Free-Floating Bike Sharing: Solving Real-life Large-scale Static Rebalancing Problems

Technical Report · October 2015

DOI: 10.13140/RG.2.1.1727.1766

2 authors, including:

Aritra Pal
University of South Florida
2 PUBLICATIONS **1** CITATION

SEE PROFILE

Free-Floating Bike Sharing: Solving Real-life Large-scale Static Rebalancing Problems

Aritra Pal[a], Yu Zhang[b]

[a] *Department of Industrial and Management Systems Engineering, University of South Florida, 4202 E. Fowler Avenue, Tampa, FL 33620, USA*

[b] *Department of Civil and Environmental Engineering, University of South Florida, 4202 E. Fowler Avenue, Tampa, FL 33620, USA*

Abstract

Free-floating bike sharing (FFBS) is an innovative bike sharing model. FFBS saves on start-up cost, in comparison to station-based bike sharing (SBBS), by avoiding construction of expensive docking stations and kiosk machines. FFBS prevents bike theft and offers significant opportunities for smart management by tracking bikes in real-time with built-in GPS. However, like SBBS, the success of FFBS depends on the efficiency of its rebalancing operations. Rebalancing refers to the reestablishment of the number of bikes at sites to desired quantities. Static rebalancing for SBBS is a challenging combinatorial optimization problem. FFBS takes it a step further, with an increase in the scale of the problem. This article is the first effort in a series of studies of FFBS planning and management, tackling static rebalancing with one vehicle. We present a hybrid nested large neighborhood search with variable neighborhood descent algorithm, which is both effective and efficient in solving static rebalancing problems for both FFBS and SBBS. Computational experiments were carried out on the 1-PDTSP instances used previously in the literature and on a new set of instances based on the Share-A-Bull FFBS (SABB) program recently launched at University of South Florida on its Tampa campus. Computational experiments on the 1-PDTSP instances demonstrate that the proposed algorithm outperforms a tabu search algorithm and is highly competitive with exact algorithms previously reported in the literature for solving static rebalancing problems in SBSS. It is able to find new solutions for 58 of 148 instances for which the optimal solution is not known and, on average, is 340 times faster than the exact algorithm that allows preemption and 580 times faster than the exact algorithm that does not allow preemption. Computational experiments on the SABB instances, consisting of up to 400 nodes and 300 bikes, demonstrate that the proposed algorithm is able to deal with the increase of scale of the static rebalancing problem pertaining to FFBS while deriving high-quality solutions in a reasonable amount of CPU time.

Keywords: Free-floating Bike Sharing, Pickup and Delivery, Granular Neighborhoods, Variable Neighborhood Descent, Large Neighborhood Search

1. Introduction

Bike sharing allows people a healthy, enjoyable and emission-free way to commute across small distances free from the worries of ownership. It also provides an alternative and attractive solution for the first- and last-mile problem in multimodal transportation. Over the years, various schemes of bike sharing have been presented, with the earliest generation dating back to July 28, 1965, in Amsterdam with Witte Fietsen (White Bikes). The next generation, coin-deposit systems, first were introduced in Denmark in Farse and Grena in 1991 and in Nakskov in 1993. A major breakthrough came when people could use a magnetic stripe card to rent a bike. This generation of bike sharing, known as the IT-based system started with Bikeabout in

1996 at Portsmouth University, in England. Interested readers are referred to [1], for an overview of various generations of bike sharing.

Free-floating bike sharing is relatively new. In this model of bike sharing, bikes can be locked to an ordinary bicycle rack (or any solid frame or standalone), thus eliminating the need for specific stations. In comparison to the prevailing SBBS, FFBS saves on start-up cost by avoiding the construction of expensive docking stations and kiosk machines. By tracking bikes in real-time with built-in GPS, FFBS provides online booking and payment, prevents bike theft, and offers significant opportunities for smart management. With FFBS, customer satisfaction levels increase because obtaining and returning the bikes becomes much more convenient compared to SBBS; the average walking distance of FFBS is shorter and customers have no worry about the shortage of a vacant spot at stations.

SocialBicycles (SoBi) is a provider of bikes for FFBS. We use SoBi bikes as an example to further illustrate how FFBS works. Each registered SoBi member gets a unique PIN and can use an app to locate available bikes. After reserving a bike, the user has 15 minutes to walk to its location. At the bike location, he/she enters the PIN on the bike's built-in keypad to unlock the bike. If he/she wants to stop somewhere quickly, the bike can be locked and placed on hold. Upon reaching the destination, the user can simply lock the bike to a bicycle rack (or any solid frame or standalone) and the bike becomes available for the next user.

During daily operation, the distribution of bikes in the system becomes skewed, often leading to a low quality of service and user satisfaction. To prevent such a scenario from prevailing, operators move bikes across the network to achieve a desired distribution. The operation of redistributing bikes across the network using a fleet of vehicle(s) is known as bike rebalancing. Rebalancing at night, when user intervention is negligible, is called static rebalancing. For a large system, bike rebalancing is a challenging combinatorial optimization problem, and the objective function is usually to minimize the financial and environmental costs of rebalancing. Different variants of the bike rebalancing problem have been proposed in the literature, see Section 2 for a detailed literature review.

The static rebalancing problem, is extremely challenging, because the number of times a station (or bicycle rack in the case of FFBS) is visited, in the optimal solution can not be determined apriori. For the same configuration, i.e., locations of stations (or bike racks in FBSS), # bikes and vehicle capacity, the complexity of the static rebalancing problem is higher for FFBS than for SBBS. Consider a bike sharing system with 100 stations (or bike racks) and 200 bikes. For SBBS, $Final\ \#\ Stations \leq Initial\ \#\ Stations \leq 100$, as bikes cannot be parked outside of stations. Whereas for FFBS, bikes can be parked anywhere in the network, either locked with bike racks or standalone. Thus, $Final\ \#\ Nodes \leq min\{Initial\ \#\ Bike\,Racks + \#\ Bikes, 2\times\ \#\ Bikes\} = min\{100 + 200, 2 \times 200\} \leq 300$.

In this study, we focused on the variant of the static rebalancing problem, as studied in [2] and [3], for SBBS. The reason for studying this variant of the static rebalancing problem over others, for FFBS, is elaborated in Section 3. [2] presents a Tabu Search algorithm and [3] presents exact algorithms, for solving the static rebalancing problem, for SBBS. However, none of these algorithms is effective, for solving static rebalancing problems even in small to medium scale FFBS, see Section 5.3 for details. Thus, we developed an algorithm, to derive high quality solutions of static rebalancing problems in FFBS, in a reasonable amount of CPU time.

The solution algorithm consists of creating an initial solution using a greedy construction heuristic and improving it until no further improvement is possible. The improvement heuristic is a hybridization of variable neighborhood descent with large neighborhood search. Four granular neighborhoods ([4]) are used for variable neighborhood descent. Four perturbation and three repairing heuristics were developed, resulting in a total of 12 large neighborhoods. Each of these is explored exhaustively before moving on to the next large neighborhood until no further improvement is possible, resulting in a nested large neighborhood search. For more details on the methodology see Section 4.

Computatonal experiments on the 1-PDTSP instances from the literature, demonstrate that the presented algorithm outperforms the tabu search algorithm and is highly competitive with the exact algorithms mentioned above, for solving static rebalancing problems in SBBS. It is able to find new solutions for 58

of 148 instances for which the optimal solution is not known and, on average, is 340 times faster than the exact algorithm that allows preemption and 580 times faster than the exact algorithm that does not allow preemption. Computational experiments on the new SABB instances, consisting of up to 400 nodes and 300 bikes, demonstrate that the presented algorithm is able to deal with, the increase in scale of the static rebalancing problem in FFBS. It also shows that the static rebalancing problem using a single vehicle and having complete rebalancing as a hard constraint is feasible for # Nodes less than or equal to 150 for the Share-A-Bull FFBS program at USF, Tampa.

The remainder of the paper is organized as follows. Section 2 presents the literature review of research conducted on Bike Sharing. Section 3 describes the static rebalancing problem and introduces the notations used throughout rest of the paper. Section 4 describes the methodology used for deriving high quality solutions for the static rebalancing problem. Section 5 summarizes the experimental results of the two case studies. Section 6 concludes the paper with directions for future research.

2. Literature Review

In recent years, with the boom of SBBS programs, extensive bike-share related research has been conducted and documented. Related to the strategic design and operational management of a bike-sharing system, the literature can be grouped into four research substreams: Strategic Design, Demand Analysis, Service-Level Analysis, and Rebalancing Operations. Strategic Design, Demand Analysis, and Service-Level Analysis are beyond the scope of this study; thus, we focus only on the literature relevant to rebalancing operations.

Rebalancing in a bike sharing system can be achieved in two ways: deploying a fleet of rebalancing trucks to manually rebalance the bikes and incentivizing users to encourage them to self-rebalance the bikes. The literature can be classified as studying either of these two or a combination of the two. Further, manual rebalancing can be classified into two subcategories: static rebalancing and dynamic rebalancing. If rebalancing is done when user intervention is negligible, it is known as static rebalancing, whereas if rebalancing is done when there is significant user intervention, the rebalancing is known as dynamic rebalancing.

Various objective functions have been proposed in the literature for static rebalancing:

1. Minimizing the tour length for a fleet of one vehicle: [2], [3]
2. Minimizing the make-span for a fleet of more than one vehicle: [5]
3. Minimizing the weighted sum of the deviation from a desired quantity of bikes in each station, the amount of pickup and drop-off, and the total travel time: [6]
4. Minimizing the sum of travel and operational costs: [7]
5. Minimizing the weighted sum of the routing cost and the customer dissatisfaction: [8], [9], [10]

[2] derives lowerbounds of the static rebalancing problem, by solving a MILP relaxation and upperbounds using Tabu Search. An auxiliary algorithm based on a maximum flow computation was used to derive loading instructions for each visit. Computational results with stations up to 100 are reported. High-quality solutions for instances with stations less than 50 were found in a reasonable amount of CPU time, but not for instances with stations greater than 50. [3] extended the work of [2], developing exact algorithms for both preemptive and non-preemptive scenarios and report computational results on instances with stations less than or equal to 60. The exact algorithm that allows preemption outperformed the exact algorithm that does not allow preemption in all respects.

[5] decomposed the static rebalancing problem with multiple rebalancing vehicles into separate single-vehicle routing problems by solving a polynomial-size clustering problem. The routing problems were handled by a clustered MIP heuristic or a constraint programming approach. [6] solved the static rebalancing problem with a time constraint using a combination of Greedy Heuristics, GRASP and VNS. [7] extended the static rebalancing problem with a single rebalancing vehicle by allowing the target inventory at a station to lie

within a given interval rather than a specific number. They report two exact algorithms, one based on traditional Branch-and-Cut and the other based on Benders decomposition and report computational results on instances up to 50 stations. [11] solved the static rebalancing problem with a fleet of both single and multiple vehicles but, in their case, stations must be visited exactly once.

[8] studied the static rebalancing problem with a fleet of both single and multiple vehicles. Routing cost and customer dissatisfaction were minimized. However, in their formulation, stations can be visited at most once, which severely limits the quality of the solutions. The time frame for rebalancing is also predetermined. They report two mathematical formulations and dominance rules and computational results on instances up to 60 stations and a fleet of 1 and 2 vehicles. [9] roposed a three-step matheuristic for the problem proposed in [8] based on clustering stations due to their geographic locations and inventory status, followed by routing the rebalancing vehicles through the clusters where each station can be visited only once. Finally, the original problem was solved with certain restrictions as an MILP by a commercial solver. They report computational results on instances of up to 200 stations and 3 repositioning vehicles and show that it outperforms the method suggested in [8]. [10] reports an iterated tabu search algorithm for solving the above variant of the static rebalancing problem with a single vehicle.

