FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Balancing Bike Sharing Systems

A Hybrid Metaheuristic Approach for the Dynamic Case

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Ing. Andreas Pinter, Bakk.techn.
Matrikelnummer 0625726

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung:

Wien, 28.11.2013

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Balancing Bike Sharing Systems

A Hybrid Metaheuristic Approach for the Dynamic Case

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering/Internet Computing

by

Ing. Andreas Pinter, Bakk.techn.
Registration Number 0625726

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance:

Vienna, 28.11.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Ing. Andreas Pinter, Bakk.techn.
Erdbergstraße 17/1/6, 1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasserin)

Danksagung

Ich möchte mich an dieser Stelle zunächst bei meinem Betreuer Günther Raidl für die Unterstützung durch Diskussionen und Feedback bedanken. Weiters möchte ich Citybike Wien und das Austrian Institute of Technology erwähnen, auf deren Vorarbeit, im geförderten Projekt 831740 der österreichischen Forschungsförderungsgesellschaft, ich aufbauen durfte.
Abschließend möchte ich sowohl meinen Eltern als auch meiner Freundin für deren Unterstützung und Motivation danken.

Abstract

Bike Sharing Systems became popular in recent years to extend the public transportation network of cities or regions. Most research in this area focuses on finding the optimal locations for bike sharing stations. Still various approaches on how to operate such a system efficiently exist. The usefulness of a bike sharing system strongly depends on the user convenience which is directly connected to the availability of bikes and parking slots when and where they are needed. To increase user convenience a fleet of vehicles (usually cars with a trailer) are used to move bikes between different stations to avoid empty or full stations.

The goal of this thesis is to provide an algorithm to efficiently calculate transportation routes for balancing such a bike sharing system to improve user convenience. This problem can be seperated into two different scenarios. The static case focuses on rebalancing the system while there is no user activity or the user activity is negligible (e.g. during night time if users are only allowed to rent bikes during the day time). This thesis is considering the dynamic case, in which the system is still online during the rebalancing process. The proposed algorithm is divided into two parts: solution search and solution evaluation. The first part is implemented using two different variants of a Variable Neighborhood Search (VNS) with an embedded Variable Neighborhood Descent (VND). While the set of neighborhood structures for the VNS variants is equal they differ in the neighborhood structures used for the VND. The solution evaluation part incorporates a Linear Programming (LP) approach to calculate the optimal set of loading instructions for a solution found by the first part. In addition a greedy approach is constructed to calculate a set of loading instructions.

Finally three variants (D: complete VND+LP; W: VND(with only two neighborhoods structures used)+LP; G: complete VND+Greedy) are tested on three different sets of test instances with 60, 90 and 120 stations to evaluate the performance of the algorithm.

One initial conclusion is that nearly all the given computation time is used to calculate optimal loading instructions with LP. Comparing variant D and W gives the impression that D is performing better although no strong statistical evidence was found. Compared to variant D and W the solutions found by variant G are between 0.5% and 5% worse. On the other hand the greedy approach evaluates about twice as much solutions than the other two methods in the same amount of time. For real world applications the greedy approach may be the better one. Although it is not guaranteed to find the optimal solution it will find good solutions relatively fast.

Kurzfassung

Bike Sharing Systeme wurden in den letzten Jahren zunehmend beliebter um das öffentliche Verkehrsnetz von Städten oder ganzen Regionen zu bereichern. Die meisten Forschungen in dem Gebiet zielten deshalb auf die optimale Positionierung von entsprechenden Stationen ab. Allerdings wurden ebenso unterschiedliche Forschungen zum effizienten Betrieb eines solchen Systems durchgeführt. Die Sinnhaftigkeit eines Bike Sharing Systems ist allerdings nur dann gegeben, wenn es von den Kunden auch genutzt wird. Dies wiederum hängt direkt damit zusammen ob zum gewünschten Zeitpunkt am gewünschten Ort ein Fahrrad bzw. ein Fahrradabstellplatz verfügbar ist. Um dies zu erreichen wird eine Flotte von Fahrzeugen (üblicherweise PKWs mit Anhängern) eingesetzt, um Fahrräder von einer Station zur anderen zu bewegen.

Diese Arbeit stellt einen Algorithmus vor, der effiziente Fahrzeugrouten berechnet um solch ein Bike Sharing System auszubalancieren und damit die Kundenzufriedenheit zu erhöhen. Grundsätzlich lässt sich das Problem in einen statischen und einen dynamischen Fall aufteilen. Im statischen Fall befindet sich das System in Ruhe, d.h es finden keine Benutzerinteraktionen statt oder die stattfindenden Benutzerinteraktionen können vernachlässigt werden (zum Beispiel in der Nacht, wenn Fahrradnutzung verboten ist). In dieser Arbeit wird der dynamische Fall betrachtet, in dem das Ausbalancieren während des Systembetriebs stattfindet. Der vorgestellte Algorithmus selbst teilt sich ebenfalls in zwei Teile: Lösungsfindung und Lösungsbewertung. Für die Lösungsfindung werden zwei Varianten einer Variable Neighborhood Search (VNS) mit integrierter Variable Neighborhood Descent (VND) eingesetzt. Beide Varianten verwenden ein identes Set an Nachbarschaften für die VNS, unterscheiden sich allerdings bei den VND Nachbarschaften. Bei der Lösungsbewertung wird ein Linear Programming (LP) Ansatz verfolgt um die optimalen Be- und Entladeanweisungen zu berechnen. Zusätzlich wird ein Greedy Ansatz zur Berechnung dieser Ladeanweisungen vorgestellt.

Schließlich wurden die Ergebnisse dreier Varianten (D: vollständiger VND+LP; W: VND(mit lediglich zwei Nachbarschaften)+LP; G: vollständiger VND+Greedy) von drei Testsets mit 60, 90 und 120 Stationen verglichen, um die Effizienz des Algorithmus zu beurteilen.

Als eine intiale Beobachtung konnte festgestellt werden, dass die LP Berechnung einen Großteil der CPU Zeit verbraucht. Vergleicht man Variante D und W miteinander entsteht der Eindruck, dass D bessere Ergebnisse liefert. Allerdings konnte kein statistischer Beweis für diese Hypothese gefunden werden. Verglichen mit Variante D und W sind die Ergebnisse von Variante G zwischen 0.5% und 5% schlechter. Allerdings wurden mit der Greedy Berechnung etwa doppelt so viele Lösungen in der gleichen Zeit evaluiert. Für einen Anwendungsfall in der realen Welt ist Variante G wohl am besten geeignet. Obwohl sie keine optimalen Lösungen garantiert, finden sich gute Lösungen relativ schnell.

Contents

CHAPTER 1

Introduction

1.1 Motivation

A bike sharing system is a system where numerous users share a set of bikes. Typically there are multiple stations distributed in the area of a city like Vienna, where users can rent bikes. The big advantage of this systems is that the bike does not have to be brought back to the same station where it was rented. Instead, any station of the system will do. In theory the system would balance itself if each station were equally desireable as a rent- or a bring back-station. In reality people prefer riding downhill over uphill, riding into the city instead of out of the city and so forth. There are lots of other reasons why one station tends to be empty while another tend to be full. The whole system can only work properly if each station can be used as a pick-up and as a drop-off location. This is absolutely essential for the usability of the system. There is nothing more frustrating than taking a detour to pick up a bike, just to realize that this particular station is currently empty. The same is true for returning a rented bike. The planing of the redistribution tours is usually done by a human operator using his/her experience to decide where to go and how many bikes to (un)load. Those plans are then executed by a fleet of vehicles to achieve a reasonably balanced system. Since most of these systems are rather small (< 50 stations) this can be done pretty easily, considering that it may be enough to simply fill the empty stations with bikes from the full ones. As those systems become increasingly popular, more and more stations are built. As the system grows, the complexity of the balancing task grows as well. Additionally, with increasing frequency of usage it may not be sufficient any more to just look at the empty/full stations, but also consider stations which are going to be empty/full in the near future. The city of Vienna is currently working with 102 stations and 1,200 bikes. Figure 1.1 shows a snapshot of the station loads in Vienna, with red dots indicating full stations and blue dots marking empty stations.

An extreme example of such a system is located in Paris with 1,225 stations and around 18,000 bikes shown in Figure 1.2

From a more technical point of view one could say that this problem is an instance of an unpaired Pickup & Delivery Problem (PDP). The PDP is a special variant of the well known

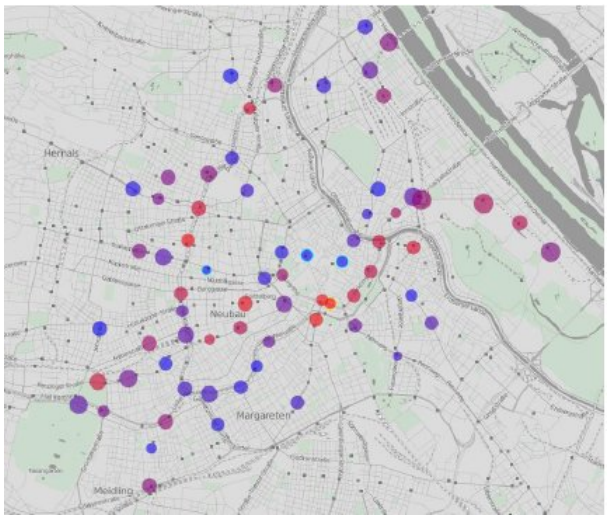


Figure 1.1: snapshot of station status in Vienna (Source: Citybike Wien)

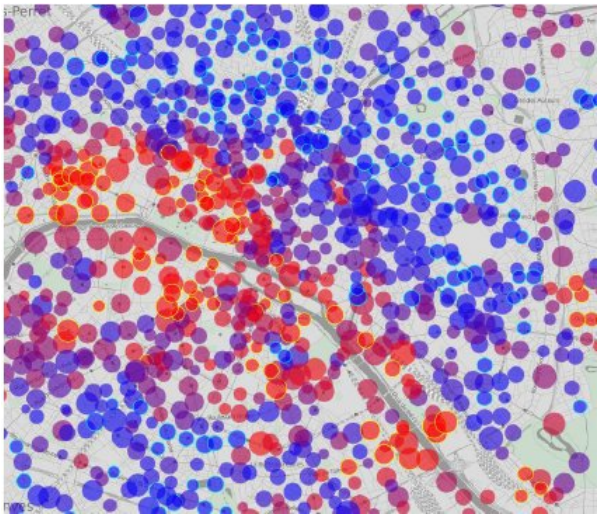


Figure 1.2: snapshot of station status in Paris (Source: Citybike Wien)

Vehicle Routing Problem (VRP). The VRP was first introduced by Dantzig and Ramser [4] in 1959 under the more specific name "The Truck Dispatching Problem". Their definition is built as an generalisation of the Traveling Salesman Problem (TSP), which consists of finding the shortest tour through a set of cities V.

Assume that the salesman carries some sample products with him. Obviously he can only carry a fixed amount C of samples with him. Additionally each city in V "consumes" a different amount of samples denoted by q_i. Those two constraints were also introduced by Dantzig and Ramser, although they were talking about product deliveries instead of samples. As long as

$C \geq \sum_{\forall i \in V} q_i$ the problem is equivalent to the TSP, because the salesman can serve the whole route before he runs out of samples. If the above equation does not hold, the salesman needs to return to some kind of depot to fill his sample bag again. Finally, assume that there is more than just one salesman and that they can all carry the same amount of samples with them. In order to separate this problem from the TSP, one talks about vehicles (or trucks) instead of salesmen, deliveries instead of samples and stations (or customers) instead of cities. The goal of the VRP is to assign all the customers to one of the vehicles in such a way that the demand q_i of all customers is satisfied and the milage of all vehicles is minimal.

The problem described above can now be extended by various additional constraints. Since the problem is described as an assignment between customers, vehicles and products, all three objects can be subject of further constraints. Customers could specify time windows in which they want to be served by the vehicles. Those time windows can be defined as hard time windows, where service needs to happen in between, and soft time windows, where service outside the time window is possible, but results in additional costs reflected in the objective function. Those time windows can also be applied to the vehicle - or for that matter its driver. Truck drivers are usually not allowed to work more than x hours a day or in a single route. Another possible constraint could be that the exact amout of products is not known in advance and needs to be covered by stochastic data. Additionally, the vehicles in the fleetcould have different capacities.

On the other hand, one may choose a different objective function than simply trying to minimize the total milage of the vehicles. A very popular objective function focuses on the minimization of the number of vehicles, because usually an additional vehicle is much more expensive than driving a sub-optimal tour with another vehicle.

The PDP further introduces a transportation request, which consists of one pickup and one delivery node. So the problem morphs from transporting goods from one central depot to multiple customers to transport goods from one location to another. This brings some additional constraints for a valid solution. Obviously, pickup node i needs to be assigned to the same route as the connected delivery node j and pickup node i need to be visited before delivery node j. Another important thing is that with these bindings it is not as trivial as in the classical VRP to define "neighbour" customers (this knowledge is usually used to find good solutions). All the above constraints can also be applied to this problem. For the unpaired PDP, the pickup and delivery locations are not linked together and can be assigned more freely to individual vehicles.

1.2 Problem Description

This paper aims to develop an algorithm for determining transportation plans to efficiently balancing bike sharing systems. This goal can be split into the static and the dynamic case. In the static case, rebalancing is done while the status of the system is not changed by any means other than the balancing process itself. An evaluation of different metaheuristic approaches (VNS, GRASP) of this case can be found in Rainer-Harbach et al. [11]. In the dynamic case, the system is active during the rebalancing process, resulting in users renting and dropping off bikes and therefore changing the current status of the system. This dynamic behavior is modeled by splitting the day into (not necessarily equal) intervals. At the end of each of those intervals user activities are cosidered. For a practical application it is assumed that information about potential

user demands is gained by statistical methods.

An instance of this problem class is defined by the following items and their properties. Note that the variable naming is similar to the ones in Rainer-Harbach et al. [11]. It should allow easier comparison between both cases for the interested reader.

- A complete directed graph $G_0 = (V_0, A_0)$ representing all the relevant locations and the shortest (in terms of costs, time or distance) connection between them. Relevant locations are all the bike stations (V) and all the depots (O).
  - Each arc $a_{u,v} \in A_0$ with $u, v \in V_0$ has
    - costs: $t_{u,v}$
  - Each station $v \in V$ has
    - a maximum number of bikes it can store $C_v > 0$
    - the number of initially stored bikes $0 \leq p_v \leq C_v$, i.e the number of bikes stored before balancing starts
    - a fixed stopping time (e. g. parking): ${t_l}^{\text{stop}} \geq 0$
- A set of vehicles L for transporting bikes between locations. Each vehicle $l \in L$ has
  - a maximum number of bikes it can store $Z_l > 0$
  - a maximal tour duration $\hat{t}_l > 0$
  - start and end points for the tour $s_l, d_l \in O$
  - the number of bikes which are already in the vehicle when it leaves the depot $\hat{b}_l \geq 0$
- A maximum amount of time $T_{\max}$ for each route to be finished.
- demands q_v^t for each station $v \in V$ at discrete times $t \in T = \{t_1, t_2, \ldots, t_{\max}\}$. q_v^t indicates the difference between bike and parking slot demands
  - $q_v^t > 0$ if more users want to rent q_v^t number of bikes at interval t from station v than bring them back
  - $q_v^t < 0$ if more users want to bring back $-q_v^t$ bikes at interval t to station v than rent some
  - $q_v^t = 0$ if bike and parking slot demands are equally high

A possible solution for such a problem consists of a set of routes R. Each route $r \in R$ has:

- a vehicle l assigned
- a total number of stops ρ^r
- An ordered list P^r specifying which stations are visited in which order. r^i denotes the i^{th} stop on the route r for $i \in \{1, \ldots, \rho^r\}$. P^r does not include the starting and endpoint of the assigned vehicle.

- a set of loading instructions Y containing y_i^r for the i^{th} stop on the route r for $i \in \{1, \ldots, \rho^r\}$
  - When $y_i^r > 0$, the instruction is to load y_i^r bikes onto the vehicle.
  - When $y_i^r < 0$ the amount of bikes should be unloaded from the vehicle at the current station.
  - When $y_i^r = 0$ no movement of bikes is necessary.

Each solution must satisfy the following condition to be considered feasible:

- each route r starts at s_l and ends at d_l.
- the amount of bikes inside a vehicle l can never be greater then Z_l or smaller than 0.
- the amount of bikes at a station v can never be greater than C_v or smaller than 0.
- the amount of bikes remaining in a vehicle l at the end of its route has to be 0.
- the duration of a route r for vehicle l can never be greater than $\hat{t}_l$.
- it is prohibited to have unsatisfied user rent demands as long as there are bikes available at the station when the demand takes place.
- it is also prohibited to have unsatisfied user bring back demands as long as there are parking slots available at the station when the demand takes place.

