Heuristics for a facility location problem with a quadratic objective function

Malena Schmidt

TU Delft

Heuristics for a facility location problem with a quadratic objective function

by

Malena Schmidt

Student number: 5447887

To obtain the degree of Master of Science in Applied Mathematics,
specialising in Discrete Mathematics and Optimisation,
at the Delft University of Technology,
to be defended publicly on 11th of July 2023.

External Supervisor:	Asst. Prof. Dr. Dr. habil. Bismark Singh
Responsible Professor:	Dr. ir. Theresia van Essen
Third Thesis Committee Member:	Dr. ir. Robbert Fokkink
Project Duration:	Nov, 2022 - July, 2023
Faculty:	Faculty of Electrical Engineering, Mathematics & Computer Science, Delft

Cover: Photo by Pourya Gohari on Unsplash
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Preface

When deciding where to go to university during Sixth Form, I was considering mostly courses that combined mathematics and computer science. The course that ended up really standing out to me was the B.Sc. Discrete Mathematics offered at the University of Warwick. This allowed me to pursue my interests in maths and computer science, giving me a good basis for further studies.

Continuing my studies at the TU Delft with the M.Sc. in Applied Mathematics has provided me with a challenge and also allowed me to view the field of discrete mathematics through a slightly different lens. As such, this thesis combines both my interest in theoretical complexity that I developed during my time at Warwick and my interest in heuristics that arose from courses I attended at the TU Delft.

I would like to thank my external supervisor, Bismark Singh, and my responsible professor at the TU Delft, Theresia van Essen, for the support and input they have provided me with throughout the thesis.

Lastly, I would like to thank my friends and family for all the support they've provided me with throughout my thesis. In particular, I would like to thank my partner, my brother and a friend who is currently working on her PhD - talking to someone else who knows the up and downs of research has been immensely helpful.

Malena Schmidt
London, June 2023

Abstract

Facility location problems are an important set of problems within the field of optimisation. These problems consider which facilities to open out of a set of possible facilities and how to assign users to the open facilities. Most of the facility location problems studied have a linear objective. In this thesis, we consider a facility location problem with a quadratic objective, the Balanced Facility Location Problem (BFLP). This problem, and facility location problems in general, quickly becomes difficult to solve for standard MIP solvers as the input size increases. The difficulty of this problem is further supported by it being a $\mathcal{NP}$-hard problem.

Hence, we develop three heuristics for the BFLP: two greedy heuristics and one local search heuristic. These heuristics are adapted from heuristics used for the standard Capacitated Facility Location Problem (CFLP).

The idea behind the first greedy heuristic is to close one facility at a time, starting with all facilities being open, until we have as many facilities open as the budget constraint of the BFLP dictates. At each step, the best facility to close is closed. Apart from adapting this heuristic to accommodate the slightly different constraints of the BFLP, we also make some further adjustments in order to reduce the running time.

The second heuristic that we adapt is a heuristic that instead of closing facilities one at a time, opens facilities one by one, until we reach the budget of open facilities that the BFLP allows. These simple heuristics perform very well in practice on both small and large instances of the BFLP.

Lastly, we discuss a local search heuristic, which attempts to improve a solution to the BFLP by opening one facility and closing another facility at each iteration. The local search only improves marginally upon the results of the greedy algorithms.

For use within the heuristics for the BFLP, we require fast heuristics to solve a subproblem of the BFLP, the problem of assigning users to facilities where the set of open facilities is fixed. Hence, we develop three heuristics for this subproblem and also adapt a previously developed heuristic to be optimised for its use within the BFLP heuristics.

Our heuristics achieve results that are similar or better than what a MIP solver achieves and find a good solution in significantly less time. Especially when the MIP solver struggles, due to the size or limited capacity of the BFLP instance, our heuristics are able to outperform the MIP solver.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
`BFLP`	Balanced Facility Location Problem
`(B)UAP`	(Balanced) User Assignment Problem
`CFLP`	Capacitated Facility Location Problem
`G(Q)AP`	Generalised (Quadratic) Assignment Problem
`MIP`	Mixed Integer Program
`PP`	Partition Problem

Symbols

Symbol	Definition
B	Budget of open facilities, $B \in \mathbb{N}$.
C_j	Capacity of facility j, measured in the number of people that can be assigned to it.
d	Depth of the local search in the `greedy_reassign_open` algorithm, Algorithm 9.
d_i	Demand of user i in the `CFLP`.
e_{ij}	Cost of serving user i from facility j in the `CFLP`.
f_j	Operating cost of facility j in the `CFLP`.
$g_{ii'jj'}$	Coefficient of the quadratic term in the `GQAP`.
I	Set of users.
J	Set of facilities.
n_c	Number of facilities considered in each iteration of the loop in the `BFLP` algorithms, Algorithms 6, 8, 10 and 11.
n_f	Number of iterations after which to recompute the user assignment in the open greedy algorithm, Algorithm 10.
n_r	Number of facilities to consider for each user in the relaxation rounding algorithm, Algorithm 4.
P_{ij}	Preference of user i for facility j. $P_{ij} \in [0, 1]$ and represents the proportion of the population that would be willing to use facility j if they were assigned to it.
S	Set of open facilities.
$T(I, J)$	The objective function value of the transportation problem with user set I and facility set J.
U_i	Population size of user i.
u_j	Utilisation of facility j, so the proportion of its capacity that is used in a solution to the `BFLP` or `BUAP`; $u_j = \frac{\sum_{i \in I} U_i P_{ij} x_{ij}}{C_j}$.

Symbol	Definition
W_{ij}	Number of people from user i that will use facility j if they are assigned to it, $W_{ij} = P_{ij}U_i$.
x_{ij}	Indicator variable of whether user i is assigned to facility j, $x_{ij} \in \{0, 1\}$.
y_j	Indicator variable of whether facility j is open, $y_j \in \{0, 1\}$.
Δ_{MIP}	Difference in objective value of the heuristic to what the `MIP` achieves after 20,000 seconds (excluding time to build the model), calculated using Equation (6.1) where obj_{MIP} is objective value of the `MIP` after 20,000 seconds.
Δ_{OPT}	Difference in objective value of the heuristic to the optimal solution, calculated using Equation (6.1) where obj_{MIP} is the optimal objective function value.
Δ_S	Difference in objective value of the heuristic to what the `MIP` achieves given the same time as the heuristic, calculated using Equation (6.1) where obj_{MIP} is objective value of the `MIP` when it is the same amount of time as the heuristic takes (including the time to build the model).

1 Introduction

Facility location problems have been of interest in the literature and practical applications since the start of the 20th century, with most work done from the 1960s onward, according to Owen and Daskin (1998). The first discussion of the facility location problem is attributed to Alfred Weber in Weber, Pick, and Friedrich (1965). The basic idea of facility location problems is that we are given a set of locations where facilities could be opened and a set of users that need to be served by these facilities. The model then needs to decide which locations to choose to open facilities at and how users are assigned to these open facilities.

Different variants of the problem exist, the most notable being capacitated versus uncapacitated facility location problems, as discussed in Sridharan (1995). In the case of the former, each facility has a capacity associated with it, which when assigning users to it may not be exceeded. Another differentiation made in Sridharan (1995) is whether each user's demand needs to be fulfilled by a single facility (single-source) or whether they can be served by multiple facilities.

Most facility location problems discussed in the literature have a linear objective function, with the aim of minimising the combined cost of opening facilities and serving a user from their assigned facility. In this thesis, we consider a single-source capacitated facility location problem with a quadratic objective function, as proposed in Schmitt and Singh (2021). The model aims to achieve low variance in the utilisation of the facilities while still ensuring good access for users, so users do not need to travel too far. Unlike in traditional facility location problems, there is a budget on how many facilities are allowed to be open. In this thesis, we investigate the complexity of this model further and develop heuristics for solving the problem.

The rest of this thesis is structured as follows: In Section 2.1, we discuss the motivation of the thesis. We then formally define the model in Section 2.2. The rest of Chapter 2 gives a more detailed overview of the heuristics that are used in the literature for problems similar to our model. Chapter 3 then shows that our problem, and also a subproblem of our problem, are $\mathcal{NP}$-hard. This provides us with further motivation to develop heuristics for the problem. In Chapter 4, we then discuss heuristics for the subproblem of assigning users to facilities. This is necessary since this subproblem needs to repeatedly be solved in the heuristics for our overall problem, meaning computationally cheap and good methods are necessary. In Chapter 5, we then discuss heuristics for the capacitated facility location problem with a quadratic objective function. The first heuristic we discuss in Section 5.1 is the close greedy heuristic, which closes facilities one by one until we reach the budget of facilities that is allowed to remain open. We then discuss the "opposite" heuristic in Section 5.2, where facilities are opened (instead of closed) one by one. Finally, we discuss a local search heuristic in Section 5.3. The results of running all heuristics can be found in Chapter 6. We conclude and summarise the findings of the thesis in Chapter 7.

2 Background

We first state the motivation of this thesis and how it extends the work done in Schmitt and Singh (2021) in Section 2.1. Then, we formally define the capacitated facility problem with a quadratic objective function in Section 2.2. Afterwards, we continue with a literature review of the Capacitated Facility Location Problem, focusing on greedy and local search heuristics in Section 2.3 to give us some ideas what heuristics exist for problems similar to our problem. Finally, we consider the literature on the Generalised (Quadratic) Assignment Problem (GQAP) in Section 2.4 since this is a subproblem of the considered facility location problem. Finding solutions to the GQAP quickly is required for greedy and local search approaches for the overall problem.

2.1. Motivation

This work expands on Schmitt and Singh (2021) who developed a Capacitated Facility Location Problem with a quadratic objective function for deciding which recycling facilities to close in Bavaria, Germany. In that thesis, the main focus was on developing the model and showing that it performs well at achieving the aim of balancing the access of users to facilities while ensuring the variation in the utilisation of the facilities is low. In addition, one simple local search heuristic was developed. This local search starts with the largest facilities being open and then tries to replace low utilisation facilities with other facilities to improve upon it. The assignment of users to facilities is done using a greedy assignment, which we discuss in more detail in Section 4.1.

Our work expands on this by first showing that the problem, and the subproblem of assigning users to facilities, is $\mathcal{NP}$-hard. Hence, we develop three new heuristics for the overall problem of deciding which facilities to open and how to assign users: A close greedy algorithm, an open greedy algorithm and a local search algorithm that differs from the one in Schmitt and Singh (2021). The motivation for developing these heuristics is that as the input instances become larger, simply solving the MIP becomes very difficult. This is due to the size of the model being proportional to $|I||J|$ where I is the set of users and J is the set of facilities. Although this is only polynomial, in practice both building the model and solving it becomes difficult, the latter especially when the capacity of the facilities becomes a limiting factor. Larger instances can be of interest in practice, for example if the problem of which recycling facilities to close were to be extended from the state of Bavaria to the whole of Germany. Apart from for closing existing facilities, the model could also be used to decide at which locations to open facilities out of a large set of possible locations. Our aim for the heuristics is that they reach an objective function value that is approximately as good as what using a MIP solver achieves after 20,000 seconds.

The time for the MIP solver does not include the time to build the model. We further hope for our heuristics to achieve their results in significantly less time. Note that even building these models and the initial relaxation that MIP solvers run can take a significant amount of time on larger instances. For example, on an instance with 5000 users and about 2500 facilities, building the model takes over an hour and solving the root relaxation takes another half an hour. Hence, there is significant room for improvement by using heuristics that do not require the model to be built. Our secondary aim is for these solutions to be better than what the MIP solver achieves. At the very least, the goal is for these to perform better than the local search heuristic in Schmitt and Singh (2021).

In addition, we investigate a subproblem of the overall problem: the problem of assigning users to facilities. Solving this subproblem multiple times is necessary for the aforementioned heuristics, hence we require very fast heuristics for this. We first show that this subproblem is $\mathcal{NP}$-hard, and then develop heuristics for it. To that extent, we adapt the greedy assignment heuristic of Schmitt and Singh (2021) to our purposes and also develop our own heuristics, a relaxation rounding heuristic and two local search algorithms. These heuristics are of interest in their own rights as well as in relation to the overall problem. Note that this subproblem is a special case of the Generalised Quadratic Assignment Problem, which has only been investigated sparsely in the literature. As such, our work also contributes to the research on the GQAP by showing that the simple heuristics we develop perform well on at least our subclass of GQAP instances.

2.2. Model

The model we are discussing here is a Capacitated Facility Location Problem with a quadratic objective function that was developed for deciding which recycling facilities to close in Bavaria by Schmitt and Singh (2021). The inputs to the model are a set of users, I , and a set of facilities, J. Each facility $j \in J$ has an associated capacity C_j with units being the number of people that can be assigned to it. Each user $i \in I$ has an associated population U_i since each user is a postcode in the original problem. Furthermore, each user $i \in I$ has a preference P_{ij} for each facility j. This is a number between 0 and 1 that is based on the distance between the user and the facility. It is interpreted as the proportion of users that will actually visit the facility if they are assigned to it. Lastly, the budget B tells us how many facilities should be opened.

$$\min_{x,y} \sum_{j\in J} C_j \left(1 - \frac{\sum_{i\in I} U_i P_{ij} x_{ij}}{C_j}\right)^2 \tag{2.1a}$$

$$\text{s.t.} \sum_{i\in I} U_i P_{ij} x_{ij} \leq C_j \qquad \forall j \in J \tag{2.1b}$$

$$\sum_{j\in J} y_j \leq B \tag{2.1c}$$

$$\sum_{j\in J} x_{ij} = 1 \qquad \forall i \in I \tag{2.1d}$$

$$x_{ij} \leq y_j \qquad \forall i \in I, j \in J \tag{2.1e}$$

$$x_{ij} \in \{0,1\} \qquad \forall i \in I, j \in J \tag{2.1f}$$

$$y_j \in \{0,1\} \qquad \forall j \in J. \tag{2.1g}$$

The variables that are used in the model are binary variables y_j for $j \in J$, which are 1 when facility $j \in J$ is open and 0 otherwise, and binary variables x_{ij} for $i \in I, j \in J$, which are 1 when user $i \in I$ is assigned to facility $j \in J$ and 0 otherwise. Now, let us consider the constraints of

the model. Constraints (2.1b) ensure that each facility does not have more users assigned to it than allowed by its capacity. Constraint (2.1c) fixes the number of facilities that are to be opened to be at most B. Together with the x_{ij} being binary, Constraints (2.1d) imply that each user is assigned to exactly one facility. Lastly, Constraints (2.1e) mean that a user only is assigned to a facility if that facility is open.
The objective function (2.1a) here has the aim of ensuring a low variance in the utilisation of the facilities, defined by $u_j = \frac{\sum_{i \in I} U_i P_{ij} x_{ij}}{C_j}$ for each $j \in J$.

We also consider a slight variant to this model, which has the same optimal solution. It has been shown by Schmitt and Singh (2021) that the y_j variables can be relaxed and this will still give an optimal solution to the original model. Additionally, we explicitly define the utilisations in Model (2.2). We mainly consider Model (2.2), except in Chapter 3 where considering Model (2.1) simplifies matters.

$$\min_{x,y} \sum_{j \in J} C_j \left(1 - u_j\right)^2 \tag{2.2a}$$

$$\text{s.t. } u_j = \frac{\sum_{i \in I} U_i P_{ij} x_{ij}}{C_j} \qquad \forall j \in J \tag{2.2b}$$

$$\sum_{j \in J} y_j \leq B \tag{2.2c}$$

$$\sum_{j \in J} x_{ij} = 1 \qquad \forall i \in I \tag{2.2d}$$

$$x_{ij} \leq y_j \qquad \forall i \in I, j \in J \tag{2.2e}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in I, j \in J \tag{2.2f}$$

$$0 \leq y_j \leq 1 \qquad \forall j \in J \tag{2.2g}$$

$$0 \leq u_j \leq 1 \qquad \forall j \in J. \tag{2.2h}$$

2.3. Literature Capacitated Facility Location Problem

We now give an overview of the literature available on problems that are similar to our problem. Model (2.2) is a Capacitated Facility Location Problem, but unlike most facility location problems, we have a quadratic objective function and not a linear objective function. There is also a small difference on the input data and consequently in two of the constraints compared to the classic Capacitated Facility Location Problem. Nevertheless, the overall idea of opening only some facilities, facilities having a certain capacity and having to assign users to facilities is the same. Hence, it is worth considering the literature on the Capacitated Facility Location Problem and the heuristics used to solve this problem.

We start by stating the Capacitated Facility Location Problem (CFLP) in the form it is usually encountered in the literature in Model (2.3), see for example Conforti, Gérard, and Zambelli (2014). For ease of comparison, the same variable names for variables that are the same in our model are used. The newly introduced data variables here are the operating cost of a facility f_j, the demand of a user d_i and the cost of transporting from user i to facility j, e_{ij}. d_i can be compared to $U_i P_{ij}$ in our model, the difference here meaning conceptually that in our model

the demand of a user changes depending on which facility fulfils it.

$$\min_{x,y} \sum_{i\in I}\sum_{j\in J} e_{ij}d_i x_{ij} + \sum_{j\in J} f_j y_j \tag{2.3a}$$

$$\text{s.t.} \sum_{j\in J} x_{ij} = 1 \qquad \forall i \in I \tag{2.3b}$$

$$\sum_{i\in I} d_i x_{ij} \leq C_j y_j \qquad \forall i \in I, j \in J \tag{2.3c}$$

$$x_{ij} \geq 0 \qquad \forall i \in I, j \in J \tag{2.3d}$$

$$y_j \in \{0,1\} \qquad \forall j \in J. \tag{2.3e}$$

Instead of having an upper bound of facilities to open, the CFLP aims to minimise the overall cost. Since opening facilities has a cost associated with it, this means that potentially not all facilities are opened. Constraints (2.1d) and Constraints (2.3b) are exactly the same. However, in Model (2.3) the x_{ij} variables are continuous, meaning the demand of one user can be fulfilled by multiple facilities. This is not the case in all Capacitated Facility Location Problems that are being discussed in the literature; for example, see Holmberg, Rönnqvist, and Yuan (1999) where a so-called single source CFLP is being discussed. Single source means that each user's demand can only be fulfilled by a single facility, which in practice means that the x_{ij} are binary variables. Lastly, Constraints (2.3c) incorporate Constraints (2.1b) and (2.1e) into a single constraint. Again, this is not the case in all capacitated facility location model formulations, for example Holmberg, Rönnqvist, and Yuan (1999) has these still as separate constraints. This is useful since having these as separate constraints strengthens the relaxations according to Sridharan (1995) and as such allows the relaxation of the y_j in our model, as done in Model (2.2) and suggested in Schmitt and Singh (2021).

The CFLP is a computationally challenging problem to solve as it is $\mathcal{NP}$-hard, and in the single source case, simply using a MIP solver can fail on larger instances according to Laporte et al. (2015). Hence, some heuristics have been developed to solve the problem, which we discuss in the following sections.

2.3.1. Greedy heuristics

There are two different greedy heuristics that are used for the CFLP, the DROP procedure, which closes facilities one by one and the ADD procedure, which opens facilities one by one. The following overview of these two heuristics is based on Jacobsen (1983), which adapted these procedures from the Uncapacitated to the Capacitated Facility Location Problem.

In both these procedures, a method that given a set of open facilities determines the objective function value and hence an assignment of users to the open facilities is required. When fixing the facilities that are open and disregarding the cost of opening facilities, the problem simplifies to the classical transportation problem; for details on the transportation problem, see Hitchcock (1941). Hence, let $T(I, J)$ denote the objective function value of the transportation problem if J are the open facilities and I all the users that need to be serviced by them. In practice, in order to save time, $T(I, J)$ might just provide a bound on what the optimal solution to the transportation problem is. Further, let S denote the set of facilities that are open at any point in the algorithm. The DROP procedure can be seen in Algorithm 1. The algorithm starts with all facilities being open (Line 1). Then, for each facility j that is open, the algorithm computes how much closing the facility would improve the objective function (Line 3). Note that δ_j here is the amount by which the objective function *decreases* – we are closing j so the cost decreases by f_j plus the difference in the objective function arising from the transportation problem. The algorithm then chooses to close the facility which leads to the largest decrease in the objective function

(Lines 4 - 6). If all single facility closures would lead to an increase in the objective function, the algorithm terminates and returns the set of open facilities that has been chosen. The last iteration's transportation problem solution can then be used to complete the solution to the CFLP.

Algorithm 1 The DROP procedure (from Jacobsen (1983))

Input: an instance of Model (2.3).
Output: a set of open facilities S.
1: $S \leftarrow J$; $\delta^* \leftarrow 1$.
2: **while** $\delta^* > 0$ **do**
3: $\quad \delta_j \leftarrow f_j + T(I, S) - T(I, S \setminus \{j\}),\ \forall j \in S$.
4: $\quad j^* \leftarrow \arg\max_{j \in S} \delta_j$; $\delta^* \leftarrow \max_{j \in S} \delta_j$.
5: $\quad$ **if** $\delta^* > 0$
6: $\quad\quad S \leftarrow S \setminus \{j^*\}$.
7: return S.