[12] studied a dynamic scenario in which borrowing and returning of bikes by users during rebalancing were taken into account. A hybrid MIP approach using Dantzig-Wolfe and Benders decomposition was applied to solve the problem. Although the upper and lower bounds of the MIP can be derived relatively quickly for instances up to 100 stations, the solution algorithm showed significant gaps that makes it ineffective for solving dynamic rebalancing problems.

[13] demonstrated, that a simple rental discount, to encourage users to return bikes to the least-loaded station between two nearby stations, can improve the bike sharing system by an exponential factor. They also calculated the ratio of bikes that have to be redistributed by trucks if a certain quality of service needs to be ensured. [14] proposed sophisticated pricing schemes to encourage self-rebalancing to reduce the need for rebalancing vehicles and labor hours, for improving the operational performance of a bike-sharing system.

[15] demonstrated that a simple rental discount, to encourage users to return bikes to the least-loaded station between two nearby stations can improve the bike sharing system by an exponential factor. They also calculated the ratio of bikes that are redistributed by trucks if a certain quality of service needs to be ensured. [16] proposed a heuristic for solving the dynamic rebalancing problem with multiple vehicles and tested it for simulated cases. In addition, they presented a dynamic pricing strategy which encourage users to return bikes to empty stations.

From the literature review, it is evident that no researcher has reported any study conducted on FFBS. However, the static rebalancing problem has a steep increase in its complexity for FFBS, owing to the inherent large-scale nature of the problem. In the existing literature, to deal with the scale of the static rebalancing problem, researchers have sacrificed solution quality by limiting the number of visits to a node (or station) to, at most, once. If multiple visits to a node (or station) are allowed, the solution algorithms are unable to cope with the scale of the problem and become ineffective for nodes (or stations) greater than 50. We bridge this gap by reporting a solution algorithm that addresses both of these issues, i.e., retaining solution quality with increase in the scale of the static rebalancing problem while allowing multiple visits to a node (or station). Thus, we are the first to report any research conducted for FFBS and to effectively deal with the increase in scale of the static rebalancing problem for both SBBS and FFBS.

3. Problem Description

3.1. Notations

Most of the notations used in this study are similar to [3], however, there are som differences. The notations used in the remainder of the paper are as follows:

- N denotes the nodes in the network, including the depot. The depot is denoted by 1, where the tour of the rebalancing vehicle starts and ends.
- $T_{i,j}$ is the distance between i^{th} and j^{th} nodes ,$\forall i, j \in N$. $T_{i,j}$ may be $\neq T_{j,i}$, for example: if there are one way lanes in the network.

- C_i denotes the capacity of the i^{th} node ,$\forall i \in N$.
- Inv_i and $Target_i$ are the initial and target quantity of bikes at the i^{th} nodes respectively, $\forall i \in N$.
- O_i is the # of operations that needs to be done at the i^{th} node, $\forall i \in N$. $O_i = Inv_i - Target_i, \forall i \in N$. Thus, O_i is $+ve$ when the i^{th} node is a pickup node and $-ve$ when the i^{th} node is a delivery node.
- E denotes the edges in the network.
- $G = (N, E)$ is a complete graph of the network.
- Q is the capacity of a rebalancing vehicle.
- $T^{'}$ and $I^{'}$ are the tour and the instruction set of the rebalancing vehicle.

The variant of static rebalancing studied in this article is similar to the one studied in [2] and [3]. In this variant, complete rebalancing is a hard constraint. This means that, the rebalancing operations terminates, only when target inventory of all the nodes in the network have been met. The necessary condition for complete rebalancing to be feasible for a particular instance, $\sum_{i \in N} O_i = 0$ must be true. Further, no time limit is imposed on the rebalancing operations, as having a time limit might not allow for complete rebalancing. At a first glance, it may seem unrealistic, but this is not true.

Ideally, complete rebalancing must be a hard constraint, as it leads to the highest level of service and minimum customer dissatisfaction. However, the time taken to completely rebalance a bike sharing system with a single vehicle, might be more than the actual time available. For example, for the Share-A-Bull FFBS program (SABB), recently started at USF, the time available for computation and rebalancing in a day, is approximately 8 hours, from 11:00 PM to 7:00 AM, as user intervention is neglible in this time period. From our experimental results (Section 5.3), we can conclude that, if $|N| \leq 150$ complete rebalancing with a single vehicle is feasible for the SABB instances. However, if $|N| \geq 200$ complete rebalancing with a single vehicle may not be feasible for the SABB instances. In this scenario, the operator has two choices. If they want to completely rebalance the system, they need at least $\lceil \frac{Rebalancing\,Time\,with\,1\,Vehicle}{Available\,Time\,for\,Rebalancing} \rceil$ vehicles to do so. If vehicles available are $<< \lceil \frac{Rebalancing\,Time\,with\,1\,Vehicle}{Available\,Time\,for\,Rebalancing} \rceil$, they must resort to partial rebalancing. In partial rebalancing, instead of meeting the target requirements at all nodes, partial requirements are met at certain nodes (or may be all of them), so that the rebalancing time is less than or equal to a desired time limit. Complete rebalancing with multiple vehicles and partial rebalancing with single and multiple vehicles are an on-going effort of our research team and will be addressed in a future article.

In this study, the objective is to minimize the total rebalancing time of the (single) rebalancing vehicle, which is the sum of travel time and loading-unloading time. Loading-unloading time is amount of bike displacements times average loading-unloading time per bike. For a particular instance of the static rebalancing problem, amount of bike displacements and average loading-unloading time per bike are fixed. Hence, if static rebalancing is conducted using a single vehicle, loading-unloading time for that particular instance is a constant. Thus, minimizing the total rebalancing time is equivalent to minimizing the total travel time. If the (average) travel speed is a constant (which is a fair assumption), minimizing the travel time is equivalent to minimizing the travel distance. Thus, our new objective is to minimize the total distance traversed by the rebalancing vehicle while deriving a sequence of nodes starting and ending at the depot and re-establishing the target number of bikes at each node by the end of the tour. Preemption is not allowed in the solution algorithm, as allowing it only increases the computational complexity of the algorithm without improving solution quality. This is proved in [3], where it is shown that, preemption at most adds a value of 0.6% in the solution quality, for the 1-PDTSP instances used in the literature.

3.2. Complete Formulation:

The complete formulation is due to [3]. Additional notations used in the complete formulation (similar to [3]) are as follows:

- $x_{i,j}$ denotes the number of times edge (i,j) is traversed, $\forall (i,j) \in E$.
- If $T_{i,j}$ satishfy the triangle inequality (which is true for both the case studies conducted), maximum number of times edge (i,j) can be traversed is $u_{i,j} = min\{|O_i|, |O_j|\}, if O_i \times O_j < 0$, otherwise $u_{i,j} = 1$ (see [3] for details).
- $x_{i,j} = \sum_{k=0}^{\lfloor log_2(u_{i,j}) \rfloor} 2^k y_{i,j,k}, y_{i,j,k} \in \{0,1\}, \forall (i,j) \in E, k \in \{0, ..., \lfloor log_2(u_{i,j}) \rfloor\}$

Let us consider a subset of nodes, $S \subseteq N \backslash \{1\}$, then

- The net imbalance of S is, $O(S) = \sum_{i \in S} O_i$.
- $\mu(S) = 1$ if there is atleast 1 node $i \in S$, such that $|O_i| > 0$, otherwise $\mu(S) = 0$.
- $r(S) = max\{\frac{\lceil |O(S)| \rceil}{Q}, \mu(S)\}$
- $\delta^+(S) = \sum_{i \in S, j \notin S} x_{i,j}$
- $\delta^-(S) = \sum_{i \notin S, j \in S} x_{i,j}$
- If y^* is an integer solution, $I_0(y^*) = \{(i,j,k) : y^*_{i,j,k} = 0\}$ and $I_1(y^*) = \{(i,j,k) : y^*_{i,j,k} = 1\}$. $I_f(y^*) = 1$ if y^* is a feasible solution for the static rebalancing problem otherwise it equals 0. It is an infeasibility indicator function.

$$minimize \sum_{(i,j) \in E} T_{i,j} \sum_{k=0}^{\lfloor log_2(u_{i,j}) \rfloor} 2^k y_{i,j,k} \tag{1}$$

$$s.t. \sum_{j:(j,i) \in \delta^-(i)} \sum_{k=0}^{\lfloor log_2(u_{j,i}) \rfloor} 2^k y_{j,i,k} = \sum_{j:(i,j) \in \delta^+(i)} \sum_{k=0}^{\lfloor log_2(u_{i,j}) \rfloor} 2^k y_{i,j,k}, \forall i \in N \tag{2}$$

$$\sum_{j:(1,j) \in \delta^+(1)} \sum_{k=0}^{\lfloor log_2(u_{1,j}) \rfloor} 2^k y_{1,j,k} \geq max\{\lceil \frac{|O_1|}{Q} \rceil, 1\} \tag{3}$$

$$\sum_{j:(i,j) \in \delta^+(S)} \sum_{k=0}^{\lfloor log_2(u_{i,j}) \rfloor} 2^k y_{i,j,k} \geq r(S), \forall S \subseteq V \backslash \{1\} \tag{4}$$

$$\sum_{(i,j,k) \in I_0(y^*)} y_{i,j,k} + \sum_{(i,j,k) \in I_1(y^*)} (1 - y_{i,j,k}) \geq 1, \forall y^* : I_f(y^*) = 1 \tag{5}$$

$$y_{i,j,k} \in \{0,1\}, \forall (i,j) \in E, k \in \{0, ..., \lfloor log_2(u_{i,j}) \rfloor\} \tag{6}$$

1 is the objective function of the static rebalancing problem, which minimizes the total traveling distance covered by the rebalancing vehicle. Constraint set 2 are the flow conservation constraints for each node in the network. Constraint 3 determines the minimum number of times the rebalancing vehicle must visit the depot, based on its imbalance, O_1. Constraint set 4 determines the minimum number of times the rebalancing vehicle must leave a subset of nodes S, where $S \subseteq N \backslash \{1\}$. Constraint set 5 separates integer feasible solutions not feasible for the static rebalancing problem but feasible for constraint sets 2 - 4. In [3], the authors report procedures, for separating integer solutions, infeasible for both, the preemptive and the non-preemptive scenario, based on combinatorial bender's cuts.

4. Methodology

There are major differences between our approach and previous approaches reported in the literature ([2]). In the algorithm presented in this paper, both the sequence of stations ($T^{'}$) and the instruction set ($I^{'}$) are computed at the onset of the solution algorithm. Further, the feasibility of the solution, i.e, $T^{'}$ & $I^{'}$ is maintained while constructing the initial solution and during a subsequent local search. This has a significant advantages and results in a decrease of the size of neighborhoods of an incumbent solution to a great extent; hence, it takes considerably less time to explore them. Further, a ranking list based on nearest neighbors is used to prioritize the edges in a tour. In a neighborhood, only moves that result in a tour whose highest edge rank is less than or equal to the highest edge rank of the current tour are allowed. This prevents exploration of unwanted regions in a neighborhood and subsequently reduces exploration time so as to make the neighborhood extremely granular ([4]). In total, four such granular neighborhoods were developed and used successively inside Variable Neighborhood Descent (VND).