From the set of possible solutions the one minimizing the amount of unsatisfied requests should be found. Such a request occurs when a station $v \in V$ at the end of an interval $t \in T$ does not have enough bikes or space to fulfill the user demand q_v^t. In a more formal way, assume that $\overline{a}_v^t$ states the amount of bikes available at station v at time t without taking q_v^t into account. Then the number of unsatisfied requests is

$$\delta_v^t = \begin{cases} q_v^t - \min(\, \overline{a}_v^t, q_v^t \,) & \text{when } q_v^t > 0 \\ -q_v^t - \min(\, C_v - \overline{a}_v^t, -q_v^t) & \text{when } q_v^t < 0 \end{cases} \tag{1.1}$$

If two solutions are equally good in reducing the number of unsatisfied requests, the one with a smaller total travel time for all vehicles is considered to be the better one. Should they also be identical in length the last distinction is made by prefering the solution with less bike movement. Solutions being equal in all three of these criterias are considered to be identical. To make this distinction the factors τ_{dist} and τ_{work} are used. Both values are set to 10^{-5} to ensure for the problem instances at hand, that their respective terms in the objective function do not exceed 1.

The complete objective function is

$$f(R, Y) = o_{\text{unsat}} + \tau_{\text{dist}} o_{\text{dist}} + \tau_{\text{work}} o_{\text{work}} \tag{1.2}$$

with

$$o_{\text{unsat}} = \sum_{\forall v \in V} (\sum_{\forall t \in T} \delta_v^t) \tag{1.3}$$

$$o_{\text{dist}} = \sum_{\forall r \in R} (\sum_{i=0}^{\rho^r - 2} t_{r^i, r^{i+1}}) + t_{s_l^r, r^0} + t_{r^{\rho^r - 1}, d_l^r} \tag{1.4}$$

$$o_{\text{work}} = \sum_{\forall r \in R} \sum_{i=0}^{\rho^r - 1} y_i^r \tag{1.5}$$

CHAPTER 2

Related Work

When researching for the 'bike sharing system' topic one quickly notices that the interest for this topic has increased in recent years. Most papers were publishing between 2010 and 2013 but not all of them are relevant to this thesis. Some are focused on the aspect of strategical planing and building such a bike sharing system subject to demands and possible connections between stations, while others consider the whole package, like Sayarshad et al. [14]. In early 2011 a small Google group was founded called "Bike Sharing Research and Practice[1]" to share ideas, publications and relevant events between researches and operators of bike sharing systems.

The most related work is from Rainer-Harbach et al. [11], who are working on metaheuristic approaches for the static case of the problem. In the static case each of the stations has an explicit target value q_v of bikes, which should be reached by the end of the algorithm. They further differentiate between the general case and a case where a special restriction named *monotonicity* is in place. Monotonicity restricts the stations to be either a pickup ($V_{pic} = \{v \in V | p_v \geq q_v\}$) or a delivery station ($V_{del} = \{v \in V | p_v \leq q_v\}$), where it is only allowed to pickup or deliver bikes. With that in mind, it is clear that the amount of bikes on any station can only increase or decrease monotonically. This addition to the problem has a major advantage since it is not important any more in which order vehicles are visiting a given station. Rainer-Harbach et al. started off with a greedy algorithm who extends a vehicle route incrementally. It computes all stations which are reachable in time and the number of bikes that could be loaded or unloaded to reach the target value q_v. Based on this value a 'balance gain per time unit' is calculated and used to choose the best station. To assure that all vehicles are empty when their route ends, there is a special calculation in place for pickup stations. For each pickup station it must be possible to drop off at least $b_l + 1$ bikes in time after the visit of the station. b_l denotes the current load of vehicle l and needs to be increased by at least 1 or a visit at that station would be pointless anyway. Aside from this, the main focus of the paper is to remove the caluclation of the loading instructions from the original problem and construct different approaches to evaluate loading

[1] https://groups.google.com/forum/?fromgroups#!forum/bikesharingsystems

instructions for a given set of routes. Three different approaches are presented. The first one is a greedy one, assuming monotonicity. Although it does not guarantee to find an optimal set of loading instructions it runs faster than the other two approaches. The second approach is based on a specifically constructed flow network and is taken and adapted from Meunier et al.[9]. It assures optimal loading instructions at the cost of an increase in average running time of about 120%. The final approach is a linear program solving a minimum cost flow problem. Its advantage compared to the other two approaches is that it works for the general case as well as for the monotonic case. Unfortunatly, this generality comes with the price of a running time of 110 times the max flow approach. According to Rainer-Harbach et al. the first two approaches are equally good with respect to the solution quality with a slight advantage for the maximum flow approach. In principle the LP is able to sometimes find a better solutions, but it is terminated by the time limit of one hour in more than 60% of the test cases.

Contardo et al. [2] were the first to focus on the dynamic case. They are specifically looking into the handling of the "peak hours" of a bike sharing system and distribute those time frames into smaller chunks of two to five minutes, when user requests happen. In contrast to this thesis, they assign each station to be a pickup or a delivery station based on geographical data and user behavior. It is assumed that the rent and bring back requests are continiously happening during those peak hours. So the size of the time periods only defines the frequency of those requests. Contardo et al. formulated two mathematical approaches to solve the problem. On one hand, they designed an arc flow formulation (AFF) based on a space-time graph of the original problem. The second approach is a column generation coupled with Benders decomposition (CG+BD). Their two approaches were tested on two instance sets created by themselves. The first set consists of randomly placed stations, alternating between pickup and delivery stations, resulting in close to equal amounts of pickup and delivery stations. The second set contains clustered stations, where each station in a cluster has the same type. Both sets are created for 25, 50 and 100 stations and a time horizon of two hours, seperated into two and five minute chunks. The fleet size is set to five. The CG+BD approach produces better lower and upper bounds than the AFF in less time. CG+BD also performs better on the clustered instances. Those are the more realistic ones, since when looking at peak hours, users tend to search for 'near' station if the current station is full or empty. Contardo et al. believe that the better performance is causeed by the additional structural information available.

A somewhat related topic was handled by Lin and Chou [8] in 2012. Based on the problem of a bike sharing system they published a paper on how to add additional reality to provide a balanced system. Most algorithms for a Vehicle Routing Problem are using Euclidean distance to incorporate the travel costs from point A to B. But in basically all real world applications this assumption is just wrong. Distances and travel times between bike stations are dependent on road conditions, traffic regulations and other geographical factors. Lin and Chou analysed the impacts of using real world travel distances on various algorithms. Instead of calculating real travel times themselves, they used the Google Directions API. It is obvious that adding much more realistic information results in better solutions for real world problems. On the other hand, adding realistic data also affects the used algorithms. Both the savings heuristic (from Clarke and Wright [1]) as well as the farthest insertion heuristic (from Rosenkrantz et al. [13]) need to be modified. Those two heuristics are calculating the value of all possible modifcations to a

route and rely on the assumption that the distance from A to B and from B to A are equivalent. Considering one-way streets and other geographical factors shows that this assumption is highly unlikely to be true in the real world. This means that e.g. the savings heuristic need to caluclate $n(n-1)$ different saving values rather than just $n(n-1)/2$. A similar increase is needed for the farthest insertion heuristic. Lin and Chou incorperated their idea into a simulation program for bike sharing system managers to simulate the resulting routes with and without real world data. As already mentioned, the addition of realistic travel times greatly increses the solution quality for a real world problem. The solutions obtained by using the classical euclidian distance were only competitive in small instances.

A recent work with similar goal is done by Schuijbroek et al.[15], who approach the problem differently. Most importantly, the evaluation of target values is included into the problem. The first step in finding a solution is to find stations which are not self-sufficient for a certain level of user satisfaction. This level is expressed by a target range rather than a fixed target value and is used as a hard constraint on the objective function. To satisfy these target values bikes are moved by multiple vehicles without the usage of a central depot; user activity is assumed to be negligible and therefore they are also working on the static case. Schuijbroek et al. implemented three different approaches to solve the problem. The first one is a pure Mixed Integer Programming approach, which they primarily used for benchmark reasons, since it is not applicable for instances with more than 50 stations and three or more vehicles. Their second approach is a Cluster-first, Route second heuristic. The idea is to group the unsatisfied stations into individual clusters and then solve the those clusters seperately. Since the satisfaction of stations is a hard constraint the minimization only considers the tour length of each cluster. In the clustering phase the algorithm tries to create feasible clusters with a minimal tour length. To minimize the tour length for the whole problem it would be necessary to know the tour length of each possible cluster of stations in advance and so they designed an algorithm to estimate the routing costs for each cluster. The estimated tour length is set to be the value of the Maximum Spanning Star, which is a Spanning Tree with depth one, of all stations in the cluster. With that assumption in mind it is clear that the found clusters may be imperfect, because the assumption about the routing costs is imperfect as well. Based on the solution and the knowledge about their optimal routing costs additional cuts can be added to the clustering to converge towards the optimal solution. To evaluate all three approaches (MIP, cluster, cluster+cuts), real life data from Boston and Washington is used. The data from Boston contains 60 stations with about 10 not self-sufficient stations and two to three vehicles. They observed that the cluster+cut approach outperformes MIP on average by about 510% with two vehicles. Important to know is that the cluster+cut approach was granted a total running time of 60 seconds, while the MIP approach was allowed to run for two hours. When three vehicles are used, cluster+cut is even stronger: 1525%! The instances from Washington consist of 135 stations and between 11 and 25 insufficient stations and 5 vehicles. Those instances are already too complex for the pure MIP approach, so cluster and cluster+cut are compared. On average the extended version outperforms the normal cluster approach by more than 40%.

Pfrommer et al. [10] worked on a different approach to balance a bike sharing system. Their main goal is to minimize the costs necessary to sustain a high service level. Besides the obvious way to minimize the vehicle usage and travel distance, Pfrommer et al. also analiyed the possi-

bility to offer an incentive to customers to help balance the system. Whenever a customer wants to bring back his rented bike, he is probably offered a bonus if he is willing to ride to a nearby (unbalanced) station to park it there. The extent of the bonus is based on the additional travel distance and the gained system balance. Alongside to this method a two step rebalance planning algorithm is designed to direct the vehicles. Their approach to balance the dynamic system is to only calculate routes for the next few steps (e.g. four) or the next few minutes of travel time (e.g. 40 minutes) and reapply the algorithm every ten minutes to account for the changed system. Due to the fact that calculating all possible four station routes is still very time consuming they reduced the algorithm to create only "promising routes". Therefore all stations are evaluated on how promising they are, comparing added system balance to needed travel time. The best n stations are then used to create the set of initial stops for a "promising route". Repeating that step leads to a reduced tree of possible routes. In a second step each of those routes is evaluated by calculating loading instructions with a quadratic program. Finally the route adding the most utility to the system is chosen. For handling multiple vehicles they choose to not co-optimize the trucks but rather optimize them sequentially so that routes from other vehicles are used as known facts. Their whole approach is based on a set of historical real world data from London and therefore tested on this information as well. Three consecutive days were simulated with various settings for the number of trucks and the amount of incentives. The not so suprising results are, that an increase in trucks or incentive values result in a much more balanced system. However as the service level increases the addition of more trucks or incentives becomes less and less effective. Another interesting observation from the simulation is that during weekend the offered bonuses could be enought to keep an acceptable service level, while during workdays the usage of trucks is absolutly necessary, because people value their time higher than the offered incentive.

An important work was done by Meunier et al. [9]. The paper discusses the static case of the Single Vehicle One-Commodity Capacitated Pickup and Delivery Problem - removing the complexitiy of multiple vehicles interfering with each other. Most importantly they propsed a polynomial algorithm to caluclate optimal loading instructions for a given vehicle route, allowing them to focus on the vehicle routing problem. This idea is also essential for this thesis and is described in more detail in section 3.3. To solve the remaining Vehicle Routing Problem two different mixed integer linear programming relaxations to obtain lower bounds for the original problem are introduced. The second relaxation is then used in a branch-and-cut approach. This approach uses linear programming to solve the problem with a subset of the constraints. When a solution is found it is checked against the whole set of constraints if any of them is violated by the current solution. If a broken constraint is found, the linear programm is extended by this constraint. If no broken constraint was found, the cutting part is finished and the branching begins. This means, that if any of the variables in the current solution is fractional, the problem is branched into two new problems. One of them is extended with a constraint stating that the fractional variable v must be $v \leq \lfloor \textit{current value} \rfloor$ and the other one requires $v \geq \lceil \textit{current value} \rceil$. Next, a tabu search algorithm is defined. In contrast to a classical local search this algorithm also allows non-improving steps during its search. To avoid the resulting possibility of cycling between multiple solutions a list of recent steps is stored in a tabu list. In the paper, the tabu list contains the removed arcs and their position in the solution. This allows the possible reinsertion

of a specific arc in a different spot. For the initial solution a bi-criteria heuristic is used. The heuristic first considers stations which can be "closed" with a single move and takes the one nearest to the current position of the vehicle as the next step of the route. A station is "closed" when its amount of bikes is equal to the target value. If no station can be closed the second criteria is used to determine the station with the greatest benefit towards the objective function. For evaluating their approaches they compared the results of the tabu search starting fromt he bi-criteria construction heuristic and the tabu search starting from the result of the branch-and-cut approach. They noticed that the tabu search works efficient on small and medium instances up to 60 stations but lose extremly fast with bigger instances. For 60 stations, the gap between the lower bound from the branch-and-cut method and the result of the tabu search is around 3% on average over all different vehicle capacities. This gap increases to around 23% for 100 station instances.

CHAPTER 3

Algorithm for the Dynamic Balancing Bike Sharing Systems

To solve the problem introduced in chapter 1.2, a Variable Neighborhood Search (VNS) was chosen. This metaheuristic was summarized by Hansen and Mladenovic in 1997 [5] and aims to provide a schema for solving optimization problems. One of the problems with optimization problems is that the solution space S - the set of possible solutions - is eincredibly big. Another one is the complexity of the problem at hand. The complexity of the problem presented in this paper is NP-hard. This means that it is believed that no deterministic algorithm exists, which can solve such a problem in a polynomial amount of computation time.

Instead of evaluating each possible solution, a single solution meta heuristic approach constructs an initial solution x and tries to improve it using a neighborhood structure. This structure is defined as a function $N(x)$, which assigns to each solution $x \in S$ a set of neighbors $N(x) \subseteq S$. A neighbor $y \in N(x)$ differs from solution x by applying a simple move operator on x. A solution x can now be improved with respect to this neighborhood structure with the simple local search algorithm shown in Algorithm 1.

Algorithm 1: Local neighborhood search

input : a solution x
output: a improved solution x with respect to the neighborhood

```
1 repeat
2 |   choose best neighbor y ∈ N(x)
3 |   if obj(y) ≤ obj(x) then
4 |   |   x ← y
5 |   end
6 until no improvement was found
```

The basic idea of the VNS is based on this local search algorithm and three facts about these neighborhoods listed in Hansen et al. [5]:

Fact 1
A local minimum with respect to one neighborhood structure is not necessarily a local minimum with another structure.

Fact 2
A global minimum is a local minimum with respect to all possible neighborhood structures.

Fact 3
For many problems a local minimum with respect to one or several neighborhoods are relatively close to each other.

Figures 3.1 and 3.2 illustrate a potential objective function for two different neighborhoods. One can see that the local mimum ① for neighborhood A is not a local minimum for neighborhood B. On the other hand both neighborhoods share a minimum ②. Since none of the two neighborhoods can further improve the solution, it is a global minimum with respect to those two neighborhoods.

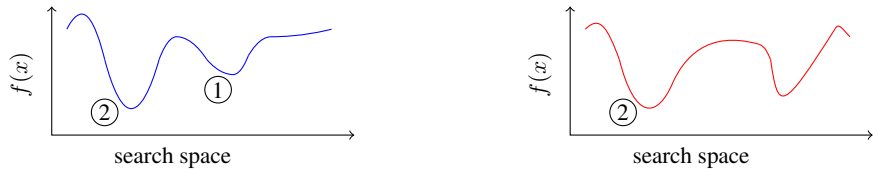


Figure 3.1: objective functions for neighborhoods A and B

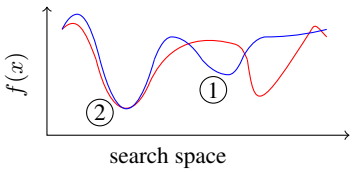


Figure 3.2: combined objective function for neighborhoods A and B

Hansen et al. [5] distinguish the usage of information into three kinds of VNS variants: deterministic, stochastic and both. The deterministic approach (Algorithm 2) is called Variable Neighborhood Descent (VND) and focuses on finding the local minimum for all the neighborhood structures based on one single initial solution x and a set of neighborhoods N_k for $k = 1, \ldots, k_{max}$.

The problem with this approach - and basically with all local search algorithms - is that the quality of the final solution strongly depends on the initial solution, because this algorithm does

Algorithm 2: Variable Neighborhood Descent (VND)

input : A solution x
input : a set of neighborhoods N
output: An improved solution x with respect to the neighborhoods in N

```
1  k ← 1
2  repeat
3  |  choose best neighbor y ∈ N_k(x)
4  |  if obj(y) ≤ obj(x) then
5  |  |  x ← y
6  |  |  k ← 1
7  |  else
8  |  |  k ← k + 1
9  |  end
10 until k = k_max
```

not have the possibility to escape local minima. Consider Figure 3.3 showing the combined objective function for various neighborhoods. Solution A marks the initial solution and since the above algorithm only moves towards better solutions, it can only move downhill to the local minimum without a chance to find the global minimum.