In Section 5.2, we discuss how we can apply this DROP procedure to our problem. As this requires being able to solve a transportation problem, which in our case is a special case of the Generalised Quadratic Assignment Problem due to our problem being single-source, we discuss the literature on how to solve the Generalised (Quadratic) Assignment Problem (GQAP) in Section 2.4. We further develop our own heuristics for our specific version of the GQAP in Chapter 4.

Now, let us discuss the ADD procedure, Algorithm 2, as seen in Jacobsen (1983). The challenge here compared to the DROP procedure is that the algorithm starts with an infeasible transportation problem since the algorithm starts with no facilities open. To circumvent this problem, the authors introduce a fake facility j' to the problem, which is open from the start. This facility has capacity $C_{j'}$ equal to the total demand of all users and very high transportation costs $e_{ij'}$ for all users $i \in I$. As long as these transportation costs are chosen high enough, by the time the algorithm terminates, no user needs to have their demand satisfied by j' anymore. The rest of the algorithm proceeds similarly to the DROP procedure, the only difference being that it opens facilities instead of closing facilities. Hence, it computes the change in objective function value of opening each facility (Line 3), chooses the facility with the largest improvement (Line 4) and opens it if that change is positive (Line 6). If no further improvement can be achieved, the algorithm terminates and returns the open facilities, after removing the fake facility j' (Line 7).

Algorithm 2 The ADD procedure (from Jacobsen (1983))

Input: an instance of Model (2.3) with an additional facility j' with $e_{i,j'} = \sum_{j \in J, i \in I} e_{i,j} \forall i \in I$ and $C_{j'} = \sum_{i \in I} d_i$.
Output: a set of open facilities S.
1: $S \leftarrow \{j'\}$; $\delta^* \leftarrow 1$.
2: **while** $\delta^* > 0$ **do**
3: $\quad \delta_j \leftarrow T(I, S) - T(I, S \cup \{j\}) - f_j,\ \forall j \in S$.
4: $\quad j^* \leftarrow \arg\max_{j \in S} \delta_j$; $\delta^* \leftarrow \max_{j \in S} \delta_j$.
5: $\quad$ **if** $\delta^* > 0$
6: $\quad\quad S \leftarrow S \cup \{j^*\}$.
7: return $S \setminus \{j'\}$.

We discuss how to apply the ADD procedure to our problem in Section 5.2.

2.3.2. Local search

The basic idea of local search algorithms is to start with a feasible solution and make small changes to it. If a change leads to a better solution, it is accepted to be the current solution. This procedure can then be repeated until some termination criteria is reached. In the case of facility location problems, these small changes to the solution usually involve opening and/or closing some facilities, see for example Zhang, Chen, and Ye (2005), Pal, Tardos, and Wexler (2001).

To get started with the simplest local search procedures for facility location problems, let us consider the two methods that are based on the `ADD` and `DROP` procedures discussed previously, as described in Jacobsen (1983). The idea of the first local search approach is to, starting with a feasible solution, choose a facility to close. Then a single iteration of the `ADD` procedure is run, i.e. the facility leading to the best improvement is opened. If this leads to a better solution than previously, this becomes the currently accepted solution. If all facilities have been tried out when closing facilities and none lead to an improvement in the objective function after running the `ADD` iteration, the algorithm terminates.

The other local search method discussed in Jacobsen (1983) is doing exactly the opposite, i.e. instead of starting by closing a facility, we start by opening a facility. Then one iteration of the `DROP` procedure is run. Overall, these two local search procedures can be summarised as closing one facility and opening one facility, where one of these choices is made in a way that is locally optimal.

Now, let us discuss some other operations that are used within local search algorithms for the `CFLP` to make small changes to the solution. In Pal, Tardos, and Wexler (2001), three different operations are used in a local search algorithm, which has an approximation ratio of $8.532 + \epsilon$. Since our objective function is different to the linear function used in this facility location problem, the proof of this approximation ratio is not transferable to our problem. However, the algorithm itself and specifically its operations are still of interest since the problems are similar enough in their constraints. The algorithm stars with a feasible solution with a set of open facilities S and a transportation problem assignment x_{ij}. Note that here again we are dealing with the version of the problem where multiple facilities can satisfy the demand of a single user. The three operations of the algorithm are:

- $add(s)$: Open a facility $s \in J$ and recompute the assignments x_{ij} assuming $S \cup \{s\}$ are the open facilities.
- $open(s, T)$: Open a facility $s \in J$ and close a set of facilities $T \subseteq S \setminus \{s\}$. All demand previously fulfilled by facilities in T is now to be fulfilled by s, without changing any other part of the assignment.
- $close(s, T)$: Close facility $s \in S$ and open a set of facilities $T \subseteq J$. All demand previously fulfilled by s is now to be fulfilled by facilities in T, without changing any other part of the assignment.

In all of these operations, it is allowed for the facilities that are being opened to already be open. The reasons for allowing this are different for each operation. In particular, $add(s)$ is the only operation that recomputes the assignment from scratch, even if s is already open, which means this operation can be used with an open facility to improve the assignment. If the facility is not open yet, the $add(s)$ operation allows us to open a facility without closing any facilities. Allowing facilities to be opened that are already open also means that both $open(s, T)$ and $close(s, T)$ can be used to close facilities without opening any new facilities by choosing s and T, respectively, as facilities that are already open. Note that the open and close operations are not always feasible to do – the capacity of the facilities we want to reassign users to might not be large enough. In order to make the algorithm more efficient, instead of calculating the exact change in objective, only bounds are calculated for the last two operations. To ensure that the

algorithm terminates in polynomial time, the authors further terminate the algorithm if, out of all the possible operations, none leads to an improvement of at least $\frac{obj(S)}{p(n,\epsilon)}$. Here, $p(n, \epsilon)$ is a polynomial in the size of the problem n and $\frac{1}{\epsilon}$, and $obj(S)$ is the objective function value that is achieved by the current solution. The authors are not clear on how they define the size of the problem, however either $n = |I| + |J|$ or $n = |I||J|$ appear to be the natural choices.

These results were further improved in Zhang, Chen, and Ye (2005) by generalising the last two operations into a single operation $multi(r, R, t, T)$, which then leads to a $5.828 + \epsilon$ approximation algorithm. In this operation, $\{r\} \cup R \subseteq S$ is the set of facilities being closed, while $\{t\} \cup T \subseteq J \setminus \{r\} \cup R$ is the set of facilities being opened. Users originally having some of their demand covered by facilities in R now have their demand satisfied by t and those originally (partially) assigned to r now have their demand fulfilled by facilities in $\{t\} \cup T$. Since this operation leads to an exponential number of operations that could be considered, Zhang, Chen, and Ye (2005) only focus on a subset of these. To this extent, for each r, t a linear programming relaxation is run for an integer program that determines what the best choice of R and T is. The authors prove that this relaxation leads to at most 2 fractional values, so it can be seen as a very close approximation to the integer program and hence allows making a good choice of R and T.

To conclude on what we have seen regarding the local search approach for the facility location problem, these algorithms all focus on closing and opening facilities, either one at a time or even opening / closing multiple facilities in a single step. Depending on the operation performed, either the assignment of users to facilities is recomputed from scratch, or it is simply slightly changed by reassigning users from now closed facilities to newly opened facilities. Note that all four operations discussed from Zhang, Chen, and Ye (2005) and Pal, Tardos, and Wexler (2001) are not applicable to our problem in their general form, since our problem has an explicit bound on the number of facilities that can be open. However, any version of these operations where the same number of facilities are opened as closed could be used in our problem.

2.4. Literature Generalised (Quadratic) Assignment Problem

As mentioned when considering the heuristics for the `CFLP`, the two greedy methods require a method to assign the users to facilities, when the set of open facilities is fixed. This can be encapsulated by Model (3.1) for our facility location problem. Since we are considering a single-source Capacitated Facility Location Problem, this problem is most similar to the Generalised Assignment Problem (`GAP`). Hence, we discuss this problem now and look at some heuristics used for this problem. The Generalised Assignment Problem, as presented by Özbakir, Baykasoğlu, and Tapkan (2010), can be seen in Model (2.4). The constraints are exactly the same as in the model we would like to solve by setting $w_{ij} = U_i P_{ij}$.

$$\min_x \sum_{i \in I} \sum_{j \in J} e_{ij} x_{ij} \tag{2.4a}$$

$$\text{s.t.} \sum_{j \in J} x_{ij} = 1 \qquad \forall i \in I \tag{2.4b}$$

$$\sum_{i \in I} w_{ij} x_{ij} \leq C_j \qquad \forall i \in I, j \in J \tag{2.4c}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in I, j \in J. \tag{2.4d}$$

However, the objective function is again linear while our objective function is quadratic. The Generalised Quadratic Assignment Problem (`GQAP`) has also been discussed in the literature, see for example Hahn et al. (2007), but to a lesser extent, with our quadratic objective function being

a special case of this more general objective seen in Equation (2.5). To see that our objective function is a special case of this, see Equation (4.2c). In most discussions of the GQAP, instead of w_{ij} the capacity required by each user does not depend on the facility, so simply w_i is used, see for example Lee and Ma (2004).

$$\sum_{i \in I} \sum_{j \in J} e_{ij} x_{ij} + \sum_{i,i' \in I, j,j' \in J} g_{ii'jj'} x_{ij} x_{i'j'}. \tag{2.5}$$

A lot of discussion of the Quadratic Assignment Problem (QAP) can be found in the literature, see Burkard et al. (1998) for an overview. This is a special case of the GQAP where every facility needs to have exactly one user assigned to them, i.e. it leads to a matching between users and facilities. Since this changes the problem significantly from our problem, we do not go into more detail on the QAP, instead focusing on the GAP and the GQAP.

2.4.1. GAP as a scheduling problem

As done in Shmoys and Tardos (1993), the GAP can be viewed as a scheduling problem. In particular, the set J is the set of machines and the set I is the set of jobs. Each job needs to be scheduled on a single machine and takes time w_{ij} on this machine. Each machine is at most allowed to take time C_j in processing. Lastly, e_{ij} is the cost incurred by scheduling job i on machine j and we are seeking to minimise the overall cost.

One other objective function that is often used in relation to (unrelated) parallel machines is to minimise the makespan – the maximum time that a machine is running in the solution, see for example Pinedo (2016), Ghirardi and Potts (2005). This objective function might be more interesting to use since it may be closer to our quadratic objective function in intention than the linear cost objective function in Equation (2.4a).

Note that even if the jobs take the same time on each machine – which would be equivalent to the discount factor $P_{ij} = 1$ in our model – the problem is strongly $\mathcal{NP}$-hard with 3 or more machines, according to Pinedo (2016). However, a simple heuristic exists for the parallel machine makespan minimisation problem: The longest processing time first rule (LPT) assigns the jobs in decreasing order of w_{ij} to the earliest available machine, which leads to a $\frac{4}{3} - \frac{1}{3m}$ approximation of the optimal solution, as proven in Pinedo (2016).

Having different processing times on each machine immediately breaks this heuristic and makes the problem harder to solve. However, a 2-approximation algorithm based on relaxing the decision version of the linear program and a binary search to determine the best makespan that can be constructed like this can be found in Lenstra, Shmoys, and Tardos (1990). The decision version is the problem of determining whether a solution with makespan at most d exists.

The question arising now is whether makespan minimisation can be somehow connected to our quadratic objective function. The problem here is that while with makespan minimisation we want to choose a machine that processes the job quickly, in our assignment problem of assigning users to facilities we want to assign them to a facility for which $U_i P_{ij}$ is large (as long as this still fulfils capacity constraints) in order to increase the utilisation of the facility and hence decrease its contribution to the objective function. Hence, in our problem, there exists some "conflict" between the capacity constraint and the objective that is not present in the makespan minimisation problem.

2.4.2. Local search heuristics for GAP

Different local search heuristics have been developed for the GAP. The first one we consider here, discussed in Amini and Racer (1994), has been shown to work very well on "small" instances, giving similar objective values as other algorithms while being faster. On "large" instances,

it performs worse, and a trade-off between time and solution quality becomes apparent. The two operations performed by the heuristic are swapping the assignment of two users and reassigning users. The algorithm works by having "major" iterations; at the end of each a change is applied, and within each "major" iteration there are multiple "minor" iterations which seek to add one move at each iteration to a sequence of moves. In each "minor" iteration, all possible swaps and reassignments are considered. Only the one with the best improvement is added to an improvement sequence. Then, the search for the best possible improvement is repeated, with users already in the sequence not allowed to be chosen again, and the best one is added to the improvement sequence. This process is repeated until no move can be added to the improvement sequence anymore without worsening the objective. Then, the subsequence of the improvement sequence that leads to the best improvement is applied. This ends a "major" iteration, and the next "major" iteration is started. The algorithm terminates if no positive change to the objective can be made.

Martello and Toth (1987) also use a simple local search procedure to improve on the solution their algorithm achieves. The local search procedure considers for each user i whether reassigning it to the facility with the minimum f_{ij} improves the solution. If it does, they accept this as the currently best solution. f_{ij} here is one (or the one that leads to the best results) of e_{ij}, $\frac{e_{ij}}{w_{ij}}$, $-w_{ij}$ or $-\frac{w_{ij}}{C_j}$.

A more generalised version of the previously discussed local search neighbourhoods can be found in Osman (1995). Given an assignment of users to facilities, Osman considers a neighbouring solution by choosing two facilities j, j'. Then, a λ-move involves reassigning at most λ users currently assigned to j to j' and at the same time reassigning at most λ users currently assigned to j' to j. These moves can generate infeasible solutions due to the capacity constraint, so the infeasible moves are disregarded in the local search procedure. Moves that lead to feasible solutions are called "admissible". Osman further considers two different approaches to choosing which admissible solution to choose. In the best-admissible strategy, all λ-moves are considered and the best one is chosen. In the first-admissible strategy, the first admissible solution that leads to an improvement in the objective function value is chosen. Both a simulated annealing strategy where worse solutions are accepted, with a decreasing probability over time, and a tabu search approach, where a move cannot be undone for a certain number of iterations are considered. Further, the author implements a simple restarting approach, in which once the local search has reached a local optimum, this solution is saved and the search is restarted from a different starting solution, until some termination criterion is reached. Considering the results of running these local search heuristics on some example instances, the first observation made in Osman (1995) is that for large instances, considering 1-moves leads to the same results as considering 2-moves, with significantly less computational effort. Further, Osman notes that the best-admissible approach leads to better results than the first-admissible strategy if the local search is not restarted when a local optimum is reached, but the performance becomes identical when this restart is used. However, due to the increased computational effort of the best-admissible strategy, for larger instances the first-admissible strategy is recommended.

Connecting this back to our problem, since we are particularly interested in instances that are significantly larger than the ones considered in all these papers, the best approach for our problem seems to be to use the simplest approach: Only consider 1-moves and use the first-admissible strategy. We do so in Section 4.3.

2.4.3. Heuristics for GQAP

Exact solution methods, such as those discussed in Lee and Ma (2004) and Hahn et al. (2007) who use branch-and-bound and linearisation techniques, are not of interest to us since these methods' primary use case seems to be on very small instances with about 20 users, while we

are interested in larger instances. Instead, we focus on heuristics developed for the GQAP. Note that in all the algorithms we discuss now, the capacity required by the user is not depended on the facility, i.e. the model uses w_i instead of w_{ij}.

The first heuristic we discuss is the GRASP with path-linking heuristic discussed in Mateus, Resende, and Silva (2010). This heuristic is a local search heuristic with multiple starting points. These starting points are generated with a randomised greedy algorithm. For each starting solution, a local search that uses 1-moves and 2-moves, but only considers a subset of a solution's neighbourhood, is run. Furthermore, a set of elite solutions P is kept. This set contains the best solutions found so far, but the set should also be diverse. This means that no solutions in the elite set are too close together, where their distance is defined by the number of move operations required to get from one solution to the other. On the solution constructed by the local search, the path linking algorithm is then run. This algorithm tries to find a path of moves between the local search solution and a random solution from the elite set. The best solution found on this path is then returned and another local search is run on this solution. If the elite set has not yet reached its capacity and the new solution is different enough from the other solutions in the set, it is added. Alternatively, if the elite set is already full, but the new solution is better than the worst solution in P, it replaces the solution in P that is closest to it among those that are worse than it. This procedure of randomly generating a greedy solution, running local search, running path-linking and then running local search again is repeated until a stopping criterion is reached.

The second heuristic we discuss, as seen in McKendall and Li (2016) has some similarities to the heuristic in Mateus, Resende, and Silva (2010) since it starts with a greedy solution and then applies a local search procedure. However, it only starts with a single greedy solution and only considers 1-moves in its best-admissible local search. Additionally, a tabu search procedure is used within the local search. This means if a user i has been reassigned from facility j, assigning user i back to facility j will not be considered for a certain number of iterations. The authors report that their algorithm reaches the same objective values as achieved in Mateus, Resende, and Silva (2010), except for one instance, in less time on average.

The last heuristic we would like to mention, whose best objective function values the previous two papers compare themselves to, is discussed by Cordeau et al. (2006). This is a genetic algorithm which after creating offspring runs a 1-move best-admissible tabu search before adding the offspring to the population if it is better than the worst element and different from the other elements. The 1-move is restricted to only reassignments and no swaps of assignments are used. Within the tabu search, solutions are allowed that are not feasible due to the capacity constraints being exceeded, but these lead to a penalty in the objective function value. If this leads to an infeasible solution on termination, an algorithm to fix this into a feasible assignment is run afterwards. The authors report that their algorithm achieves the optimal solution on very small instances where it is feasible to solve the model, and leads to results that are better than what the linearisation of the problem achieves on larger instances.

To conclude, all the heuristic methods that are not aiming to lead to a provable optimal solution use some form of local search, either within the main algorithm or on a greedy solution. Within the local search approaches, at most 2-moves are considered.

This concludes our review of the literature on problems similar to our problem(s). In the next chapter, we show that our problems are $\mathcal{NP}$-hard, before developing heuristics which use some of the ideas that we discussed in this literature review.

3

Complexity

As discussed in Section 2.3, several heuristics and greedy methods to solve capacitated facility location problems rely on subroutines that assess how beneficial a particular facility is. To this end, in this section, we consider the following subproblem of our quadratic optimisation Model (3.1): assignment of users to a *known* set of open facilities, $S \subseteq J$. A solution obtained from this problem extends to Model (2.1) by setting $y_j = 0, \forall j \in J \setminus S$. Our hope is that solving this subproblem cheaply would allow use of a heuristic that solves this subproblem repeatedly in an algorithm for the overall problem. Our subproblem, for a given set of open facilities S, is as follows.

$$\overline{z} = \min_{x} \sum_{j \in S} C_j \left(1 - \frac{\sum_{i \in I} U_i P_{ij} x_{ij}}{C_j}\right)^2 \tag{3.1a}$$

$$\text{s.t.} \sum_{i \in I} U_i P_{ij} x_{ij} \leq C_j \qquad \forall j \in S \tag{3.1b}$$

$$\sum_{j \in S} x_{ij} = 1 \qquad \forall i \in I \tag{3.1c}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in I, j \in S. \tag{3.1d}$$

Unfortunately, as we show in the next section, Model (3.1) is $\mathcal{NP}$-hard. We prove that this is true even for two facilities and even in the absence of an objective function; i.e., determining a feasible solution to the decision version of Model (3.1) itself is $\mathcal{NP}$-complete. We show this from scratch in order to gain some intuition as to what makes the problem difficult to solve. Furthermore, the proof ideas we gain from this help us in showing that our specific version of the GQAP, the decision version of Model (3.1), is $\mathcal{NP}$-complete, even if we had sufficient capacity.

3.1. Complexity UAP

We first define the decision version of the problem to determine feasibility of Model (3.1). For simplicity in this section, we let $W_{ij} = U_i P_{ij}, \forall i \in I, j \in S$.

Definition 1. *The User Assignment Problem (UAP)*

Instance: Given a set of users $i \in I = \{i_1, i_2, \ldots, i_{|I|}\}$, $|I| \geq 2$, a set of facilities $j \in S = \{j_1, j_2, \ldots, j_{|S|}\}$ with corresponding capacities $C_j \in \mathbb{Z}^+$, and weights $W_{ij} \in \mathbb{R}^+$ of assigning user $i \in I$ to facility $j \in J$.

Question: Do there exist subsets of users $I_j \subseteq I, \forall j \in S$ such that the I_j form a partition of I (i.e., $\bigcup_{j \in S} I_j = I$ and $I_j \cap I_{j'} = \emptyset$ for all $j \neq j' \in S$) with $\sum_{i \in I_j} W_{ij} \leq C_j, \forall j \in S$?

Proposition 1. *The* UAP *formulates the feasible region given by Equations* (3.1b)-(3.1d) *correctly.*

Proof. Consider a solution x_{ij} that is feasible for Constraints (3.1b)-(3.1d). Then, from Constraints (3.1c)-(3.1d), each user $i \in I$ is assigned uniquely to one and only one facility $j \in J$. We define $I_j = \{j \in S : x_{ij} = 1\}$; then, the I_j form a partition of I. From Constraints (3.1b), we then have $\sum_{i \in I_j} W_{ij} \leq C_j$ which completes the equivalence.