With the decrease in the size of the neighborhoods used in VND, finding high-quality solutions becomes extremely challenging. To overcome this, VND is hybridized with Large Neighborhood Search (LNS). Multiple large neighborhoods based on the structure of the problem were developed that explore local optimas that are either clustered together or are present in valleys far away from each other. Further, these large neighborhoods are nested together to increase the effectiveness of the presented algorithm. The whole process is elaborated in great detail in subsequent sections.

For instances, where a non-zero operation has to be performed at the depot, i.e., $O_1 \neq 0$, a pseudo node is created in the network. The modified network has $N+1$ nodes, $N+1$ being the index of the pseudo node created. In the modified network,

$$O^{'}_1 = 0 \tag{7}$$
$$O^{'}_{N+1} = O_1 \tag{8}$$
$$O^{'}_i = O_i, \quad \forall i \in \{2, ..., N\} \tag{9}$$
$$T^{'}_{i,j} = T_{i,j}, \quad \forall i \in \{1, ..., N\}, j \in \{1, ..., N\} \tag{10}$$
$$T^{'}_{i,N+1} = T_{i,1}, \quad \forall i \in \{1, ..., N\} \tag{11}$$
$$T^{'}_{N+1,i} = T_{1,i}, \quad \forall i \in \{1, ..., N\} \tag{12}$$
$$\tag{13}$$

. The solution algorithm will be used to solve the modified network instead of the original network. In the solution of the modified network, all $'N+1'$ in the tour will be replaced by $'1'$ to obtain a solution feasible for the original network.

4.1. Initial Solution

An initial solution is created using a greedy construction heuristic. Unlike [2], both the sequence of the nodes and the loading / unloading instructions are created simultaneously. Algorithm 1 is the pseudocode of the greedy construction heuristic. On lines 3 and 9, in Algorithm 1, the function "Maximum operations for every other node()" computes the maximum operation that can be performed at a node other than the current node, if that node is visited next from the current node. When computing maximum operation, only operations remaining at a node are taken into consideration. The "Nearest Neighbor Function()" in Algorithm 1 can be computed in three different ways. For all three different functions, nodes where a non-zero operation is left, are considered.

1. Nearest Neighbor 1: The nearest neighbor 1 of a node, is the node with the minimum (traveling) cost from the current node ,i.e, $argmin\left(T_{i,j}\right), j \in N/\{i\}, i$ being the current node.
2. Nearest Neighbor 2: The nearest neighbor 2 of a node, is the node that has the maximum value of $\frac{|Max\ Ops_j|}{T_{i,j}}, \forall j \in N/\{i\}, i$ being the current node.
3. Nearest Neighbor 3: The nearest neighbor 3 of a node, is a random node other than the current node and the depot.

Algorithm 1: Greedy Construction Heuristic

Data: $O_i, Q, T_{i,j}, Nearest\ Neighbor\ Function$
Result: $(T^{'}, I^{'})$

$T^{'} \longleftarrow [Depot]$
$I^{'} \longleftarrow [0]$
$Max\ Ops \longleftarrow Maximum\ operations\ for\ every\ other\ node(O_i, Q, sum(I^{'}))$
$Next\ Node \longleftarrow Randomly\ select\ any\ node\ from\ 2$ **to** N
$Add\ Next\ Node\ at\ the\ end\ of\ T^{'}$
$Add\ Max\ Ops_{Next\ Node}\ at\ the\ end\ of\ I^{'}$
$O_{Next\ Node} = O_{Next\ Node} - Max\ Ops_{Next\ Node}$
while $Number\ of\ non\ zero\ elements\ in\ O_i > 0$ **do**
    $Max\ Ops \longleftarrow Maximum\ operations\ for\ every\ other\ node(O_i, Q, sum(I^{'}))$
    $Next\ Node \longleftarrow Nearest\ Neighbor\ Function(O_i, T_{i,j}, Next\ Node, Max\ Ops)$
    $Add\ Next\ Node\ at\ the\ end\ of\ T^{'}$
    $Add\ Max\ Ops_{Next\ Node}\ at\ the\ end\ of\ I^{'}$
    $O_{Next\ Node} = O_{Next\ Node} - Max\ Ops_{Next\ Node}$
end
$Add\ Depot\ at\ the\ end\ of\ T^{'}$
$Add\ 0\ at\ the\ end\ of\ I^{'}$

4.2. Repairing a partial / infeasible solution:

Algorithm 2 is the pseudocode of the repairing mechanism used for repairing a partial or an infeasible solution of the static rebalancing problem. It is also based on the greedy construction heuristic described in Section 4.1. As, three different functions (Nearest Neighbor 1,2 and 3 described in Section 4.1) can be used as the nearest neighbor function (line 18), three different solutions can be constructed from an initial partial solution. This feature comes in handy while repairing an infeasible solution from a perturbation (Section 4.3.3) on a feasible solution.

4.3. Neighborhoods of a solution:

The neighborhoods of a solution used in the presented algorithm can be classified into two categories:

1. Neighborhoods used for descent / local search / intensification (Section 4.3.1)
2. Neighborhoods used for perturbation / diversification (Section 4.3.3)

4.3.1. Neighborhoods used for descent

In the following neighborhoods, when an operation is performed on $T^{'}$, the corresponding operation is also performed on $I^{'}$ and vice versa to keep the solution feasible at all time.

Algorithm 2: Repair Tour

Data: $T^{'}, O_i, Q, T_{i,j}, Nearest\ Neighbor\ Function$
Result: $(T^{'}, I^{'})$

if $T_1^{'} is\ not\ Depot$ **then**
  $Depot\ is\ added\ at\ the\ begining\ of\ T^{'}$
end
$I^{'} \longleftarrow [0]$
$i \longleftarrow 2$
while $i \leq length(T^{'})$ **do**
  $Max\ Ops \longleftarrow Maximum\ operations\ for\ every\ other\ node(O_i, Q, sum(I^{'}))$
  if $Max\ Ops_{T_i^{'}} = 0$ **then**
    $Delete\ station\ at\ location\ i\ of\ T^{'}$
  else
    $Add\ Max\ Ops_{T_i^{'}}\ at\ the\ end\ of\ I^{'}$
    $O_{T_i^{'}} \longleftarrow O_{T_i^{'}} - Max\ Ops_{T_i^{'}}$
    $i \longleftarrow i + 1$
  end
end
while $Number\ of\ non\ zero\ elements\ in\ O_i > 0$ **do**
  $Max\ Ops \longleftarrow Maximum\ operations\ for\ every\ other\ node(O_i, Q, sum(I^{'}))$
  $Next\ Node \longleftarrow Nearest\ Neighbor\ Function(O_i, T_{i,j}, Next\ Node, Max\ Ops)$
  $Add\ Next\ Node\ at\ the\ end\ of\ T^{'}$
  $Add\ Max\ Ops_{Next\ Node}\ at\ the\ end\ of\ I^{'}$
  $O_{Next\ Node} = O_{Next\ Node} - Max\ Ops_{Next\ Node}$
end
$Add\ Depot\ at\ the\ end\ of\ T^{'}$
$Add\ 0\ at\ the\ end\ of\ I^{'}$

4.3.1.1. 3-OPT / Delete-Reinsert:. In a 3-OPT neighborhood, 3 edges are broken and 3 new edges are inserted. In this case, one node is deleted from a location in the tour and inserted in another location in the tour while maintaining feasibility of the solution. At each iteration, starting from the left (or beginning) of a tour, each node (other than the depot) is inspected to determine if it can be deleted and reinserted either ahead or behind its current location in the tour. If a valid location(s) is (are) found that reduces the cost of the current tour, then the (best) move is made. An iteration is completed on reaching the penultimate location in the tour. This procedure is continued until no further improvement is possible.

4.3.1.2. 4-OPT Neighborhoods:. In a 4-OPT neighborhood, 4 edges are broken and 4 new edges are inserted.

4.3.1.2.1. Node Swapping: Two 4-OPT neighborhoods based on Node Swapping have been developed. Let i & j be the positions of two nodes in a tour such that $i \neq j, i < j, i \neq 1$ & $j \neq$ *length of the tour*. If $I_i^{'} = I_j^{'}$ & $T_i^{'} \neq T_j^{'}$, then the 2 nodes at i & j can be swapped to obtain a new feasible tour. If the swapping results in a tour with a lower cost, the swapping is confirmed. At each iteration, each location (i) of a tour is inspected for a possible swap. If a valid swap(s) is(are) found that reduces the current cost of the tour, the (best) swap is made. An iteration is completed on reaching the penultimate location in the tour. This procedure is continued until no further improvement is possible.

Another variation of the above procedure is, let i & j be the positions of two nodes in a tour such that $i \neq j, i < j, i \neq 1$ & $j \neq$ *length of the tour*. If $I_i^{'} \neq I_j^{'}$ & $T_i^{'} \neq T_j^{'}$, and swapping the two nodes at i & j results in a new feasible tour with a lower cost, the swapping is confirmed. Everything else is exactly same as the above procedure.

4.3.1.2.2. Arc Exchange: Let us consider four edges e_i, e_j, e_k, e_l, connecting nodes at location i, j, k, l to nodes at location $i+1, j+1, k+1, l+1$ in a tour respectively, such that $i < j < k < l$. If the flow of bikes (= number of bikes carried by the rebalancing vehicle) on edges e_i & e_k and that on edges e_j & e_l are equal, the tour segment $(i+1) \rightarrow j$ can be swapped with the tour segment $(k+1) \rightarrow l$ without losing feasibility. If such a move results in a tour with a lower cost than the current one, that move is confirmed. At each iteration, a step ($= j - i - 1$) is fixed and all possible values of k & l are checked for possible exchanges. If no improvement is possible for the current value of step, its value is incremented by 1. The value of step is initialized with 1 and can at most be increased to *length of the current tour* $- 5$, otherwise there will be overlapping of the tour segments. On reaching this value of step, the procedure is terminated.

4.3.1.3. Variable-OPT / Adjust Instructions:. In this neighborhood, the number of edges (it is variable) broken is equal to the number of edges inserted. The number of edges broken or inserted, can be represented as $2n$, where $n \in Z^{+}$. This neighborhood comprises of nodes, that are visited multiple times in the current tour. The objective of this neighborhood is to drive the loading-unloading instructions, at viable locations in the current tour, of such nodes towards 0. If this can be achieved for one or more locations, the corresponding locations can be removed from the tour. This is because, the instances (used in this study) are metric, and removing nodes at locations in the current tour, for which the instruction is 0, reduces the cost of the tour without making it infeasible.

Let a node be present at the i^{th} and j^{th}($i < j$) locations in a tour. If $|I_i^{'}| \leq |I_j^{'}|$, $I_i^{'}$ is driven towards 0 by transferring instructions between $I_i^{'}$ & $I_j^{'}$, while maintaining feasibility of the flow of bikes in the tour segment $i \longrightarrow j$ and vice versa. This procedure is executed for all possible combinations of i & j for each node with multiple visits in a tour. This neighborhood also serves a secondary purpose. The instructions at some locations in the current tour are altered, which might create new moves for neighborhoods described earlier.