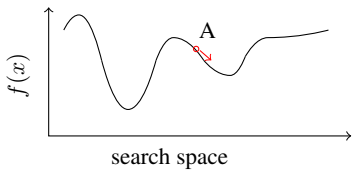


Figure 3.3: a local search sinking into a local optimum

The stochastic approach (Algorithm 3) tries to do the same with the help of probability and is called Reduced VNS (RVNS) by Hansen et al. [5].

According to Hansen this variant is useful when operating with very large instances, since it only needs to evaluate one individual instead of the whole neighborhood structure. For the stopping criteria used in the algorithm a maximum number of iterations without improvement is adviced.

The final approach (Algorithm 4) combines those two into the Basic VNS (VNS) method. This method capitalizes on the probability of the RVNS to escape local minima and the power of a local search (like VND) to find the said minimum.

The idea behind Algorithm 4 is to use neighborhoods for the operation in line 4, which are capable of making big moves in the search space. Those big moves usually come with the price of additional running time. Since the algorithm is not evaluating the whole neighborhood but only one random instance, that is not a problem. A useful stopping criteria could once again be a set number of iterations or an amount of nonimproving iterations.

Algorithm 3: Reduced Variable Neighborhood Search

input : a solution x
input : a set of neighborhoods N
output: an improved solution x with respect to the neighborhoods in N

```
while stopping criteria not met do
    k ← 1
    repeat
        choose a random neighbor y ∈ N_k(x)
        if obj(y) ≤ obj(x) then
            x ← y
            k ← 1
        else
            k ← k + 1
        end
    until k = k_max
end
```

Algorithm 4: Basic Variable Neighborhood Search

input : a solution x
input : a set of neighborhoods N
input : a local search procedure LS
output: an improved solution x

```
while stopping criteria not met do
    k ← 1
    repeat
        choose a random neighbor y ∈ N_k(x)
        y' ← LS(y)
        if obj(y') ≤ obj(x) then
            x ← y'
            k ← 1
        else
            k ← k + 1
        end
    until k = k_max
end
```

For this paper a Basic VNS with an incorporated VND as a local search component is choosen. In the following sections the individual parts for those two algorithms are defined. Chapter 3.1 covers the creation of the initial solution and outlining possible improvements and modifcations. The chapter thereafter 3.2 describes the used neighborhoods for the VNS and the VND part. In addition to the basic VNS algorithm this paper utilizes the idea of Meunier et al.[9] and removes the loading instructions from the original problem. Instead a sub-algorithm is constructed to calculate the optimal set of loading instructions for a given route. The final chapter 3.3 of this section describes two different approaches for this algorithm.

3.1 Construction Heuristic

The first step in finding a good or optimal solution for such a problem is to create a starting solution. Creating a rather good starting solution usually improves the run time of the whole algorithm. The better the initial solution is the less improvement is necessary. A greedy algorithm is used to create this initial solution. The algorithm (seen in Algorithm 5) starts off with empty routes. These routes are extended step by step. For each step the algorithm chooses the best possible extension, with respect to the number of unsatisfied requests and the current situation. For evaluating this best extension the following variables are needed:

- the current location $r \in V$ of vehicle l
- the current load $b \in \{0, Z_l\}$ of vehicle l
- the distance between station r and $s \in V \setminus \{r\}$
- the demands at all stations $v \in V \setminus \{r\}$
- the available bikes b_s^t at station $s \in V$ at time $t \in [0, T_{\max}]$ without considering loading instructions from the visit of vehicle l.
- the time when the next user demand is considered $t_n = \min(t_i \in T | t_i > t)$

For each station which is reachable before t_n the algorithm evaluates how much demand can be satisfied if this particular station would be visited next. This calculation is done once with the assumption of loading bikes onto the vehicle and once with unloading bikes from the vehicle (seen in Algorithm 6). How much demand is satisfiable depends on the number of bikes the vehicle can (un)load without interfering with other vehicles. In the following code fragment the idea of this caluclation is illustrated. Further details are presented afterwards.

The amount of moveable bikes at station s to satisfy user demands in the future is denoted as sat_s. The amount of remaining usable bikes at the station s is denoted by $rest_s$, while $rest_l$ denotes the amount of remaining usable bikes in vehicle l. The term "usable" means that those bikes could be moved without interfering with other vehicles in the future. Moving more than $rest_s$ or $rest_l$ bikes would interfer with other vehicles in such a way that their loading instructions cannot be processed correctly any more.

Starting from the arival time of vehicle l at station s each event in the future is analysed to get the maximal amount of satisfiable demands. Depending on the type of the event the previous

Algorithm 5: Basic Construction Heuristic

input : An instance I
output: A solution S for instance I

foreach $l \in L$ **do**
  $S \leftarrow$ empty solution
  $finished \leftarrow$ false
  while *not* $finished$ **do**
    $c \leftarrow$ `calculateCandidates(`*I,S*`)`
    if *c is empty* **then**
      $finished \leftarrow$ true
    else
      $s \leftarrow$ `selectExtension(`S, c`)`
      `addSelection(`S, s`)`
    end
  end
end

Algorithm 6: Calculate Candidates

input : An instance I
input : An unfinished solution S
output: A set of candidates C to extend solution S

$C \leftarrow \emptyset$
$v \leftarrow$ `calculateVehicleData(`*I,S*`)`
$t_{next} \leftarrow$ `min(`$t \in T | t > v.t$`)`
repeat
  foreach $s \in V$ **do**
    if `IsReachableBefore(`*s*,t_{next}`)` *and* `IsFeasibleReachable(`*s*`)` **then**
      $C \leftarrow C \cup$ `calculateStationLoading(`*s,v,*t_{next}`)`
      $C \leftarrow C \cup$ `calculateStationUnloading(`*s,v,*t_{next}`)`
    end
  end
  $t_{next} \leftarrow$ `min(`$t \in T | t > t_{next}$`)`
until $t_{next} \geq T_num$ *or* $C \neq \emptyset$

values are changed accordingly (pseudo code is shown in Algorithm 7). The amout of bikes can either be increased, by unloading vehicle visits as well as users bringing back bikes, or decreased, by loading vehicle visits and users reting bikes.

Algorithm 7: Calculate Station Loading

input : A station s
input : A vehicle data object v
input : A time t
output: An data object i containing a potential loading instruction

```
1  rest_s ← StationLoad(s,t)
2  rest_l ← v.C − v.b
3  sat_s ← 0
4  R ← ∅
5  foreach e ∈ E =all events at station s do
6      if e.t < t then continue
7      if e = other vehicle then
8          if vehicle is loading then rest_s ← rest_s − e.bikes
9      else // e = a user request
10         if unsatRentRequests(e) > 0 then
           rest_s ← rest_s−unsatRentRequests(e)
11         else
12             add ← min(unsatBringRequests(e), min(rest_s, rest_l))
13             sat_s ← sat_s + add
14             rest_s ← rest_s − add
15             rest_l ← rest_l − add
16         end
17         R ← R ∪ {(sat_s, rest_s)}
18     if rest_s ≤ 0 then break
19     if rest_l ≤ 0 then break
20 end
21 choose best tuple from R based on sat_s
22 i.fulfill ← best.sat_s
23 i.bikes ← min(best.sat_s + best.rest_s, Z_l − v.b)  i.greedy ← i.fulfill/t_{v.r,s}
24 i.distance ← 1/t_{v.r,s}
```

Let the current assumption be that vehicle l is unloading bikes onto station s. The algorithm iterates through all events starting from the arival time t_l of vehicle l at station s. The initial amount of sat_s is obviously 0, while $rest_s = C_s - \overline{a}_s^{t_l}$. When the next event is another visiting vehicle, $rest_s$ has to be reduced by the amount of bikes the other vehicle is unloading to station s, because the alogithm needs to leave this space empty, if it does not want to interfere with the other vehicles routes. If the vehicle is picking up some bikes, nothing changes for the current evaluation, since the maximum amount of bikes the algorithm could unload is still at least capped

by the current amount of empty spaces. When the next event is a user demand it gets more interesting. If it is a bring back demand, the algorithm needs to adjust $rest_s$ similar to the previous case. But if it is a rent demand the number of unsatisfied requests $unsat_s^t$ has to be evaluated. Those are requests where users want to rent bikes, but there are no bikes left on station s. Since the vehicle l is currently unloading bikes to station s those demands could possibly be satisfied. The amount of satisfiable demands for this time interval t is cacluclated by

$$sat_s^t = \min(unsat_s^t, \min(rest_s, b)) \tag{3.1}$$

This value is used to update all the other variables. The number of satisfiable demands sat_s increases by sat_s^t, the number of bikes in the vehicle b as well as the additional available amount of moveable bikes $rest_s$ decreases by sat_s^t. sat_s, t and $rest_s$ are stored as a triple until the algorithm finishes. This end is reached when either $rest_s$ or b are 0 or if there are no further events in the future for station s. From all those possible triples the most satisfying one is chosen. Let SAT_s be the set of all those triples then the most satisfying one is

$$\text{bestAction}_s = \max((sat_s * t) \qquad \forall sat_s \in SAT_s \text{ where } t = \text{index } of SAT_s) \tag{3.2}$$

When the assumption is made that vehicle l is loading bikes from station s a similar logic can be used to calculate the best bike movement. The main difference is that the algorithm looks at the remaining space in the vehicle and the remaining bikes in the station.

At this point for each reachable station the best action is known and the best overall station can be chosen (see Algorithm 8). The best station is defined as having the biggest greedy value, which is calculated by

$$\text{greedy}_s = sat_s / t_{l_v,s}. \tag{3.3}$$

In addition to the greedy value a distance value is calculated by

$$\text{dist}_s = 1 / t_{l_v,s}. \tag{3.4}$$

This values is used if none of the reachable stations can satisfy any demands to choose the nearest station to the current location as the next step. In both cases the amount of additional $bikes_s$ is used to distinct multiple stations with equal greedy or distance values.

When there are no reachable stations left, the tour for the current vehicle is finished and the route for the next vehicle will be constructed.

Possible improvements

The above algorithm could be possibly improved in various ways. One idea is to combine the selection of the "best" action and the selection of the best station into one calculation. Doing so would increase the amount of information available for the calculation and therefore allow a possibly better selection to be made. Looking at all the possible actions may lead to the selection of an action which would have been ruled out due the above algorithm. Another possible weakness of the above algorithm is that it first focuses on stations reachable until the next demand interval for evaluation. Only if none of these stations can be used to statisfy any demands the

Algorithm 8: Select Extension

input : A set of possible candidates C
output: The choosen candidates instruction

```
1  greedyIndex ← 0
2  greedyValue ← 0
3  greedyBike ← 0
4  distIndex ← 0
5  distValue ← 0
6  distBike ← 0
7  foreach i ∈ C do
8      if i.greedy > greedyValue or (i.greedy = greedyValue and
       i.bikes > greedyBike) then
9          greedyIndex = i
10         greedyValue = i.greedy
11         greedyBike = i.bikes
12     end
13     if i.distance > distValue or (i.distance = distValue and i.bikes > distBike)
       then
14         distIndex = i
15         distValue = i.distance
16         distBike = i.bikes
17     end
18 end
19 if greedyValue = 0 then return C[distIndex]
20 return C[greedyIndex]
```

algorithm considers stations reachable until the next demand interval. If an unsatisfied station is located a little bit away from the other stations, it may not be possible to reach it within one interval. As long as there are stations reachable until the next interval, with unsatisfied demands, it may not be evaluated at all. The graph in Figure 3.4 illustrates such an example.

This could be improved by looking at each station no matter if they could be reached prior to the next interval or not. This fact would have to be included into the greedy and distance value calculation to account for the increased effort to reach that special station. Removing that filtering would increase the runtime of the algorithm by a small amount. Most of the time the distant station would still not be selected, since it is more expensive to go there than to some other station with similar amount of satisfiable demands. Additionally, a real world bike sharing system can be assumed to try to distribute their stations evenly around the city map anyways.

Possible modifications

One possible modification for the above algorithm is the support for parallel route construction. The algorithm is currently constructing one route at a time and starts the next one when the

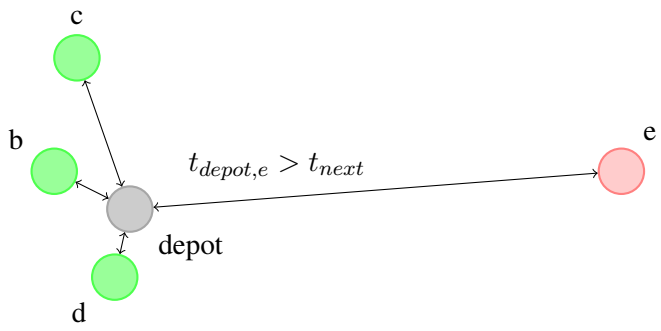


Figure 3.4: a station which is usually not reachable until the next interval

previous one is finished. A parallel construction will bring up a different initial solution since the values and variables for deciding a next step will be different. In addition it would ease up the calculation of possible next steps. The algorithm could simply ignore all stations which are already visited by some other vehicle in the current time frame. Or at least it only needs to care about a vehicle visit in the current time frame and not in the next one, since no vehicle is currently visiting anything in the next time frame. All in all it may produce an even better initial solution. Another possible modification could be to add an α value to the selection function. This α value would be used to "randomize" the selection of the next step. The algorithm could randomly choose one of the stations with a greedy value not more than $\alpha\%$ away from the best solution. This would allow to use this construction heuristic in a GRASP (greedy randomized adaptive search procedure [6][12]) solution approach.

3.2 Neighborhoods

VND Neighborhoods

In this section the various neighborhoods used for the VND part of the algorithm are explained. Most of them were adopted from Rainer-Harbach et al. [11] and have proven to be viable neighborhoods in Vehicle Routing Problems or have been specifically designed by Rainer-Harbach et al. for the Balancing Bike Sharing Systems Problem. The neigbhorhoods are used in this specific static order. Only INSERT-SAT is discarded since it showed to be counterproductive, since the number of satisfied stations increases with the solution quality. Therefore this neighborhoods running time increased continuously with little to no increase in objective function.

Remove visit (REMOVE): This neighborhood works on a single route and tries to remove each of the visits indidivually. Its goal is to remove visits which do not provide anything useful for the overall solution quality. Consider Figure 3.5 representing the route of a vehicle. The number on the arcs between the visits represents the current vehicle load. The outgoing arcs of station 4 and 5 respesent the amount of unsatisfied pick up requests at that station. At station 3 the vehicle is picking up 2 bikes, which are then carried to station 5 without satisfying any requests. Removing that node would yield an overall better route.

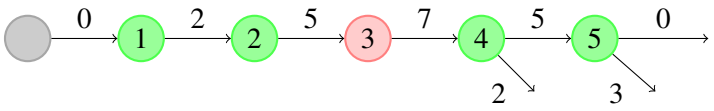
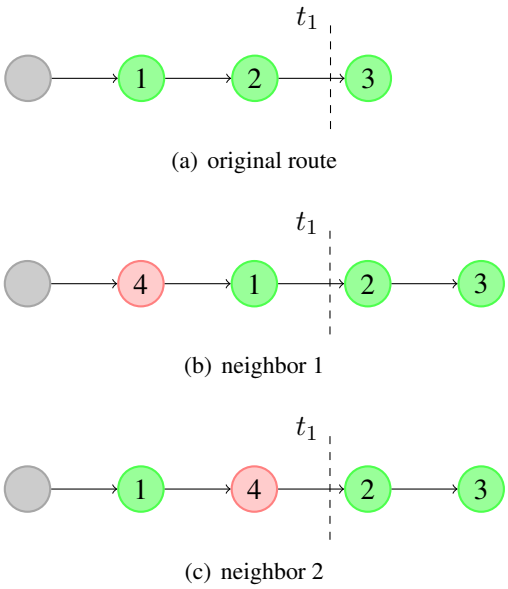


Figure 3.5: a suboptimal visit at station 3

Note that one could argue that station 1 does not provide anything to the solution as well and could be removed instead of station 3. However, the neighborhood removes the stations which results in the shortest route possible.

Insert unsatisfied requests (INSERT-UNSAT): This neighborhood also works on a single route. First, all the unsatisfried requests in the current solution are collected. Then the neighborhood tries to insert a visit to the stations where these requests occur prior to the request time. This is done for each available route and each available insert position. In the following example, station 4 has an unsatisfied request at time t_1, which is not handled by the shown route in 3.6(a). Routes 3.6(b) and 3.6(c) are two neighbors, which are evaluated through this neighborhood.



(a) original route

(b) neighbor 1

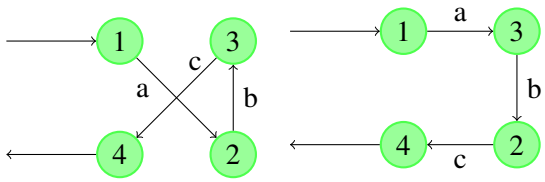
(c) neighbor 2

Figure 3.6: original route and two of its neighbors through INSERT-UNSAT

Note that this is just a simplified example and that it may be possible that visiting station 4 prior to t_1 would push both other visits over the edge of t_1.