Conversely, let I_j form a YES instance of the UAP. Set

$$x_{ij} = \begin{cases} 1 & \text{if } i \in I_j \\ 0 & \text{otherwise} \end{cases}, \forall j \in J, i \in I. \tag{3.2}$$

Then, Constraints (3.1c)-(3.1d) capture the mutually exclusive and exhaustive partitions of I. Further, $\sum_{i \in I} U_i P_{ij} x_{ij} = \sum_{i \in I_j} W_{ij} \leq C_j$ shows that Constraints (3.1b) are valid. Thus, the feasible solution x_{ij} correctly formulates the UAP. □

In what follows, we show that the Partition Problem, whose definition and complexity we state here, reduces to the UAP.

Definition 2. *The Partition Problem (PP)*

Instance: Given a set of positive integers $T = \{t_1, ..., t_{|T|}\}$, $|T| \geq 2$; *i.e.,* $t_k \in \mathbb{Z}^+, \forall k = 1, 2, \ldots, |T|$.

Question: Does there exist a subset $L \subseteq \{1, ..., |T|\}$ *such that* $\sum_{k \in L} t_k = \frac{1}{2} \sum_{t \in T} t = \sum_{k \in \{1,...,|T|\} \setminus L} t_k$?

Lemma 1 (Karp (1972))**.** *The Partition Problem is* $\mathcal{NP}$*-complete.*

Theorem 1. *The* UAP *is* $\mathcal{NP}$*-complete.*

Proof. Given an instance of PP we construct an instance of UAP as follows:

- $I \leftarrow \{1, ..., |T|\}$;
- $S \leftarrow \{1, 2\}$;
- $C_1 = C_2 \leftarrow \left\lfloor \frac{1}{2} \sum_{t \in T} t \right\rfloor$;
- $W_{i1} = W_{i2} \leftarrow t_i$ for all $i \in I$.

We first observe that the UAP is in $\mathcal{NP}$ and that the construction of the instance of the UAP is polynomial in the input size $|T|$. Next, we show that an instance of the PP is a YES instance, if and only if the transformed instance is a YES instance for the UAP.

$\Longrightarrow$ First, consider a YES instance of the PP given by a subset $L \subseteq \{1, ..., |T|\}$. The existence of a partition and $t_k \in \mathbb{Z}^+$ implies that $\sum_{t \in T} t$ is an even integer; hence, $\left\lfloor \frac{1}{2} \sum_{t \in T} t \right\rfloor = \frac{1}{2} \sum_{t \in T} t$. We then construct subsets of the users leading to a YES instance of the UAP as follows: $I_1 \leftarrow L$ and $I_2 \leftarrow \{1, ..., |T|\} \setminus L$. Then, $I_1 \cup I_2 = L \cup (\{1, ..., |T|\} \setminus L) = \{1, ..., |T|\} = I$ and $I_1 \cap I_2 = L \cap (\{1, ..., |T|\} \setminus L) = \emptyset$; thus, this assignment is a partition of I. Further, we have

$$\sum_{i \in I_1} W_{i1} = \sum_{k \in L} t_k = \frac{1}{2} \sum_{t \in T} t = C_1; \tag{3.3a}$$

$$\sum_{i \in I_2} W_{i2} = \sum_{k \in \{1,...,|T|\} \setminus L} t_k = \frac{1}{2} \sum_{t \in T} t = C_2 \tag{3.3b}$$

where the first and third equality in both equations follow from construction, while the second holds since the PP instance is a YES instance. Thus, the UAP instance is also a YES instance.

$\Longleftarrow$ Next, consider a YES instance of the UAP given by two subsets I_1 and I_2. First, we have

$$\sum_{t \in T} t = \sum_{i \in I_1} W_{i1} + \sum_{i \in I_2} W_{i2} \leq C_1 + C_2 = 2 \left\lfloor \frac{1}{2} \sum_{t \in T} t \right\rfloor. \tag{3.4}$$

Therefore, $\frac{1}{2}\sum_{t\in T} t \leq \left\lfloor \frac{1}{2}\sum_{t\in T} t \right\rfloor$ which can only hold if $\frac{1}{2}\sum_{t\in T} t = \left\lfloor \frac{1}{2}\sum_{t\in T} t \right\rfloor$ and hence $\sum_{t\in T} t$ is an even integer. In Equation (3.4), the first equality holds by construction and since the UAP instance is a YES instance, the inequality holds since UAP is a YES instance, and the second equality holds by construction. We now construct a partition leading to a YES instance of the PP. Consider $L \leftarrow I_1$. It follows that $\{1,...,|T|\} \setminus L = I_2$ since I_1, I_2 partition $I = \{1,..,|T|\}$. It remains to be shown that $\sum_{k\in L} t_k = \sum_{k\in T\setminus L} t_k$. We further have

$$\frac{1}{2}\sum_{t\in T} t = C_1 \geq \sum_{i\in I_1} W_{i1} = \sum_{k\in L} t_k; \tag{3.5a}$$

$$\frac{1}{2}\sum_{t\in T} t = C_2 \geq \sum_{i\in I_2} W_{i2} = \sum_{k\in\{1,...,|T|\}\setminus L} t_k. \tag{3.5b}$$

In both equations, the first and third equality follow from the construction, while the inequality follows since UAP is a YES instance.

However, by definition,

$$\sum_{k\in L} t_k + \sum_{k\in\{1,...,|T|\}\setminus L} t_k = \sum_{t\in T} t. \tag{3.6}$$

Assume $\frac{1}{2}\sum_{t\in T} t > \sum_{k\in L} t_k$. Then,

$$\sum_{k\in L} t_k + \sum_{k\in\{1,...,|T|\}\setminus L} t_k < \frac{1}{2}\sum_{t\in T} t + \frac{1}{2}\sum_{t\in T} t = \sum_{t\in T} t. \tag{3.7}$$

Equation (3.7) contradicts Equation (3.6); hence, from Equation (3.5a) we have $\sum_{k\in L} t_k = \frac{1}{2}\sum_{t\in T} t$. With an analogous argument from Equation (3.5b) it follows that $\sum_{k\in\{1,...,|T|\}\setminus L} t_k = \frac{1}{2}\sum_{t\in T} t$. Thus, the subset L provides a YES instance of the PP. □

In Appendix A, we provide an example of this mapping.

3.2. Complexity SCUAP

In Section 3.1, we showed that determining a feasible solution for Model (3.1) is, in general, $\mathcal{NP}$-complete. However, solutions to special cases of instances of the UAP are easy. For example, if the capacities are large enough to accommodate any user, so $C_j \geq \sum_{i\in I} W_{ij}, \forall j \in S$, a trivial feasible solution is obtained by assigning any user to any facility. However, as we show in this section, determining an optimal set of facilities is still hard even when the capacities are sufficient. Indeed, in Appendix A, we provide two examples for how an intuitively determined solution is still suboptimal. To this end, we now define the decision version of Model (3.1) assuming that capacities are sufficient.

Definition 3. *The Sufficient Capacity User Assignment Problem (SCUAP)*

Instance: Given a set of users $i \in I = \{i_1, i_2, \ldots, i_{|I|}\}$, $|I| \geq 2$, a set of facilities $j \in S = \{j_1, j_2, \ldots, j_{|S|}\}$ with corresponding capacities $C_j \in \mathbb{Z}^+$, weights $W_{ij} \in \mathbb{R}^+$ of assigning user $i \in I$ to facility $j \in J$ s.t. $C_j \geq \sum_{i\in I} W_{ij}, \forall j \in S$ and a number $M \in \mathbb{R}^+$.

Question: Do there exist subsets of users $I_j \subseteq I, \forall j \in S$ such that the I_j form a partition of I (i.e., $\bigcup_{j\in S} I_j = I$ and $I_j \cap I_{j'} = \emptyset$ for all $j \neq j' \in S$) such that $\sum_{j\in S} C_j \left(1 - \frac{\sum_{i\in I_j} W_{ij}}{C_j}\right)^2 \leq M$?

To prove that the SCUAP is $\mathcal{NP}$-complete, we need Proposition 2; hence, we state and prove it now before proving that the SCUAP is $\mathcal{NP}$-complete.

Proposition 2. *Given $y \in \mathbb{R}^+$ the function $f(x_1, x_2) = (1 - \frac{x_1}{y})^2 + (1 - \frac{x_2}{y})^2$ subject to $x_1 + x_2 = y$, where $x_1, x_2 \in \mathbb{R}^+$, is uniquely minimised at $x_1 = \frac{1}{2}y = x_2$, and hence $f(\frac{1}{2}y, \frac{1}{2}y) = \frac{1}{2}$.*

Proof. We reformulate the given function by setting $a = \frac{x_1}{y}$. Then, the problem is to show that the function $(1-a)^2 + (1-(1-a))^2$, subject to $a \in [0,1]$, is uniquely minimised at $a = \frac{1}{2}$. Setting the derivative of the function to zero, we have $a = \frac{1}{2}$ as required. This is indeed a unique minimum since the second derivative is $4 > 0$. Hence, $f(x_1, x_2)$ is uniquely minimised at $x_1 = \frac{1}{2}y = x_2$ which gives $f(\frac{1}{2}y, \frac{1}{2}y) = \frac{1}{2}$. □

Theorem 2. *The SCUAP is $\mathcal{NP}$-complete.*

Proof. Given an instance of PP, we construct an instance of SCUAP as follows:

- $I \leftarrow \{1, ..., |T|\}$;
- $S \leftarrow \{1, 2\}$;
- $C_1 = C_2 \leftarrow \sum_{t \in T} t$;
- $W_{i1} = W_{i2} \leftarrow t_i$ for all $i \in I$;
- $M \leftarrow \frac{1}{2} \sum_{t \in T} t$.

Note that our requirement of having sufficient capacity is satisfied since $\sum_{i \in I} W_{ij} = \sum_{k \in I} t_k = \sum_{t \in T} t = C_j, \forall j \in S$.
Also, observe that the SCUAP is in $\mathcal{NP}$ and that the construction of the instance of the SCUAP is polynomial in the input size $|T|$. Next, we show that an instance of the PP is a YES instance, if and only if the transformed instance is a YES instance for the SCUAP.

$\Longrightarrow$ First, consider a YES instance of the PP given by a subset $L \subseteq \{1, ..., |T|\}$. We then construct subsets of the users leading to a YES instance of the SCUAP as follows: $I_1 \leftarrow L$ and $I_2 \leftarrow \{1, ..., |T|\} \setminus L$. Then, $I_1 \cup I_2 = L \cup (\{1, ..., |T|\} \setminus L) = \{1, ..., |T|\} = I$ and $I_1 \cap I_2 = L \cap (\{1, ..., |T|\} \setminus L) = \emptyset$; thus, this assignment is a partition of I. This solution has objective value

$$\sum_{j \in S} C_j \left(1 - \frac{\sum_{i \in I_j} W_{ij}}{C_j}\right)^2 = \left(1 - \frac{\sum_{i \in I_1} W_{i1}}{C_1}\right)^2 C_1 + \left(1 - \frac{\sum_{i \in I_2} W_{i2}}{C_2}\right)^2 C_2 \tag{3.8a}$$

$$= \left(1 - \frac{\sum_{k \in L} t_k}{\sum_{t \in T} t}\right)^2 \sum_{t \in T} t + \left(1 - \frac{\sum_{k \in \{1,...,|T|\} \setminus L} t_k}{\sum_{t \in T} t}\right)^2 \sum_{t \in T} t \tag{3.8b}$$

$$= \left(1 - \frac{\frac{1}{2}\sum_{t \in T} t}{\sum_{t \in T} t}\right)^2 \sum_{t \in T} t + \left(1 - \frac{\frac{1}{2}\sum_{t \in T} t}{\sum_{t \in T} t}\right)^2 \sum_{t \in T} t \tag{3.8c}$$

$$= 2\left(1 - \frac{1}{2}\right)^2 \sum_{t \in T} t = \frac{1}{2} \sum_{t \in T} t \tag{3.8d}$$

$$= M. \tag{3.8e}$$

Equation (3.8a) follows from the definition of the SCUAP instance, Equation (3.8b) holds by construction, Equation (3.8c) holds since the PP instance is a YES instance, Equation (3.8d) is a simplification, while Equation (3.8e) holds by construction from the definition of M. As equality holds throughout, the constructed instance is a YES instance of the SCUAP.

$\Longleftarrow$ Next, consider a YES instance of the SCUAP given by two subsets I_1 and I_2. We now construct a partition leading to a YES instance of the PP. Consider $L \leftarrow I_1$. It follows

that $\{1,\ldots,|T|\} \setminus L = I_2$ since I_1, I_2 partition $I = \{1,..,|T|\}$. It remains to be shown that $\sum_{k\in L} t_k = \sum_{k\in\{1,\ldots,|T|\}\setminus L} t_k$.

Consider the optimal objective function value of this SCUAP instance.

$$\frac{1}{2}\sum_{t\in T} t = M \geq \sum_{j\in S} C_j\left(1 - \frac{\sum_{i\in I_j} W_{ij}}{C_j}\right)^2 \tag{3.9a}$$

$$= C_1\left(1 - \frac{\sum_{i\in I_1} W_{i1}}{C_1}\right)^2 + C_2\left(1 - \frac{\sum_{i\in I_2} W_{i2}}{C_2}\right)^2 \tag{3.9b}$$

$$= \sum_{t\in T} t\left(1 - \frac{\sum_{k\in L} t_k}{\sum_{t\in T} t}\right)^2 + \sum_{t\in T} t\left(1 - \frac{\sum_{k\in\{1,\ldots,|T|\}\setminus L} t_k}{\sum_{t\in T} t}\right)^2. \tag{3.9c}$$

Equation (3.9a) follows since the optimal objective function value is at most M since this is a Yes instance, where $M = \frac{1}{2}\sum_{t\in T} T$ by construction. Equations (3.9b) and (3.9c) follow from the definition of the SCUAP instance and our definition of L. Dividing throughout by $\sum_{t\in T} t > 0$ simplifies Equation (3.9) to

$$\frac{1}{2} \geq \left(1 - \frac{\sum_{k\in L} t_k}{\sum_{t\in T} t}\right)^2 + \left(1 - \frac{\sum_{k\in\{1,\ldots,|T|\}\setminus L} t_k}{\sum_{t\in T} t}\right)^2. \tag{3.10}$$

We now use Proposition 2 with $y \leftarrow \sum_{t\in T} t > 0$. Then, the function

$$\left(1 - \frac{x_1}{\sum_{t\in T} t}\right)^2 + \left(1 - \frac{x_2}{\sum_{t\in T} t}\right)^2, \tag{3.11}$$

subject to $x_1 + x_2 = \sum_{t\in T} t$, with $x_1, x_2 \in \mathbb{R}^+$ is uniquely minimised at $x_1^* = x_2^* = \frac{1}{2}\sum_{t\in T} t$, giving a value of $\frac{1}{2}$. For the feasible solution $x_1 \leftarrow \sum_{k\in L} t_k$ and $x_2 \leftarrow \sum_{k\in\{1,\ldots,|T|\}\setminus L} t_k$, we then have

$$\frac{1}{2} \leq \left(1 - \frac{\sum_{k\in L} t_k}{\sum_{t\in T} t}\right)^2 + \left(1 - \frac{\sum_{k\in\{1,\ldots,|T|\}\setminus L} t_k}{\sum_{t\in T} t}\right)^2. \tag{3.12}$$

Since the minimum is uniquely attained by Proposition 2, Equations (3.10) and (3.12) together yield that $x_1^* = x_2^* = \sum_{k\in L} t_k = \sum_{k\in\{1,\ldots,|T|\}\setminus L} t_k$; i.e., the considered set L indeed defines a YES instance of the PP. □

In Appendix A, we provide an example of this mapping.

3.3. Complexity BUAP and BFLP

The above discussion leads us to our main result of this section that both Model (2.1), the Balanced Facility Location Problem, and Model (3.1), the Balanced User Assignment Problem, are $\mathcal{NP}$-complete. We formally define the decision versions of these problems.

Definition 4. *The Balanced User Assignment Problem (BUAP)*

Instance: Given a set of users $i \in I = \{i_1, i_2, \ldots, i_{|I|}\}$, $|I| \geq 2$, a set of facilities $j \in S = \{j_1, j_2, \ldots, j_{|S|}\}$ with corresponding capacities $C_j \in \mathbb{Z}^+$, weights $W_{ij} \in \mathbb{R}^+$ of assigning user $i \in I$ to facility $j \in J$ and a number $M \in \mathbb{R}^+$.

Question: Do there exist subsets of users $I_j \subseteq I, \forall j \in S$ such that the I_j form a partition of I (i.e., $\bigcup_{j\in S} I_j = I$ and $I_j \cap I_{j'} = \emptyset$ for all $j \neq j' \in S$) and the capacity constraints are satisfied (i.e. $\sum_{i\in I_j} W_{ij} \leq C_j, \forall j \in S$) such that $\sum_{j\in S} C_j\left(1 - \frac{\sum_{i\in I_j} W_{ij}}{C_j}\right)^2 \leq M$?

Definition 5. *The Balanced Facility Location Problem (BFLP)*

Instance: Given a set of users $i \in I = \{i_1, i_2, \ldots, i_{|I|}\}$, $|I| \geq 2$, *a set of facilities* $j \in J = \{j_1, j_2, \ldots, j_{|J|}\}$ *with corresponding capacities* $C_j \in \mathbb{Z}^+$, *weights* $W_{ij} \in \mathbb{R}^+$ *of assigning user* $i \in I$ *to facility* $j \in J$, *a budget* $B \leq |J| \in \mathbb{Z}^+$ *and a number* $M \in \mathbb{R}^+$.

Question: Do there exist a subset of facilities $S \subseteq J$ *s.t.* $|S| \leq B$, *subsets of users* $I_j \subseteq I, \forall j \in S$ *such that the* I_j *form a partition of* I *(i.e.,* $\bigcup_{j \in S} I_j = I$ *and* $I_j \cap I_{j'} = \emptyset$ *for all* $j \neq j' \in S$*) and the capacity constraints are satisfied (i.e.* $\sum_{i \in I_j} W_{ij} \leq C_j, \forall j \in S$*) such that* $\sum_{j \in J} C_j \left(1 - \frac{\sum_{i \in I_j} W_{ij}}{C_j}\right)^2 \leq M$*?*

Theorem 3. *The BUAP is* $\mathcal{NP}$*-complete.*

Proof. This follows directly from Theorem 2 since the SCUAP is a special case of the BUAP. □

Theorem 4. *The BFLP is* $\mathcal{NP}$*-complete.*

Proof. This follows directly from Theorem 3 since the BUAP is a special case of the BFLP. In particular, an instance of the BUAP with given inputs I', S' and M' reduces to the BFLP by setting $I \leftarrow I'$, $J \leftarrow S'$, $B \leftarrow |J|$ and $M \leftarrow M'$. □

This concludes our discussion on the theoretical complexity of both Model (2.1) and Model (3.1). In the following sections, we develop heuristics to solve these models. Fortunately, we find that despite the theoretical hardness of the BUAP and the BFLP, we can still obtain feasible and high-quality solutions via our heuristics; our computational experiments in Chapter 6 further confirm this. This observation is similar to finding solutions for the Partition Problem, which despite its hardness allows for high-quality heuristic solutions.

4

Heuristics for BUAP

In this chapter, we present four heuristics for Model (3.1): the greedy algorithm proposed in Schmitt and Singh (2021), an algorithm based on rounding a fractional solution and two local search algorithms. Note that in the objective function value that these algorithms return, we include the closed facilities. We do so since this is the form of the objective function we require when using these heuristics as subroutines for solving the BFLP in Chapter 5.

4.1. Greedy algorithm of Schmitt and Singh

Algorithm 3 The `greedy_assignment` algorithm (adapted from Schmitt and Singh (2021))

Input: an instance of Model (3.1); a function `sort`$(L, A_l,$ ascending/descending) that sorts elements $l \in L$ of the vector A_l in ascending or descending order.

Output: status f = either `feasible` or `infeasible` for the given inputs; if `feasible`, an assignment x for the input instance with corresponding utilisation u and objective function value $\overline{z}$.

Initialise: $I' \leftarrow I$; $x_{ij} \leftarrow 0,\ \forall i \in I, j \in J$; $R_j \leftarrow C_j,\ \forall j \in S$.
while $I' \neq \emptyset$ **do**
  $M_j \leftarrow \emptyset\, \forall j \in S$.
  for $i \in I'$ **do**
    $j' \leftarrow$ `random_choice`$(\arg\max_{\{j \in S: W_{ij} \leq R_j\}} \{P_{ij}\})$.
    $M_{j'} \leftarrow M_{j'} \cup \{i\}$.
  if $M_j = \emptyset,\ \forall j \in S$, return $f \leftarrow$`infeasible`; "heuristic failed".
  for $j \in S$ **do**
    $I'' \leftarrow$ `sort`$(M_j, P_{ij},$ descending).
    for $i \in I''$ **do**
      if $W_{ij} \leq R_j$
        $R_j \leftarrow R_j - W_{ij}$; $I' \leftarrow I' \setminus \{i\}$; $x_{ij} \leftarrow 1$.
return $f \leftarrow$ `feasible`; x; $u_j \leftarrow \frac{\sum_{i \in I_j} W_{ij} x_{ij}}{C_j},\ \forall j \in J$; $\overline{z} \leftarrow \sum_{j \in J} C_j (1 - u_j)^2$.