4.3.2. Variable Neighborhood Descent (VND)

VND is successively exploring the neighborhoods (described in sections 4.3.1.1, 4.3.1.2.1, 4.3.1.2.2 and 4.3.1.3) of an incumbent solution until no further improvement is possible. The order in which the neighborhoods are explored is a crucial factor for the search to be effective and is based on the computational complexity and the execution time of an individual neighborhood. The order used is as follows:

1. Variable-OPT: Adjust Instructions
2. 3-OPT: Delete-Reinsert
3. 4-OPT:
    1. Node Swapping 1
    2. Node Swapping 2
    3. Arc Exchange

Algorithm 3 is the pseudocode for VND, used in the presented algorithm.

Algorithm 3: Variable Neighborhood Descent

Data: $T^{'}, I^{'}, O_i, Q, T_{i,j}$
Result: $(T^{'}, I^{'})$
while $Cost(T^{'})! = Cost(T^{'}_{previous})$ **do**
  for $i \leftarrow 1$ **to** 5 **do**
    $(T^{''}, I^{''}) \longleftarrow Neighborhood_i(T^{'}, I^{'}, O_i, Q, T_{i,j})$
    if $Cost(T^{''}) < Cost(T^{'})$ **then**
      $T^{'} \longleftarrow T^{''}$
      $I^{'} \longleftarrow I^{''}$
    end
  end
end

4.3.3. Neighborhoods used for Perturbation

Perturbations used in the presented algorithm are greatly influenced by Chained Lin-Kernighan used for solving large-scale Traveling Salesman Problems ([17]). However, there is a major difference, in Chained Lin-Kernighan selected edges are destroyed whereas in the presented algorithm selected stations are destroyed.

Locations in a tour are ranked for the purpose of perturbation.

$$Rank\ of\ location\ i = \frac{Rank\ of\ Edge_{i-1,i} + Rank\ of\ Edge_{i,i+1}}{2}, \forall i \in [2, length\ of\ tour - 1]$$

and

$$Average\ Rank\ of\ a\ Tour = \frac{\sum_{i \in [2, length\ of\ tour-1]} Rank\ of\ location\ i}{length\ of\ tour - 2}$$

. Lower rank signifies desirable configuration whereas higher rank signifies undesirable configuration and more likely to be perturbed.

In the upcoming Large Neighborhoods (Sections 4.3.3.1 and 4.3.3.2), two functions are used *Sorted Location Rank List* and *Reverse Sorted Location Rank List*. *Sorted Location Rank List* takes a tour and a rank list as its inputs and computes the rank of all the locations (except the first and the last) in the tour. It then sorts the locations in the tour in a descending order of their respective ranks. The sorted list and the number of locations whose rank is above the average rank of the tour is returned.

Reverse Sorted Location Rank List is similar to *Sorted Location Rank List*, except the locations in the tour are sorted in an ascending order of their respective ranks. The sorted list and the number of locations whose rank is less than or equal to the average rank of the tour is returned.

4.3.3.1. Large Neighborhood 1 and 2 :. Large Neighborhoods 1 and 2 are complements of each other. In Large Neighborhood 1, local optimas in valleys clustered together, are explored systematically by destroying locations in a tour with undesirable configurations, followed by repairing the tour and VND. If cost of the new tour is lower than that of the current tour, the new tour becomes the current tour, or else the amount of perturbation is incremented. It continues until the value of perturbation equals that of maximum perturbation. Similarly, in Large Neighborhood 2, local optimas in valleys far away from each other, are explored systematically by destroying locations in a tour with undesirable configurations, followed by repairing the tour and VND. If cost of the new tour is lower than that of the current tour, the new tour becomes the current tour, or else the amount of perturbation is incremented. It continues until the value of perturbation equals that of maximum perturbation.

With increase in the # Nodes, execution time of VND and the # maximum perturbation increases significantly. Thus, to keep the exploration time of the large neighborhoods reasonable, without hampering the quality of the solutions found, perturbation is limited to only $\frac{15}{\#\,Nodes}\%$ of the maximum perturbation. Further, perturbation is varied between its minimum and maximum values simultaneously, i.e, the value of perturbation =

$$\{1, \#\,Maximum\ Perturbation, 2, \#\,Maximum\ Perturbation - 1, ...\}$$

rather than

$$\{1, 2, ..., \#\,Maximum\ Perturbation - 1, \#\,Maximum\ Perturbation\}$$

. This is based on the observation that, perturbation is most effective while near its extreme values.

Algorithm 4 is the pseudocode for Large Neighborhood 1. The pseudocode for Large Neighborhood 2 is exactly similar to Algorithm 4, except *Sorted Location Rank List* and *Number of Locations above Average Rank* are replaced by *Reverse Sorted Location Rank List* and *Number of Locations below Average Rank* respectively.

4.3.3.2. Large Neighborhood 3 and 4 :. As with Large Neighborhoods 1 and 2, Large Neighborhoods 3 and 4 are also complements of each other. In Large Neighborhood 3, node(s) right of a location in the tour is (are) destroyed, followed by repairing the tour and VND. If cost of the new tour is lower than that of the current tour, the new tour becomes the current tour, or else the amount of perturbation is incremented. It continues until the value of perturbation equals that of maximum perturbation. Similarly, in Large Neighborhood 4, node(s) left of a location in the tour is (are) destroyed, followed by repairing the tour and VND. If cost of the new tour is lower than that of the current tour, the new tour becomes the current tour, or else the amount of perturbation is incremented. It continues until the value of perturbation equals that of maximum perturbation. The locations chosen in these large neighborhoods are locations with rank greater than the average rank of the tour.

Algorithm 5 is the pseudocode for Large Neighborhood 3. Pseudocode for Large Neighborhood 4 is similar to Algorithm 5, except " > " in lines 5 and 23 and *Delete Nodes in* $T^{'}$ *right of (Location to Destroy[Pertubation])* in line 13 is replaced by " < " and *Delete Nodes in* $T^{'}$ *left of (Location to Destroy[Pertubation])* respectively.

4.4. Overall Algorithm

Algorithm 6 is the pseudocode of the complete algorithm. The initial solution is constructed using Algorithm 1 (line 1). Lines 2 to 20 is the pseudocode for the nested large neighborhood search algorithm. $Repairing\ Mechanism_j$ in line 6 in Algorithm 6, denotes that $Nearest\ Neighbor\ j()$ is used in Algorithm 2 for repairing the perturbed solution. $T^{'}_{-12}$ is the 12^{th} previous tour after perturbation.

Algorithm 4: Large Neighborhood 1

Data: $T^{'}, I^{'}, O_i, Q, T_{i,j}, Repairing\ Mechanism$

Result: $(T^{'}, I^{'})$

$Location\ Rank\ List, Number\ of\ Locations\ above\ Average\ Rank = Sorted\ Location\ Rank\ List(T^{'}, Rank\ List)$

$Perturbation \longleftarrow 1$

while $Perturbation <= \dfrac{Number\ of\ Locations\ above\ Average\ Rank \times 15}{Number\ of\ Nodes}$ **do**

  if $Perturbation\ is\ Odd$ **then**

    $Locations\ to\ Destroy \longleftarrow Location\ Rank\ List\left[1 : \dfrac{Perturbation + 1}{2}\right]$

  else

    $Locations\ to\ Destroy \longleftarrow$
    $Location\ Rank\ List\left[1 : Number\ of\ Locations\ above\ Average\ Rank + 1 - \dfrac{Perturbation}{2}\right]$

  end

  $T^{''} \longleftarrow Delete\ Nodes\ at\ locations\ (Locations\ to\ Destroy)\ of\ T^{'}$

  $(T^{''}, I^{''}) \longleftarrow Repair\ Tour(T^{''}, O_i, Q, T_{i,j}, Repairing\ Mechanism)$

  $(T^{''}, I^{''}) \longleftarrow Variable\ Neighborhood\ Descent(T^{''}, I^{''}, O_i, Q, T_{i,j})$

  if $Cost(T^{''}) < Cost(T^{'})$ **then**

    $T^{'} \longleftarrow T^{''}$

    $I^{'} \longleftarrow I^{''}$

    $Location\ Rank\ List, Number\ of\ Locations\ above\ Average\ Rank =$
    $Sorted\ Location\ Rank\ List(T^{'}, Rank\ List)$

    $Perturbation \longleftarrow 1$

  else

    $Perturbation \longleftarrow Perturbation + 1$

  end

end

Algorithm 5: Large Neighborhood 3

Data: $T^{'}, I^{'}, O_i, Q, T_{i,j}, Repairing\ Mechanism$
Result: $(T^{'}, I^{'})$

$Location\ Rank\ List, Number\ of\ Locations\ above\ Average\ Rank = Sorted\ Location\ Rank\ List(T^{'}, Rank\ List)$
$Location\ Rank\ List \longleftarrow Location\ Rank\ List[1 : Number\ of\ Locations\ above\ Average\ Rank]$
$i \longleftarrow 2$
while $i <= length(Location\ Rank\ List)$ **do**
    if $Location\ Rank\ List[i] > Location\ Rank\ List[i-1]$ **then**
        $deleteat(Location\ Rank\ List, i)$
    else
        $i \longleftarrow i + 1$
    end
end
$Perturbation \longleftarrow 1$
while $Perturbation <= length(Location\ Rank\ List)$ **do**
    $T^{''} \longleftarrow Delete\ Nodes\ in\ T^{'}\ right\ of\ (Location\ to\ Destroy[Pertubation])$
    $(T^{''}, I^{''}) \longleftarrow Repair\ Tour(T^{''}, O_i, Q, T_{i,j}, Repairing\ Mechanism)$
    $(T^{''}, I^{''}) \longleftarrow Variable\ Neighborhood\ Descent(T^{''}, I^{''}, O_i, Q, T_{i,j})$
    if $Cost(T^{''}) < Cost(T^{'})$ **then**
        $T^{'} \longleftarrow T^{''}$
        $I^{'} \longleftarrow I^{''}$
        $Location\ Rank\ List, Number\ of\ Locations\ above\ Average\ Rank =$
        $Sorted\ Location\ Rank\ List(T^{'}, Rank\ List)$
        $Location\ Rank\ List \longleftarrow Location\ Rank\ List[1 : Number\ of\ Locations\ above\ Average\ Rank]$
        $i \longleftarrow 2$
        while $i <= length(Location\ Rank\ List)$ **do**
            if $Location\ Rank\ List[i] > Location\ Rank\ List[i-1]$ **then**
                $deleteat(Location\ Rank\ List, i)$
            else
                $i \longleftarrow i + 1$
            end
        end
        $Perturbation \longleftarrow 1$
    else
        $Perturbation \longleftarrow Perturbation + 1$
    end
end

Algorithm 6: Overall Algorithm

Data: $O_i, Q, T_{i,j}, Nearest\ Neighbor\ Function$
Result: $(T^{'}, I^{'})$

```
(T', I') ⟵ Greedy Construction Heuristic(O_i, Q, T_{i,j}, Nearest Neighbor Function)
STOP ⟵ FALSE
while STOP = FALSE do
    for i ← 1 to 4 do
        for j ← 1 to 3 do
            (T'', I'') ⟵ Large Neighborhood i(T', I', O_i, Q, T_{i,j}, Repairing Mechanism_j)
            if Cost(T'') < Cost(T') then
                T' ⟵ T''
                I' ⟵ I''
            end
            if Cost(T') = Cost(T'_{-12}) then
                STOP ⟵ TRUE
                break
            end
        end
        if STOP = TRUE then
            break
        end
    end
end
```

5. Numerical Experiments:

5.1. Experimental Setup:

The algorithm has been coded in Julia (v 0.3.10) ([18]). The Julia implementation of the algorithm has been made as efficient as possible following recommendations from "http://docs.julialang.org/en/release-0.3/manual/performance-tips/" and "http://docs.julialang.org/en/release-0.3/manual/profile/". Experiments are conducted with the garbage collector turned on and the "@inbounds" macro added before each function definition. Turning off the garbage collector, results in an average speed up of 50% for individual instances, but results in a stack overflow when multiple instances are run sequentially. The computational experiments are carried out on a workstation powered by an Intel Core i7-4790 CPU @ 3.6 GHz with a total RAM of 16 GB running an Ubuntu 14.10 64 bit operating system.