Intra-route 2-OPT (INTRA-2-OPT): This is the first neighborhood restructuring the solution rather than changing the used stations. It operates on one individual route and performs the well know 2-OPT operation introduced by Croes et al. [3] for the Traveling Salesman Problem. The idea of this optimization is to swap two arcs, which are overlapping in the plane of the problem yielding a shorter overall route. Figure 3.7 shows the stations and the vehicle route in a

geographical viewpoint before and after the 2-opt is done. As one can see the endpoint of arc a becomes c's startpoint and vice versa. Note that the orientation of arc b needs to be reversed to create a valid route again. This is true for each arc between arcs a and c.

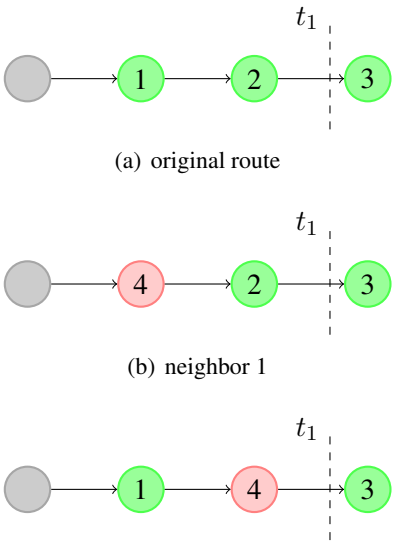


(a) original route (b) after 2 opt

Figure 3.7: an example of a 2-opt swap

With this neighborhood each possible pair of arcs is used for a 2-opt operation. The resulting route is then evaluated regarding its objective value compared to the original route.

Replace visit (REPLACE): This neighborhood aims to combine the neighborhoods REMOVE and INSERT-UNSAT. Especially in very tight routes it may not be possible to insert another visit, without exceeding the time budget, although it might be a good move to replace one visit with another. The neighborhood evaluates all possible replacements of current visits with those of still unsatisfied requests. Using the example from INSERT-UNSAT and executing REPLACE instead would result the following two neighbors.



(a) original route

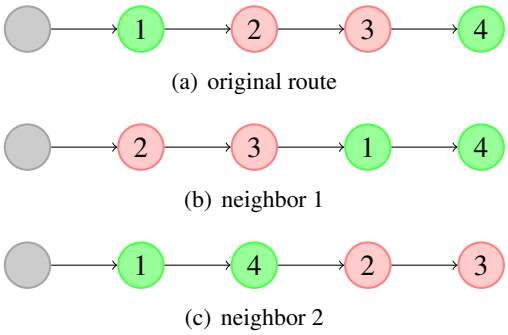
(b) neighbor 1

(c) neighbor 2

Figure 3.8: original route and two of its neighbors through REPLACE

Intra-route OR-OPT (INTRA-OR-OPT): This neighborhood tries to use good subroutes to produce better solutions. It moves a subroute of one, two or three consecutive visits to a

different location in the current route. This allows visits which benefit much from each other - either because their stations are close to each other or they satisfy each others needs - to stay together in the resulting route. In Figure 3.9 the nodes 2 and 3 are kept together and moved around in the route.



(a) original route

(b) neighbor 1

(c) neighbor 2

Figure 3.9: original route and two of its neighbors through INTRA-OR-OPT

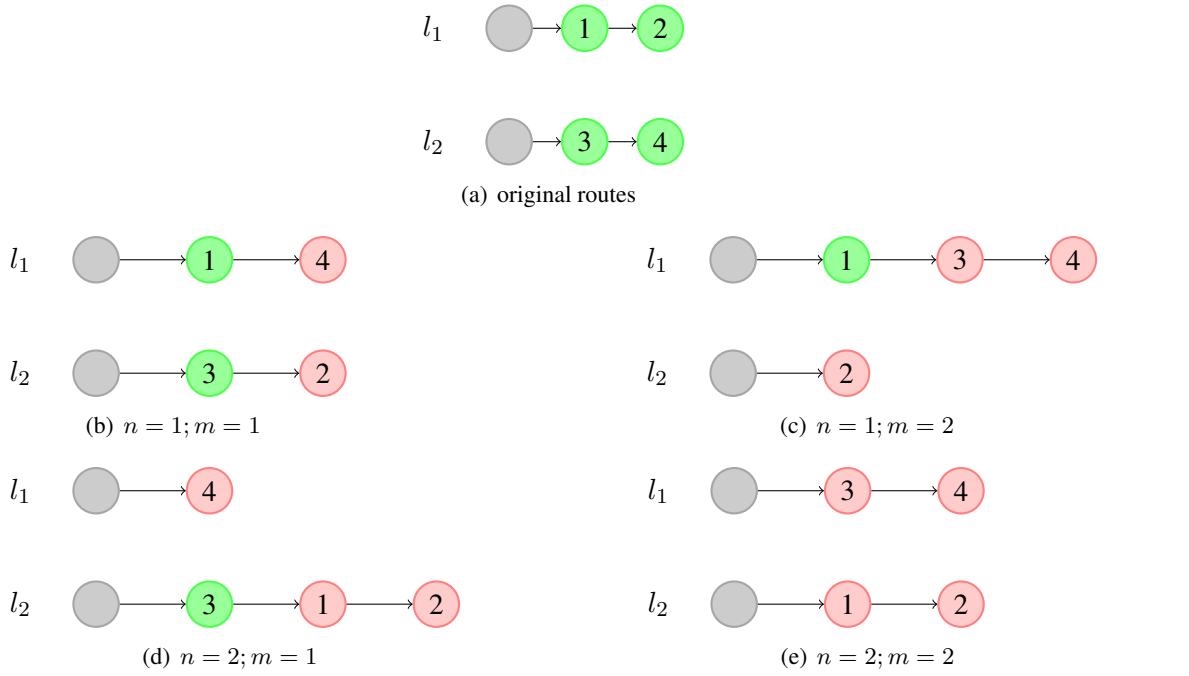
Note that the neighborhood evaluates all possible blocks of sizes one, two and three at each possible insert location.

Inter-route 2-OPT STAR (INTER-2-OPT-STAR): This is the only neighborhood working on multiple routes. Therefore it will not yield any improvements for instances with only one vehicle. For multiple vehicle instances the last n visits of one routes are swapped with the last m visits of another route. Each possible combination of two routes are evaluated with n and m ranging from 1 to the actual length of the respective routes. Figure 3.10 shows all possible neighbors for two short routes.

Based on the example it is clear that there are a lot of possible routes to evaluate in this neighborhood. Therefore the algorithm breaks the search for some n if the addition of m makes the route infeasible with regards to the time budget. If $m - n$ additonal visits exceed the time budget $m - n + 1$ additional visits are not going to be any different.

Intra-route 3-OPT (INTRA-3-OPT): This neighborhood should be consideres as a fall-back. It evaluates many posibilities, which are left after iterating through the other neighborhoods. The given route is seperated into four parts a, b, c, d and reassembled as a, c, b, d. Note that parts a and d can be empty, while parts b and c need to have minimum length of 4 each. With this restriction the neighborhood avoids neighbors which were already evaluated through INTRA-OR-OPT. Figure 3.11 showes one exemplary neighbor.

Insert satisfied Stations (INSERT-SAT): This last neighborhood is basically the same as INSERT-UNSAT, unless it tries to insert already satisfied requests rather than unsatisfied ones. The goal is to insert satisfied stations, so they can provide additional bikes or spots, which in turn let the vehicle satisfy other requests down the road. Unfortunatly this neighborhood showed to not be very effective. Even as first overall neighborhood it does not yield very good results. In addition to its poor effect on the objective function, its time consumption increases, since the better the solutions become the more satisfied requests are available to be inserted.



(a) original routes

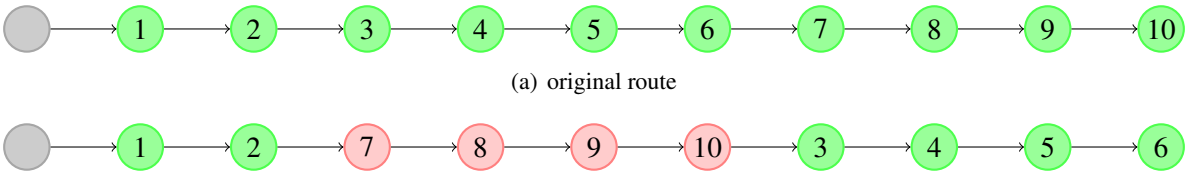
(b) $n = 1; m = 1$

(c) $n = 1; m = 2$

(d) $n = 2; m = 1$

(e) $n = 2; m = 2$

Figure 3.10: original route and its four neighbors through INTER-2-OPT-STAR



(a) original route

(b) exemplary neighbor with $a = \{1, 2\}; b = \{3, 4, 5, 6\}; c = \{7, 8, 9, 10\}; d = \{\}$

Figure 3.11: original route and an exemplary neighbor with INTRA-3-OPT

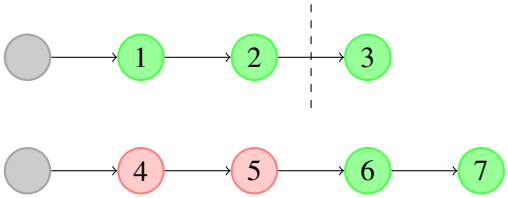
VNS Neighborhoods

For the VNS part of the algorithm three different neighborhoods are used. Each of them can be modified by a parameter, increasing the number of neighborhoods effectively to up to 18 (each parameter is used with 6 different values).

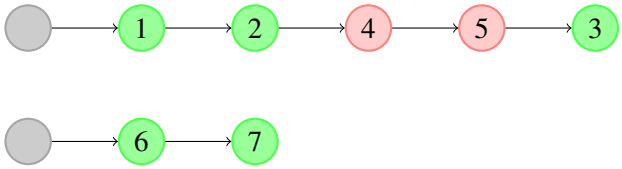
Remove stations (p): This very basic neighborhood removes a certain stop from a route with the given probability p. For each visit of each route in the solution this probability is used to determine if this visit should be removed. Through the parameter p the probability to remove a station constantly increases in six steps. Initially p is set to 0.1. With each subsequent neighborhood p increases by 0.04 stopping at 0.3. So in the final call nearly $\frac{1}{3}$ of the stations are going to be removed from the routes.

This is the only neighborhood called for all solutions. The other neighborhoods are only useful when more than one vehicle is involved.

Move sequence (l): In this VNS Neighborhood a fragment of stops are randomly moved from one route to another. The values needed for this operation are determined by randomization: the source and target routes of the fragment as well as the starting position in the source and the insert position in the target. Parameter l controls the maximum length of the moving fragment, but the real length is still randomly choosen between 1 and l, only limited by the length of the source route. With a probability of 0.1 the moved fragment is inserted in reverse order into the target route. Figure 3.2 illustrates the operation.



(a) randomly chosen routes, the chosen fragment is marked, as well as the target location

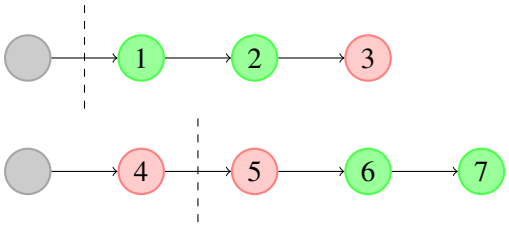


(b) resulting neighbor

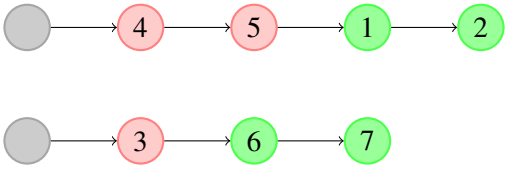
Figure 3.12: original route and an exemplary neighbor with MOVE SEQUENCE

This neighborhood is used with six different values for l. The first five calls are issued with sizes 1 to 5, while the last call uses 99999. This allows the neighborhood to basically try to merge two routes together. Figure 3.2 illustrates the neighborhood with l being set to 3.

Exchange sequence (l): With this neighborhood two randomly selected sequences of two randomly selected routes are exchanged. The parameter l defines the maximal length of the chosen fragments. Similar to the previous neighborhood this one is called with six different values for l increasing from 1 to 5 and finally being set to 9999. Figure 3.2 illustrates the operation with l being set to 2



(a) randomly chosen routes, the chosen fragments are marked, as well as the target locations



(b) the resulting neighbor

Figure 3.13: original route and an exemplary neighbor with EXCHANGE SEQUENCE

3.3 Optimal Loading Instructions

In this section an algorithm for calculating optimal loading instructions based on given routes is presented. The idea to create an own algorithm for this subproblem is coming from Meunier et al. [9]. It is important to note that [9] works on a restricted static problem with only one vehicle - both constraints this paper aims to avoid. For sake of completeness and understanding, the idea of [9] is recapped here: A special flow network is constructed based on the input instance and a given route r for vehicle l using the following rules:

1. add a node for each occurrence of a station v
2. add a node S and a node E for the starting and end point of the vehicle l
3. add arc (v, u) with capacity Z_l if the vehicle is going from v to u in the given route
4. add arc (v_i^j, v_i^{j+1}) with capacity C_v, with j indicating the j^{th} occurrence of station v_i on route r
5. add arc (S, v_i^1) with capacity p_v for the first occurrence of station v_i on route r
6. add arc (v_i^n, E) with capacity q_v for the last occurrence of station v_i on route r

The resulting flow network (as shown in figure 3.14) can be solved by any max flow algorithm. Arcs created by rule 3 are modeling the movement of the vehicle l. The difference between the incoming flow and the outgoing flow on these arcs denote the needed loading instructions. Rule 4 models the remaining bikes for a station, which is visited multiple times by the vehicle. Finally rules 5 and 6 are modeling the initial amount of bikes for each station and the desired target amount respectively.

Example 1 In this example the flow network for an instance with 5 stations and a vehicle route of $a \to b \to d \to a \to c$ is shown. Red arcs are created by rule 3, the black one by rule 4, the blue ones by rule 5 and the orange ones by rule 6.

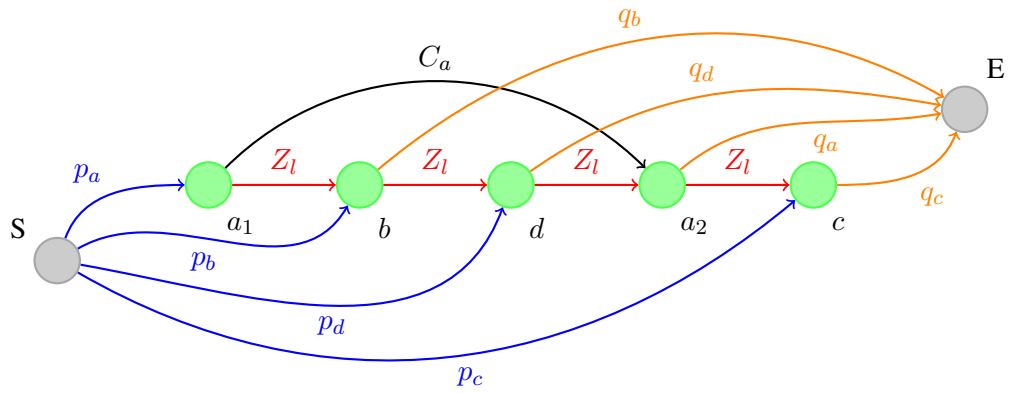
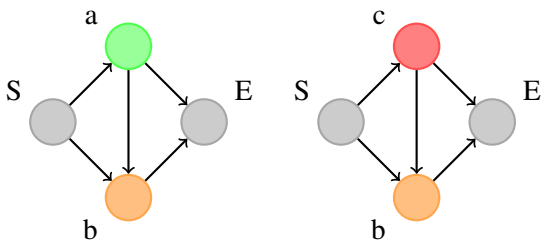


Figure 3.14: resulting flow network for an example instance

Based on this idea a solution for dealing with multiple vehicles and the dynamic case needs to be found. Adding additional vehicles (and their routes) to the problem introduces additional constraints to consider. A naive idea would be to calculate the loading instructions for each vehicle sequentially, which may result in a problem illustrated in example 2.

Example 2



(a) $r_1 = \{a, b\}$ (b) $r_2 = \{c, b\}$

Figure 3.15: optimizing vehicle routes sequentially

Consider, two routes $r_1 = \{a, b\}$ and $r_2 = \{c, b\}$ as seen in Figure 3.15. Station a is already satisfied, while station b is missing some bikes and station c is holding too many bikes. If r_1 is evaluated first it will probably load all bikes at station a and bring them to station b. This would leave r_2 with nothing useful to do since station b is already satisfied. If r_2 is evaluated first it would take bikes from station c and load them to b to satisfy both stations.

To avoid such problems the evaluation of routes sharing at least one station needs to be done in a combined way. Ignoring the time constraint for a single vehicle one could argue that a

sequential evaluation is nothing different than one vehicle handling both routes one after another. But there is a difference. When optimizing them as "one route" (as seen in figure 3.16(b)) both networks are considered when finding a maximum flow, while in the sequential case 3.16(a) only the result of the first network (which cannot be changed any more) is used for the second network.

