Relaxed(n::Int64,k::Int64, nbp::Vector{Any}, capa::Vector{Any},
    ideal::Vector{Any})



m = Model(Cbc.Optimizer)

#varibales definition

    @variable(m, x[i in 1:n, j in 1:n],Bin)
    @variable(m, load[j in 0:n], Int)
    @variable(m, drop[i in 1:n, j in 1:n],Int)
    @variable(m, imbalance[i in 1:n], Int)

    #constraints definition
    @constraint(m, c1[i in 1:n], sum(x[i,j] for j in 1:n) == 1)
    @constraint(m, c2[j in 1:n], sum(x[i,j] for i in 1:n) == 1)
```

```
    @constraint(m, c3[j in 0:n], load[j] <= k )
    @constraint(m, c4[j in 1:n], load[j] == load[j-1]-sum(drop[i,j] for i in 1:n))
    @constraint(m, c5[i in 1:n,j in 1:n], drop[i,j] <= (capa[i] - nbp[i]) * x[i,j])
    @constraint(m, c6[i in 1:n,j in 1:n], -nbp[i] * x[i,j]  <= drop[i,j])
    @constraint(m, c7[i in 1:n], nbp[i]
    + sum(drop[i,j] for j in 1:n) - ideal[i] <= imbalance[i])
    @constraint(m, c8[i in 1:n], -nbp[i]
    - sum(drop[i,j] for j in 1:n) + ideal[i] <= imbalance[i])
    @constraint(m, c10[j in 0:n], load[j] >= 0)
    @constraint(m, c11[i in 1:n], imbalance[i] >= 0)


    #objective function
    @objective(m, Min, sum(imbalance[i] for i in 1:n))

    #start chronometer
    start = time()

    #less talking, more doing
set_silent(m)

    #relax integrality
    relax_integrality(m);



    #solve the model
    optimize!(m)

    #finish chronometer
    finish = time()

    #lower bound for imbalance is the minimized value

    lower_bound_imbalance = round(objective_value(m))
```

```
    println("We took ", finish-start, " seconds to finish.")
    println("The lower bound for imbalance : ", lower_bound_imbalance)
    # Getting the status of the solution
status = termination_status(m)
isOptimal = status == MOI.OPTIMAL # true if the problem has been optimally solved

if isOptimal println("The problem was solved to the optimum.")
else println("The problem wasn't solved to the optimum.")
end

end
```