5.2. Case Study 1: 1-PDTSP Instances

The pupose of Case Study 1 is to compare the performance of the presented algorithm with exact algorithms from [3] and Tabu Search algorithms from [2].

The instances are similar to those used in [2] and [3] and adapted from the 1-PDTSP instances introduced in [19]. Computational experiments are carried out on a subset of the above instances with $\alpha = \{1, 3\}$, $|N| = \{20, 30, 40, 50, 60\}$, $Q = \{10, 15, 20, 25, 30, 35, 40, 45, 1000\}$ and $|N| = 100$, $Q = \{10, 30, 45, 1000\}$. Further, each of these configurations have 10 independent instances, resulting in a total of 980 instances. Each instance consist of Euclidean coordinates of the nodes, and an integer demand $O_i = \{-10, ..., 10\}, \forall i \in N$. For each instance $Inv_i = \alpha \times 10$, $Target_i = \alpha \times (10 - O_i)$, $C_i = \alpha \times 20$, $\forall i \in N$ and $T_{i,j}$ is the euclidean distance between the i^{th} and the j^{th} nodes, $\forall i, j \in N$.

5.2.1. Terminology:

- **TS1:** Tabu search algorithm initialized with the solution from a greedy heuristic ([2])
- **TS2:** Tabu search algorithm initialized with the Eulerian circuit from the MILP relaxation ([2])
- **RB:** Reliability branching used for exploring the Branch-and-bound tree while solving the relaxation of the static rebalancing problem ([2]).
- **DB:** Degree branching used for exploring the Branch-and-bound tree while solving the relaxation of the static rebalancing problem ([2]).
- **PEA:** Exact algorithm that allows preemption ([3])
- **NPEA:** Exact algorithm that does not allow preemption ([3])
- **NLNS+VND:** The hybrid nested large neighborhood search with variable neighborhood descent algorithm presented in this paper.
- **LB:** Lower bounds
- **UB:** Upper bounds

Algorithms for which more than one trial has been taken,

- **Best UB:** Minimum upper bound found by the algorithm in all of its trials
- **Avg UB:** Average of the upper bounds found by the algorithm in all of its trials.

For each instance:

- TS1 has two trials.
- TS2 has one trial each, with RB and DB as the branching strategy.
- NPEA and PEA have one trial each.
- NLNS+VND has 30 trials, using each of the 3 nearest neighborhood functions to create an initial solution for 10 trials.

5.2.2. Results:

For each instance, we compute:

1. Best known lowerbound (BEST LB) =
$$maximum\{NPEA_{LB}, PEA_{LB}, RB_{LB}, DB_{LB}\}$$
2. Best known upperbound (BEST UB) =
$$minimum\{NPEA_{UB}, PEA_{UB}, TS1_{Best\,UB}, TS2_{Best\,UB}\}$$
3. Best known non-preemptive upperbound (BEST NP UB) = $NPEA_{UB}$
4. Best upperbound found (BEST UB Found) = minimum upperbound found by NLNS+VND in all 30 trials.
5. Average upperbound found (AVG UB Found) = average of the upperbounds found by NLNS+VND in all 30 trials.

5.2.2.1. New Solutions found by NLNS+VND:. In Tables 1 and 2, columns $3-10$ refer to the value of Q. "-" in Table 1 represent the configurations, for which previous upperbounds are not available. Tables 1 and 2 refer to the number of instances out of 10 for each configuration, for which BEST UB Found $<$ BEST UB and BEST UB Found $<$ BEST NP UB respectively. From Tables 1 and 2, it is evident that NLNS+VND is more effective than the algorithms presented in the literature for realistic instances,i.e., instances with $|N| \geq 50$ and $|Q| \leq 20$.

Table 1: New Solutions Found

α	$\|N\|$	10	15	20	25	30	35	40	45	1000
1	50	1	0	0	0	0	0	0	0	0
1	100	10	-	-	-	4	-	-	1	4
3	50	1	0	1	0	0	0	0	0	0
3	60	2	0	0	2	0	1	0	0	0
3	100	10	-	-	-	10	-	-	10	1

Table 2: New Non-preemptive Solutions Found

α	$\|N\|$	10	15	20	25	30	35	40	45	1000
1	40	1	0	0	0	0	0	0	0	0
1	50	1	1	0	0	0	1	0	0	1
1	60	2	2	0	0	1	2	1	2	3
3	40	1	0	1	1	1	1	0	0	0
3	50	3	3	4	1	1	1	1	1	1
3	60	4	1	2	5	2	2	1	2	2

5.2.2.2. Comparison of NLNS+VND with Exact Algorithms ([3]):. Tables 3 and 4 compare the performance NLNS+VND with NPEA and PEA and summarizes the GAP in % and Computational Time in seconds of the three algorithms respectively.

Table 3: Comparison of NLNS+VND with Exact Algorithms - GAP(%)

Category	*Value*	*PEA*	*NPEA*	*NLNS + VND*	
				Best	*Average*
Overall	-	-0.04	2.0	2.03	4.14
α	1	-0.33	0.98	1.83	3.83
α	3	0.25	3.03	**2.23**	4.45
$\|N\|$	20	-0.58	0.0	0.78	1.51
$\|N\|$	30	-0.15	0.0	1.23	2.45
$\|N\|$	40	-0.38	1.05	2.15	4.32
$\|N\|$	50	0.08	3.27	**2.24**	5.13
$\|N\|$	60	0.82	5.68	**3.73**	7.27
Q	10	0.51	4.28	**1.97**	**3.93**
Q	15	0.15	3.0	**1.43**	3.15
Q	20	-0.03	2.21	2.77	5.42
Q	25	0.11	2.38	**2.24**	4.59
Q	30	0.06	1.39	2.01	4.28
Q	35	0.02	0.83	2.32	4.53
Q	40	-0.32	0.53	2.13	4.47
Q	45	-0.39	1.79	**1.79**	3.86
Q	1000	-0.47	1.61	**1.58**	2.99

In Table 3, columns PEA and NPEA refer to $\frac{PEA_{UB} - BEST\ LB}{BEST\ LB} \times 100$ and $\frac{NPEA_{UB} - BEST\ LB}{BEST\ LB} \times 100$ respectively. Columns Best and Average under NLNS+VND refer to $\frac{BEST\ UB\ Found - BEST\ LB}{BEST\ LB} \times 100$ and $\frac{AVG\ UB\ Found - BEST\ LB}{BEST\ LB} \times 100$ respectively. Negative values in column PEA denotes the added value of preemption for that configuration. From Table 3 it is evident that, PEA is more effective than NLNS+VND as it allows preemption. However, on average NLNS+VND is able to find solutions better than NPEA, for instances with $\alpha = \{3\}$, $|N| \geq 50$ and $Q = \{10, 15, 25, 45, 1000\}$.

Table 4: Comparison of NLNS+VND with Exact Algorithms - Computational Time(seconds)

Category	*Value*	*PEA*	*NPEA*	*NLNS+VND*	*Time Ratio*	
					PEA	*NPEA*
Overall	-	978.06	1663.74	**2.87**	341.37	580.68
α	1	696.29	1545.52	**0.71**	**980.89**	**2177.25**
α	3	1259.84	1781.95	**5.02**	250.94	354.94
$\|N\|$	20	0.61	28.9	**0.25**	2.45	116.51
$\|N\|$	30	113.24	142.74	**0.75**	150.61	189.83
$\|N\|$	40	123.58	1105.43	**1.63**	75.65	676.67
$\|N\|$	50	1806.71	2939.74	**3.85**	469.62	764.14
$\|N\|$	60	2846.17	4101.88	**7.84**	362.8	522.87
Q	10	1205.56	1657.67	**14.18**	84.99	116.86
Q	15	868.66	1430.1	**3.78**	229.69	378.15
Q	20	868.77	1522.94	**2.35**	369.09	647.01
Q	25	779.46	1272.11	**1.49**	522.18	852.22
Q	30	771.69	1504.66	**0.93**	828.17	**1614.79**
Q	35	1221.36	1986.39	**0.93**	**1308.83**	**2128.65**
Q	40	1207.66	1709.21	**0.84**	**1444.85**	**2044.89**
Q	45	862.39	1658.9	**0.74**	**1160.4**	**2232.15**
Q	1000	1016.99	2231.63	**0.53**	**1923.0**	**4219.75**

In Table 4, columns PEA and NPEA refer to the time taken by the respective algorithms in seconds to execute on Iridis 4 Computing Cluster. Column NLNS+VND refers to the average of the computational time in seconds of the 30 trials of NLNS+VND. The performance of Iridis 4 Computing Cluster and Intel i7-4790 is quite similar, so the computational times are not normalized. Columns PEA and NPEA under column Time Ratio is the ratio $\frac{PEA_{Time}}{(NLNS+VND)_{Average\ Time}}$ and $\frac{NPEA_{Time}}{(NLNS+VND)_{Average\ Time}}$ respectively. They signify on average how fast NLNS+VND is compared to PEA and NPEA respectively. From Table 4 it is evident that, NLNS+VND is extremely efficient compared to PEA and NPEA. For instances with $\alpha = 1$ or $Q \geq 30$, NLNS+VND is three orders of magnitude faster than both PEA and NPEA.

5.2.2.3. Comparison of NLNS+VND with Tabu Search ([2]):. Tables 5 and 6 compare the performance NLNS+VND with TS1 and TS2 and summarizes the GAP in % and Computational Time in seconds, of the three algorithms respectively.

Table 5: Comparison of NLNS+VND with Tabu Search Algorithms - GAP(%)

Category	*Value*	*TS*1		*TS*2		*NLNS* + *VND*	
		Best	*Average*	*Best*	*Average*	*Best*	*Average*
Overall	-	14.79	16.28	5.21	7.15	**4.65**	**7.04**
α	1	11.92	13.42	3.7	4.93	4.36	6.78
α	3	17.65	19.13	6.72	9.37	**4.9**	**7.3**
$\|N\|$	20	1.23	2.07	-0.61	-0.6	0.67	1.23
$\|N\|$	40	3.25	4.24	-0.27	-0.15	2.01	4.12
$\|N\|$	60	12.32	13.94	2.02	2.98	3.27	6.56
$\|N\|$	100	42.34	44.87	19.69	26.36	**12.64**	**16.25**
Q	10	22.55	24.1	8.1	12.13	**5.28**	**7.45**
Q	30	15.52	17.15	6.36	8.1	**5.33**	**8.06**
Q	45	12.71	14.09	5.14	6.21	**5.02**	7.61
Q	1000	8.37	9.78	1.24	2.14	2.97	5.04

In Table 5, columns Best and Average under columns TS1 and TS2 refer to $\frac{TS1_{BEST\,UB} - BEST\,LB}{BEST\,LB} \times 100$, $\frac{TS1_{AVG\,UB} - BEST\,LB}{BEST\,LB} \times 100$, $\frac{TS2_{BEST\,UB} - BEST\,LB}{BEST\,LB} \times 100$ and $\frac{TS2_{AVG\,UB} - BEST\,LB}{BEST\,LB} \times 100$ respectively. Columns Best and Average under column NLNS+VND refer to $\frac{BEST\,UB\,Found - BEST\,LB}{BEST\,LB} \times 100$ and $\frac{AVG\,UB\,Found - BEST\,LB}{BEST\,LB} \times 100$ respectively. TS1 and TS2 both allow preemption, thus negative values under column TS2 denotes the added value of preemption for that configuration. From Table 5 it is evident that, NLNS+VND is more effective than TS1 for all instances whereas NLNS+VND is more effective than TS2, for instances with $\alpha = 3$ or $|N| > 60$ or $Q = \{10, 30, 45\}$.