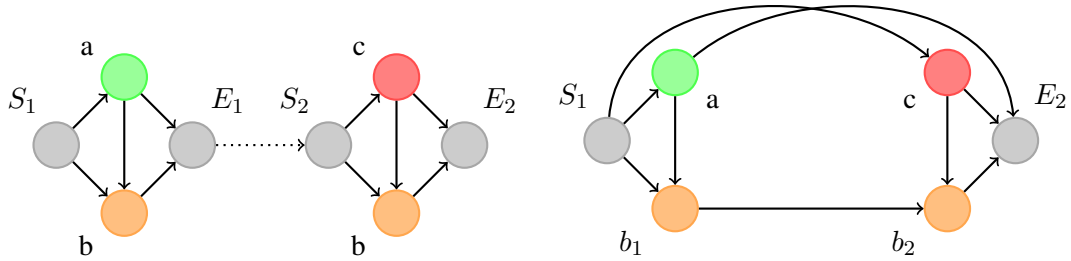


Figure 3.16: combining two routes

Transforming network 3.16(a) into network 3.16(b) can be done by first combining node E_1 with S_2 making the end node of one network the new start node from another network. In a next step this new node can be completely replaced by additional arcs using rule 4 from the initial rule set.

To extend the idea of Meunier et al. [9] for the dynamic case a similar approach can be used. Each time interval of the dynamic problem could be seen as a single static problem. Similar to the above case those subproblems cannot be solved individually because the result of the first interval influences the calculations for the second interval.

Flow network approach

Putting all the above together a modified flow network for the dynamic case with multiple vehicles can be constructed. The following inputs are used:

- a problem instance graph $G_0 = (V_0, A_0)$
- the set of vehicle L
- the set of time intervals $T = \{t_0, \ldots, t_{max}\}$
- a given solution containing a set of routes $r_l = (r_l^1, \ldots, r_l^{p_l})\ \forall l \in L, r_l^i \in V$
- a function to calculate the (absolute) arrival time of a vehicle l reaching its i^{th} station $ta(r_l^i)$

The resulting graph $G_f = (V_f, A_f)$ is constructed by the following rules

1. add a starting node S and a target node D for the whole graph.

2. add a node for demand n at station at each time interval: $n_v^t | t \in T, v \in V$
3. add a node for each visit of some vehicle at a station: $e_v^t | t = \text{ta}(r_l^i), v = r_l^i, l \in L$
4. add arcs with capacity p_v from the starting node S to the starting node of each station: $(S, n_v^{t_0}) | v \in V$
5. add arcs with capacity C_v from the last interval node of each station to the target node D: $(n_v^{t_{max}}, D) | v \in V$
6. add arcs with capacity q_v^t from the starting node S to all bring back demands at a time interval: $(S, n_v^t) | v \in V, t \in T \setminus \{t_0\}, q_v^t > 0$
7. add arcs with capacity q_v^t from all rent demands at a time interval to the target node D: $(n_v^t, D) | v \in V, t \in T \setminus \{t_0\}, q_v^t < 0$
8. add arcs with capacity Z_l from one stop of a vehicle to the next: $(e_v^t, e_u^{t'}) | t = \text{ta}(r_l^{i-1}), t' = \text{ta}(r_l^i), v = r_l^{i-1}, u = r_l^i, i = 2, \ldots, p_l, l \in L$
9. add arcs with capacity C_v from one event at a station (demand or vehicle visit) to the next:

    - Let V_v^{ord} be the set of all nodes n_v^t and e_v^t for station v ordered by increasing t. Further let o^j be the j^{th} element of this set and j^{max} the total amount of elements in the set. Then the needed arcs can be constructed by: $(o^{j-1}, o^j) | o \in V_v^{ord}, j = 2, \ldots, j^{max}$

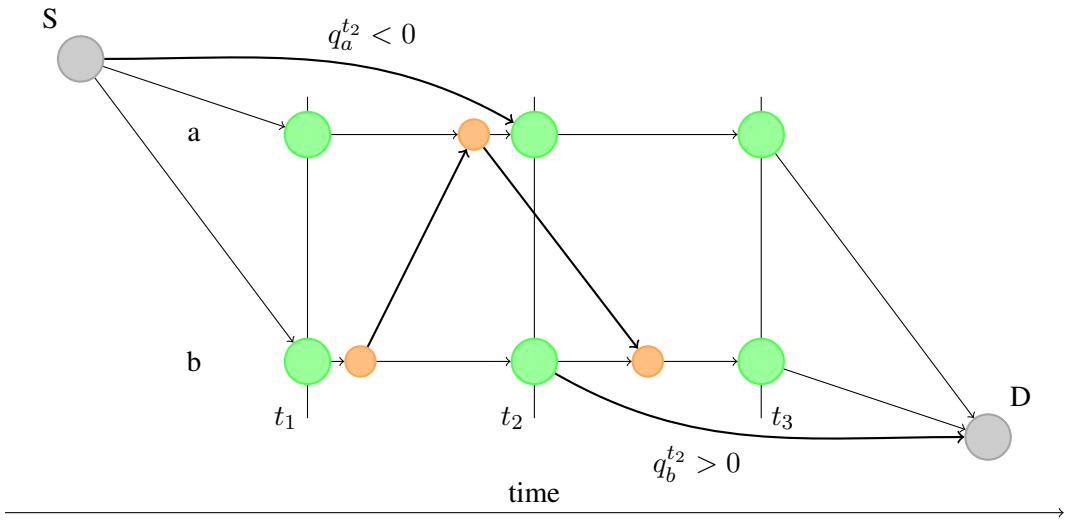
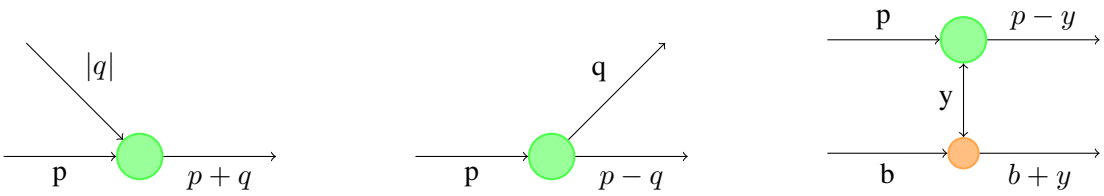


Figure 3.17: an example for a dynamic flow network

Figure 3.17 illustrates such a graph. Most of the nodes in the graph can be classified in one of the following three types. The list gives an explanation for each type and the meaning of each of their arcs.

1. a bring back demand (figure 3.18(a)) with
   - the current load p of station v prior to the demand (ingoing)
   - the amount of bikes q which should be brought back (ingoing)
   - the load of station v after the demand took place (outgoing)
2. a rent demand (figure 3.18(b)) with
   - the current load p of station v prior to the demand (ingoing)
   - the amount of bikes q which should be rented (outgoing)
   - the load of station v after the demand took place (outgoing)
3. a vehicle visit (figure 3.18(c)) with
   - the current load p of station v prior to the visit of vehicle l (ingoing)
   - the current load b of vehicle l prior to visiting station v (ingoing)
   - the load of station v after vehicle l visited (outgoing)
   - the load of vehicle l after it visited station v (outgoing)



(a) bring back demand (b) rent demand (c) vehicle visit

Figure 3.18: illustrations of node types

This problem could be solved by a maximum flow algorithm. Unfortunately it could still produce invalid solutions, since the algorithm needs to make sure that all the arcs from rule 6 and 7 are prioritized. Example 3 illustrates the problem.

Example 3 Let v be a station with $q_v^t < 0$ and an current station load of $b_v \geq -q_v^t$. There would be enough bikes to satisfy the user demand, when interval t happens. Unfortunately the flow algorithm may choose to increase the arc (o^j, o^{j+1}) - the outgoing arc for remaining bikes at the station - instead of the arc (n_v^t, D) - the demand arc -, because it is more benefitial for the maximum flow down the road. Transfered to the real world the system would hold back a number of bikes to satisfy other demands later on. While this may be the optimal solution it is not a realistic one, since the system cannot (and should not) prohibit the rent (or return) of bikes if there are enough bikes (or slots) available.

To solve this issue, one could try to create a specialized max flow algorithm. This approach is not pursued further, since it would be another research topic to construct such an algorithm.

Another idea is to create a repair algorithm to transform infeasible solutions into feasible ones with minimal (or zero) increase in the objective function. This algorithm needs to push back reserved bikes (or slots) up to the point where they would have been used in the first place. Such an algorithm could work as follows:

1. Set t_{curr} to t_0.
2. Find first unvisited, unsatisfied demand q_v^t where $t >= t_{curr}$.
3. If a unsatisfied demand was found, construct a residual network G'; Otherwise end algoritm.
4. Cut all arcs starting or ending prior to t.
5. Include a new target node K.
6. Replace arc (D, n_v^t) in G' with arc (n_v^t, K) using $(-q_v^t - f_{(n_v^t, D)})$ as capacity; or arc (n_v^t, S) in G' with arc (n_v^t, K) and capacity $q_v^t - f_{(S, n_v^t)}$ respectively.
7. Find an augmented path in G' from D to K.
8. If no path exists, continue with 12.
9. If such a path exists reduce $f_{(i,j)}$ along the path according to the size of the augmented flow.
10. If q_v^t is not satisfied, continue with 7.
11. If $t == t_{max}$ end the algorithm.
12. Set t_{curr} to t and start again with 2 .

This algorithm iterates through the graph in a timely order and repairs each infeasible flow at a demand arc. Theorem 1 argues that each possible infeasible solution can be repaired with this algorithm.

Theorem 1. *The proposed repair algorithm always creates a feasible solution from an infeasible maxflow solution.*

Proof. Assume an arbitrary infeasible solution, which holds back an arbitrary number of bikes x at station s at time t instead of satisfying the current user request. According to the rule set above those bikes amplify the flow to the next event for station s. From there on the bikes have three possible ways to go:

stay at station s

When they stay at the station, they will either move on to the next event or to the sink of the flow network if this is the last event for station s. They may even create unsatisfied bring back requests since x additional slots are occupied at station s, ultimatively increasing the objective function.

get loaded onto a vehicle
: When loaded onto a vehicle the bikes will be moved to some different station at some point in time. Since there is no arc from the vehicle routes to the sink it is not possible for the bikes to stay in the vehicle. Sooner or later they will be dropped of at some station v.

used for a user request s
: If the bikes are used for the next user request they are moved along this arc to the sink. Since the amount of additional statisfied request must be $\leq x$ the objective function is not significantly changed.

As one can easily see repeating that process ensures that the bikes are moved to the sink in any case. What one can also see is, that those bikes are never moved along arcs which start prior to t. Therefore it is clear that there must be a set of augmented paths in G' from the sink to the event at station s at time t with a total size of x. □

At this point it seems that a maxflow network approach works for the static case (as in Meunier et al. [9]) but adds a lot of complexity for the dynamic case. Therefore this approach is not continued further. Since the constructed network is used as a base for the LP approach it is essential to see the idea behind this network.

Linear Programming approach

Using the above flow network a linear (integer) program can be constructed to calculate the optimal loading instructions for a given set of routes. For the sake of readability the rules for constructing the network are reformulated in a set notation here.

$G_f = (V_f, A_f)$ with

- $V_f = \{S, D\} \cup V_t$
- $V_t = V_{station} \cup V_L$
- $V_{station} = \{n_v^t | t \in T, v \in V\}$
- $V_L = \bigcup_{l \in L} V_l$
- $V_l = \{e_v^t | t = \text{ta}(r_i^l)\, v = r_l^i, l \in L\}$

and

- $A_f = A_{start} \cup A_{end} \cup A_{demand} \cup A_L$
- $A_{start} = \{(S, n_v^{t_0}) | v \in V\}$
- $A_{end} = \{(n_v^{t_{max}}, D) | v \in V\}$
- $A_{demand} = A_{q^+} \cup A_{q^-}$
- $A_{q^+} = \{(S, n_v^t) | v \in V, i \in T \setminus \{t_0\}, q_v^i > 0\}$

- $A_{q^-} = \{(n_v^t, D) | v \in V, i \in T \setminus \{t_0\}, q_v^i < 0\}$
- $A_L = \bigcup_{l \in L} A_l$
- $A_l = \{(e_v^t, e_u^{t'}) | t = \text{ta}(r_l^{i-1}), t' = \text{ta}(r_l^i), v = r_l^{i-1}, u = r_l^i, i = 2, \ldots, p_l, l \in L\}$
- $A_V = \bigcup_{v \in V} A_v$
- $A_v = \{(o^{j-1}, o^j) | o \in V_v^{ord}, j = 2, \ldots, j^{max}\}$ (see rule 9 from the flow network for definition of V_v^{ord} and j^{max})

In addition decision variables

- $f_{u,v} \geq 0 \, \forall (u, v) \in A_f$

are used.

The LP is then

$$\min \sum_{\forall (S, n_v^t) \in A_{q^+}} q_v^t - f_{S,n_v^t} + \sum_{\forall (n_v^t, D) \in A_{q^-}} -q_v^t - f_{n_v^t, D} + \tau (\sum_{\forall (l,e) \in V_L} y_e^{l^+} + y_e^{l^-}) \tag{3.5}$$

subject to

$$f_{S,n_v^{t_0}} = p_v \qquad \forall (S, n_v^{t_0}) \in A_{start} \tag{3.6}$$

$$f_{S,n_v^t} \leq q_v^t \qquad \forall (S, n_v^t) \in A_{q^+} \tag{3.7}$$

$$f_{n_v^t, D} \leq -q_v^t \qquad \forall (n_v^t, D) \in A_{q^-} \tag{3.8}$$

$$f_{u,v} \leq C_v \qquad \forall (u, v) \in \{A_{end} \cup A_V\} \tag{3.9}$$

$$f_{u,v} \leq C_l \qquad \forall (u, v) \in A_l \tag{3.10}$$

$$\sum_{(u,v) \in \{A_{start} \cup A_{q^+} \cup A_L\}} f_{u,v} = \sum_{(v,w) \in \{A_{end} \cup A_{q^-} \cup A_L\}} f_{v,w} \qquad \forall v \in V_t \tag{3.11}$$

$$y_e^l = f_{u,e} - f_{e,w} \qquad \forall (l, e) \in V_L \tag{3.12}$$

$$y_e^l = y_e^{l^+} - y_e^{l^-} \qquad \forall (l, e) \in V_L \tag{3.13}$$

$$y_e^{l^+} \geq 0 \qquad \forall (l, e) \in V_L \tag{3.14}$$

$$y_e^{l^-} \geq 0 \qquad \forall (l, e) \in V_L \tag{3.15}$$

Equation (3.6) ensures that the initial value of each bike station is fixed. (3.7) and (3.8) limit the maximal flow allowed at the demand arcs. (3.9) and (3.10) limit the flow on all the remaining arcs. Finally (3.11) ensures flow conservation. Constraints (3.12) creates additional variables holding the necessary loading instructions. Since those can be positive or negative (3.13), (3.14) and (3.15) ensure positive values for usage in the objective function. The factor τ needs to make sure that the second part of the objective function is between 0 and < 1. That way a better solution (with less unsatisfied demands) is always prefered over a worse solution with less loading instructions.

This linear program is still able to produce infeasible solutions (See Example 3.19). By adding appropriate weights ω_{t_i} to all the demands and to the objective function as well, the simplex algorithm is forced to satisfy demands if bikes or slots are available. These weights are decreasing along the time axis, so that satisfing early requests is more benefitial to the objective function than saving bikes for later use. Equation (3.16) denotes the extended objective function (LP+).

$$\min \sum_{\forall(S,n_v^t)\in A_{q^+}} \omega_{t_i}(q_v^t - f_{S,n_v^t}) + \sum_{\forall(n_v^t,D)\in A_{q^-}} \omega_{t_i}(-q_v^t - f_{n_v^t,D}) + \tau(\sum_{\forall(l,e)\in V_L} y_e^{l^+} + y_e^{l^-}) \tag{3.16}$$

Example 4

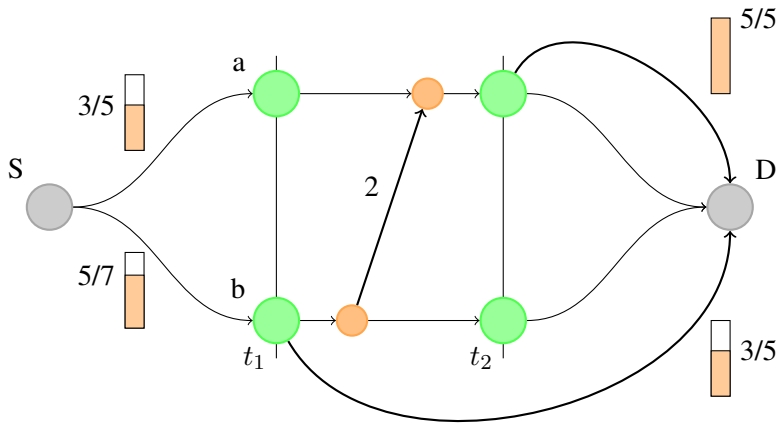


Figure 3.19: illustration of an unrealistic solution

Assume two stations a and b. a holds 3 bikes, while b has 5 bikes. b has a rent-request of 5 bikes at t_1 and a has a rent-request of 5 bikes at t_2. Additionally there is a route from b to a between t_1 and t_2 (see figure 3.19). The initial solution uses 3 bikes for the rent-request of b and keeps 2 bikes to deliver to a for its rent-request. Using the LP extension all the bikes at b would be used to satisfy the rent-request and no bikes are kept to send them to a. In this scenario we reduce the number of unnecessary loading actions.