We begin by reproducing the greedy algorithm proposed in Schmitt and Singh (2021); we later compare our results with those obtained by this algorithm and also further adapt it to our purposes. The algorithm of Schmitt and Singh (2021) seeks to provide a feasible solution to the complete Model (2.1); however, in this section, we are only interested in a solution for Model (3.1).

Thus, we extract the relevant parts of this algorithm, and summarise it in Algorithm 3. For each user, the algorithm first determines their most preferred facility from those that have sufficient capacity to accommodate it (Line 5). If no such facility is available, we terminate reporting infeasibility. Else, we consider each facility sequentially, and assign users to it in order of their preference for this facility until its capacity is exhausted (Lines 8 - 12). We repeat this assignment until all users are allocated. For details, see Schmitt and Singh (2021).

4.2. Basic rounding algorithm

We next consider the continuous relaxation of Model (3.1) given by:

$$\min_{x} \sum_{j \in S} C_j \left(1 - \frac{\sum_{i \in I} U_i P_{ij} x_{ij}}{C_j}\right)^2 \tag{4.1a}$$

$$\text{s.t. } (3.1b) - (3.1c) \tag{4.1b}$$

$$0 \leq x_{ij} \leq 1 \; \forall i \in I, j \in S. \tag{4.1c}$$

The quadratic objective function of Model (4.1), which is of the form $c + b^T x + x^T A x$, is positive semi-definite. To see this, we note:

$$\sum_{j \in S} C_j \left(1 - \frac{\sum_{i \in I} W_{ij} x_{ij}}{C_j}\right)^2 \tag{4.2a}$$

$$= \sum_{j \in J} C_j \left(1 - 2\frac{\sum_{i \in I} W_{ij} x_{ij}}{C_j} + \left(\frac{\sum_{i \in I} W_{ij} x_{ij}}{C_j}\right)^2\right) \tag{4.2b}$$

$$= \sum_{j \in J} C_j - 2 \sum_{j \in J} \sum_{i \in I} W_{ij} x_{ij} + \sum_{j \in J} \frac{\left(\sum_{i \in I} W_{ij} x_{ij}\right)^2}{C_j}, \tag{4.2c}$$

where the last term of Equation (4.2c) is the quantity of our interest. We rewrite it as follows:

$$\sum_{j \in J} \frac{\left(\sum_{i \in I} W_{ij} x_{ij}\right)^2}{C_j} = \sum_{j \in J} \sum_{i_1 \in I} \sum_{i_2 \in I} \frac{x_{i_1 j} W_{i_1 j} W_{i_2 j} x_{i_2 j}}{C_j}. \tag{4.3}$$

Since $W_{ij} \geq 0, \forall i \in I, j \in J$, for each j, we have Gram matrices $(A_j)_{kl} = \frac{W_{kj} W_{lj}}{C_j}$ that are positive semi-definite. The matrix A then contains these matrices on its diagonal and is otherwise zero. Hence, A is also positive semi-definite. Thus, a standard mixed integer program solver such as Gurobi is sufficient to solve Model (4.1) to optimality according to Wolfe (1959). Next, we provide a basic algorithm based on rounding the fractional x solution obtained from Model (4.4). We present this in Algorithm 4, and provide an explanation next.

Let us first explain one input of Algorithm 4, n_r, the **n**umber of facilities considered for each user in the **r**elaxation. We reduce the number of x_{ij} variables to consider in the continuous relaxation to the $n_r \leq |S|$ facilities that a user prefers the most; i.e., we reduce the $|I||S|$ combinations to $|I|n_r$. Hence, for each user $i \in I$ let $T_i = \{j \in S : P_{ij}$ is among the n_r largest $P_{ij'}$ for $j' \in J\}$ and $V_j = \{i \in I : j \in T_i\}$. The model then needs to be adapted as seen in Model (4.4) in order to allow for less x_{ij} being defined. We make this adjustment since we are interested only in a good quality fractional solution that might even be suboptimal. The choice of n_r is arbitrary – fewer facilities are required if they are sufficiently spread out. In our computational experiments, we find $n_r = 20$ or 50 perform well on our instances.

$$\overline{z} = \min_{x} \sum_{j \in S} C_j \left(1 - \frac{\sum_{i \in V_j} U_i P_{ij} x_{ij}}{C_j}\right)^2 \tag{4.4a}$$

$$\text{s.t.} \sum_{i \in V_j} U_i P_{ij} x_{ij} \leq C_j \qquad \forall j \in S \tag{4.4b}$$

$$\sum_{j \in T_i} x_{ij} = 1 \qquad \forall i \in I \tag{4.4c}$$

$$0 \leq x_{ij} \leq 1 \qquad \forall i \in I, j \in T_i. \tag{4.4d}$$

Once we have the solution to this relaxation, to determine binary assignments of the x_{ij} variables from the solution to the continuous relaxation, we begin by assigning each user i to the facility with the largest x_{ij} value (Line 2). If there is more than one such solution for the argmax, we arbitrarily pick one. We denote the subset of users assigned to j by $I_j \subseteq I$. We adapt this solution if the capacities of some facilities are exceeded. We denote by S' the subset of facilities whose capacities are exceeded (Line 5). We then seek to reassign users of facilities in the set S' to other facilities that have available capacity. Consider a given $j \in S'$. We begin the reassignment by computing the subset of users assigned to this j that have $W_{ij} > C_j$ since these users cannot be assigned to j in a binary solution; we denote this subset as I'_j (Line 7). Then, all users in the set I'_j need to be reassigned. We reassign them to the facility other than j that they most prefer and that still has capacity for them in Lines 8-12. If we are now within the capacity for this facility j, we continue to the next facility (Line 13). If we are still over capacity, we further reassign users, but now do so seeking to keep assignments in order of preferences; i.e., we first remove users that have low preferences for facility j (Lines 14-17). We stop this reassignment for j when the capacity constraint is satisfied, and go to the next facility (Line 20).

Algorithm 4 can fail to determine a feasible solution even if the input instance is feasible. This failure is determined in Lines 10 or 21; i.e., if a facility's violated capacity cannot be reassigned, we exit immediately and report a failure.

Algorithm 4 is similar to the `greedy_assignment` algorithm, Algorithm 3. However, the key difference is that unlike Algorithm 3, Algorithm 4 begins with a complete although infeasible assignment. This assignment comes from a fractional solution that we now seek to make feasible. The similarity is in the steps taken to make this solution feasible; specifically, we (re)assign users to facilities with the highest preference that have sufficient capacity for them. As with the greedy algorithm, our algorithm might fail to find a feasible solution even if such a solution exists. Furthermore, our approach of using n_r is different from that proposed in Schmitt and Singh (2021), where a cut-off for the preference is used in the original model. Specifically, in Schmitt and Singh (2021) only (i, j) pairs where $P_{ij} \geq 0.2$ are considered, and further, Constraints (2.2d) are relaxed to an inequality. This requires Schmitt and Singh (2021) to perform a post-processing model as some users are left unassigned. Our approach obviates this need, and requires neither the post-processing model nor the relaxation to an inequality.

Algorithm 4 The `relaxation_rounding` algorithm

Input: an instance of Model (3.1); a scalar $n_r \leq |S|$; a function `sort`$(L, A_l,$ ascending/descending) that sorts elements $l \in L$ of the vector A_l in ascending or descending order.

Output: status f = either `feasible` or `infeasible` for the given inputs; if `feasible`, an assignment x for the input instance with corresponding utilisation u and objective function value $\overline{z}$.

1: solve Model (4.1) with n_r most preferred facilities for each user considered in x_{ij}, get continuous x.
2: $a_i \leftarrow \{\arg\max_{j \in S}\{x_{ij}\}\},\ \forall i \in I$.
3: $I_j \leftarrow \{i \in I : a_i = j\} \subseteq I,\ \forall j \in S$.
4: $R_j \leftarrow C_j - \sum_{i \in I_j} W_{i,j},\ \forall j \in S$.
5: $S' \leftarrow \{j \in S : R_j < 0\} \subseteq S$.
6: **for** j in S' **do**
7: $\quad I'_j \leftarrow \{i \in I_j : W_{ij} > C_j\} \subseteq I_j$.
8: $\quad$ **for** $i \in I'_j$ **do**
9: $\quad\quad K \leftarrow \arg\max_k\{P_{ik} : R_k - W_{ik} \geq 0, k \neq j\}$.
10: $\quad\quad$ **if** $K = \emptyset$, return $f \leftarrow$ `infeasible`; "heuristic failed".
11: $\quad\quad k \leftarrow$ `random_choice`(K).
12: $\quad\quad I_k \leftarrow I_k \cup \{i\}$; $I_j \leftarrow I_j \setminus \{i\}$; $R_k \leftarrow R_k - W_{ik}$; $R_j \leftarrow R_j + W_{ij}$.
13: $\quad$ **if** $R_j \geq 0$, break. Continue with the next iteration of the outer `for` loop (Line 6).
14: $\quad I''_j =$ `sort`$(I_j \setminus I'_j, P_{ij},$ ascending$)$
15: $\quad$ **for** $i \in I''_j$ **do**
16: $\quad\quad K \leftarrow \arg\max_k\{P_{ik} : R_k - W_{ik} \geq 0, k \neq j\}$.
17: $\quad\quad$ **if** $K = \emptyset$, break. Continue with the next iteration of the `for` loop (Line 15).
18: $\quad\quad k \leftarrow$ `random_choice`(K).
19: $\quad\quad I_k \leftarrow I_k \cup \{i\}$; $I_j \leftarrow I_j \setminus \{i\}$; $R_k \leftarrow R_k - W_{ik}$; $R_j \leftarrow R_j + W_{ij}$.
20: $\quad\quad$ **if** $R_j \geq 0$, break. Continue with the next iteration of the `for` loop (Line 6).
21: $\quad$ **if** $R_j < 0$, return $f \leftarrow$ `infeasible`; "heuristic failed".
22: return $f \leftarrow$ `feasible`; $x_{ij} \leftarrow 1,\ \forall i \in I_j$, else $x_{ij} \leftarrow 0$; $u_j \leftarrow \frac{\sum_{i \in I_j} W_{ij} x_{ij}}{C_j},\ \forall j \in J$; $\overline{z} \leftarrow \sum_{j \in J} C_j (1 - u_j)^2$.

4.3. Local search approaches

In this section, we describe two local search heuristics to improve a given feasible solution for Model (3.1). We seek to ensure feasibility of the solution in each refinement, and consider two algorithms motivated by the work in Osman (1995). The neighbourhoods used by these two local search approaches are:

- reassigning a single user to a different facility, and
- swapping the assignments of two users.

Such simple algorithms are well-known in the literature on local search algorithms, see for example Montes de Oca, Ner, and Cotta (2011), and we adapt these to our problem. In our algorithms, we consider a user and a facility we want to try assigning them to; in the case of swapping assignments, we need to choose two users and facilities. After checking that there is enough capacity for this reassignment or swap, we calculate the change in objective function

value arising from this change to the assignment. If this change indicates that it is better than our current assignment, we perform this reassignment or swap. Further, in our implementation, we only consider a pool of facilities that a user can be reassigned to based on the user's preferences; i.e., we include a cutoff of $P_{ij} = 0.2$ to consider j for user i; depending on the instance this cut-off should be changed accordingly. This choice seeks to ensure a greater likelihood of obtaining improved solutions and significantly reduces computational effort; such cut-offs have been implemented before in Risanger et al. (2021). We continue these random reassignments or swaps until a certain time limit is reached or until no significant improvements have been found after checking every single facility for whether reassigning a user of that facility to a different facility could lead to an improvement. As we show in Section 6.2, these local search algorithms are computationally cheap. We provide the pseudocodes and specific details for both these heuristics in Appendix B.

5

Heuristics for BFLP

Since we have now developed good and fast heuristics for the BUAP, we return to the main goal of the thesis in this chapter: developing heuristics for the BFLP, Model (2.2). Recall that in the model we are aiming to only have at most B facilities open – we want to determine the set of facilities to keep open, S, in such a way that still allows for a "good" user assignment. As such, the heuristics developed in Chapter 4 will be used to determine which facilities are "good" to close and also to compute the final assignment once the set of open facilities has been determined.

As we will make calls to the user assignment algorithms in Chapter 4, for consistency we define these as a single oracle in Algorithm 5. Here, the procedure takes as input a set of $|I|$ users and $|S|$ facilities and constructs the assignment using method m; choices of m include those we study in Chapter 4, such as `greedy_assignment`, `greedy_assignment` with `local_search_reassign`, `relaxation_rounding`, or `relaxation_rounding` with `local_search_swap`. In the following discussion, by "assignment" we refer to the solution x returned by `user_assignment`. We further define a "partial assignment" as a solution to Model (3.1), x, where Constraints (3.1c) are changed to $\sum_{j \in S} x_{ij} \leq 1$.

Algorithm 5 The `user_assignment` procedure(I, S, m)

Input: an instance of Model (3.1); a method m to construct a feasible solution of the instance.
Output: status f = either `feasible` or `infeasible` for the given inputs; if `feasible`: solution x, utilisation u, and objective function value $\overline{z}$; if `infeasible`: $x \leftarrow 0, u \leftarrow 0, \overline{z} \leftarrow +\infty$.

1: Run assignment method m on the input instance and return the result.

The first heuristic for Model (2.2) that we discuss is the close greedy algorithm, based on the ADD procedure discussed in Section 2.3.1 and in Jacobsen (1983). The basic idea of this approach is that we close one facility at a time, each time the facility that leads to the best objective function value. We build up to the final algorithm, which means we start with this very basic idea and then discuss the improvements we made one by one. How the algorithm performs after each improvement can be seen in Section 6.3. One of these improvements to the algorithm, discussed in Section 5.1.2, brings us back to the BUAP, as we adapt the `greedy_assignment` algorithm, Algorithm 3, specifically for the purpose of close greedy.

The second heuristic for Model (2.2) that we discuss is the open greedy algorithm, based on the DROP procedure discussed in Section 2.3.1 and in Jacobsen (1983). We immediately use what we have learned from developing close greedy. As such, we also need to separately adapt the `greedy_assignment` algorithm for the purposes of open greedy. We do so in Section 5.2.1.

Results of the open greedy algorithm can be found in Section 6.4.

The final heuristic we discuss is a local search heuristic in Section 5.3. This local search heuristic tries to improve a given solution, by trying closing one facility and opening another facility to see if the resulting solution is better than the current solution. The algorithm combines the local search procedures described in Jacobsen (1983), which we also discuss at the start of Section 2.3.2, and adapts them to our problem.

5.1. Close greedy algorithm

In this section, we build up to the final version of our close greedy algorithm, which is based on the DROP procedure of Jacobsen (1983).

5.1.1. Recomputing assignment every iteration

Our first adaptation of the DROP procedure can be seen in Algorithm 6. Recall that the idea here is to close the facilities one by one, at each step the facility which is the best to close based on the objective function value. We discuss the algorithm in more detail now. The inputs are an instance of Model (2.2), a user assignment method m to be used, and a (potentially different) second user assignment method m'. For more details on the `user_assignment`, see Algorithm 5 and Chapter 4. We first initialise the set of open facilities, S, to contain all facilities and then run the user assignment method m with all facilities open (Line 1). In the `while` loop, we close one facility in each iteration (Lines 2-9). At each iteration, we only consider a subset J' of all the currently open facilities, based on which n_c facilities have the lowest utilisation in the current assignment (Line 3). The idea of only considering the facilities with the lowest utilisation is similar to the idea used in the local search algorithm in Schmitt and Singh (2021). We do not want to consider every facility at each iteration since this could lead to very high run times. However, this means that we might not make the best choice at each step.

In the following for loop (Lines 5-7), we try closing a single facility in each iteration, computing the new assignment with method m (Line 6). We update the best solution found so far, if we find a better solution (Line 7). This for loop could be parallelised to speed up the algorithm; however, this was not implemented since other methods that we discuss later to speed up the algorithm already work well. Note that if none of the facilities could be closed with a feasible assignment, at this point the algorithm has to report a failure to construct a feasible solution (Line 8). If a feasible solution was achieved for at least one of the possible closures, the "best" facility – so the facility that increases the objective function value the least – is closed and all other information regarding the current assignment is updated (Line 9). The `while` loop continues until enough facilities are closed. Then, we run the final assignment method m' and the solution is updated if this leads to a better solution (Line 10). This is done to allow the final solution to be as good as possible: we might want to use a quick method m when closing facilities throughout the algorithm, e.g. `greedy_assignment`, but once we know the open facilities we can then run a better but slower method m', e.g. `relaxation_rounding` with `local_search_reassign`. Finally, the constructed solution is returned (Line 12).

In our implementation of Algorithm 6 that we use for the numerical experiments in Section 6.3.1, we compute the quantity $W_{ij} = P_{ij}U_i$ beforehand and further use preference cutoffs of 0.2 in the BUAP local search approaches. Additionally, when using $m =$ `relaxation_rounding`, we build the model only once and edit it in future iterations, thereby saving computational effort. In particular, when wanting to compute the assignment to $S \setminus \{j\}$ when, in a previous iteration, we computed it for S, we remove j from the set of open facilities and add it to the set of closed facilities. In addition, the implementation of Equation (3.1c) has its right-hand size set by a parameter. Hence, we update this parameter to 0 for j. Finally, the x_{ij} values for the newly closed facility are fixed to 0. In case the previous version of the model includes a

Algorithm 6 The `close_greedy_basic` algorithm

Input: an instance of Model (2.2); methods m, m' for Oracle `user_assignment` defined in Algorithm 5; scalar $n_c \leq |J|$.

Output: status f = either `feasible` or `infeasible` for the given inputs; if `feasible`, a set of open facilities S, an assignment x for the input instance, and the corresponding objective function value $\overline{z}$.

1: **Initialise:** $S \leftarrow J$; $[f, x, u, \overline{z}] \leftarrow$ `user_assignment`(I, S, m).
2: **while** $|S| > B$ **do**
3: $J' \leftarrow \{$indices of smallest n_c values of $u_j, j \in S\} \subseteq J$.
4: $[f^*, x^*, u^*, z^*, j^*] \leftarrow$ [`infeasible`, $0_{|I|,|J|}, 0_{|J|}, +\infty$, "$None$"].
5: **for** $j' \in J'$ **do**
6: $[f', x', u', z'] \leftarrow$ `user_assignment`$(I, S \setminus \{j'\}, m)$.
7: **if** f' = `feasible` and $z' < z^*$, $[f^*, x^*, u^*, z^*, j^*] \leftarrow [f', x', u', z', j']$.
8: **if** f^* = `infeasible`, return $f \leftarrow$ `infeasible`; "heuristic failed".
9: $S \leftarrow S \setminus \{j^*\}$; $[f, x, u, \overline{z}] \leftarrow [f^*, x^*, u^*, z^*]$.
10: $[f', x', u', z'] \leftarrow$ `user_assignment`(I, S, m').
11: **if** f' = `feasible` and $z' < \overline{z}$, $[f, x, u, \overline{z}] \leftarrow [f', x', u', z']$.
12: return f; S; x; $\overline{z}$.

facility in the set of closed facilities that we now want to be open, we additionally need to do the opposite for this facility as well, e.g. unfix the x_{ij} values. Within relaxation rounding, we further consider only a set of 20 most preferred facilities for each i; this removes a significant number of x_{ij} variables from the model. If the corresponding models are infeasible, we rebuild with a new set of 20 preferred facilities – as over multiple iterations all of the most preferred 20 facilities might have been closed.

The strength of this algorithm rests on being able to compute good user assignments multiple times and speedily; it is here that the quick implementation methods we discuss in Chapter 4 are useful. However, overall they are still slower than we would hope for. For example, in our computational experiments, we find each `user_assignment` call takes about two–thirds of a second for instances with $|I| = 2060, |J| = 1394$ for m = `greedy_assignment`. Reducing run times is necessary since our computational results in Section 6.3.1 show that for the aforementioned instance we can only consider 5 facilities at each iteration if we want the algorithm to run in a "reasonable" amount of time. This leads to the objective function value being too far away from the optimal solution. Hence, we will consider how to speed up Line 6, the call to `user_assignment` within the loops, in the next section.