Table 6: Comparison with Tabu Search Algorithms - Computational Time(seconds)

Category	*Value*	*Average Time (seconds)*			*Time Ratio*	
		*TS*1	*TS*2	*NLNS* + *VND*	*TS*1	*TS*2
Overall	-	801.02	4557.35	**25.47**	$\geq$ 7.86	$\geq$ 44.73
α	1	616.4	3591.69	**2.67**	$\geq$ **57.73**	$\geq$ **336.41**
α	3	985.65	5523.0	**48.27**	$\geq$ 5.11	$\geq$ 28.61
$\|N\|$	20	17.79	6.0	**0.32**	$\geq$ 13.83	$\geq$ 4.66
$\|N\|$	40	355.02	2980.28	**2.18**	$\geq$ **40.71**	$\geq$ **341.71**
$\|N\|$	60	1038.14	5285.57	**11.61**	$\geq$ **22.35**	$\geq$ **113.8**
$\|N\|$	100	1793.14	9957.55	**87.76**	$\geq$ 5.11	$\geq$ 28.37
Q	10	1353.86	7279.15	**93.83**	$\geq$ 3.61	$\geq$ 19.39
Q	30	598.61	4904.72	**3.66**	$\geq$ **40.87**	$\geq$ **334.88**
Q	45	639.63	3766.67	**2.7**	$\geq$ **59.28**	$\geq$ **349.11**
Q	1000	611.99	2278.86	**1.68**	$\geq$ **91.03**	$\geq$ **338.95**

In Table 6, columns TS1 and TS2 under column Average Time refer to the average time taken in seconds by the respective algorithms to execute on a workstation powered by an AMD Athlon 5600+ processor @ 2.8 Ghz. Column NLNS+VND under Average Time refers to the average of the computational time of all 30 trials of NLNS+VND in seconds. Unfortunately, we did not have access to a workstation powered by an

AMD Athlon 5600+ processor, so it was impossible to compute the Time Ratio exactly. However, a trial run on a Notebook powered by an AMD A8-6410K APU @ 2.0Ghz turned out to be, on average, 4 times slower than that on our workstation. As the clock speed of AMD Athlon 5600+ (2.8 Ghz) is greater than that of AMD A8-6410K (2.0 Ghz), a workstation powered by an AMD Athlon 5600+ CPU should be k times slower than a workstation powered by an Intel i7-4790 CPU, where $k < 4$. Thus, while computing the time ratio for TS1 and TS2, their reported times were divided by 4 to compute a lower bound of the respective Time Ratios. Columns TS 1 and TS 2 under Time Ratio is the ratio $\frac{TS1_{Time}}{4 \times NLNS + VND_{AVG\,Time}}$ and $\frac{TS2_{Time}}{4 \times NLNS + VND_{AVG\,Time}}$ respectively. Thus, they signify on average at least how fast NLNS+VND is compared to TS1 and TS2 respectively. For instances with $\alpha = 1$ or $Q \geq 30$ or $|N| = \{40, 60\}$, NLNS+VND is one and two orders of magnitude faster than TS1 and TS2 respectively.

5.2.2.4. Comparison of performance of NLNS+VND with different initial solutions:. Table 7 summarizes the GAP in % and Computational Time in seconds respectively, of NLNS+VND with different initial solutions. In Table 7, columns $NN\,i, \forall i \in \{1, 2, 3\}$ under Average Gap and Average Time refers to the scenario when $Nearest\ Neighbor\ i()$ is used for creating the initial solution.

Table 7: Comparison of NLNS+VND with different initial solutions

Category	*Value*	*Average Gap (%)*			*Average Time (seconds)*		
		$NN1$	$NN2$	$NN3$	$NN1$	$NN2$	$NN3$
Overall	-	5.13	5.13	**5.11**	9.72	**9.59**	9.71
α	1	4.81	4.79	**4.75**	**1.28**	1.32	1.38
α	3	**5.45**	5.47	5.48	18.17	**17.85**	18.04
$\|N\|$	20	1.52	1.53	**1.48**	**0.24**	0.25	0.25
$\|N\|$	30	**2.39**	2.49	2.47	**0.75**	**0.75**	0.76
$\|N\|$	40	4.33	4.35	**4.28**	**1.6**	1.62	1.68
$\|N\|$	50	5.19	**5.1**	**5.1**	3.88	**3.77**	3.89
$\|N\|$	60	**7.26**	7.29	**7.26**	7.75	**7.65**	8.13
$\|N\|$	100	16.27	**16.17**	16.29	87.1	**85.84**	85.86
Q	10	**6.2**	6.23	6.27	65.86	**64.57**	65.06
Q	15	3.16	3.18	**3.12**	3.79	**3.69**	3.87
Q	20	5.48	5.48	**5.3**	**2.31**	2.35	2.4
Q	25	4.6	4.64	**4.54**	**1.45**	1.49	1.53
Q	30	**6.68**	6.73	6.72	**2.65**	**2.65**	2.85
Q	35	4.58	**4.51**	**4.51**	**0.92**	0.93	0.95
Q	40	**4.45**	4.5	4.46	**0.82**	0.84	0.85
Q	45	6.22	**6.21**	6.22	2.0	**1.98**	2.07
Q	1000	4.24	**4.14**	4.27	**1.16**	1.34	1.33

From Table 7, it is evident that the performance of NLNS+VND is independent of the initial solution used, as with all the three different "Nearest Neighbor ()" used for creating initial solutions, NLNS+VND has quite similar performance.

From Tables 4 and 6, it is evident that NLNS+VND is the most efficient of all the five algorithms, as on average it has the lowest runtime. From Tables 1, 2, 3 and 5, it is evident that PEA outperforms NLNS+VND in terms of finding higher quality solutions for instances with $|N| \leq 60$. One of the reasons for this is, PEA allows preemption, whereas NLNS+VND does not allow preemption. For realistic instances, NLNS+VND outperforms NPEA, TS1 and TS2 in terms of finding higher quality solutions. Futher, the performance of

PEA, NPEA and TS2 depends on the performance of a commercial MILP solver whereas that of NLNS+VND does not.

5.3. Casestudy 2: Share-A-Bull Instances

The University of South Florida's Tampa campus covers 1,700 acres and houses more than 320 buildings. Walking from one building to another during the short breaks between classes is challenging, owing to weather conditions and the heavy weight of textbooks. An annual Tampa campus transportation and parking survey shows that those who drive to campus make, on average, one across-campus trip per day (between buildings or to lunch). Given that there are more than 38,000 students and 1,700 faculty and staff on the Tampa campus, across-campus driving trips can lead to significant fuel consumption and greenhouse gas (GHG) emissions. Thus, USF collaborated with Social Bicycles (SoBi) and developed the Share-A-Bull FFBS program (SABB). Phase I of the program was launched in September 2015 with 100 bicycles. With Phases II and III in next few years, the program will be expanded to 300 bicycles and cover both the Tampa campus and student housing in the vicinity of the campus. The program is expected to be integrated with parking management and other multimodal transportation initiatives on the campus. SABB provides an excellent case study for our bike rebalancing research.

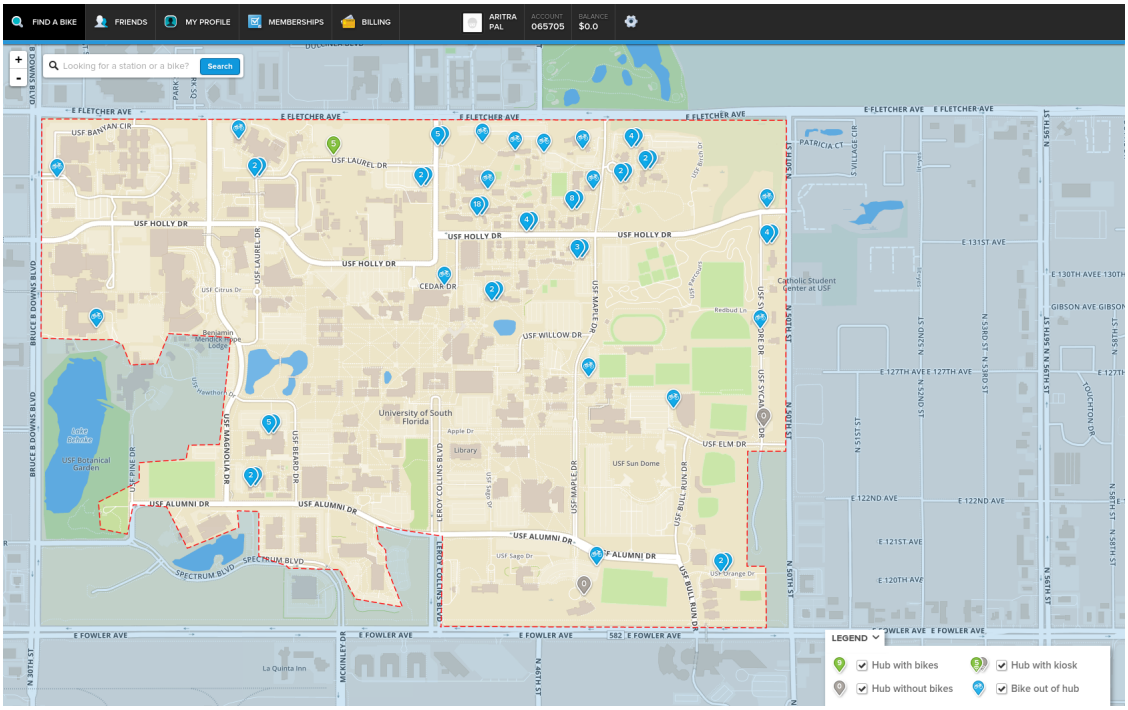

Figure 1: **Map of Share-A-Bull on October** 8^{th}**, 2015 at 12 P.M.**

Originally, there were 147 nodes (bike racks) on the USF Tampa campus. An additional 303 artificial nodes at realistic locations on the campus were added to generate large-scale instances. The capacities of these artificial nodes were generated from $N(\mu, \sigma)$, where μ and σ are the mean and standard deviation of the capacities of the original 147 nodes (bike racks). The bicycle rack located at the campus recreation center is treated as the Depot. Testcases are designed to simulate Phase I, II and III of the Share-A-Bull program, i.e., for 100, 200 and 300 bikes. For each scenario, the maximum $|N| = min\{2 \times \#\,Bikes, \#\,Bikes + \#\,Original\ Nodes\}$. Thus for 100, 200 and 300 bikes the maximum $|N|$ are 200, 347 and 447 respectively. We used the following configurations for our experiments:

Phase	# Bikes	$\|N\|$
I	100	100,150,200
II	200	100,150,200,300
III	300	100,150,200,300,400

For configurations with $|N| \leq 147$, $|N| - 1$ was selected randomly from the set of 146 original nodes (excluding the Depot), otherwise all the original 146 nodes (excluding the depot) were selected, and the remaining $|N| - 147$ nodes, were selected randomly from the 303 artificial nodes. Loading-unloading operations at each node was randomly assigned such that

1. $O_1 = 0$
2. $O_i \neq 0, \forall i \in \{2, 3, ..., |N|\}$
3. $|O_i| \leq C_i$.
4. $\sum_i O_i = 0$
5. $\sum_i |O_i| = 2 \times \ \# \ Bikes$.