The question arises if it is necessary to define specific value for ω. Theorem 2 suggests that it is sufficient to choose ω_{t_i} in such a way that they decrease along the time axis of the problem but are ≥ 1 at all times.

Theorem 2. *The above LP, with extension (LP+), is not able to create an optimal solution containing more unsatisfied requests than the solution created by the basic LP as long as ω values decrease along the time axis and be ≥ 1 at all times.*

Proof. Assume $\omega_{t_i} \gg \omega_{t_j}$ when $t_j \gg t_i \forall t_i, t_j \in T$.

To create an additional unsatisfied request LP+ needs to take away (at least) one bike before a pickup-requests takes place. In doing so, the objective function of LP+ is punished with ω_{t_i}. To lower the objective function again the taken bikes needs to be used to satisfy another request. Since our vehicles are only moving forward in time, it is not possible to move the bikes to an earlier station and all further incoming requests have $\omega_{t_j} \ll \omega_{t_i}$. So the LP+ is not taking away the bikes in the first place. If the bikes are not used to satisfy another request the punishment for the pickup is even bigger.

A similar logic can be used for a delivery-request. In this case LP+ is adding another (arbitrary) bike to a station, so the delivery-requests cannot be fully satisfied resulting in a penalty of ω_{t_i}. In the best case this missing bike leads to a state where another delivery-request can be satisfied. This would reduce the penalty by $\omega_{t_j} \ll \omega_{t_i}$. In the other cases it either does not influence the result in any way or it creates another unsatisfied requests because this missing bike should have been picked up later on.

Changing the assumption from the beginning would cause the LP to prioritize the later user demands over the early ones. Additionally we need to specify $\omega_{t_i} \geq 1 \forall t_i \in T$. This ensures that the first part of the objective function is always more valuable than the second part factored by τ. Choosing a value ≤ 1 would make an additional satisfied request probably less valuable than a shorter route. □

Greedy approach

Based on the shown flow network and the presented greedy construction heuristic another approach for calculating the loading instructions was pursued. The idea is to reduce the obvious drawback of an LP appraoch - the runtime - at the cost of not getting an optimal solution.

The greedy heuristic to calculate the loading instructions is entirely based on the already presented methods used in the constructions heuristics. Still there is one main difference between those two: the routes are already given and cannot be changed anymore.

Algorithm 9 illustrates the idea. Important to note here, is that the method *getNextEvent* is returning all events from all routes in their chronological order. This is done so that the calculation of loading instructions does not need to care about interfering with other vehicles in the future, since those visits are all set to have a loading instruction of 0. Another adventage of this approach is that the algorithm can keep a local copy of the current load of each station and each vehicle. A sequential approach on the individual routes would require to update all the supplementary structures, which is a rather costly operation. To do so L and V are initialized with the initial vehicle and station load, while operations *updateVehicleData* and *updateStationData* are executing the choosen loading instruction on the local copy of the vehicle loads and station loads respectively.

One problem with this approach remains. Since the heuristic is always moving as many bikes as possible it may occur that there are bikes left in the vehicles at the end of their routes. While this is not a problem itself it still reduces the meaningfulness of a comparison between the LP and the greedy approach. Algorithm 10 shows how the resulting solution is fixed, if there are any bikes left in the vehicles.

The algorithm is iterating the solution from the back. Method *getPrevEvent* is the opposite to

Algorithm 9: Calculate Greedy Loading Instructions

input : An instance I
input : A solution S with routes but without loading instructions
output: A finished solution with loading instructions

```
1  currL ← initializeVehicleData
2  currV ← initializeStationData
3  e ← getNextEvent(S)
4  while e ≠ NULL do
5      foreach visit ∈ e.visits do
6          if visit.l = −1 then updateStationData(currV, visit.bikes)
7          else
8              loading ← calculateStationLoading(visit.s, currL[visit.l],
               visit.t)
9              unloading ← calculateStationUnLoading(visit.s, currL[visit.l],
               visit.t)
10             best ← selectExtension(S, {loading, unloading})
11             updateVehicleData(currL, best.bikes)
12             updateStationData(currV, best.bikes)
13         end
14     end
15 end
```

getNextEvent and provides the events in timely reversed order.Only pickup actions are modified. Each left over bike was previously picked up on the route of this vehicle. Since it is still available in the vehicle at the end of the route, it was picked up unnecessarily in the first place and can safely be left at the station.

Algorithm 10: Repair Solution

input : An instance I
input : A solution S
output: A solution without bikes left in the vehicles at the end of their routes

```
1  currL ← initializeFinalVehicleData(S)
2  finished ← 0
3  foreach l ∈ I.L do
4  |  if currL[l] = 0 then finished++
5  end
6  while finished < I.L do
7  |  e ← getPrevEvent(S)
8  |  if e =NULL then break
9  |  foreach visit ∈ e.visits do
10 |  |  l ← visit.l
11 |  |  if l ≠ −1 and currL[l] > 0 then
12 |  |  |  if visit.y > 0 then
13 |  |  |  |  yMod ← min(currL[l], visit.y])
14 |  |  |  |  visit.y ← visit.y − yMod
15 |  |  |  |  currL[l] ← currL[l] − yMod
16 |  |  |  |  if curr[l] ≤ 0 then finished++
17 |  |  |  end
18 |  |  end
19 |  end
20 end
```

CHAPTER 4

Computationl Results

This section should illustrate the results of the proposed algorithm for a set of instances. Due to the lack of a second algorithm for comparison, diverse variants of the VNS algorithm are compared with each other. In the next section the testing instances are described, followed by the different VNS variants. Finally in section 4.3 the results of each of the variants are compared.

4.1 Instances

Instead of creating own instances the set of instances from Rainer-Harbach et al. [11] is used. This set can be found at the web page of the Institute of Computer Graphics and Algorithms of the university of technology in Vienna[1] and was constructed as follows:

the set of stations V
: consists of 92 real world stations from CityBike Wien[2] and 664 fictional stations which were evaluated by the Austrian Institute of Technology (AIT)[3] to be located at realistic locations throughout the city of Vienna.

travel time $t_{u,v}$
: is calculated by the average travel time from u to v plus an estimated amount of time for parking the vehicle and performing the (un)loading operation.

initial load p_v
: is taken from a snap shot of the real stations. For the artificial stations the value is taken randomly from a real station in a randomly choosen snap shot.

time intervals T
: are set to happen every hour for the duration of one work day (lasting 8 hours), resulting

[1] Instances: https://www.ads.tuwien.ac.at./w/Research/Problem Instances
[2] http://www.citybikewien.at
[3] http://www.ait.ac.at

in $8n$ demands. This can be easily extended to intervals happening every 15 minutes, increasing the number of demands to $32n$.

demands q_v^t
are generated in two steps. At first each station is assigned a type, indicating if the station load tends to increase or decrease over the course of a day. This is not to be confused with monotonicity, since the station load still increases/decreases but is more likely to be below/above the starting value at the end of the day. The probability for each type is 0.5. In the next step the demand values for a single station v are calculated using a beta distribution on the interval of $[-C_v/2, C_v/2]$. Parameters are set to $p = 20$ and $q = 25$ for a station tending to increase its load and to $p = 25, q = 20$ respectively.

Based on this information a single instance with n stations is created with the following "algorithm".

1. Select the first station randomly out of the set of 756 stations.
2. Add the n nearest stations based on the Euclidean distance to the initial station.
3. Select one station randomly from the set of $n + 1$ stations to be the depot.

For each size n, 30 independent instances are created and available in the mentioned set. For the purpose of this paper instances with 60, 90 and 120 stations are primarily used. This decision is based on the fact that the city of Vienna is currently (as of 2012) maintaining 102 stations. As for the vehicle fleet, a homogeneous fleet with capacity of $Z_l = 20 \; \forall l \in L$ is assumed. The size of the fleet varies in $|L| \in \{1, 2, 3, 5\}$ to show the effects of additional vehicles on the solution quality. The range of this set is based on the real world scenario of CityBike Wien, which uses two vehicles to balance the system. Finally the time budget for the fleets is set to $\{2, 4, 8\}$ hours to show the effects of additional time on the solution quality. For the objective function each unsatisfied request increases the value by 1, while the milage is added with a factor of 1^{-5} to always prioritize solutions with less unsatisfied requests over shorter routes.

4.2 Variants

Executing a single algorithm on a set of test instances does not yield much insight about the solution quality produced by said algorithm. Therefore three different variants of the VNS algorithm are tested, to overcome that fact to a certain degree.
The variant D is using all of the available VND neighborhoods except INSERT-SAT. With these VND neighborhoods in use the algorithm tries to improve a specific solution as far as possible before trying to escape the current minimum with a VNS step resulting in a "depth-focused" search.
Variant W is only using REMOVE and INSERT-UNSAT. Those two neighborhoods consume the least computational time to explore them completely. Therefore the algorithm has more time to explore the search space with the VNS neighborhoods resulting in a "width-focused" search.
During the evaluation of the above two variants a third variant emerged (variant G). Instead

of using the LP it is using the greedy approach to calculate the loading instructions. Since the greedy algorithm is expected to be faster than the LP it is using all of the neighborhoods to offset the fact that it is not necessarily calculating the optimal set of loading instructions.

4.3 Results

The proposed algorithm was implemented in C++ using GCC 4.6.3. Each of the test runs was executed on a single core of an Intel Xeon E5540 machine with 2.53 GHz and 3 GB RAM. The LP part of the algorithm was solved with CPLEX 12.4 with default settings and the restriction to only use one thread. The algorihm was stasted 10 times for each of the 30 instances with every possible combination of the above parameters. Each of those runs was granted a time limit of one hour to complete.

The most important observation from tests is that nearly all runs are terminated by the deadline of 1 hour CPU-time. Only some instances with a time budget of two hours and a fleet size of 1 or 2 vehicles are finished earlier. In each of those instances the best solution was found very quickly, which means that the initial greedy solution was improved a little bit but could not be improved any further. As soon as the algorithm has enough "room" to be able to improve its solution through VND and VNS neighborhoods the running time always hits the deadline of two hours. This is due the usage of a LP to evaluate the optimal loading instructions, which is done for every evaluated solution. Tables 4.1, 6.1 and 6.2 show the average total cpu time T, the average cpu time used in CPLEX T_{cplex} and the average amount of solutions evaluated in this time. Each of those solutions was first constructed through one of the neighborhoods, so completing a constructed routing-solution with (optimal) loading instructions increases the time consumption for evaluating one neighbor by a factor of δ.

$$t_{create} = T - T_{cplex} \tag{4.1}$$

$$\delta = \frac{T_{cplex}}{t_{create}} \tag{4.2}$$

δ increases with the complexity of the instance up to a factor of over 100. These numbers can be seen as a lower bound, since T_{cplex} counts the total time consumption of CPLEX but not every solution created is evaluated with the LP. Some solutions get discarded earlier because their route length exceeds the maximum time budget. So t_{create} also includes solutions which were created but not evaluated with the LP, while T_{cplex} only counts evaluated solutions. Therefore t_{create} would be even smaller when only containing evaluated solutions. This observation is independent of the two VNS variants although it leads to the fact that the factor is higher with variant D, where more solutions are produced through VND neighborhoods and therefore more solutions get discarded prior to CPLEX.

Based on this observation it is somewhat clear that variant D is performing a little bit better than variant W. Using the limited time left outside of CPLEX is better spent improving the current solution with different neighborhoods instead of trusting probability to find a better solution. This behavior is reflected in the average solution quality as well as the best and worst solutions found. Tables 4.2 and 4.3 show the results for all instances with 90 and 120 stations

instances		variant W				variant D			
$\|L\|$	$\widehat{t}$	T	T_{cplex}	δ	#sol	T	T_{cplex}	δ	#sol
1	120	2760.34	2457.18	8.10	673,417	3439.37	3115.04	9.60	832,993
1	240	3604.16	3437.39	20.61	715,188	3606.46	3454.65	22.75	697,542
1	480	3615.83	3534.97	43.71	516,473	3624.94	3547.61	45.87	495,902
2	120	3555.25	3175.44	8.36	718,853	3603.72	3302.39	10.95	692,780
2	240	3610.23	3497.92	31.14	481,133	3617.88	3516.51	34.69	465,191
2	480	3629.50	3575.26	65.91	327,966	3661.18	3613.88	76.40	311,989
3	120	3602.57	3290.07	10.52	572,972	3605.46	3358.03	13.57	564,860
3	240	3618.08	3529.55	39.86	374,724	3625.03	3548.88	46.60	356,620
3	480	3643.98	3589.15	65.46	265,843	3705.80	3660.15	80.18	246,340
5	120	3603.46	3344.65	12.92	443,763	3611.97	3423.82	18.19	432,210
5	240	3632.67	3563.40	51.44	275,440	3649.63	3601.12	74.22	250,940
5	480	3666.32	3615.23	70.75	239,615	3720.13	3681.50	95.29	221,262

Table 4.1: CPLEX time consumption for 90 stations

respectively on a time budget of 8 hours. These are the most realistic scenarios for the city of Vienna.

Note that obj^+ is the best objective value found in all test runs, while $\overline{obj}$ is the average and obj^- is the worst objective function. For instances with 90 stations, two vehicles and 8 hours budget a Wilcox signed-rank test (illustrated in Table 4.4)supports the hypothesis that variant D performs better than variant W. With sample data from the same instances with five vehicles (seen in Table 4.5 the result is not significant anymore leading to the conclusion that both variants would perform equally good given enough ressources.

For a Wilcox signed-rank test the following steps were executed:

1. calculate $|\overline{obj_W} - \overline{obj_D}|$ for each pair values
2. calculate $\mathrm{sgn}(\overline{obj_W} - \overline{obj_D})$ for each pair of values
3. assign rank numbers from 1 to N with N the number of pairs, in ascending order of the absolute difference from step 1. Let R_i be the assigned rank and sgn_i the sign from step 2 for the i^{th} pair.
4. calculate $W = |\sum_{i=1}^{N}(R_i sgn_i)|$

The null hypothesis H_0 stating that the mean difference between the average results of variant W and D is zero can be rejected, when $W > W_{\alpha=p,N}$.

Table 4.6 illustrates how often each variant is superior to the other. The first set of columns focuses on the best objective value found, while the second set uses the average solution quality for the given parameters. In both cases variant D seem to be better for the problem instances at hand.