5.1.2. Reusing previous assignment

As mentioned in the previous section, even with the fastest user assignment method, Algorithm 6 is still too slow. Hence, we need to improve upon this. The idea we discuss now is to reuse the previous iteration's assignment; as such, we wish to optimise Algorithm 3 for its use within the close greedy algorithm. At the start of Algorithm 6, we compute a user assignment assuming all facilities are open (Line 1). Instead of recomputing the assignment from scratch in Line 6, we only wish to adapt the assignment from the previous iteration's assignment. We are only closing a single facility j' at a time, so the only changes that need to be made to the assignment to make it feasible is to reassign the users currently assigned to facility j'. Hence, we want to design a quick algorithm that given an assignment x of users to facilities in S constructs an assignment of users to facilities in $S \setminus \{j'\}$. The original `greedy_assignment` algorithm can be

adapted to do so by starting it with the partial assignment and only asking it to assign the users that need reassigning, i.e. the users previously assigned to j', greedily to the facilities in $S \setminus \{j'\}$. The changes that need to be made to the `greedy_assignment` algorithm, Algorithm 3, can be seen in Algorithm 7. Note that these changes only concern the initialisation steps before going into the main `while` loop: the users that need to be assigned are now only the ones currently assigned to j', we need to remove j' from the set of open facilities and mark all users assigned to it to have no assignment yet, and finally we need to calculate how much capacity each facility still has left given the partial assignment (Line 1).

Algorithm 7 The `greedy_reassign` algorithm

Input: an instance of Model (3.1); the facility to close j'; an assignment x of I to S; a function sort(L, A_l, ascending/descending) that sorts elements $l \in L$ depending on their value A_l in ascending or descending order.

Output: status f = either `feasible` or `infeasible` for the given inputs; if `feasible` returns an assignment x for the input instance with corresponding utilisation u, and objective function value $\overline{z}$.

1: **Initialise**: $I' \leftarrow \{i \in I : x_{ij'} = 1\}$; $x_{ij'} \leftarrow 0$, $\forall i \in I'$; $S \leftarrow S \setminus \{j'\}$; $R_j \leftarrow C_j - \sum_{j \in I} U_i P_{ij} x_{ij}$, $\forall j \in S$.

2: Lines 2 - 13 from Algorithm 3.

Then, the only change we are making to Algorithm 6 is to replace Line 6 with a call to Algorithm 7. In case these assignments get worse over time since the reassignment might be worse than completely recomputing the solution, one could completely recompute the assignment after some number of facilities have been closed as the last thing to do in the `while` loop – in practice this did not lead to any visible improvement, so we will not discuss this in any more detail.

As seen in Section 6.3.2, this section's improvements allow us to increase n_c to 200 and still have a reasonable running time. This leads to an improvement in the objective function value. However, the heuristic still leads to results that are significantly worse than what the `MIP` solver achieves on larger instances. Hence, we discuss one last improvement to the algorithm in the next section.

5.1.3. Further improvements on choosing facilities

Since the heuristic performs very well on a smaller instance, where we are able to consider all facilities in every iteration, the way we choose which facilities to consider might not focus us on the actual relevant facilities. This leads us to our last idea for where we can improve the heuristic: We need to reconsider how to make a good choice of the facilities we consider at each iteration. Currently, this choice is made based on which facilities have the lowest utilisation, in the hope that closing a barely utilised facility has little effect on the objective function, as also done in Schmitt and Singh (2021). However, while the utilisation is indirectly related to the objective function value, this might not be the best indicator to use. For example, there might be some high utilisation facilities that we can close and reassign their users to a nearby facility without affecting the objective function value too negatively.

The change to the algorithm we are making now is hence in the way we choose the facilities to consider. Algorithm 8 includes these changes to the way the facilities to consider at each step are chosen, as well as using the `greedy_reassign` method discussed in the previous section. The basic idea of how we choose which facilities to consider is to choose the ones that we found to be "good" to close in a previous iteration, but that we did not close yet.

Let us now explain the changes made regarding choosing the facilities to consider in more

detail. In the first iteration, we simply consider all facilities, instead of just n_c facilities. This can be seen in the algorithm in Line 1, where the set of the facilities to consider, J', is initialised to J. We save the change in objective function that closing facility j would lead to in δ_j (Line 7). Then, in every next iteration, we consider the n_c facilities that are still open that previously led to the smallest change in objective function value (Line 10). Note that we are considering facilities with low δ_j since this indicates the objective function is increased by the smallest amount possible, which is what we are aiming for in this minimisation problem. When a facility is considered, its δ_j value is also updated. This means that some δ values will be more "outdated" than others.

Algorithm 8 The `close_greedy` algorithm

Input: an instance of Model (2.2); methods m, m' for Oracle `user_assignment` defined in Algorithm 5; scalar $n_c \leq |J|$.

Output: status f = either `feasible` or `infeasible` for the given inputs; if `feasible`, a set of open facilities S, an assignment x for the input instance, and the corresponding objective function value $\overline{z}$.

1: **Initialise:** $S \leftarrow J$; $J' \leftarrow J$; $\delta_j \leftarrow 0$, $\forall j \in J$; $[f, x, u, \overline{z}] \leftarrow$ `user_assignment`(I, S, m).
2: **while** $|S| > B$ **do**
3: $\quad [f^*, x^*, u^*, z^*, j^*] \leftarrow$ [`infeasible`, $0_{|I|,|J|}, 0_{|J|}, +\infty$, "$None$"].
4: $\quad$ **for** $j' \in J'$ **do**
5: $\quad\quad [f', x', u', z'] \leftarrow$ `greedy_reassign`(I, S, j', x).
6: $\quad\quad$ **if** $f' =$ `feasible` and $z' < z^*$, $[f^*, x^*, u^*, z^*, j^*] \leftarrow [f', x', u', z', j']$.
7: $\quad\quad \delta_{j'} \leftarrow z' - \overline{z}$.
8: $\quad$ **if** $f^* =$ `infeasible`, return $f \leftarrow$ `infeasible`; "heuristic failed".
9: $\quad S \leftarrow S \setminus \{j^*\}$; $[f, x, u, \overline{z}] \leftarrow [f^*, x^*, u^*, z^*]$.
10: $\quad J' \leftarrow \{j \in S :$ indices of smallest n_c values of $\delta_j\} \subseteq J$.
11: $[f', x', u', z'] \leftarrow$ `user_assignment`(I, S, m').
12: **if** $f' =$ `feasible` and $z' < \overline{z}$, $[f, x, u, \overline{z}] \leftarrow [f', x', u', z']$.
13: return f; S; x; $\overline{z}$.

We are hoping that if a facility was bad to close in an iteration, how bad it is to close is unlikely to get any better in the next few iterations. This is especially the case for these large instances, where a closure of a facility is unlikely to influence how bad it is to close a facility far away from that closed facility.

However, the previous is not a mathematical precise statement and there exist counterexamples to this general intuition as to why this method might work well. In particular, δ_j is not guaranteed to only increase from one iteration to the next; it is possible for it to decrease, as seen in Example 1. It would be good if it could only increase since this would mean that even if a facility was bad in the first iteration, we would start considering it again if everything else is worse after a few iterations of updating δ and making potentially suboptimal choices. If δ_j could decrease, that could mean that a facility that would be good to close in iteration 100 would never be considered again if it was bad to close in the first iteration. Note however that Example 1 tells us more about how the `greedy_assignment` method (and hence the `greedy_reassign` method) works than about the overall structure of the close greedy algorithm – it exploits that a user's most preferred facility may not be the best facility to assign it to.

Hence, the statement that δ_j can only increase could be true if the optimal assignment was to be computed at every iteration. Note that what we are trying to prove here about the function with input S defined by an optimal solution to the `BUAP`, including the closed facilities in the objective function, is that it is supermodular. See Schrijver (2003) for a more detailed discussion

on supermodularity and submodularity. In other words, if the function is supermodular, δ_j cannot decrease by definition. See Proposition 3 for the formal statement of this. However, we can find a counterexample, showing that δ_j can decrease even if the assignments are made optimally, as seen in Example 2. This example exploits that the capacity of the facilities is limited. Hence, the function described in Proposition 3 is not supermodular.

Example 1. *We would like to show that δ_j can decrease from one iteration to the next in Algorithm 8. This is assuming the starting solution is created with the* `greedy_assignment` *method.*

For simplicity, we define $W_{ij} = U_i P_{ij}$ again. We consider an instance with $I = \{1,2,3,4\}$ and $J = \{1,2,3,4\}$. Let $C_1 = 10$, $C_2 = 10$, $C_3 = 25$ and $C_4 = 40$. Let the W_{ij} be defined as below, with each user having $U_i = 10$:

$$\begin{aligned} W_{1,1} &= 10 & W_{1,2} &= 0 & W_{1,3} &= 9 & W_{1,4} &= 0 \\ W_{2,1} &= 0 & W_{2,2} &= 10 & W_{2,3} &= 8 & W_{2,4} &= 7 \\ W_{3,1} &= 0 & W_{3,2} &= 0 & W_{3,3} &= 10 & W_{3,4} &= 0 \\ W_{4,1} &= 0 & W_{4,2} &= 0 & W_{4,3} &= 0 & W_{4,4} &= 10. \end{aligned}$$

We use superscripts to denote the iteration of the algorithm we are in, both for the objective and the values of δ. At the start of the algorithm, the `greedy_assignment` *algorithm assigns user i to facility i. Now, the objective value of this assignment is*

$$obj^0 = 0 + 0 + 25\left(\frac{15}{25}\right)^2 + 40\left(\frac{3}{4}\right)^2 = 31.5. \tag{5.1}$$

If we close facility 1, user 1 would be reassigned to facility 3 in the `greedy_reassign` *algorithm. This leads to*

$$\delta_1^1 = 10 + 0 + 25\left(\frac{6}{25}\right)^2 + 40\left(\frac{3}{4}\right)^2 - obj^0 = 33.94 - 31.5 = 2.44. \tag{5.2}$$

Similarly, if we close facility 2, user 2 is reassigned to facility 3. Hence,

$$\delta_2^1 = 0 + 10 + 25\left(\frac{7}{25}\right)^2 + 40\left(\frac{3}{4}\right)^2 - obj^0 = 34.46 - 31.5 = 2.96. \tag{5.3}$$

Note that $\delta_3^1, \delta_4^1 > \delta_1^1$ simply since closing these facilities increases the objective function value by 16 / 17.5 respectively since when their users are reassigned, they contribute 0. So, in the first iteration, we close facility 1 and have an objective function value of $obj^1 = 33.94$ now.
We now consider what δ_2^2 is. Since the capacity of facility 3 is now more filled, user 2 has to be reassigned to facility 4 instead:

$$\delta_2^2 = 10 + 10 + 25\left(\frac{6}{25}\right)^2 + 40\left(\frac{23}{40}\right)^2 - obj^1 = 34.665 - 33.94 = 0.725 < \delta_2^1. \tag{5.4}$$

Hence, we conclude that δ_j can decrease from one iteration to the next in Algorithm 8.

Proposition 3. *δ_j cannot decrease from one iteration to the next in Algorithm 8, assuming the user assignments are made optimally, if $f(S)$ defined by*

$$f(S) = \min_x \sum_{j\in J} C_j \left(1 - \frac{\sum_{i\in I} U_i P_{i,j} x_{i,j}}{C_j}\right)^2 \tag{5.5a}$$

$$s.t. \sum_{i\in I} U_i P_{i,j} x_{i,j} \le C_j \qquad \forall j \in S \tag{5.5b}$$

$$\sum_{j\in S} x_{i,j} = 1 \qquad \forall i \in I \tag{5.5c}$$

$$x_{ij} = 0 \qquad \forall j \in J \setminus S \tag{5.5d}$$

$$x_{i,j} \in \{0,1\} \qquad \forall i \in I, j \in J. \tag{5.5e}$$

is supermodular.

Proof. Let us first formalise the statements in question. δ_j cannot decrease from one iteration to the next if $\delta_j^t \le \delta_j^{t+1}$, where t indicates which iteration this δ was calculated. This can be expanded as

$$\delta_j^t = f(S^t \setminus \{j\}) - f(S^t) \le f(S^t \setminus \{j, j^*\}) - f(S^t \setminus \{j^*\}) = f(S^{t+1} \setminus \{j\}) - f(S^{t+1}) = \delta_j^{t+1}, \tag{5.6}$$

where S^t is the set of open facilities at iteration t and j^* is the facility that was chosen to be closed at iteration t. Furthermore, the function $f(T)$ is supermodular if, according to an equivalent definition in Schrijver (2003),

$$f(T \cup \{x\}) + f(T \cup \{y\}) \le f(T) + f(T \cup \{x, y\}),\ \forall T \subset J, x \ne y \in J \setminus T. \tag{5.7}$$

Setting $T = S^t \setminus \{j, j^*\}$, $x = j$, $y = j^*$ this results in

$$f(S^t \setminus \{j^*\}) + f(S^t \setminus \{j\}) \le f(S^t \setminus \{j, j^*\}) + f(S^t). \tag{5.8}$$

Rearranging Equation (5.8) results in Equation (5.6), which completes the proof. □

Example 2. *We provide a counterexample for $\delta_j^t \le \delta_j^{t+1}$ in Algorithm 8, assuming the assignments are calculated optimally. Consider an instance of Model (2.2) with $I = \{1,2,3,4\}$ and $J = \{1,2,3,4\}$. Let $C_j = 20$ for all $j \in J$. For simplicity, let $W_{ij} = U_i P_{ij}$ and define W_{ij} by*

$$W_{ij} = \begin{cases} 20 & \text{if } i = j, \\ 5 & \text{otherwise.} \end{cases} \tag{5.9}$$

When all facilities are open, the optimal assignment assigns user i to facility j, which leads to an objective value of 0. Without loss of generality, assume that facility 1 is closed in the first iteration. Closing a single facility leads to an optimal solution of $20 + 0 + 20(\frac{10}{20})^2 + 20(\frac{15}{20})^2 = 20 + 5 + 11.25 = 36.25$. Hence, $\delta_i^1 = 36.25$ for all $i \in I$. Now, once another facility is closed, the optimal objective function value becomes $20 + 20 + 20(\frac{10}{20})^2 + 20(\frac{10}{20})^2 = 50$. Hence, $\delta_i^2 = 50 - 36.25 = 13.75 < 36.25 = \delta_i^1$ for $i \in \{2,3,4\}$. We have shown that $\delta_j^t > \delta_j^{t+1}$ is possible and therefore, according to Proposition 3, $f(S)$ is not supermodular.

5.1.4. Conclusion

Since we have reached a good solution quality and runtime as seen in Section 6.3.3, we conclude on the close greedy approach now. Any further change to the heuristic is unlikely to lead to a significant improvement, and a better idea is to just run a local search once this close greedy algorithm has terminated. This is discussed in Section 5.3. Other small improvements that could be considered for close greedy are:

- Also considering a certain number of random facilities at each iteration.
- Parallelise the `for` loop in Algorithm 8.

5.2. Open greedy algorithm

The second approach we consider is based on the `ADD` algorithm by Jacobsen (1983). Instead of closing facilities one by one as in the previous section, we now open facilities one by one. This is more difficult than closing facilities one by one for two reasons:

- At the beginning, the problem is infeasible, i.e. when there are zero open facilities, no users can be assigned to a facility. Even when we start opening facilities, the `BUAP` might not be feasible since there are too few facilities available.
- Unlike when closing facilities, we cannot easily adapt the `greedy_assignment` algorithm, Algorithm 3, as done in Algorithm 7 by just changing the initialisation to deal with a partial assignment.

The first problem was solved in Sridharan (1995) by adding an artificial facility with large capacity and which has a large cost of assigning users to it. Since our objective function is slightly differently structured, and we have no cost for assigning users to facilities, instead, we allow some users not to be assigned in the first few iterations of the algorithm. Since assigning users that are previously unassigned to a facility can only decrease the objective function value, this is reasonable.

The second problem we address in more detail in Section 5.2.1, before discussing the open greedy algorithm in Section 5.2.2. We present the results of the open greedy algorithm in Section 6.4.

5.2.1. Adapting greedy assignment to open facilities

In this section, we adapt Algorithm 3 to allow for its use in the open greedy algorithm in a way that reuses the previous iteration's assignment. To this extent, the algorithm needs to deal with two things:

- If there are unassigned users, it should try to assign them to the newly opened facility.
- Afterwards or if there are no unassigned users, it needs to try and reassign some already assigned users to the newly opened facility.

We now discuss how Algorithm 9 achieves these goals. Note that the inputs to the algorithm are a (partial) assignment of users to S and a facility j' to open. The algorithm then needs to adapt this assignment to account for this newly opened facility, returning a new assignment of users to facilities in $S \cup \{j'\}$. The algorithm first initialises the set of unassigned users I' as the set of users that has no j for which $x_{ij} = 1$ and sets R_j to the capacity remaining in facility j given the partial assignment (Line 1). Next, Lines 2-12 from Algorithm 3 with Line 7 replaced by "break" are run. This tries to assign the unassigned users in a greedy way. We do not want to return the solution in Line 7 like in Algorithm 3 since we still want to do some reassignment to the newly opened facility. That is what the rest of the algorithm does. Apart from assigning

Algorithm 9 The `greedy_reassign_open` algorithm

Input: an instance of Model (3.1); the facility to open $j' \in J \setminus S$; a (partial) assignment of I to S x; the depth d of the local search; a function `sort`$(L, A_l,$ ascending/descending) that sorts elements $l \in L$ depending on their value A_l in ascending or descending order.

Output: status f = either `feasible` or `infeasible` for the given inputs; if `feasible`, returns an assignment x for the input instance with corresponding utilisation u and objective function value $\overline{z}$. If `infeasible`, it also returns these, but the assignment x does not assign every user to a facility.

1: **Initialise:** $S \leftarrow S \cup \{j'\}$; $I' \leftarrow \{i \in I : x_{ij} = 0,\ \forall j \in S\}$; $R_j \leftarrow C_j - \sum_{j \in I} U_i P_{ij} x_{ij},\ \forall j \in S$.
2: Lines 2-12 from Algorithm 3 with Line 7 replaced by "break to Line 3 in this algorithm".
3: $J' \leftarrow \{j'\}$; $k \leftarrow 0$.
4: **while** $k < d$ **do**
5: $\quad J'_k \leftarrow \emptyset$.
6: $\quad$ **for** $j^* \in J'$ **do**
7: $\quad\quad I'' \leftarrow$ `sort`$(\{i \in I : U_i P_{ij^*} \leq R_j, x_{ij^*} = 0\}, P_{ij^*},$ descending).
8: $\quad\quad$ **for** $i \in I''$ **do**
9: $\quad\quad\quad$ **if** $i \in I'$ and $U_i P_{ij^*} \leq R_{j^*}$
10: $\quad\quad\quad\quad x_{ij^*} \leftarrow 1$; $R_{j^*} \leftarrow R_{j^*} - U_i P_{ij^*}$; $I' \leftarrow I' \setminus \{i\}$.
11: $\quad\quad\quad$ **else**
12: $\quad\quad\quad\quad j'' \leftarrow \arg\max_{j \in J}\{x_{ij}\}$ ▷ This gets the facility i is assigned to.
13: $\quad\quad\quad\quad$ **if** `if_reassignment_better`(i, j^*, j'', R_j) = `True`
14: $\quad\quad\quad\quad\quad x_{ij''} \leftarrow 0$; $x_{ij^*} \leftarrow 1$; $R_{j''} \leftarrow R_{j''} + U_i P_{ij''}$; $R_{j^*} \leftarrow R_{j^*} - U_i P_{ij^*}$.
15: $\quad\quad\quad\quad\quad J'_k \leftarrow J'_k \cup \{j''\}$.
16: $\quad J' \leftarrow J'_k$; $k \leftarrow k + 1$.
17: **if** $|I'| = 0$, $f \leftarrow$ `feasible`, **else** $f \leftarrow$ `infeasible`.
18: return f; x; $u_j \leftarrow \frac{\sum_{i \in I} W_{ij} x_{ij}}{C_j},\ \forall j \in J$; $\overline{z} \leftarrow \sum_{j \in J} C_j (1 - u_j)^2$.

users to j' we also want to potentially try to reassign users to facilities that had users taken away from them (to be assigned to j') since they now might have enough capacity for some other users. This is the role of the parameter d: it determines how "deep" the local search is. $d = 1$ means we only try to reassign users to j'; $d = 2$ means we afterwards also try to reassign users to the facilities that lost users to j'; $d = 3$ means we additionally also try to reassign users to facilities that lost users in the previous iteration and so forth.

Hence, we initialise the set of facilities J' that we are trying to reassign users to as just j' and set the counter k that determines how "deep" we are into the local search to 0 (Line 3). We then start a while loop that terminates when the input depth d is reached (Line 4). Next, we initialise the set of facilities we wish to consider in the next iteration, J'_k to the empty set (Line 5). For each facility $j^* \in J'$, we then determine the set of users that we want to try to reassign to j^*. To this extent, we only want to consider users for which j^* has enough capacity, and we want to first consider users that have a higher preference for j^* (Line 7). Then, we distinguish between two cases: users that are unassigned (Line 9) and users that are already previously assigned to a facility (Line 11). In the former case, we simply check that the facility has enough capacity for the user (Line 9) and then assign it to the facility, updating the remaining capacity R_{j^*} in the process and removing the user from the set of unassigned users (Line 10). This is guaranteed to improve the objective function since an unassigned user had no effect on the objective previously and now decreases it. In the latter case, we determine the facility that the user is currently assigned to (Line 12) and then use Algorithm 13 to determine if the reassignment of user i to

j^* is feasible and leads to a better solution (Line 13). If it does, we update the assignment and the remaining capacities of the two involved facilities (Line 14). We also update the facilities to be considered in the next iteration to include facility j'', which is the facility from which we just took a user away. Once all facilities in J' are considered, we update the set of facilities to consider for the next iteration and increase the counter (Line 16).