The initial inventory for all the nodes, $= -O_i$ if $O_i \leq 0$, else it is $= 0$. Similarly, the final target level for all the nodes $= O_i$ if $O_i \geq 0$, else it is $= 0$. The distance matrix between the nodes, i.e., $T_{i,j}$ was computed from an Open Street Map of the USF, Tampa campus. For each unique configuration, 10 independent instances were created. For each instance, NLNS+VND (with Nearest Neighbor 2() as the initial solution creator) was executed once with $Q = \{5, 8, 10\}$. This is because, the project advisory board for the Share-A-Bull program, is considering these three capacities to invest in.

As mentioned earlier, approximately 8 hours (from 11:00 pm to 7:00 am) is available per day for computation and rebalancing. One of the objectives of this case study was to determine for which of this configurations complete rebalancing with a single vehicle is feasible. Another important objective was to determine if NLNS+VND is capable of dealing with increase in the complexity of the static rebalancing problem (when $|N| \geq 100, Q \leq 10$) for FFBS.

Figures 2, 3 and 4 demonstrate, the variation of Rebalancing Time with Q, Average Speed of the Rebalancing Vehicle, Average Loading-Unloading Time per Bike (seconds), # Bikes, and |N| for the SABB instances.

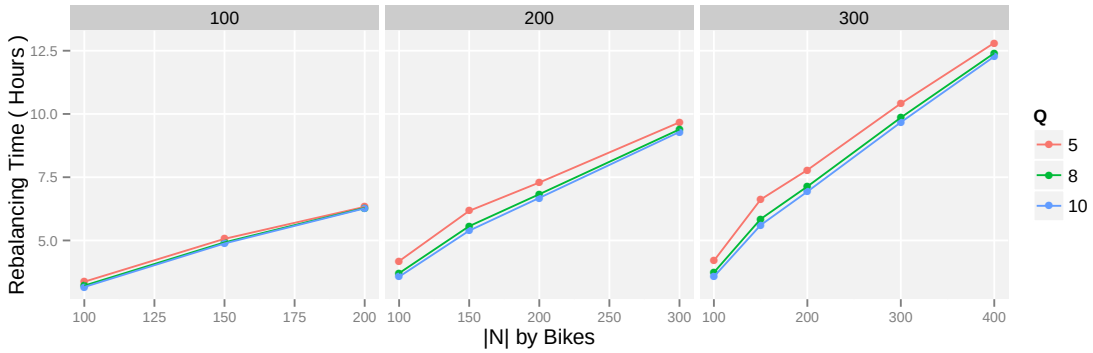

Figure 2: **Variation of Rebalancing Time with $Q, \#\ Bikes$ and $|N|$ for SABB Instances**

Although the speed limit on the USF, Tampa campus is 25 *mph*, the average speed of the rebalancing vehicle varies from $\{10, 15, 20\}$ *mph* (Figure 3), taking into consideration, acceleration and deceleration of the rebalancing vehicle.

The average loading-unloading time per bike (LUL Time) is varied from $= \{30, 45, 60\}$ *seconds* (Figure 4).

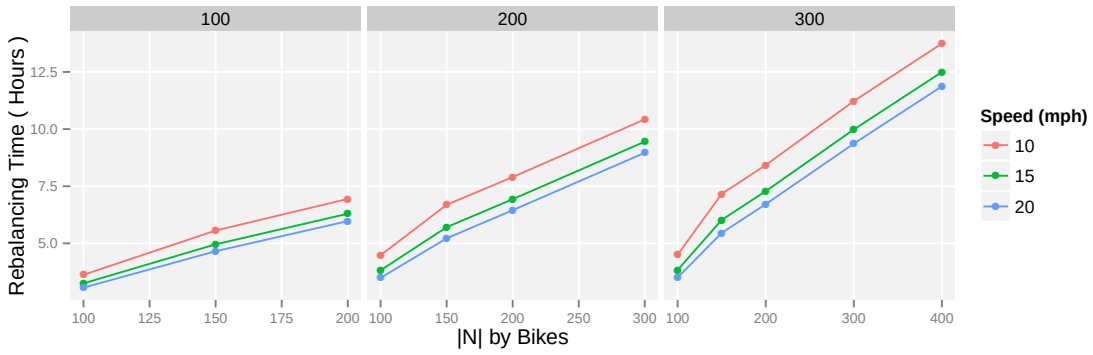

Figure 3: **Variation of Rebalancing Time with Average Speed of the Rebalancing Vehicle (mph), # *Bikes* and $|N|$ for SABB Instances**

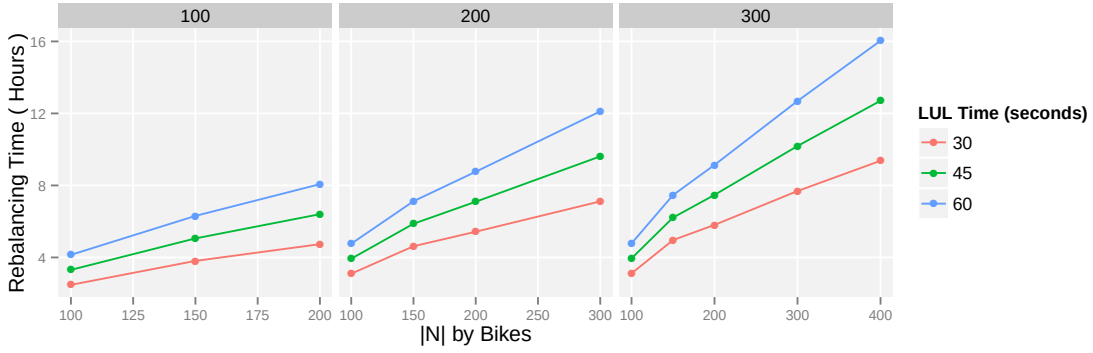

Figure 4: **Variation of Rebalancing Time with Average Loading-Unloading Time per Bike (seconds), # *Bikes* and $|N|$ for SABB Instances**

From Figures 2, 3 and 4, it is evident that out of all the five factors LUL Time has the most effect on Rebalancing Time. This is because, first the total loading-unloading time is directly propotional to LUL Time and second, the total loading-unloading time contributes considerably towards the total rebalancing time.

Figures 5 and 6 demonstrate, the effectiveness of NLNS+VND in finding high quality solutions for the SABB instances.

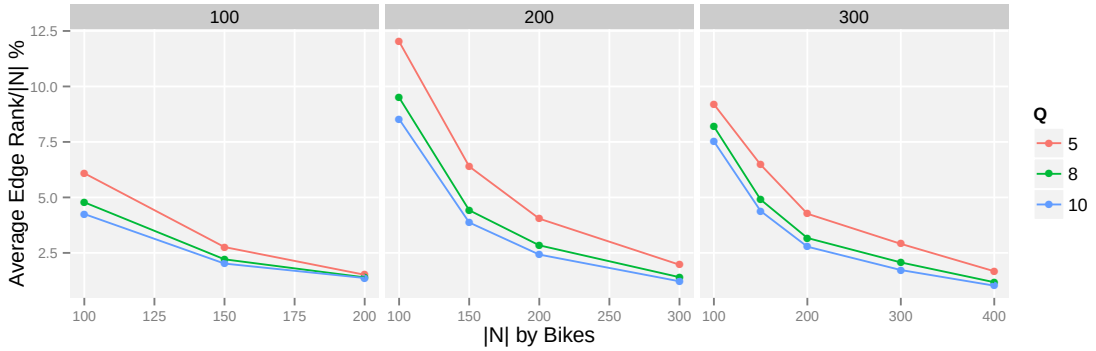

Figure 5: **Variation of** $\frac{Average\ Edge\ Rank}{|N|}\%$ **with** $Q, \#\ Bikes$ **and** $|N|$ **for SABB Instances**

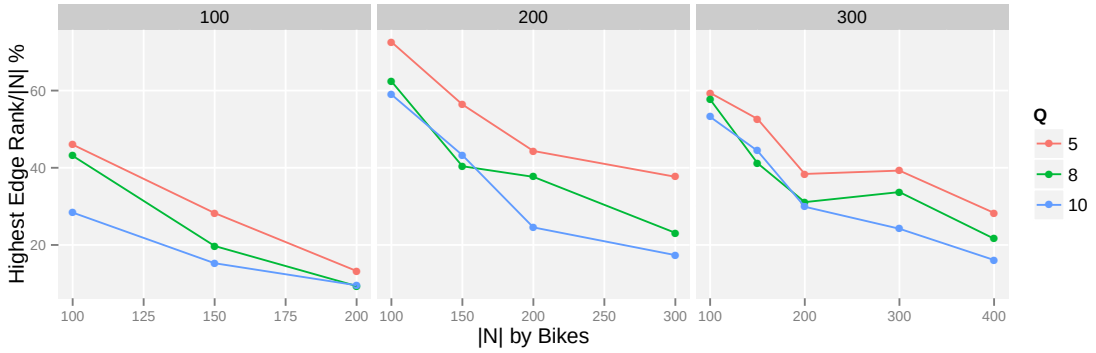

Figure 6: **Variation of** $\frac{Highest\ Edge\ Rank}{|N|}\%$ **with** $Q, \#\ Bikes$ **and** $|N|$ **for SABB Instances**

Like gap between a lowerbound and an upperbound, $\frac{Average\ Edge\ Rank}{|N|}\%$ and $\frac{Highest\ Edge\ Rank}{|N|}\%$ can be used as a measure of the quality of the solution. From our experiments, we noticed that $80-85\%$ of the edges in a tour, have Edge Rank $\leq$ Average Edge Rank of the tour. Intuitively we can claim that, lower the average and the highest edge rank of a tour, higher is the quality of that tour. From figures 5 and 6, it is evident that the ratio is inversely propotional to $\#\ Bikes$ and $|N|$, and directly propotional to Q. Thus, NLNS+VND is not only effective at the lower end ($|N| = \{100, 150\}$), but also at the higher end ($|N| = \{300, 400\}$) of the spectrum. Further, $\frac{Highest\ Edge\ Rank}{|N|}\%$ demonstrate that high quality solutions can be obtained, by working only with edges $\leq \frac{Highest\ Edge\ Rank}{|N|}\%$. This information will come in handy when working with large instances.

Figure 7 demonstrate the efficiency of NLNS+VND, in finding high quality solutions for the SABB instances.