It is obvious that the amount of unsatisfied requests is reduced if additional vehicle(s) or additional time budget is available. Table 4.7 compares the solution quality with one vehicle

instance		variant W			variant D		
#	initial	obj^+	obj^-	$\overline{obj}$	obj^+	obj^-	$\overline{obj}$
00	284	42.0784	45.0797	43.4786	37.0819	42.0770	40.3785
01	299	49.0738	53.0777	50.1759	45.0758	49.0769	45.9773
02	290	53.0756	59.0737	55.1759	48.0774	53.0772	50.0774
03	343	81.0808	91.0837	86.1802	75.0825	82.0834	78.1840
04	327	62.0794	72.0772	67.4769	56.0778	65.0722	60.7760
05	332	51.0771	60.0811	52.9799	43.0801	50.0794	46.5811
06	313	53.0751	58.0806	55.4773	44.0820	51.0816	47.6779
07	288	43.0793	45.0765	44.1757	38.0769	43.0717	40.8773
08	329	65.0710	69.0735	66.7740	55.0825	63.0753	58.5759
09	328	58.0818	67.0766	61.3799	56.0786	61.0769	57.6802
10	341	58.0870	66.0791	62.1828	54.0862	58.0834	56.3845
11	297	38.0839	49.0801	43.5806	34.0830	38.0833	36.0825
12	329	51.0786	57.0814	54.0793	48.0812	52.0777	49.6806
13	298	51.0875	57.0837	54.1832	47.0896	53.0781	49.7829
14	244	50.0739	53.0774	51.9756	46.0750	49.0798	47.4776
15	407	105.0771	121.0793	114.6778	103.0820	112.0806	109.6813
16	239	12.0721	15.0703	13.5708	10.0723	13.0713	12.2701
17	281	45.0737	52.0770	46.7771	39.0799	45.0798	42.6789
18	281	29.0739	33.0749	31.2742	24.0757	30.0728	26.5743
19	234	27.0734	31.0758	29.0728	25.0766	28.0699	26.5729
20	322	43.0841	51.0818	48.8793	42.0815	47.0778	44.3809
21	201	15.0697	17.0705	15.8721	13.0714	15.0750	13.7724
22	323	80.0746	85.0766	82.6752	73.0751	78.0762	75.7764
23	221	26.0774	32.0805	29.1791	26.0806	29.0797	27.5789
24	280	19.0778	24.0814	21.3813	18.0808	23.0781	20.2803
25	322	51.0791	56.0772	54.4777	45.0794	50.0726	47.2793
26	357	75.0912	85.0832	79.3860	66.0868	74.0869	70.0865
27	301	63.0748	66.0789	64.2763	59.0742	63.0770	61.6755
28	307	59.0823	67.0828	63.9816	56.0837	61.0846	58.9841
29	256	31.0835	35.0841	33.2826	29.0787	31.0774	29.4820

Table 4.2: Best, worst and average solutions for instances with 90 stations, 2 vehicles and a time budget of 480 minutes per vehicle

when additional time is added, while table 4.8 shows the effects of additional vehicles for both variants. As one can see additional time increases the solution quality faster than adding vehicles. Doubling the time budget increases the average solution quality faster than adding another vehicle with a short time window. This may be due the fact that some unsatisfied requests are not satisfiable with a small time window, because they occur late in the day and are therefore not reachable with this model of the problem. In this algorithm a short time budget means that

instance		variant W			variant D		
#	initial	obj^+	obj^-	$\overline{obj}$	obj^+	obj^-	$\overline{obj}$
00	407	129.0717	136.0744	132.7718	118.0779	128.0689	123.1739
01	377	114.0747	130.0721	122.1745	107.0809	114.0750	110.4788
02	376	106.0732	114.0709	109.7733	96.0785	104.0715	99.6754
03	400	116.0818	132.0830	124.3809	106.0861	115.0802	111.7834
04	404	132.0742	143.0768	138.6744	123.0736	132.0763	128.5750
05	439	160.0777	169.0802	164.7788	151.0807	156.0790	153.1804
06	472	153.0741	164.0794	159.7778	144.0766	158.0695	149.6786
07	449	148.0838	158.0820	152.0818	136.0834	144.0843	141.3842
08	372	107.0775	114.0721	111.5741	101.0789	107.0796	105.3779
09	383	100.0798	113.0781	106.5773	92.0787	101.0787	96.5782
10	310	68.0767	74.0794	70.9791	62.0822	69.0789	66.7793
11	476	178.0793	191.0720	186.0758	164.0790	176.0775	171.0773
12	406	118.0812	125.0799	121.2817	109.0849	116.0783	113.2832
13	332	82.0725	88.0783	84.9760	75.0763	82.0777	78.8756
14	427	127.0743	135.0806	130.4805	115.0817	123.0741	118.6809
15	410	112.0849	120.0804	114.8846	103.0828	111.0803	106.2839
16	452	153.0801	162.0760	158.9774	145.0797	153.0784	149.0790
17	500	172.0816	182.0784	177.6789	157.0792	162.0783	160.5787
18	389	90.0804	101.0797	94.5848	84.0834	91.0785	88.0830
19	364	88.0851	97.0801	93.3792	81.0836	94.0781	85.5812
20	343	83.0732	92.0719	87.4743	79.0766	83.0731	80.8741
21	415	131.0849	139.0830	135.3837	117.0870	126.0890	123.2874
22	314	64.0721	70.0726	67.0741	59.0810	63.0805	61.3791
23	321	87.0717	91.0689	88.5700	80.0701	85.0730	82.9736
24	441	132.0789	139.0767	135.7762	119.0797	123.0751	120.7787
25	394	122.0850	134.0734	126.7782	118.0792	121.0828	119.7815
26	372	82.0783	90.0791	85.9791	75.0835	83.0806	77.6805
27	453	160.0836	170.0823	165.4807	145.0847	154.0820	150.4830
28	293	63.0683	70.0690	67.2684	56.0723	62.0638	57.6697
29	461	150.0858	159.0787	155.6827	134.0868	144.0880	140.0878

Table 4.3: Best, worst and average solutions for instances with 120 stations, 2 vehicles and a time budget of 480 minutes per vehicle

the vehicles can only be moved during the first $\hat{t}_l$ minutes of the day, but still needs to satisfy the whole day of requests. If some station is self-sufficient during this time it usually is not a good idea to visit this station, while other unsatisfied stations are reachable. In addition, those self-sufficient stations may become nearly full or empty due to user requests (but without unsatisfied requests) therefore adding or removing bikes to satisfy requests later in the day, may cause unsatisfied requests right now, ultimatively gaining nothing. With a lager time budget

W	465
$W_{\alpha=0.05,30}$	137
$W_{\alpha=0.01,30}$	109

Table 4.4: Wilcox signed-rank test for 90 stations, **2 vehicles** and 8 hours time

W	67
$W_{\alpha=0.05,30}$	137
$W_{\alpha=0.01,30}$	109

Table 4.5: Wilcox signed-rank test for 90 stations, **5 vehicles** and 8 hours time

these requests are then reachable and satisfiable.

In addition to increasing the time budget or the number of vehicles, one can also increase the total running time of the algorithm. Table 4.9 compares the average objective value of 10 test runs with 90 stations, 2 vehicles and a time budget of 480 minutes. The only difference is the running time of the algorithm being 24 hours compared to only one hour. Note that none of the (600) test runs terminated naturally but was stopped after 24 hours. The third column indicates the difference in solution quality for the individual instances. As one can see the average quality increase for both variants is located in the range from 10% to 15%. This means that the results obtained after only one hour are relatively good considering the tradeoff between running time and solution quality. Another interesting thing to see is that even after 24 hours, variant D seems to be superior when the amount of vehicles is too small for the problem instance.

Tables 4.10 and 4.11 show average relative success rates of the various VND neighborhoods. Those values are average values over all testruns and instances with 90 stations. As one can see neighborhood 7 (3-OPT) does not yield better results. This is due its position at the end of the VND search. Short spot tests showed that 3-OPT does yield improvements when executed earlier in the search. Not very surprisingly, neighborhood 2 (INSERT-UNSAT) performs very well in all cases, as one of two neighborhoods capable of inserting new nodes. Although one can see that INSERT-UNSAT is becoming less effective when routes are getting more complex. In those cases the restructuring neighborhoods (INTRA-2-OPT, INTRA-OR-OPT and INTER-2-OPT-STAR) become more successful. Another observation is the decreasing success of neighborhood 4 (REPLACE). In routes with a short time budget it may not be possible to add another station to the route without exceeding the time budget. Those are the cases were REPLACE shines, since it combines neighborhood 1 and 2. As soon as the time budget (combined with the number of vehicles) gets bigger this neighborhood cannot find as much of an improvement any more. In table 4.10 we can see that the success rate of REPLACE drops from 43.25% to 10.71% and further to just 0.45% by simply increasing the time budget of 5 vehicles. The drop is not as fast with only 3 vehicles, but still there. Similar neighborhoods 3 (INTRA-2-OPT) and 5 (INTRA-OR-OPT) increase their successrates with more complex routes to optimize.

Respectively Table 4.12 shows average relative success rates for all three VNS neighborhoods. Since each of those neighborhoods is called 6 times with different parameters the table sums up all 6 calls. The table should show how successful a single neighborhood concept is, re-

gardless of its respective parameters. While Neighborhood 1 (REMOVE STATIONS) becomes more successful with an increasing time budget neighborhood 2 (MOVE SEQUENCE) declines. Neighborhood 3 (EXCHANGE SEQUENCE) does not seem to have a correlation to the vehicles time budget. Comparing with Table 6.8 and 6.9 only reveals that neighborhood 3 seems to work best with a time budget of 4 hours as long as the number of vehicles is < 5, in which case it declines with the vehicles time budget.

Finally Table 4.13 compares the solution quality of the individual variants with a vehicle time budget of 8 hours (the most realistic case). The % values indicate the reduction of unsatisfied user request compared to the initial status of the system. As one can see the greedy approach is between 0.5% and 5% worse than the LP when the number of vehicles is suitable for the problem. As soon as there are enough vehicles to satisfy all the requests even the greedy approach finds a very good solution. Tables 6.10 and 6.11 illustrate a similar behaviour with two and four hours of operating time.

Table 4.14 compares the average number of solutions evaluated by each variant and gives a difference in percent. As one can see the Greedy approach evaluates usually about twice as much solutions than one of the other two variants. When there is only one vehicle and a time budget of two hours the numbers are not that impressive, but still bigger than for one of the other two variants. When the number of vehicles increases the difference increases as well. The same is true for an increased time budget. Both observations can be traced back to the increased complexity (and therefore running time) for the LP in such cases.

			obj^+			$\overline{obj}$		
$\|V\|$	$\|L\|$	T_{max}	W	D	equal	W	D	equal
60	1	120	1	7	22	1	9	20
60	1	240	2	13	15	2	25	3
60	1	480	1	20	9	0	30	0
60	2	120	2	7	21	2	17	11
60	2	240	2	20	8	1	29	0
60	2	480	6	24	0	2	28	0
60	3	120	2	12	16	2	25	3
60	3	240	1	28	1	1	29	0
60	3	480	1	29	0	4	26	0
60	5	120	7	13	10	7	23	0
60	5	240	0	30	0	2	28	0
60	5	480	0	30	0	1	29	0
90	1	120	1	9	20	1	10	19
90	1	240	2	13	15	2	26	2
90	1	480	1	28	1	0	30	0
90	2	120	3	16	11	3	19	8
90	2	240	1	29	0	0	30	0
90	2	480	1	29	0	0	30	0
90	3	120	2	18	10	2	26	2
90	3	240	0	30	0	0	30	0
90	3	480	3	27	0	1	29	0
90	5	120	5	19	6	6	24	0
90	5	240	0	30	0	0	30	0
90	5	480	5	25	0	12	18	0
120	1	120	2	13	15	3	12	15
120	1	240	3	18	9	1	27	2
120	1	480	0	27	3	0	30	0
120	2	120	2	16	12	3	21	6
120	2	240	0	29	1	0	30	0
120	2	480	0	30	0	0	30	0
120	3	120	3	17	10	5	24	1
120	3	240	0	30	0	0	30	0
120	3	480	0	30	0	0	30	0
120	5	120	3	23	4	2	28	0
120	5	240	0	30	0	0	30	0
120	5	480	8	22	0	8	22	0
Total			70	791	219	74	914	92

Table 4.6: Comparison on how often each variant yields the better result

	variant W			variant D		
$\|V\|$	120	240	480	120	240	480
60	23.24 %	48.66 %	73.12 %	23.66 %	49.09 %	74.03 %
90	15.00 %	33.98 %	53.62 %	15.19 %	34.58 %	54.78 %
120	11.77 %	27.44 %	43.68 %	11.89 %	28.10 %	44.94 %

Table 4.7: Average solution qualitiy increase in % due to a bigger time budget (T_{max}) for one vehicle

	variant W				variant D			
$\|V\|$	1	2	3	5	1	2	3	5
60	23.24 %	42.03 %	56.56 %	77.95 %	23.66 %	42.69 %	57.20 %	78.79 %
90	15.00 %	28.49 %	39.80 %	58.37 %	15.19 %	29.06 %	40.23 %	58.87 %
120	11.77 %	22.37 %	31.33 %	46.47 %	11.89 %	22.90 %	31.91 %	47.39 %

Table 4.8: Average solution qualitiy increase in % due to additional vehicles with a two hour time budget

	variant W			variant D		
instance	1h	24h	δ	1h	24h	δ
00	43.47	39.78	8.51%	40.37	36.98	8.42%
01	50.17	44.87	10.56%	45.97	40.97	10.88%
02	55.17	48.47	12.14%	50.07	43.28	13.58%
03	86.18	77.18	10.44%	78.18	72.88	6.78%
04	67.47	58.37	13.49%	60.77	55.58	8.55%
05	52.97	45.68	13.78%	46.58	42.08	9.66%
06	55.47	46.97	15.32%	47.67	41.08	13.84%
07	44.17	38.88	11.99%	40.87	37.88	7.34%
08	66.77	61.67	7.64%	58.57	54.17	7.52%
09	61.37	54.28	11.57%	57.68	52.48	9.01%
10	62.18	56.18	9.65%	56.38	51.88	7.98%
11	43.58	35.48	18.58%	36.08	33.28	7.76%
12	54.07	49.08	9.25%	49.68	45.48	8.46%
13	54.18	48.18	11.07%	49.78	44.48	10.64%
14	51.97	45.68	12.12%	47.47	43.88	7.58%
15	114.67	107.88	5.93%	109.68	101.68	7.30%
16	13.57	10.67	21.35%	12.27	9.97	18.72%
17	46.77	38.18	18.38%	42.67	36.57	14.30%
18	31.27	25.27	19.18%	26.57	21.77	18.05%
19	29.07	25.07	13.76%	26.57	22.07	16.93%
20	48.87	43.68	10.63%	44.38	40.88	7.88%
21	15.87	13.07	17.64%	13.77	11.47	16.68%
22	82.67	73.07	11.61%	75.77	71.77	5.28%
23	29.17	25.27	13.37%	27.57	22.88	17.04%
24	21.38	18.08	15.43%	20.28	16.18	20.20%
25	54.47	46.97	13.77%	47.27	42.37	10.37%
26	79.38	69.09	12.97%	70.08	64.59	7.85%
27	64.27	60.47	5.92%	61.67	58.27	5.52%
28	63.98	58.58	8.44%	58.98	56.28	4.58%
29	33.28	29.28	12.02%	29.48	26.58	9.83%
avg			12.55%			10.61%

Table 4.9: Average solution qualitiy increase in % due to a longer running time for 90 stations, 2 vehicles and an 8 hour time budget

$\|L\|$	$\widehat{t}$	REMOVE	INSERT-UNSAT	INTRA-2-OPT	REPLACE	INTRA-OR-OPT	INTER-2-OPT-STAR	INTRA-3-OPT
1	120	7.41	76.56	4.90	7.78	0	0	0
1	240	7.15	57.10	23.22	11.36	1.14	0	.0002
1	480	7.36	52.99	28.17	8.60	2.85	0	.0054
2	120	8.93	47.87	12.74	28.19	.15	2.09	0
2	240	6.25	41.34	32.28	15.16	2.46	2.47	.0032
2	480	11.26	36.68	35.52	8.86	4.97	2.68	.0041
3	120	8.68	41.26	11.59	35.49	.09	2.86	0
3	240	5.88	40.33	32.02	13.35	3.13	5.27	.0017
3	480	13.37	30.26	37.38	4.69	6.51	7.76	.0020
5	120	7.41	32.43	11.49	43.25	.19	5.19	0
5	240	5.79	34.85	32.77	10.71	4.51	11.34	.0006
5	480	11.06	23.51	37.18	.45	7.20	20.56	0

Table 4.10: Average relative VND Neighborhood success rates for 90 stations.

$\|L\|$	$\widehat{t}$	REMOVE	INSERT-UNSAT
1	120	8.45	84.88
1	240	11.67	88.32
1	480	12.35	87.64
2	120	16.90	83.09
2	240	13.71	86.28
2	480	22.19	77.80
3	120	18.83	81.16
3	240	12.95	87.04
3	480	25.63	74.36
5	120	20.16	79.83
5	240	13.19	86.80
5	480	25.10	74.89

Table 4.11: Average relative VND Neighborhood success rates for 90 stations.

$\|L\|$	$\widehat{t}$	REMOVE	MOVE	EXCHANGE
1	120	5.00	0	0
1	240	13.33	0	0
1	480	16.11	0	0
2	120	3.53	5.13	.68
2	240	7.31	3.84	2.52
2	480	8.39	3.16	1.98
3	120	3.65	7.25	1.33
3	240	8.02	3.87	2.27
3	480	9.23	3.19	.93
5	120	4.49	7.99	2.59
5	240	8.41	3.63	1.73
5	480	9.78	2.45	.37

Table 4.12: Average relative VNS Neighborhood success rates for 90 stations.

$\|V\|$	$\|L\|$	Constr.+VND	Variant W	Variant D	Greedy
60	1	63.59 %	73.12 %	74.03 %	70.80 %
60	2	88.37 %	96.99 %	97.28 %	94.64 %
60	3	96.39 %	99.30 %	99.33 %	98.77 %
60	5	98.79 %	99.80 %	99.80 %	99.45 %
90	1	48.58 %	53.62 %	54.78 %	52.27 %
90	2	75.61 %	82.94 %	84.51 %	80.30 %
90	3	89.78 %	95.46 %	96.03 %	92.95 %
90	5	97.28 %	99.18 %	99.14 %	98.46 %
120	1	40.16 %	43.68 %	44.94 %	42.44 %
120	2	65.35 %	69.80 %	72.25 %	67.72 %
120	3	80.93 %	86.69 %	88.75 %	83.01 %
120	5	94.58 %	98.20 %	98.32 %	95.80 %

Table 4.13: Average solution qualitiy increase in % for different techniques with a vehicle time budget of 480 minutes.

|V|	$\|L\|$	$\widehat{t}$	Variant W	Variant D	Greedy
60	1	120	543,760	871,322	1,042,558
60	1	240	820,337	777,516	1,440,833
60	1	480	572,413	532,012	1,333,627
60	2	120	757,600	798,079	1,378,567
60	2	240	530,245	497,889	1,294,001
60	2	480	392,928	358,595	1,156,529
60	3	120	629,255	629,347	1,338,262
60	3	240	411,275	378,352	1,171,423
60	3	480	361,518	327,081	1,076,731
60	5	120	497,069	470,588	1,248,190
60	5	240	345,945	310,065	1,044,502
60	5	480	343,759	306,998	1,018,749
90	1	120	673,417	832,993	956,876
90	1	240	715,188	697,542	975,451
90	1	480	516,473	495,902	911,010
90	2	120	718,853	692,780	950,909
90	2	240	481,133	465,191	886,425
90	2	480	327,966	311,989	779,453
90	3	120	572,972	564,860	915,365
90	3	240	374,724	356,620	805,457
90	3	480	265,843	246,340	697,107
90	5	120	443,763	432,210	859,558
90	5	240	275,440	250,940	698,324
90	5	480	239,615	221,262	629,627
120	1	120	652,267	743,132	750,099
120	1	240	627,791	618,484	741,754
120	1	480	476,806	463,189	700,681
120	2	120	624,159	607,085	723,260
120	2	240	441,482	428,559	686,400
120	2	480	308,914	298,925	608,626
120	3	120	509,773	509,644	699,588
120	3	240	350,785	336,230	630,268
120	3	480	242,309	227,936	546,439
120	5	120	405,919	400,727	656,353
120	5	240	260,188	240,800	544,597
120	5	480	202,849	187,202	472,036

Table 4.14: average number of solutions evaluated.