Since the assignment could be infeasible if not enough facilities are open for all the users to be assigned with this greedy routine, we check if all users are assigned to determine if the assignment is feasible (Line 17). Finally, we return the result (Line 18). Even if the assignment is not feasible, we want to return it since the partial assignment can be used in future iterations of the open greedy algorithm, where this algorithm is used.

Overall, this algorithm can be best described as combining the `greedy_assignment` algorithm, Algorithm 3, by Schmitt and Singh (2021) with a more targeted version of the `local_search_reassign` algorithm, Algorithm 12. This local search is necessary for opening facilities but not for closing facilities, since when opening facilities, there is no clear set of users that need to be reassigned. Without this local search, no users would be assigned to the newly opened facility if the input assignment was already feasible, which defeats the purpose of assessing how good the facility would be to open. In the next section, we discuss the open greedy algorithm that uses Algorithm 9 as a subroutine.

5.2.2. Open greedy algorithm

Now that we have adapted the `greedy_assignment` algorithm in a way that allows us to build upon a previous assignment when opening a facility, we discuss the overall open greedy algorithm. This idea is based on the `DROP` approach discussed in Jacobsen (1983). We directly use what we have learned from the close greedy algorithm:

- Using the previous iteration's assignment speeds up the algorithm.
- Considering facilities that were good to consider in previous iterations works well.

This results in Algorithm 10. The algorithm starts by initialising everything we need: the set S of open facilities is initialised to the empty set, the set J' of the facilities to consider each iteration is initialised to all facilities, the change in objective function for each facility is initialised to 0 and the solution is initialised to be infeasible (Line 1). Then, we start opening facilities one by one, at each step choosing the best facility within the set J' to open. Note that we accept solutions that are infeasible, unless we already reached a feasible solution in the previous iteration (Line 6). This is because the algorithm starts with no facilities, so for the first few iterations the assignments are infeasible. Every time we recalculate the objective function when considering to open j', we additionally update $\delta_{j'}$ to the change in objective function that is observed at this point in the algorithm (Line 7). After updating the assignment and the open facilities with the best facility to open (Line 8), we update the set of facilities to consider in the next iteration to be the ones that were the best to open in previous iterations, as indicated by δ_j (Line 9). We include the idea of recomputing the assignment from scratch every n_f iterations in Lines 10-12, in case the assignment done by the `greedy_reassign_open` algorithm needs improvement. Finally, once all the facilities have been opened, the final assignment method is run and the assignment is updated if this leads to a better assignment (Lines 13-14).

As with close greedy, we would like for δ to only increase in order for us to reconsider opening facilities again that might only have been bad to open at the beginning, but might be good to open once more facilities are already open. However, we can find a counterexample that shows that δ can decrease from one iteration to the next. This again exploits greedy (re)assignment not leading to the optimal solution, see Example 3. Note that again, as in Proposition 3, δ_j not being able to decrease is captured by the definition of supermodularity.

Algorithm 10 The `open_greedy` algorithm

Input: an instance of Model (2.2); methods m, m' for Oracle `user_assignment` defined in Algorithm 5; scalar $n_c \leq |J|$ for the number of facilities to consider each iteration; scalar depth $d \geq 1 \in \mathbb{Z}^+$ for the `greedy_reassign_open` algorithm; scalar $n_f \leq |J|$ for the number of iterations after which to fix the assignment.

Output: status f = either `feasible` or `infeasible` for the given inputs; if `feasible`, a set of open facilities S, an assignment x for the input instance and the corresponding objective function value $\overline{z}$.

Initialise: $S \leftarrow \emptyset$; $J' \leftarrow J$; $\delta_j \leftarrow 0,\ \forall j \in J$; $[f, x, u, \overline{z}] \leftarrow [\texttt{infeasible}, 0_{|I|,|J|}, 0_{|J|}, \sum_{j \in J} C_j]$.
while $|S| < B$ **do**
  $[f^*, x^*, u^*, z^*, j^*] \leftarrow [\texttt{infeasible}, 0_{|I|,|J|}, 0_{|J|}, +\infty, \text{“}None\text{”}]$.
  for $j' \in J'$ **do**
    $[f', x', u', z'] \leftarrow \texttt{greedy\_reassign\_open}(I, S, j', x, d)$.
    if (f' = `feasible` or $f' = f$) and $z' < z^*$, $[f^*, x^*, u^*, z^*, j^*] \leftarrow [f', x', u', z', j']$.
    $\delta_{j'} \leftarrow z' - \overline{z}$.
  $S \leftarrow S \cup \{j^*\}$; $[f, x, u, \overline{z}] \leftarrow [f^*, x^*, u^*, z^*]$.
  $J' \leftarrow \{j \in J \setminus S$: indices of smallest n_c values of $\delta_j\} \subseteq J$.
  if $|S| \equiv 0 \mod n_f$
    $[f', x', u', z'] \leftarrow \texttt{user\_assignment}(I, S, m)$.
    if f' = `feasible` and $z' < \overline{z}$, $[f, x, u, \overline{z}] \leftarrow [f', x', u', z']$.
$[f', x', u', z'] \leftarrow \texttt{user\_assignment}(I, S, m')$.
if f' = `feasible` and $z' < \overline{z}$, $[f, x, u, \overline{z}] \leftarrow [f', x', u', z']$.
return f; S; x; $\overline{z}$.

The same instance as in Example 2 can be used to show that, even if the assignments are made optimally, δ can decrease in the open greedy algorithm from one iteration to the next.

Example 3. *We would like to show that δ_j can decrease from one iteration to the next in Algorithm 10.*

We consider an instance with 3 facilities and 4 users. Let $C_1 = 20$, $C_2 = 10$ and $C_3 = 8$. Further, let $U_1 = 8$, $U_2 = 12$, $U_3 = 8$ and $U_4 = 8$. Let the P_{ij} be defined as below:

$$P_{1,1} = \frac{1}{2} \qquad P_{1,2} = \frac{5}{8} \qquad P_{1,3} = 1$$
$$P_{2,1} = 1 \qquad P_{2,2} = \frac{3}{4} \qquad P_{2,3} = \frac{2}{3}$$
$$P_{3,1} = \frac{1}{2} \qquad P_{3,2} = \frac{5}{8} \qquad P_{3,3} = 1$$
$$P_{4,1} = 1 \qquad P_{4,2} = \frac{5}{8} \qquad P_{4,3} = 1$$

We use superscripts to denote the iteration of the algorithm we are in, both for the objective and the values of δ. At the start of the algorithm no facilities are open and no users assigned, so the objective value is

$$obj^0 = 20 + 10 + 8 = 38. \tag{5.10}$$

If we open facility 1, users 2 and 4 would be assigned to it since they have the highest preference for facility 1 and after assigning them, the facility's capacity is reached. This leads to

$$\delta_1^1 = 10 + 8 - 38 = -20. \tag{5.11}$$

Similarly, if we open facility 2*, user* 2 *would be assigned to facility* 2 *and then not enough capacity is left for any of the other users to be assigned to facility* 2*. Hence,*

$$\delta_2^1 = 20 + 10\left(\frac{1}{10}\right)^2 + 8 - 38 = -9.9. \tag{5.12}$$

Note that $\delta_3^1 > \delta_1^1$ *since opening facility* 3 *decreases the objective function by at most* 8*. So, in the first iteration, we open facility* 1 *and have an objective function value of* $obj^1 = 18$*.*
We now consider δ_2^2*. Since users* 2 *and* 4 *are now already assigned, instead, Algorithm 9 will assign users* 1 *and* 3 *to facility* 2*. Hence, we have*

$$\delta_2^2 = 0 + 0 + 8 - 18 = -10 < -9.9 = \delta_2^1. \tag{5.13}$$

Hence, we conclude that δ_j *can decrease from one iteration to the next in Algorithm 10. Note that this example falls apart if the user assignments were done optimally since then* $\delta_2^1 = -10 = \delta_2^2$ *by assigning users 1 and 3 to facility 2. The same instance as in Example 2 can be used to show that, even if the assignments are made optimally,* δ *can decrease in the open greedy algorithm from one iteration to the next.*

5.2.3. Conclusion

We discuss the performance of Algorithm 10 in Section 6.4. On the two instances we consider first, it generally performs better than close greedy. On one instance we consider in Section 6.6, it performs worse and on the last instance it performs better than close greedy at lower budgets. As with the close greedy algorithm, possible improvements to the algorithm include considering a certain number of random facilities and parallelising the `for` loop.

5.3. Local search

In this section, we combine the two local search algorithms discussed in Sridharan (1995) based on `ADD` and `DROP` (or open greedy and close greedy in our case) into a single local search algorithm. We state and explain the algorithm in this section and present the results in Section 6.5.

The basic idea of the local search algorithm is to at each step open a facility and close a facility, where one of these is done in the best possible way. If this leads to a better solution, this is accepted as the current solution.

We now discuss the details of the `BFLP_local_search`, Algorithm 11. The algorithm starts by calling Algorithm 16, `initialise_change`. This algorithm simply computes the change δ_j in objective function value if a single facility j is closed / opened, depending on if it is currently opened or closed. Additionally, the set of facilities we want to consider is initialised to the whole set of facilities and the counter for the number of iterations of the while loop, k, is initialised to 0 (Line 1). The while loop terminates if either all facilities have been considered without any change occurring (i.e. $|J'| = 0$) or if the iteration limit l that is an input to the algorithm is reached. The latter could also be replaced by a time limit, but an iteration limit was chosen in order to make the results more comparable between similar instances.

At each iteration of the while loop, we first choose a facility using Algorithm 17 (Line 3). This algorithm chooses a facility that according to δ is good to consider, i.e. has small δ. Since δ is based on closing the facility if it is open and opening if it is closed, it would not make sense to simply choose the facility with the smallest δ: in that case as long as there are still closed facilities in J', one of those would be chosen since opening a facility cannot increase the objective. This is because in `greedy_reassign_open`, we only accept changes that improve the objective. Hence, in Algorithm 17 we first make a random choice, based on how many open facilities there are out of the total number of facilities, of whether to consider an open or closed facility. Then,

Algorithm 11 The `BFLP_local_search` algorithm

Input: an instance of Model (2.2); a feasible solution to the model (assignment x, the set of open facilities S, the objective function value $\overline{z}$); method m' for Oracle `user_assignment` defined in Algorithm 5; the number of facilities to consider each iteration, scalar $n_c \leq |J|$; scalar depth d for Algorithm 9; an iteration limit l for the main `while` loop of the algorithm.

Output: status f = `feasible`; if `feasible`, a set of open facilities S, an assignment x for the input instance and the corresponding objective function value $\overline{z}$.

1: **Initialise:** $\delta \leftarrow$ `initialise_change`$(x, S, \overline{z}, d)$; $J' \leftarrow J$; $k \leftarrow 0$.
2: **while** $|J'| > 0$ and $k < l$ **do**
3: $\quad j' \leftarrow$ `choose_fac_based_on_change`(J, J', S, δ); $J' \leftarrow J' \setminus \{j'\}$; $k \leftarrow k + 1$.
4: $\quad [f^*, x^*, u^*, z^*, j^*] \leftarrow$ [`infeasible`, $0_{|I|,|J|}, 0_{|J|}, +\infty$, "$None$"].
5: $\quad$ **if** $j' \in S$
6: $\quad\quad [f', x', u', z'] \leftarrow$ `greedy_reassign`(I, S, j', x); $\delta_{j'} \leftarrow z' - \overline{z}$.
7: $\quad\quad J'' \leftarrow \{j \in J \setminus S :$ indices of smallest n_c values in $\delta_j\}$.
8: $\quad\quad$ **for** $j'' \in J''$ **do**
9: $\quad\quad\quad [f'', x'', u'', z''] \leftarrow$ `greedy_reassign_open`$(I, S \setminus \{j'\}, j'', x', d)$; $\delta_{j''} \leftarrow z'' - \overline{z}$.
10: $\quad\quad\quad$ **if** f'' = `feasible` and $z'' < z^*$, $[f^*, x^*, u^*, z^*, j^*] \leftarrow [f'', x'', u'', z'', j'']$.
11: $\quad\quad$ **if** f^* = `feasible` and $z^* < \overline{z}$
12: $\quad\quad\quad S \leftarrow (S \cup \{j^*\}) \setminus \{j'\}$; $[f, x, u, \overline{z}] \leftarrow [f^*, x^*, u^*, z^*]$.
13: $\quad\quad\quad J' \leftarrow J$; $\delta_{j'} \leftarrow -\delta_{j'}$; $\delta_{j^*} \leftarrow -\delta_{j^*}$.
14: $\quad$ **else**
15: $\quad\quad [f', x', u', z'] \leftarrow$ `greedy_reassign_open`(I, S, j', x, d); $\delta_{j'} \leftarrow z' - \overline{z}$.
16: $\quad\quad J'' \leftarrow \{j \in S :$ indices of smallest n_c values in $\delta_j\}$.
17: $\quad\quad$ **for** $j'' \in J''$ **do**
18: $\quad\quad\quad [f'', x'', u'', z''] \leftarrow$ `greedy_reassign`$(I, S \cup \{j'\}, j'', x')$; $\delta_{j''} \leftarrow z'' - \overline{z}$.
19: $\quad\quad\quad$ **if** f'' = `feasible` and $z'' < z^*$ $[f^*, x^*, u^*, z^*, j^*] \leftarrow [f'', x'', u'', z'', j'']$.
20: $\quad\quad$ **if** f^* = `feasible` and $z^* < \overline{z}$
21: $\quad\quad\quad S \leftarrow (S \cup \{j'\}) \setminus \{j^*\}$; $[f, x, u, \overline{z}] \leftarrow [f^*, x^*, u^*, z^*]$.
22: $\quad\quad\quad J' \leftarrow J$; $\delta_{j'} \leftarrow -\delta_{j'}$; $\delta_{j^*} \leftarrow -\delta_{j^*}$.
23: $[f', x', u', z'] \leftarrow$ `user_assignment`(I, S, m').
24: **if** f' = `feasible` and $z' < \overline{z}$, $[f, x, u, \overline{z}] \leftarrow [f', x', u', z']$.
25: return f; S; x; $\overline{z}$.

we consider the best open / closed facility in the list of facilities we want to consider, J', based on it having the smallest δ. Coming back to the main algorithm, once this facility has been chosen, we remove it from the facilities to consider J' (Line 3).

We then split into two cases, depending on whether the chosen facility is open (Lines 5-13) or closed (Lines 14-22). Let us consider the case that j' is open; the other case follows analogously. First, we compute the assignment assuming j' is closed and update $\delta_{j'}$ accordingly (Line 6). Then, we choose the n_c facilities which we want to consider opening from the set of closed facilities based on which facilities have the lowest δ_j value (Line 7). We refer to this set as J''. We consider each facility in J'' and find the one to open that leads to the smallest objective function value, based on j' being closed and the corresponding assignment x' (Lines 8). If the best objective function value found this way is smaller than our current solution, we update the set of open facilities, S, by closing j' and opening j^* and update the assignment (Line 12). Additionally, we update J' to include all facilities again and negate the values of δ for the two facilities involved (Line 13). We update J' since now that we made a change, facilities that we

considered opening or closing previously might lead to an improvement if we consider them again. We update the δ values since the facilities have swapped from being open to being closed (or vice versa), so the change in objective function of undoing this is exactly the opposite.

Once the while loop terminates, due to either all facilities having been considered without any decrease in the objective function or the maximum iteration limit being reached, we recompute the assignment from scratch (Line 23). If this leads to a better assignment, we update our assignment (Line 24). Recall that the assignment methods are not guaranteed to find a feasible solution, even if one exists. Hence, it is possible for the call to `user_assignment` to return `infeasible` despite keeping a feasible assignment throughout the local search algorithm. Hence, the check for feasibility in Line 24 is necessary. Finally, the resulting set of open facilities S, the assignment x and the objective function value is returned (Line 25).

In Section 6.5, the results of this local search show that it performs well at improving bad solutions significantly. On the good solutions achieved by open and close greedy, the local search algorithm only makes minimal improvements before being stuck in a (local) minimum. Some changes that could be made to the algorithm to improve it include:

- Start with low n_c and increase it every n_i iterations by some value.
- Randomly allow some changes that make the solution worse, to see if we can escape a local optimum.

6

Results and discussion

In this chapter, we present and discuss the results of the heuristics discussed in Chapter 4 and Chapter 5. We start by discussing the input data that we run the algorithms on in Section 6.1. In Section 6.2, we then discuss the results of the heuristics for the Balanced User Assignment Problem, whose theory was discussed in Chapter 4. We then move on to the heuristics for our main problem, the Balanced Facility Location Problem. We first present and discuss the results of the close greedy algorithm, Algorithm 6, on two instances after each of the improvements that are discussed in Section 5.1, hence showing that these are indeed improvements to the algorithm. This can be found in Section 6.3. We then discuss the results of the open greedy algorithm, Algorithm 10, in Section 6.4. Then, we see if the `BFLP` local search can improve upon the results of open and close greedy in Section 6.5. We compare how the different heuristics perform on different instances, also comparing their performance to the algorithm developed in Schmitt and Singh (2021), in Section 6.6. Finally, we summarise our findings in Section 6.7.

In all these sections, we compare the results our heuristics achieve to the solution of the `MIP` solver, either after a certain time running it or in case it converges, the optimal solution. Hence, we use Equation (6.1) to calculate the difference to the `MIP` solution, where obj_{MIP} is the objective function value achieved by the `MIP` and obj_h is the objective function value achieved by the heuristic. Positive values indicate that the heuristic performs better than the `MIP`.

$$100 \times \frac{obj_{MIP} - obj_h}{obj_{MIP}} \tag{6.1}$$

We denote the result of Equation (6.1) by Δ_{MIP} if obj_{MIP} is the result the `MIP` achieves after 20,000 seconds. When obj_{MIP} is the solution of the `MIP` when it is given the same amount of time as the heuristic, we denote the result of Equation (6.1) by Δ_S. This is useful to see whether we can at least achieve a better result than the `MIP` given that amount of time when we are worse than the `MIP` after 20,000 seconds. Lastly, we use Δ_{OPT} if the obj_{MIP} value is the optimal objective function value.

All computations for this work are performed on the DelftBlue system with 1 core on an Intel XEON E5-6248R 24C 3.0GHz processor; see Delft High Performance Computing Centre (DHPC) (2022) for details on the system. The code and data used for the computations can be found at `https://github.com/Malena205/heuristics_quadratic_facility_location`.

6.1. Data

We now discuss the data we used. The main data set we use is the data set of users and recycling facilities in Bavaria, as created by Schmitt and Singh (2021). This data set consists of 2060 user

postcodes and 1394 postcodes with facilities. Note that if multiple facilities are in the same postcodes, they are counted as a single facility located at their centroid. We also consider a subset of this data set which consists of all the data for postcodes starting with 90, 91 or 92. This allows us to see how the algorithms scale from a smaller to a larger instance. This smaller instance consists of 368 users and 234 facilities. In the following sections, these are referred to as the large(r) and small(er) instance, respectively.

When testing the `BUAP` heuristics, we additionally need to fix the open facilities. To that extent, we create a total of 4 instances out of the above two instances: For each of them, we create one instance where the open facilities are the ones that the `MIP` of the `BFLP` when run for 20,000 seconds with a budget of 30% of all facilities opens and one instance where this is set to 90%. We look at 30% and 90% of facilities open since while 90% means there are more facilities to consider which could lead to longer run times, at 30% a solution might be more difficult to find due to limited capacity. We denote these instances by small-30, small-90, large-30 and large-90.

In addition, we also create some artificial data to allow us to compare how the different heuristics for the `BFLP` perform on different instances. These instances are only considered in Section 6.6. They are created by choosing n random longitudes and latitude pairs within a "rectangle" that either includes Germany or a subarea of Germany. By varying the "rectangle" size, we can create instances that have a different spread of users. For each chosen location, the user is assigned a population from a range uniformly at random. With probability r, a facility is chosen to be close to each user, i.e. to the longitude and latitude of the user a small Gaussian distribution with parameters 0, 0.01 is added, and a facility is placed there. The capacity of this facility is again chosen uniformly at random from a range of numbers. Finally, the parameters P_{ij} are calculated based on the distance between user and facility using the exponential decay formula used in Schmitt and Singh (2021). For more details on the data generation, please refer to Appendix D.