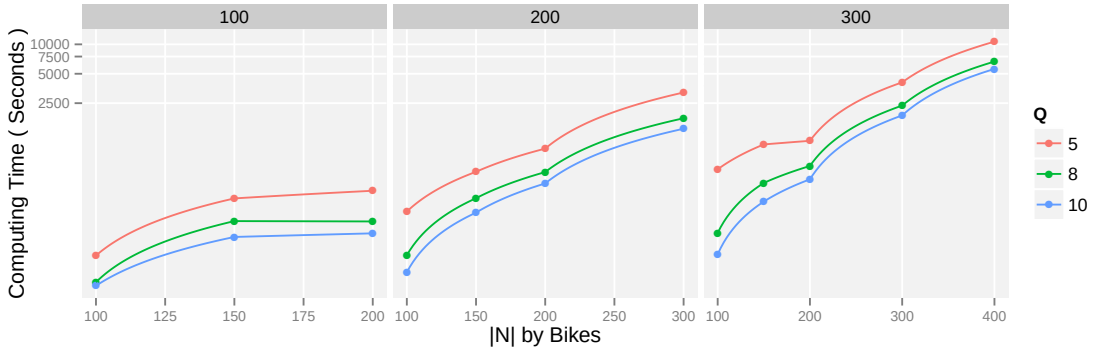

Figure 7: **Variation of Computing Time with $Q, \#\ Bikes$ and $|N|$ for SABB Instances**

The scale on the y-axis for Computing Time (seconds), in Figure 7 is in log10 scale. From Figure 7, it can be concluded that the computational time increases with increase in $\#\ Bikes$ and $|N|$ and decreases with increase in Q. Of these three parameters, $|N|$ has the most effect on the computational time. It is evident that NLNS+VND can handle instances with $|N| \geq 100$. However, it loses its efficiency when $|N| > 200$. We must note that, complete rebalancing with a single vehicle for the SABB instances is not feasible, when $|N| \geq 200$ (Figure 8).

Total Time (in Figure 8) is the summation of the Computing Time and the Rebalancing Time. The purpose of Figure 8 is to determine, for which configurations, complete rebalancing with a single vehicle is feasible for the SABB instances, in the extreme scenario. Complete rebalancing with a single vehicle is feasible, for any configuration below or on the desired time line (= 8 Hours). From Figure 8 it is evident that, configurations with $|N| \leq 150$ are almost always feasible for complete rebalancing with a single vehicle. If $|N| > 150$, the operator has to resort to either complete rebalancing with multiple vehicles, or partial rebalancing, depending on the time budget and number of vehicles available for rebalancing.

6. Conclusion

In this article, we presented a hybrid nested large neighborhood search with variable neighborhood descent algorithm for solving static bike rebalancing problems both effectively and efficiently for free-floating and station-based bike sharing. Computational experiments on 1-PDTSP instances, previously used the literature, demonstrate that the presented algorithm outperforms tabu search and is highly competitive with exact algorithms reported in the literature. The fact that it was able to find new solutions for 58 of 148 instances (for which the optimal solution is not known) and new non-preemptive solutions for 60 of 160 instances (for which the non-preemptive optimal solution is not known) show that the presented algorithm is more robust than the algorithms previously reported in the literature. Further, the presented algorithm is, on average, 340 times faster than the exact algorithm that allows preemption and 580 times faster than the exact algorithm does not allow preemption. To the best of our knowledge, we are the first to solve static rebalancing problems, with instances having nodes greater than or equal to 50 and vehicle capacity less than 30 both effectively and efficiently.

Computational experiments on the new SABB instances consisting of up to 400 nodes and 300 bikes demonstrate that the presented algorithm is able to deal with the increase in scale of the static rebalancing problem for FFBS. It also shows that the static rebalancing problem using a single vehicle and having complete rebalancing as a hard constraint is feasible when $|N| \leq 150$ for the Share-A-Bull FFBS program at USF, Tampa.

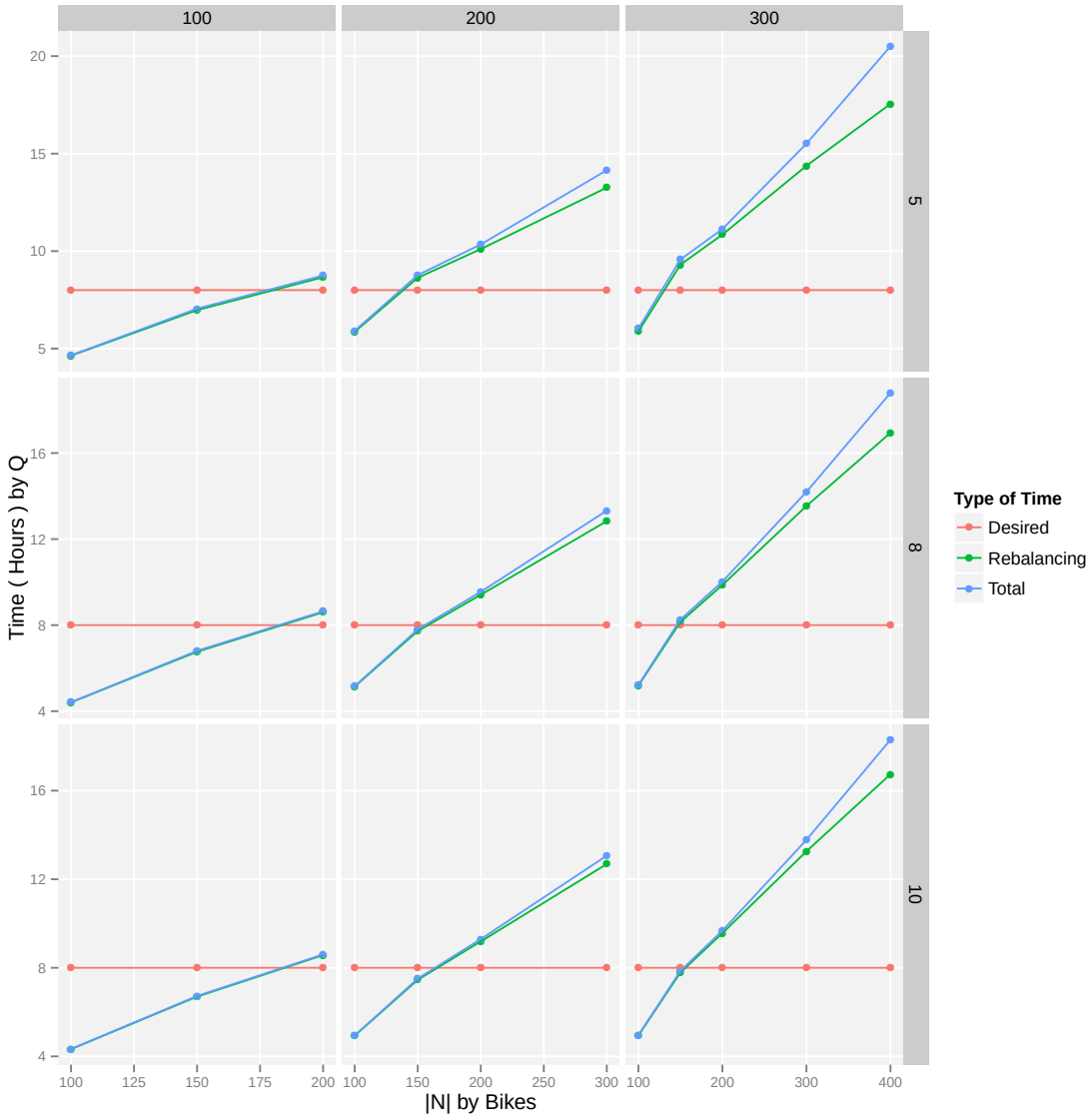

Figure 8: **Plot of Total Time and Rebalancing Time vs $Q, \#\, Bikes$ and $|N|$ when Speed = 10mph and LUL Time = 60 Secs for SABB instances**

In future articles, we will present solution algorithms for dealing with static complete rebalancing for FFBS with multiple vehicles and static partial rebalancing for FFBS with single and multiple vehicles.

7. Acknowledgements

This study was supported by the Student Green Energy Fund (SGEF) at University of South Florida. The authors thank Joseph Fields for creating an Open Street Map of the USF Tampa campus.

8. References

[1] DeMaio P. Bike-sharing: History, impacts, models of provision, and future. Journal of Public Transportation 2009;12(4):41–56.
[2] Chemla D, Meunier F, Wolfler Calvo R. Bike sharing systems: Solving the static rebalancing problem. Discrete Optimization 2013;10(2):120–46.
[3] Erdoğan G, Battarra M, Calvo RW. An exact algorithm for the static rebalancing problem arising in bicycle sharing systems. European Journal of Operational Research 2015;.
[4] Toth P, Vigo D. The granular tabu search and its application to the vehicle-routing problem. Informs Journal on computing 2003;15(4):333–46.
[5] Schuijbroek J, Hampshire R, van Hoeve WJ. Inventory rebalancing and vehicle routing in bike sharing systems 2013;.
[6] Rainer-Harbach M, Papazek P, Raidl GR, Hu B, Kloimüllner C. Pilot, grasp, and vns approaches for the static balancing of bicycle sharing systems. Journal of Global Optimization 2014;:1–33.
[7] Erdoğan G, Laporte G, Calvo RW. The static bicycle relocation problem with demand intervals. European Journal of Operational Research 2014;238(2):451–7.
[8] Raviv T, Tzur M, Forma IA. Static repositioning in a bike-sharing system: models and solution approaches. EURO Journal on Transportation and Logistics 2013;2(3):187–229.
[9] Forma IA, Raviv T, Tzur M. A 3-step math heuristic for the static repositioning problem in bike-sharing systems. Transportation Research Part B: Methodological 2015;71:230–47.
[10] Ho SC, Szeto W. Solving a static repositioning problem in bike-sharing systems using iterated tabu search. Transportation Research Part E: Logistics and Transportation Review 2014;69:180–98.
[11] Dell'Amico M, Hadjicostantinou E, Iori M, Novellani S. The bike sharing rebalancing problem: Mathematical formulations and benchmark instances. Omega 2014;45:7–19.
[12] Contardo C, Morency C, Rousseau LM. Balancing a dynamic public bike-sharing system; vol. 4. CIRRELT; 2012.
[13] Fricker C, Gast N. Incentives and regulations in bike-sharing systems with stations of finite capacity. arXiv preprint arXiv:12011178 2012;.
[14] Haider Z, Nikolaev A, Kang JE, Kwon C. Inventory rebalancing through pricing in public bike sharing systems. Ph.D. thesis; STATE UNIVERSITY OF NEW YORK AT BUFFALO; 2014.
[15] Chemla D, Meunier F, Pradeau T, Calvo RW, Yahiaoui H, et al. Self-service bike sharing systems: Simulation, repositioning, pricing 2013;.
[16] Pfrommer J, Warrington J, Schildbach G, Morari M. Dynamic vehicle redistribution and online price incentives in shared mobility systems 2013;.
[17] Applegate D, Cook W, Rohe A. Chained lin-kernighan for large traveling salesman problems. INFORMS Journal on Computing 2003;15(1):82–92.
[18] Bezanson J, Karpinski S, Shah VB, Edelman A. Julia: A fast dynamic language for technical computing. arXiv preprint arXiv:12095145 2012;.
[19] Hernández-Pérez H, Salazar-González JJ. A branch-and-cut algorithm for a traveling salesman problem with pickup and delivery. Discrete Applied Mathematics 2004;145(1):126–39.