CHAPTER 5

Conclusion and Future Work

This paper presented an algorithm to solve the Dynamic Balancing Bike Sharing System Problem heuristically. It utilizes a VNS metaheuristic with an embedded VND local search. The neighborhoods contain classical VRPs neighborhoods as well as some special neighborhoods taken from [11]. Additionally a LP and a Greedy algorithm was used to calculate (optimal) loading instructions for constructed solutions and further evaluate the objective value for those solutions. Based on a set of semi-realistic example instances three variants of the algorithm were tested. The first one (variant D) uses almost all available VND neighborhoods while the second one (variant W) only uses two VND neighborhoods to utilize the VNS part more. Since most of the computation time is used in the LP no strong statistical evidence was found for one variant beeing superior to the other. The third one offsets this flaw by using the Greedy algorithm to calculate the loading instructions.
For a real world application the Greedy approach is probably the best choice. Although it does not provide an optimal solution, it finds good solutions in a relatively short time. Future reasearch may focus on further improving the Greedy approach to produce better solutions without increasing the running time too much. Another focus for future work could be the problem itself. A realistic addition would be to add a starting time ts_l to each vehicle to indicate the time when vehicle l starts its tour. This would allow modeling multiple vehicle crews sharing vehicles and eliminiate the need to satisfy stations early through out the whole day.

CHAPTER 6

Appendix

In this section additional tables are included for the sake of completeness.

instances		variant W				variant D			
$\|L\|$	$\widehat{t}$	T	T_{cplex}	δ	#sol	T	T_{cplex}	δ	#sol
1	120	1834.20	1700.71	12.74	543,760	3103.07	2931.62	17.09	871,322
1	240	3602.38	3508.58	37.40	820,337	3603.41	3523.20	43.92	777,516
1	480	3606.99	3567.83	91.11	572,413	3614.28	3580.44	105.78	532,012
2	120	3223.25	2998.21	13.32	757,600	3602.00	3405.58	17.33	798,079
2	240	3605.02	3542.61	56.76	530,245	3608.42	3556.99	69.15	497,889
2	480	3610.63	3567.33	82.37	392,928	3635.83	3600.12	100.81	358,595
3	120	3462.26	3258.25	15.97	629,255	3603.04	3440.83	21.21	629,347
3	240	3607.49	3553.79	66.18	411,275	3614.46	3575.65	92.12	378,352
3	480	3614.05	3566.42	74.87	361,518	3637.52	3596.61	87.90	327,081
5	120	3601.59	3431.72	20.20	497,069	3606.16	3490.86	30.27	470,588
5	240	3613.08	3556.66	63.04	345,945	3625.60	3583.85	85.82	310,065
5	480	3620.31	3566.04	65.71	343,759	3635.39	3592.78	84.31	306,998

Table 6.1: CPLEX time consumption for 60 stations

instances		variant W				variant D			
$\|L\|$	$\widehat{t}$	T	T_{cplex}	δ	#sol	T	T_{cplex}	δ	#sol
1	120	3186.70	2760.00	6.46	652,267	3601.48	3186.72	7.68	743,132
1	240	3606.30	3384.34	15.24	627,791	3609.76	3402.79	16.44	618,484
1	480	3623.02	3504.95	29.68	476,806	3635.78	3525.27	31.90	463,189
2	120	3602.69	3131.40	6.64	624,159	3605.20	3219.32	8.34	607,085
2	240	3615.64	3460.22	22.26	441,482	3622.57	3482.90	24.93	428,559
2	480	3650.70	3570.36	44.44	308,914	3682.17	3607.61	48.38	298,925
3	120	3603.24	3201.43	7.96	509,773	3607.59	3283.86	10.14	509,644
3	240	3625.43	3500.88	28.10	350,785	3635.89	3528.27	32.78	336,230
3	480	3691.99	3623.24	52.70	242,309	3716.85	3656.51	60.59	227,936
5	120	3604.97	3264.29	9.58	405,919	3618.01	3368.22	13.48	400,727
5	240	3651.72	3553.96	36.35	260,188	3669.93	3597.13	49.41	240,800
5	480	3743.88	3675.65	53.87	202,849	3795.40	3735.14	61.97	187,202

Table 6.2: CPLEX time consumption for 120 stations

instance		variant W			variant D		
#	initial	obj^+	obj^-	$\overline{obj}$	obj^+	obj^-	$\overline{obj}$
00	212	2.0491	6.0444	3.8488	2.0490	3.0502	2.3494
01	128	.0372	1.0357	.2379	.0370	1.0356	.1371
02	169	.0664	1.0590	.9571	1.0519	1.0574	1.0538
03	174	1.0575	2.0583	1.2581	1.0572	1.0577	1.0573
04	238	11.0659	14.0656	12.4650	10.0649	12.0649	10.8643
05	141	2.0400	2.0406	2.0402	2.0398	2.0399	2.0398
06	202	8.0636	12.0653	8.8648	8.0631	9.0670	8.2662
07	172	2.0630	3.0620	2.9598	2.0667	3.0589	2.9588
08	182	7.0580	9.0566	7.6577	7.0579	7.0591	7.0583
09	173	6.0630	9.0548	8.0579	7.0617	9.0538	7.9581
10	159	1.0427	1.0449	1.0436	1.0425	1.0475	1.0448
11	223	7.0655	12.0665	9.1657	7.0650	9.0657	7.3655
12	151	2.0445	3.0445	2.1450	2.0443	2.0469	2.0451
13	222	4.0692	5.0815	4.8695	3.0703	4.0720	3.7684
14	224	13.0742	17.0724	15.5720	12.0731	15.0702	13.4723
15	173	2.0497	4.0486	3.0489	3.0482	3.0488	3.0485
16	198	.0478	1.0485	.1485	.0475	1.0476	.1479
17	207	6.0738	9.0748	7.4733	6.0733	8.0692	6.9718
18	161	4.0516	5.0574	4.4520	4.0520	5.0488	4.3511
19	219	8.0612	11.0609	9.2629	8.0590	10.0592	8.7602
20	261	21.0732	25.0667	22.7732	20.0740	23.0719	21.5730
21	230	12.0770	16.0751	13.4776	11.0786	13.0753	12.5745
22	209	8.0645	9.0643	8.7638	8.0632	9.0629	8.2635
23	241	15.0626	20.0628	17.3647	14.0614	16.0652	14.9634
24	191	2.0570	4.0554	3.1570	2.0543	4.0525	2.4549
25	135	.0561	1.0525	.5546	.0559	1.0511	.5535
26	165	3.0540	4.0535	3.1547	3.0532	3.0544	3.0539
27	163	2.0600	4.0592	3.2611	2.0595	4.0575	2.6602
28	175	2.0513	3.0542	2.2530	2.0516	3.0534	2.1530
29	253	9.0684	14.0725	10.7692	8.0695	10.0656	9.0678

Table 6.3: Best, worst and average solutions for instances with 60 stations, 2 vehicles and a time budget of 480 minutes per vehicle

$\|L\|$	$\widehat{t}$	REMOVE	INSERT-UNSAT	INTRA-2-OPT	REPLACE	INTRA-OR-OPT	INTER-2-OPT-STAR	INTRA-3-OPT
1	120	8.23	78.82	6.06	6.49	.37	0	0
1	240	6.68	58.85	26.04	6.88	1.52	0	0
1	480	7.80	57.42	26.82	4.51	3.42	0	.0076
2	120	9.74	46.63	12.57	27.71	.28	3.03	0
2	240	6.39	42.41	34.02	10.96	3.24	2.94	.0030
2	480	14.54	37.31	34.52	3.15	5.77	4.67	.0041
3	120	8.93	39.86	13.52	33.47	.38	3.81	0
3	240	6.87	40.12	32.91	10.07	3.91	6.09	.0010
3	480	12.46	33.81	34.58	.84	5.97	12.30	.0040
5	120	7.50	29.86	12.95	40.67	.40	8.58	0
5	240	7.77	39.34	27.66	2.81	3.77	18.62	0
5	480	8.67	32.61	30.70	.11	4.57	23.30	.0003

Table 6.4: Average relative VND Neighborhood success rates for 60 stations.

$\|L\|$	$\widehat{t}$	REMOVE	INSERT-UNSAT
1	120	9.02	90.97
1	240	10.39	89.60
1	480	11.64	88.35
2	120	19.19	80.80
2	240	13.61	86.38
2	480	26.03	73.96
3	120	19.87	80.12
3	240	14.24	85.75
3	480	23.99	76.00
5	120	22.47	77.52
5	240	14.98	85.01
5	480	18.53	81.46

Table 6.5: Average relative VND Neighborhood success rates for 60 stations.

$\|L\|$	$\widehat{t}$	REMOVE	INSERT-UNSAT	INTRA-2-OPT	REPLACE	INTRA-OR-OPT	INTER-2-OPT-STAR	INTRA-3-OPT
1	120	11.00	67.72	11.41	9.85	0	0	0
1	240	7.68	52.97	24.65	13.69	.98	0	0
1	480	8.81	51.90	27.46	9.48	2.32	0	.0040
2	120	11.53	47.50	11.97	27.20	.21	1.58	0
2	240	6.25	40.40	31.46	17.41	2.06	2.39	.0008
2	480	11.51	36.47	32.94	12.63	3.95	2.46	.0029
3	120	9.66	42.00	11.21	34.54	.30	2.27	0
3	240	5.84	38.67	31.97	16.00	2.59	4.90	.0017
3	480	11.60	31.57	35.92	10.15	5.23	5.50	.0037
5	120	7.76	34.05	10.79	42.31	.38	4.68	0
5	240	5.39	34.66	32.83	14.11	3.59	9.39	.0013
5	480	11.95	21.28	40.15	1.26	7.85	17.48	0

Table 6.6: Average relative VND Neighborhood success rates for 120 stations.

$\|L\|$	$\widehat{t}$	REMOVE	INSERT-UNSAT
1	120	11.93	84.73
1	240	12.45	87.54
1	480	14.35	85.64
2	120	18.69	81.30
2	240	14.25	85.74
2	480	22.96	77.03
3	120	19.44	80.55
3	240	13.37	86.62
3	480	23.59	76.40
5	120	21.12	78.87
5	240	12.77	87.22
5	480	27.73	72.26

Table 6.7: Average relative VND Neighborhood success rates for 120 stations.

$\|L\|$	$\widehat{t}$	REMOVE	MOVE	EXCHANGE
1	120	3.33	0	0
1	240	12.22	0	0
1	480	16.66	0	0
2	120	4.61	5.60	1.08
2	240	7.23	4.37	3.02
2	480	7.67	3.91	2.81
3	120	3.92	7.96	2.21
3	240	7.59	3.99	2.60
3	480	8.41	3.86	1.60
5	120	3.80	8.14	2.58
5	240	7.74	4.23	1.95
5	480	8.44	3.97	1.08

Table 6.8: Average relative VNS Neighborhood success rates for 60 stations.

$\|L\|$	$\widehat{t}$	REMOVE	MOVE	EXCHANGE
1	120	4.44	0	0
1	240	13.33	0	0
1	480	16.66	0	0
2	120	3.79	5.74	.97
2	240	6.94	4.24	2.53
2	480	8.56	3.27	1.59
3	120	3.95	8.24	1.73
3	240	7.74	3.41	2.10
3	480	8.95	3.17	.96
5	120	4.33	8.53	2.62
5	240	8.61	3.02	1.22
5	480	11.16	1.33	.11

Table 6.9: Average relative VNS Neighborhood success rates for 120 stations.

$\|V\|$	$\|L\|$	Constr.+VND	Variant W	Variant D	Greedy
60	1	18.86 %	23.24 %	23.66 %	21.42 %
60	2	32.07 %	42.03 %	42.69 %	38.38 %
60	3	41.95 %	56.56 %	57.20 %	51.35 %
60	5	58.21 %	77.95 %	78.79 %	66.87 %
90	1	13.14 %	15.00 %	15.19 %	13.98 %
90	2	23.81 %	28.49 %	29.06 %	25.76 %
90	3	31.55 %	39.80 %	40.23 %	35.23 %
90	5	44.86 %	58.37 %	58.87 %	50.95 %
120	1	9.98 %	11.77 %	11.89 %	10.67 %
120	2	17.65 %	22.37 %	22.90 %	20.22 %
120	3	24.45 %	31.33 %	31.91 %	27.70 %
120	5	35.96 %	46.47 %	47.39 %	39.52 %

Table 6.10: Average solution qualitiy increase in % for different techniques with a vehicle time budget of 120 minutes.

$\|V\|$	$\|L\|$	Constr.+VND	Variant W	Variant D	Greedy
60	1	42.99 %	48.66 %	49.09 %	47.79 %
60	2	68.29 %	77.68 %	78.44 %	76.11 %
60	3	83.11 %	93.39 %	94.12 %	91.17 %
60	5	95.15 %	99.77 %	99.78 %	99.14 %
90	1	30.30 %	33.98 %	34.58 %	33.23 %
90	2	51.47 %	57.94 %	59.11 %	57.38 %
90	3	66.62 %	75.33 %	76.73 %	73.69 %
90	5	85.31 %	94.60 %	95.68 %	91.95 %
120	1	24.57 %	27.44 %	28.10 %	26.93 %
120	2	42.37 %	47.67 %	49.01 %	47.00 %
120	3	56.50 %	63.30 %	64.94 %	61.62 %
120	5	75.88 %	84.34 %	86.26 %	80.84 %

Table 6.11: Average solution qualitiy increase in % for different techniques with a vehicle time budget of 240 minutes.

Bibliography

[1] G. Clarke and J.W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.

[2] C. Contardo, C. Morency, and L.M. Rousseau. Balancing a dynamic public bike-sharing system. Tech. Rep. CIRRELT-2012-09, CIRRELT, Montréal, Canada, 2012.

[3] G.A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[4] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Management Science*, 6:80–91, 10 1959.

[5] P. Hansen and N. Mladenovic. Variable neighborhood search. *Computers and Operations Research*, 24(11):1097–1100, 1997.

[6] P. J. Hart and A. W. Shogan. Semi-greedy heuristics: An empirical study. *Operations Research Letters*, 6(3):107–114, 1987.

[7] F.S. Hillier and G.J Lieberman. Introduction to linear programming. In *Introduction to Operations Research*, pages 31–43. McGraw-Hill Education, 9. edition, 2010.

[8] J.-H. Lin and T.-C. Chou. A geo-aware and VRP-based public bicycle redistribution system. *International Journal of Vehicular Technology*, Volume 2012:14 pages, 2012. Article ID 963427.

[9] F. Meunier, R. Wolfler Calvo, and D. Chemla. Bike hiring system: solving the rebalancing problem in the static case. *Discrete Optimization*, 10(2):120–146, 2013.

[10] J. Pfrommer, J. Warrington, G. Schildbach, and M. Morari. Dynamic vehicle redistribution and online price incentives in shared mobility systems. *arXiv preprint arXiv:1304.3949*, 2013.

[11] M. Rainer-Harbach, P. Papazek, B. Hu, and G. Raidl. Balancing bicycle sharing systems: A variable neighborhood search approach. In *M. Middendorf, C. Blum (eds.) Evolutionary Computation in Combinatorial Optimization*, volume 7832, pages 121–132. Springer Berlin Heidelberg, 2013.

[12] M. G. C. Resende and C. C. Ribeiro. Greedy randomized adaptive search procedures. In *Handbook of metaheuristics*, pages 219–249. Springer, 2003.

[13] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581, 1977.

[14] H. Sayarshad, S. Tavassoli, and F. Zhao. A multi-periodic optimization formulation for bike planning and bike utilization. *Applied Mathematical Modelling*, 36(10):4944–4951, 2012.

[15] J. Schuijbroek, R. Hampshire, and W.J. van Hoeve. Inventory rebalancing and vehicle routing in bike sharing systems. Tech.Rep. 1491, Tepper School of Business, 2013. http://repository.cmu.edu/tepper/1491.