In this way, we produce two artificial instances. Artificial Instance 1 is created from a rectangle of size 9.2 times 3.1 (the units here arise from these being longitudes and latitudes, the area is approximately the size of Germany) with 5000 users and 2497 facilities. Artificial Instance 2 is created from a rectangle of size 1.5 times 1.7, approximately a fourth of the size of Bavaria, with 1500 users and 425 facilities. The first instance was created in order to test the heuristics on an even larger instance. The second instance was created to test the algorithms on a more "dense" instance – the users and facilities are in a smaller area here, which leads to larger values of P_{ij} and more users per facility. Both instances are created with user population ranges of 0 to 20,000 and facility capacity ranges of 0 to 80,000.

6.2. Results heuristics `BUAP`

We now discuss the performance of the `BUAP` heuristics that are discussed in Section 4. Table 6.1 shows the results for four different instances. Note that we run `relaxation_rounding` with $n_r = 20$, so for each user we only consider 20 x_{ij} variables in the relaxation. The first observation we make is that for all these methods and instances, the objective value is close to the optimal objective function value, and the heuristics are significantly faster than solving the `MIP`.

Overall, the differences to the optimal solution, or to the `MIP` when the `MIP` does not find an optimal solution within 20,000 seconds, are smaller when more facilities are open, i.e. smaller for 90 instances than for 30 instances, most likely due to a feasible solution being easier to find. Recall that the last two instances in the table have about 5.6 times more users than the first two instances. As such, we are able to solve the smaller instances to optimality with the `MIP`. Also, the time savings are not quite as large for the smaller instances as for the larger instances, but still very clear. Note that the run time is larger for the 90 instances than the 30 instances for `greedy_assignment`, but for `relaxation_rounding` this is the other way around.

This can be explained by the relaxation being more difficult to solve at lower budgets and more reassignments being required to make the solution feasible after rounding. As in both cases $n_r = 20$, the size of the model is the same for the 30 instance as for the corresponding 90 instance.

The `greedy_assignment` algorithm here does the worst and `relaxation_rounding` with `local_search_reassign` does the best. The `relaxation_rounding` heuristic even achieves the same result as the optimal solution, up to the accuracy we are considering here, on the smaller instances. Although `relaxation_rounding` does very well regarding its objective function value, the run time is slower than we would want because we want to re-run this heuristic multiple times. In particular, `relaxation_rounding` takes about 10 seconds for the larger two instances. Some time savings can be made by only building the model once when running similar models, as discussed in Section 5.1. However, we will see later that this is not sufficient for `relaxation_rounding` to be used as the main user assignment method m in the basic version of the close greedy algorithm. Comparing the two different local search heuristics, the `local_search_reassign` heuristic does better than the `local_search_swap` heuristic in the same amount or even less amount of time. The latter only leads to a very small improvement.

Table 6.1: Results of the different `BUAP` heuristics on different instances. Instances large-30 and large-90 are solved to a `MIP` gap of 0.02% and 0.35%, respectively, and all other `MIP` models achieve optimality in less than 20,000 seconds. Specifically, the optimal solution is reached in 15 seconds for small-30 and 28 seconds for small-90. Δ_{MIP} and Δ_{OPT} are defined in Equation (6.1). `relaxation_rounding` is run with $n_r = 20$. For details, see Section 6.2.

		small-30		small-90		large-30		large-90	
Heuristic	**Local search**	**Run time [s]**	Δ_{OPT} **[%]**	**Run time [s]**	Δ_{OPT} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
relaxation_ rounding	**none**	6.80	0.00	2.03	0.00	11.57	-0.28	10.07	-0.09
	reassign	6.81	0.00	2.05	0.00	11.77	-0.09	10.40	-0.05
	swap	6.81	0.00	2.05	0.00	11.82	-0.26	10.44	-0.05
greedy_ assignment	**none**	0.01	-1.05	0.02	-0.71	0.28	-1.31	0.67	-1.13
	reassign	0.02	-0.21	0.04	-0.03	0.48	-0.25	1.05	-0.17
	swap	0.03	-1.04	0.05	-0.68	0.52	-1.12	1.10	-0.99

To conclude, `relaxation_rounding` consistently leads to better objective function values than the `greedy_assignment` method, but takes significantly longer. Regardless, to make a final assignment once it is decided which facilities are open `relaxation_rounding` is useful, even if it is not useful if an assignment needs to be computed many times. Regarding the local search heuristics, reassigning users performs consistently significantly better than swapping users and both are computationally cheap.

6.3. Results close greedy algorithm

In this section, we discuss the results of the close greedy algorithm from Section 5.1 and how the changes made to it in that section improve the results. We consider the results on the smaller and the larger instance.

6.3.1. Results basic close greedy algorithm

First, we discuss the results of the basic close greedy algorithm, Algorithm 6. We vary the budget factor, which indicates the proportion of facilities that should remain open, to assess how

the algorithm performs at these different budgets. We compare the results of our close greedy heuristic to the objective value that the MIP program based on Model 2.2 reaches after 20,000 seconds. In the second column in Table 6.2, the gap reached by the MIP program after 20,000 seconds can be seen. This is the gap between the highest lower bound and the best solution the MIP finds. The second column for each user assignment method m of the table indicates how far the heuristic is from the MIP after 20,000 seconds. This is calculated using Equation (6.1) and denoted as Δ_{MIP}. The third column for each m is calculated analogously but with obj_{MIP} being the objective reached by the MIP in the time it takes the heuristic to reach its solution. The time for the MIP here includes the time to build the model. This is denoted by Δ_S.

Table 6.2: Results of close greedy algorithm, Algorithm 6, with m = `greedy_assignment` or m = `relaxation_rounding` throughout and m' = `relaxation_rounding` with `local_search_reassign` as the final assignment method. This is on the smaller instance. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the gap between the solution and the lower bound the MIP solver reaches after 20,000 seconds. The first column for each method m shows the run time of the close greedy heuristic. The second and third column for each method m are calculated based on Equation (6.1) with obj_{MIP} being the objective value reached after running the MIP for 20,000 seconds and running it for the number of seconds indicated in the second column, respectively. For details, see Section 6.3.1.

		m = **greedy_assignment** $n_c = \|J\|$			m = **relaxation_rounding** $n_c = 5$		
Budget factor	**MIP gap after 20k s [%]**	**Run time [s]**	Δ_{MIP} [%]	Δ_S [%]	**Run time [s]**	Δ_{MIP} [%]	Δ_S [%]
0.9	0.57	123	-0.14	-0.14	64	-1.80	-1.80
0.8	1.41	224	-0.09	-0.04	125	-5.70	-5.65
0.7	2.19	306	-0.11	-0.10	185	-10.00	-9.79
0.6	3.05	367	-0.10	0.26	247	-18.60	-18.17
0.5	3.53	413	-0.20	0.25	287	-22.44	-21.89
0.4	3.93	445	-0.15	0.72	394	-31.68	-30.54
0.3	4.12	466	-0.28	0.08	395	-32.34	-31.86
0.2	4.64	498	-0.09	0.98	494	-32.57	-31.15
0.1	4.18	500	0.00	0.02	537	-22.34	-22.32

Let us first consider the results for the smaller instance, as seen in Table 6.2. The objective function values are very close to what is achieved by the MIP for all budget factors when using `greedy_assignment` and $n_c = |J|$. Generally, for the middle budgets the heuristic is further away from the MIP result but still very close. In addition, for lower budgets the heuristic performs better than the MIP given the same amount of time, as indicated by the numbers in the fifth column becoming positive. This is due to the MIP becoming more difficult to solve at lower budgets, as indicated by the MIP gap in the second column. Therefore, the heuristic is able to reach a better solution in the same amount of time. However, the run time of our heuristic is slower than we would want, considering this is the smaller instance. Hence, for the larger instance, it seems not sensible to consider all facilities that are still open at each iteration since this would lead to too large run times. Also, recall that here we are using the `greedy_assignment` method in each iteration. When using the `relaxation_rounding` method at each iteration while still considering all facilities at each step, even at a budget of 0.9, the algorithm had not terminated after 7 hours. Hence, instead, we show the results at $n_c = 5$ when using `relaxation_rounding`. This performs significantly worse but leads to very similar run times. This performing worse is due to the decreased n_c, but increasing n_c further would lead to even longer run times, which considering the size of this instance does not appear reasonable.

Now, let us consider the larger instance. Since the run times are already high for the smaller instance, we only consider 5 facilities in each iteration when running the larger instance, as seen in Table 6.3. Compared to the smaller instance at $n_c = |J|$, the objective function value reached compared to the `MIP` is a lot worse, both comparing this to the `MIP` after 20,000 seconds and at the same time. It can also be seen that while `relaxation_rounding` takes longer than the `greedy_assignment` method, it leads to better results in the objective function for all but a budget of 0.1. That means that having a better estimator of how good closing a facility is helps to make a better choice. Lastly, let us note the worsening of the difference to the `MIP` as the budget decreases down to a budget of 0.3, which indicates that we are making more and more suboptimal choices when closing facilities. For the budgets of 0.2 and 0.1 this is not the case, possibly because it becomes harder for the `MIP` to solve such low budgets. Additionally, the difference to the `MIP` at the same time shows how slow the heuristic is: The `MIP` performs better than the heuristics given the same amount of time at all budgets where the `MIP` has found a solution at that point in time. Using the `greedy_assignment` method however is "fast enough" that the `MIP` has not found a solution yet at most budgets. This is indicated by "NA" in the table.

Since `greedy_assignment` is significantly faster than `relaxation_rounding`, we give the results for Algorithm 6 with $n_c = 20$ and $m =$ `greedy_assignment` in Table E.1. This increases the run time to be more similar to the time the algorithm takes with `relaxation_rounding` and $n_c = 5$. The algorithm with `greedy_assignment` and $n_c = 20$ performs better regarding its difference to the `MIP` than it does with `relaxation_rounding` and $n_c = 5$ at most budgets. However, the performance is still significantly worse than the `MIP`.

Table 6.3: Results of close greedy algorithm, Algorithm 6, comparing using $m =$ `greedy_assignment` throughout and $m =$ `relaxation_rounding`. The final assignment is done with $m' =$ `relaxation_rounding` with `local_search_reassign` in both cases. This is with $n_c = 5$ and on the larger instance. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the gap between the solution and the lower bound the `MIP` solver reaches after 20,000 seconds. For each method, the run time, the difference to the `MIP` after 20,000 seconds and at the same time, calculated using Equation (6.1), are displayed. "NA" indicates that the `MIP` has not found a solution yet by the time the heuristic terminates. For details, see Section 6.3.1.

		`greedy_assignment`			**`relaxation_rounding`**		
Budget factor	**`MIP` gap after 20k s [%]**	**Run time [s]**	Δ_{MIP} **[%]**	Δ_S **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	Δ_S **[%]**
0.9	1.33	479	-3.87	NA	1780	-3.64	-3.45
0.8	2.36	855	-9.08	NA	3620	-8.23	-8.23
0.7	3.53	1260	-16.09	-15.50	4389	-14.60	-14.60
0.6	5.07	1544	-23.13	-22.21	5432	-20.27	-20.23
0.5	7.49	1833	-28.78	-28.50	6627	-23.15	-23.05
0.4	6.66	2379	-38.28	NA	10540	-31.09	-31.09
0.3	7.42	2785	-42.66	NA	12315	-33.32	-33.32
0.2	7.71	3181	-36.71	NA	15284	-32.66	-32.66
0.1	7.84	3443	-23.11	NA	13624	-25.21	-23.71

To conclude, we can see that this version of close greedy works well but is slower than we would want on the smaller instance, while for the larger instance there is a trade-off between quality of results and run time. The parameter n_c also highly influences the quality of results and to get good results regarding the objective function, the results of this algorithm suggest that we need to consider a significant proportion of all facilities to make the best choice.

6.3.2. Results close greedy with reusing previous assignment

We now discuss the results of close greedy after the improvement that was made in Section 5.1.2. This improvement is that instead of recomputing the assignment of users to facilities, we use the previous iterations assignment and only reassign users that are currently assigned to the facility that we wish to close. Since we are now only using `relaxation_rounding` to make the final assignment and not also throughout the algorithm, we have decided to increase the number of facilities we consider for each user, n_r, from 20 to 50. This increases the run time of the final assignment, but since we are only running `relaxation_rounding` once, making sure each user has enough "choice" to lead to a good solution is deemed more important.

Table 6.4: Results of close greedy algorithm, Algorithm 6, when Line 6 is replaced with a call to Algorithm 7. The initial assignment method is m = `greedy_assignment` with `local_search_reassign`, the final assignment is m' = `relaxation_rounding` with `local_search_reassign`. This is with $n_c = |J|$ and on the smaller instance. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the gap between the solution and the lower bound the `MIP` solver reaches after 20,000 seconds. The third column shows the run time of the heuristic. The last two columns are calculated based on Equation (6.1). "NA" indicates that the `MIP` has not found a solution yet by the time the heuristic terminates. For details, see Section 6.3.2.

Budget factor	`MIP` gap after 20k s [%]	Run time [s]	Δ_{MIP} [%]	Δ_S [%]
0.9	0.57	7	-0.01	NA
0.8	1.41	12	-0.02	0.17
0.7	2.19	17	-0.07	0.32
0.6	3.05	19	-0.10	0.61
0.5	3.53	21	-0.14	52.31
0.4	3.93	23	-0.05	49.97
0.3	4.12	29	-0.26	46.50
0.2	4.64	25	-0.09	41.32
0.1	4.18	27	-0.22	26.38

Again, let us consider the results of the smaller instance first, as seen in Table 6.4. Comparing this to Table 6.2 with `greedy_assignment` at $n_c = |J|$ we make two observations: This has improved run times by an order of magnitude and the objective function value has barely changed at all. Note that in these results, we run the `local_search_reassign` heuristic after we make the first greedy assignment and use $n_r = 50$, which we do not do in the previous section. This explains why at some budgets this performs better. Without these changes, no change in objective function value was observed between reusing the assignment and computing the assignment from scratch every iteration. Hence, this indicates that the reassignment gives (almost in all cases) the same results as re-running the whole assignment.

As can be seen from the last column in Table 6.4, the heuristic is performing slightly better than the `MIP` at the same point in time for higher budgets and significantly better at lower budgets. For the budget factor of 0.9 the `MIP` has not found a solution yet by the time the heuristic has returned a solution. The large difference at lower budgets is not as significant as it appears – less than a minute and in some cases just a few seconds after the time the heuristic finishes, the `MIP` significantly improves.

For the larger instance, it would still take too much time to consider every single facility at each step. However, we have significantly increased n_c to 200 for the results in Table 6.5 compared to 5 in the previous section since this approach is significantly faster than recomputing the assignment at each step. Considering more facilities at each step has improved the objective significantly, but still not far enough for this to be very close to the `MIP` for most budgets. This improvement is most obvious when considering the difference to the `MIP` as seen in the last two

columns, which has been more than halved compared to the results seen in Table 6.3 for most budgets. At very low budgets, the MIP becomes very difficult to solve, so at a 0.1 budget the heuristic is able to perform better than the MIP.

Table 6.5: Results of close greedy algorithm, Algorithm 6, when Line 6 is replaced with a call to Algorithm 7. The initial assignment method is m = `greedy_assignment` with `local_search_reassign`, the final assignment is m' = `relaxation_rounding` with `local_search_reassign`. This is with $n_c = 200$ and on the larger instance. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the gap between the solution and the lower bound the MIP solver reaches after 20,000 seconds. The third column shows the run time of the heuristic. The last two columns are calculated based on Equation (6.1). "NA" indicates that the MIP has not found a solution yet by the time the heuristic terminates. For details, see Section 6.3.2.

Budget factor	MIP gap after 20k s [%]	Run time [s]	Δ_{MIP} [%]	Δ_S [%]
0.9	1.33	486	-0.59	NA
0.8	2.36	920	-2.02	NA
0.7	3.53	1323	-3.87	-3.34
0.6	5.07	1733	-5.44	-4.65
0.5	7.49	2093	-5.76	-5.68
0.4	6.66	2396	-10.24	-10.24
0.3	7.42	2697	-9.60	-8.54
0.2	7.71	3021	-4.01	NA
0.1	7.84	3148	0.33	NA

To conclude, we can see a significant improvement in run time without a negative effect on the objective function value by re-using the assignment. However, for the larger instance, the performance regarding the objective function value is still not as good as one would hope. Not considering every single facility, or potentially the way we choose the facilities we consider, is what most likely leads to this.

6.3.3. Results close greedy with reusing previous assignment and considering previously good facilities

We now discuss how choosing the facilities to consider based on how good they were to close in a previous iterations performs. Hence, the results in this section are for our final version of the close greedy algorithm, Algorithm 8. Since we are already getting good results for the smaller instance in the previous section by considering all facilities at each iteration, we start by considering the larger instance. Note that for the smaller instance, all we achieve with this improvement to the close greedy algorithm is an improvement in the run time. For the larger instance, we consider the results for three different values of n_c, 1, 5 and 50. Recall that the instance has 1394 facilities, so these numbers are more than an order of magnitude smaller than the total number of facilities we could consider. As in the previous section, we run `relaxation_rounding` and the `local_search_reassign` heuristic once the algorithm terminates to get the final assignment of users to facilities.

We focus on $n_c = 5$ and 50 for now. The improvement we can see in Table 6.6 compared to Table 6.5 is large: We are only considering a fortieth / fourth of the facilities at each step compared to how many facilities we consider in Table 6.5 which means the run time is significantly shorter. At the same time, the objective value reached is significantly better. At all budgets except for 0.9, the solution reached by the close greedy algorithm is even better than the result the MIP arrives at after 20,000 seconds. At 0.9, the solution is only slightly worse than the MIP. This is in stark contrast to a difference to the MIP up to 10% observed in the previous section. Note that it does

Table 6.6: Results of close greedy algorithm, Algorithm 8. The initial assignment method is m = `greedy_assignment` with `local_search_reassign`, the final assignment is m' = `relaxation_rounding` with `local_search_reassign`. This is with $n_c = 1$, $n_c = 5$ and $n_c = 50$ on the larger instance. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the gap between the solution and the lower bound the `MIP` solver reaches after 20,000 seconds. For each n_c, the first column shows the run time. The second column is calculated based on Equation (6.1). For details, see Section 6.3.3.

		$n_c = 1$		$n_c = 5$		$n_c = 50$	
Budget factor	**`MIP` gap after 20k s [%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
0.9	1.33	70	-0.08	75	-0.01	185	0.00
0.8	2.36	67	-0.07	84	0.04	306	0.04
0.7	3.53	68	-0.12	102	0.13	415	0.13
0.6	5.07	83	0.22	106	0.51	520	0.52
0.5	7.49	71	1.53	117	1.94	618	1.93
0.4	6.66	76	-0.28	122	0.11	671	0.13
0.3	7.42	77	-0.24	137	0.35	724	0.36
0.2	7.71	77	-0.79	143	0.24	775	0.25
0.1	7.84	78	-0.49	151	0.39	824	0.35

not make sense to compare these results to how the `MIP` does at the same point in time since it takes almost 600 seconds to even build these models, so the `MIP` has not found a solution yet by the time the heuristic terminates. Additionally, the `MIP` then spends 400 seconds on the root relaxation and for low budgets it also takes a significant amount of time to find its first solution, e.g. for a 0.4 budget it takes 2000 seconds to find the first solution, not including the time to build the model and solving the root relaxation. Hence, the `MIP` has not found a solution by the time this heuristic terminates at all budgets.

The budget factors of 0.5 and 0.4 appear to be outliers with how much better than the `MIP` they perform. For a 0.5 budget, this is significantly better with the difference to the `MIP` being close to 2%, while it only performs slightly better for a 0.4 budget. This appears to be more connected with how well the `MIP` does than with how well the close greedy heuristic works: looking at the `MIP` gap after 20,000 seconds, the `MIP` struggles more with a 0.5 budget, while the close greedy heuristic performs consistently well. The gap referred to here is the gap the solver has between its best solution and the best lower bound.

Now, comparing $n_c = 5$ and $n_c = 50$, we can see that the results are very similar with $n_c = 50$ performing slightly better on most budgets, but performing marginally worse for 0.5 and 0.1 budgets. Since no significant improvement in objective function value can be seen while the run time is more than double, it seems more sensible to choose a lower value for n_c and then potentially use a local search approach on that solution. $n_c = 5$ is a surprisingly low number of facilities to consider each iteration, especially since there are 1394 facilities in total and since only considering 5 facilities worked badly in Section 6.3.1. Hence, let us consider what happens at $n_c = 1$, which means we close all the facilities whose closure at the point when all facilities are open seems best. This leads to the heuristic performing worse than the `MIP` across all budgets except 0.5 and 0.6, but not significantly so, with differences of at most about -0.8%. The time savings compared to $n_c = 5$ are not as significant anymore since the final assignment and computing the results of the closure of all facilities one by one in the first iteration takes a significant proportion of the time. Hence, it seems better to have $n_c \geq 5$ for this instance. However, it is surprising how well just closing the facilities that seem best to close in the first

iteration performs.

As a concern previously has been that the assignment might be getting worse as more facilities close and that might influence the choice we are making at each step, we also tried completely recomputing the assignment with the `greedy_assignment` heuristic (not just the `greedy_reassign` heuristics) after every 50 facilities that have been closed. This however did not have any effect on the solution: The only time this re-computation lead to a better assignment was after 400 facilities were closed, so it appears simply reassigning greedily gives the same result as completely recomputing most of the time.

Table 6.7: Results of close greedy algorithm, Algorithm 8. The initial assignment method is m = `greedy_assignment` with `local_search_reassign`, the final assignment is m' = `relaxation_rounding` with `local_search_reassign`. This is with $n_c = 5$ and $n_c = 50$ on the smaller instance. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the gap between the solution and the lower bound the `MIP` solver reaches after 20,000 seconds. The first column for each n_c value shows the run time. The second column for each n_c is calculated based on Equation (6.1). For details, see Section 6.3.3.

		$n_c = 5$		$n_c = 50$	
Budget factor	**MIP gap after 20k s [%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
0.9	0.57	3	-0.01	4	-0.01
0.8	1.41	3	-0.02	5	-0.02
0.7	2.19	3	-0.06	6	-0.07
0.6	3.05	3	-0.09	7	-0.10
0.5	3.53	3	-0.16	8	-0.14
0.4	3.93	3	-0.08	8	-0.05
0.3	4.12	4	-0.26	9	-0.26
0.2	4.64	3	-0.09	10	-0.09
0.1	4.18	2	-0.22	9	-0.22

For another indicator of the performance of this heuristic, even with low n_c, we consider how well Algorithm 8 performs on the smaller instance. See Table 6.7 for details. At $n_c = 50$ (recall that this instance has 234 facilities), the results achieved are completely identical to those achieved when considering all still open facilities at each iteration, as done in Table 6.4. At a budget of 0.1, this heuristic took only 9 seconds, compared to 23 seconds in Table 6.4. At $n_c = 5$, the objective function value is identical at 5 of the 9 budgets. At two budgets the objective is worse at $n_c = 5$ than considering all facilities and at two budgets it is better, both only marginally so with the difference to the `MIP` changing by at most 0.02 percentage points. This is a strong indicator that we are now choosing the right facilities to consider at each iteration, i.e. we are choosing the right set J'. Regarding run time, at all budgets, the heuristic now takes less than 4 seconds when $n_c = 5$.

To conclude, this new method of deciding which facilities to consider closing performs significantly better than considering facilities with low utilisations. At the budgets where the `MIP` starts to struggle to converge, this heuristic outperforms what the `MIP` achieves in 20,000 seconds in a significantly shorter time.

6.3.4. Conclusion close greedy algorithm

As we can see from the results in the previous sections, both changes to the algorithm discussed in Section 5.1 significantly improve the performance of the algorithm. Not completely recomputing the assignment but instead adapting it leads to a significant decrease in run time, while choosing which facilities to consider based on their performance in previous iterations leads to a significant

improvement in the objective function value achieved. With this latter improvement, even when only 5 facilities are considered each iteration, the results are very good and further increasing n_c only improves the solution quality marginally.

6.4. Results open greedy

We now discuss the results of Algorithm 10. We have fixed m to `greedy_assignment` with `local_search_reassign` and m' to `relaxation_rounding` with `local_search_reassign` throughout this section. The parameters whose effect we need to consider now are the number of facilities to consider each iteration n_c, the depth of the local search within the `greedy_reassign_open` algorithm d and the number of iterations after which we recompute the assignment n_f.

Table 6.8: Results of open greedy algorithm, Algorithm 10, with m = `greedy_assignment` with `local_search_reassign`, m' = `relaxation_rounding` with `local_search_reassign`. This is with $d = 1$ and $n_f = |J|$ on the smaller instance. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the gap between the solution and the lower bound the `MIP` solver reaches after 20,000 seconds. For each number of facilities to consider, n_c, the first column shows the run time. The second column is calculated based on Equation (6.1). For details, see Section 6.4.

		$n_c = 5$		$n_c = 50$		$n_c = \|J\|$	
Budget factor	**`MIP` gap after 20k s [%]**	**Run time [s]**	Δ_{MIP} [%]	**Run time [s]**	Δ_{MIP} [%]	**Run time [s]**	Δ_{MIP} [%]
0.9	0.57	4	-0.03	16	0.00	45	0.00
0.8	1.41	4	-0.01	15	0.00	40	0.00
0.7	2.19	4	-0.07	13	-0.06	38	-0.06
0.6	3.05	4	-0.02	12	-0.01	35	-0.01
0.5	3.53	4	-0.02	10	-0.02	31	-0.02
0.4	3.93	3	-0.07	8	-0.01	26	-0.01
0.3	4.12	3	-0.12	7	-0.07	22	-0.07
0.2	4.64	3	-0.04	5	0.00	15	0.00
0.1	4.18	2	-0.09	3	-0.03	7	-0.03

For the smaller instance, we consider $d \in [1, 2, 3]$, $n_c \in [5, 50, |J|]$ and $n_f \in [10, 50, |J|]$; for the larger instance, we consider $d \in [1, 2, 3]$, $n_c \in [5, 50]$ and $n_f \in [50, |J|]$. Note that $n_f = |J|$ means that the assignment is not recomputed from scratch during the main while loop. Our first observation is that for the smaller instance the runs with $n_c = 50$ and $n_c = |J|$, with the other two parameters being fixed, result in the same objective function value. An example of this can be seen in Table 6.8, where $d = 1$ and $n_f = |J|$. This is the same as we observed in the close greedy algorithm, which suggests it is not necessary to consider all facilities at each iteration due to us choosing the right subset of facilities to consider. However, some difference can be observed between $n_c = 5$ and $n_c = 50$, with $n_c = 50$ performing better. Considering the difference to the `MIP` after 20,000 seconds, Δ_{MIP}, this improves by at most 0.06 percentage points by increasing n_c from 5 to 50. For the larger instance, an improvement in the Δ_{MIP} value when increasing n_c from 5 to 50 of at most 0.14 percentage points can be observed by comparing the results in Table 6.9 and Table F.2.

Comparing the results in Table 6.8 to the ones for the close greedy algorithm in Table 6.7, at the same value of n_c, open greedy performs generally better than close greedy except for at $n_c = 5$ at higher budgets. The pattern of open greedy generally performing better than close greedy applies to the larger instance at $n_c = 5$ and $n_c = 50$ as well by comparing Table 6.6 and Table 6.9 / Table F.2.

Secondly, recomputing the assignment every $n_f < |J|$ iterations in the open greedy algorithm leads to a better assignment than what reassigning users with `greedy_reassign_open` achieves multiple times throughout the algorithm. Especially at depth $d = 1$, the complete reassignment often outperforms what is achieved by `greedy_reassign_open`. Recall that in the close greedy algorithm, the complete recalculation of the assignment only led to an improvement once on the large instance at a budget factor of 0.3. Hence, this suggests the reassignments arising from opening facilities are worse than those made when closing a facility. However, this does not appear to have any effect on the choices of the facilities to open that the open greedy algorithm makes since the final objective function value is identical in most cases, no matter the value of n_f. When it differs between larger and smaller values of n_f there is no clear trend: For some budgets, having lower n_f performs better and for other budgets having higher n_f performs marginally better. Hence, we have omitted the results where we consider different values of n_f here, concluding that recomputing the assignment from scratch every n_f iterations does not improve the algorithm.

Thirdly, the objective function value reached is generally better at higher d, as seen for the larger instance in Table 6.9. At the same time, the run time when going from $d = 1$ to $d = 2$ is approximately doubled. The improvement in objective value, as represented here by the difference to the `MIP` after 20,000 seconds, is generally larger when going from $d = 1$ to $d = 2$ than it is when increasing it further to $d = 3$. Note that at larger values of d open greedy becomes slower than close greedy. The general pattern of larger d leading to better results can also be observed for the smaller instance in Table F.1. At $d = 2$ and $d = 3$ this is the first time our heuristic performs slightly better than the `MIP` on the smaller instance at some budgets.

Table 6.9: Results of open greedy algorithm, Algorithm 10, with m = `greedy_assignment` with `local_search_reassign`, m' = `relaxation_rounding` with `local_search_reassign`. This is with $n_c = 50$ and $n_f = |J|$ on the larger instance. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the gap between the solution and the lower bound the `MIP` solver reaches after 20,000 seconds. For each depth d, the first column shows the run time. The second column is calculated based on Equation (6.1). For details, see Section 6.4.

		$d = 1$		$d = 2$		$d = 3$	
Budget factor	**`MIP` gap after 20k s [%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
0.9	1.33	1030	0.00	1811	0.00	2459	0.00
0.8	2.36	901	0.05	1621	0.09	2216	0.09
0.7	3.53	750	0.14	1425	0.18	2089	0.19
0.6	5.07	629	0.60	1222	0.63	1839	0.65
0.5	7.49	491	2.00	1054	2.06	1566	2.07
0.4	6.66	376	0.20	849	0.26	1343	0.28
0.3	7.42	271	0.45	634	0.55	1061	0.55
0.2	7.71	182	0.29	439	0.31	767	0.34
0.1	7.84	106	0.40	234	0.47	434	0.47

To conclude, the open greedy algorithm overall performs well on our instances. Higher n_c and higher d generally lead to better results but also longer run times. However, there seems to be some ceiling for both of these values as to how much better increasing them can make the results: For n_c, this is clear by considering the smaller instance and noting that $n_c = 50$ and $n_c = |J|$ lead to the same results; for d, this can be seen by considering that the improvement in objective value between $d = 2$ and $d = 3$ for the larger instance is already very small, while it is generally larger between $d = 1$ and $d = 2$. Recomputing the assignment from scratch every n_f

iterations appears to have little to no effect on the objective function value. Overall, the open greedy algorithm generally performs marginally better than the close greedy algorithm on these instances. Like the close greedy algorithm, it hence performs better than the MIP after 20,000 seconds in a fraction of the time.

6.5. Results BFLP local search

We start by discussing the smaller instance; the results of the local search on the smaller instance can be found in Appendix G, Table G.2 and Table G.3. The starting solutions of the local search we consider are the results of Algorithm 8, the close greedy algorithm, with $n_c = 5, 50$ as seen in Table 6.7 and the results of Algorithm 10, the open greedy algorithm, with $n_c = 5, 50$, $d = 2$ and $n_f = |J|$ as seen in Table F.1. We consider these starting solutions to answer the following questions: Does running local search on the close greedy solution improve the objective function value beyond the objective function value achieved by open greedy? Is it more sensible to run open/close greedy with $n_c = 5$ and then run a BFLP local search, or to run open/close greedy with $n_c = 50$ (with or without a local search afterwards)? The answer to the first question is that open greedy still performs marginally better than close greedy together with the local search, where both open and close greedy are run with $n_c = 50$. Regarding the second question, for open greedy there is no clear answer since for some budgets open greedy with $n_c = 5$ and then running a BLFP local search performs better, while for other budgets open greedy with $n_c = 50$ performs better. However, for most budgets they lead to the same objective. Note that the local search starting on the open greedy solution with $n_c = 50$ does not find any improvement. When starting with the close greedy algorithm solution of $n_c = 5$, running local search leads to a better solution than simply using close greedy with $n_c = 50$. When also running local search starting on the $n_c = 50$ close greedy solution, this leads to exactly the same objective value as local search starting on the $n_c = 5$ close greedy solution for most budgets.

Table 6.10: Results of BFLP local search algorithm, Algorithm 11. The starting solution is from Algorithm 6 with m = `greedy_assignment`, m' = `relaxation_rounding` with `local_search_reassign`, $n_c = 5$ on the larger instance whose results can be found in Table 6.3. Local search is run with $n_c = 50$, $d = 2$ and m' = `relaxation_rounding` with `local_search_reassign`. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the difference to the MIP before the local search is run. We compare Algorithm 11 to its variant where j' and J'' are chosen randomly (without δ). For each number of iterations l, the first column shows the run time. The second column is calculated based on Equation (6.1). For details, see Section 6.5.

		With δ				**With random choices**			
		$l = 200$		$l = 1000$		$l = 200$		$l = 1000$	
Budget factor	Δ_{MIP} **before local search [%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
0.9	-3.87	340	-0.09	1370	-0.06	313	-0.67	1339	-0.13
0.8	-9.08	348	-0.19	1415	-0.07	305	-1.72	1348	-0.15
0.7	-16.09	344	-0.42	1392	-0.002	353	-3.42	1515	-0.20
0.6	-23.13	340	-0.77	1329	0.37	281	-6.15	1153	0.16
0.5	-28.78	326	0.08	1214	1.62	271	-6.19	1089	1.28
0.4	-38.28	512	-2.20	1493	-0.25	380	-13.20	1485	-0.96
0.3	-42.66	483	-1.93	1457	-0.11	355	-15.80	1224	-0.76
0.2	-36.71	379	-0.67	1089	0.02	271	-16.78	949	-1.19
0.1	-23.11	288	0.14	820	0.28	192	-9.28	654	-1.36

Note that we run the BFLP local search with the following parameters on the smaller instance: $n_c \in [50, |J|]$, $d \in [1, 2]$ and $l = \infty$. Setting l to infinity means that the algorithms terminates when no improvement has been made by considering every single facility to open or close. The questions we would like to answer by considering these different values are: Do we need to consider all facilities in each iteration or are 50 sufficient, as is the case in the open and close greedy algorithms? Does increasing d improve the solution quality? Regarding the first question, having $n_c = 50$ or $n_c = |J|$ leads to exactly the same solution except for two cases (i.e. specific values of d) where $n_c = 50$ performs better and two cases where $n_c = |J|$ performs better. Hence, we can conclude that it is again not necessary to consider all facilities at every step. Considering the effect of d on the local search, as with the open greedy algorithm, the improvement is minimal with a change of at most 0.1 percentage points when starting on close greedy and of at most 0.02 percentage points when starting on open greedy.

We now consider the larger instance. Firstly, we would like to discuss how the BFLP local search performs when given a starting solution that is far away from the optimal solution. To this extent, we take as our starting solution the results in Table 6.3, specifically the results of the basic close greedy algorithm using `greedy_assignment` as the main user assignment method. We run local search with this starting solution, both as described in Algorithm 11 and with all choices of facilities being made randomly instead of based on δ. This allows us to conclude whether making choices based on δ improves the algorithm. We consider $l \in [200, 1000]$, $n_c = 50$ and $d = 2$; the results can be seen in Table 6.10.

Our first takeaway is that choosing facilities based on δ improves the algorithm significantly if the algorithm is run for only 200 iterations and still slightly if it is run for 1000 iterations. Hence, by considering the best facilities based on δ, we are making larger improvements earlier on in the local search. In addition, using δ does increase the run time somewhat, but not to the extent that it would be better not to use it. Our second observation is that after 1000 iterations, with using δ, the algorithm achieves solutions that are still slightly worse than those achieved by the improved closed greedy algorithm and the open greedy algorithm with $n_c = 50$, as seen in Table 6.6 and Table 6.9. However, it is now a lot closer to these results than before the local search.

Now that we established that the local search succeeds at improving a bad solution significantly, let us consider how the local search heuristic performs when it gets given a good starting solution. Hence, we consider how it performs given the best solution we have found so far for the large instance, the open greedy solution with $d = 3$, $n_c = 50$ and $n_f = |J|$ as given in Table 6.9. This also tells us how close Algorithm 10 is to a local optimum. The results of running local search on this with $d = 2$ and $n_c = 50$ can be seen in Table 6.11. Our first observation is that there are three budgets, 0.1, 0.2 and 0.4 where the local search does not find any improvement. For all other budgets, the improvement from local search is small: The largest improvement in the difference to the MIP after 20,000 seconds by running local search is an increase by 0.009 percentage points at a budget of 0.8. Considering the number of iterations that the local search is run for, there are only two budgets for which running local search for 100 instead of 20 iterations lead to an improvement (0.5 and 0.7) and there are no budgets for which running the local search for 200 iterations instead of 100 iterations leads to an improvement. Hence, we can conclude that the solution achieved by open greedy is very close to a local optimum and local search can find any small improvements within the first few iterations.

We now briefly comment on how much the local search improves the solutions found with the close greedy algorithm at $n_c = 5$ and $n_c = 50$, i.e. with the starting solution given by the last four columns in Table 6.6. The local search results can be found in Table 6.12. Local search improves these solutions, but even after 200 iterations of local search with $n_c = 50$, $d = 2$ the solution reached is still worse than simply running open greedy, apart from at budget factor

Table 6.11: Results of `BFLP` local search algorithm, Algorithm 11. The starting solution is from Algorithm 10 with m = `greedy_assignment` with `local_search_reassign`, m' = `relaxation_rounding` with `local_search_reassign`, $n_c = 50$, $d = 3$, $n_f = |J|$ on the larger instance, as seen in Table 6.9. Local search is run with $n_c = 50$, $d = 2$ and m' = `relaxation_rounding` with `local_search_reassign`. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the difference to the `MIP` after 20k seconds before the local search is run, so of the starting solution, calculated based on Equation (6.1). For each number of iterations l, the first column shows the run time. The second column is calculated based on Equation (6.1). For details, see Section 6.5.

		$l = 20$		$l = 100$		$l = 200$	
Budget factor	Δ_{MIP} **before local search [%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
0.9	0.001	98	0.007	239	0.007	381	0.007
0.8	0.089	106	0.098	249	0.098	419	0.098
0.7	0.187	103	0.189	233	0.190	403	0.190
0.6	0.648	96	0.650	204	0.650	367	0.650
0.5	2.075	98	2.075	204	2.077	360	2.077
0.4	0.279	99	0.279	196	0.279	320	0.279
0.3	0.550	82	0.555	154	0.555	257	0.555
0.2	0.342	86	0.342	156	0.342	221	0.342
0.1	0.470	91	0.470	126	0.470	169	0.470

0.9. However, it is now closer to that solution, with differences in the Δ_{MIP} being at most 0.13 percentage points and most of them being about 0.05 percentage points worse, when comparing local search starting on the $n_c = 50$ close greedy solution to Table 6.11. Hence, we can conclude that the best solution we can get is by running the best possible algorithm to create the solution, which is `greedy_open` for this instance, and then running the local search for a few iterations. However, note that the results of the local search starting from close greedy with $n_c = 5$ and close greedy with $n_c = 50$ are very similar – for some budgets the first is better, for the other the second. Hence, this shows that it makes more sense to run close greedy with $n_c = 5$ first and then run local search, instead of running close greedy simply with $n_c = 50$, due to the former having lower run times.

We further note by considering Table G.1, where the local search is run on the same starting solutions but with $n_c = 5$, $d = 1$, that on average the improvement made after 20 iterations is the same as when using $n_c = 50$, $d = 2$ for 20 iterations. However, using these smaller parameters leads to less improvement than using the larger parameters after 100 and 200 iterations. Since most improvements are made in the first 20 iterations, this still means local search can also be run with smaller parameters without significantly affecting the performance.

To conclude, the local search algorithm, when started with a bad solution, improves that solution significantly. Using δ ensures that large improvements can already be seen after few iterations. When giving the local search algorithm a good solution, it finds most improvements in the first 100 iterations. However, the algorithm is not able to improve the slightly worse close greedy solution to make it better than the open greedy solution, but is able to improve close greedy $n_c = 5$ to the levels of close greedy $n_c = 50$. All of these improvements and differences are small – hence, our open and close greedy solutions already are very close to a local optimum.

Table 6.12: Results of BFLP local search algorithm, Algorithm 11. The starting solution is from Algorithm 8 with m = `greedy_assignment` with `local_search_reassign`, m' = `relaxation_rounding` with `local_search_reassign`, $n_c = 5$ or $n_c = 50$ on the larger instance. Local search is run with $n_c = 50$, $d = 2$ and m' = `relaxation_rounding` with `local_search_reassign`. The budget factor indicates the proportion of all facilities that should remain open. The third column shows the difference to the MIP after 20k seconds before the local search is run, so of the starting solution, calculated based on Equation (6.1). For each number of iterations l, the first column shows the run time. The second column is calculated based on Equation (6.1). For details, see Section 6.5.

			$l = 20$		$l = 100$		$l = 200$	
Starting instance	**Budget factor**	Δ_{MIP} **before local search [%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
Close greedy with $n_c = 5$	0.9	-0.006	90	0.015	206	0.016	357	0.016
	0.8	0.041	92	0.070	206	0.077	356	0.081
	0.7	0.126	90	0.154	212	0.169	349	0.175
	0.6	0.508	141	0.581	291	0.597	452	0.616
	0.5	1.936	131	1.994	241	2.039	398	2.056
	0.4	0.106	117	0.153	216	0.226	408	0.224
	0.3	0.353	136	0.419	238	0.471	296	0.478
	0.2	0.237	117	0.293	159	0.301	302	0.313
	0.1	0.388	98	0.436	148	0.437	181	0.436
Close greedy with $n_c = 50$	0.9	-0.004	124	0.015	213	0.016	354	0.016
	0.8	0.044	120	0.067	225	0.078	442	0.083
	0.7	0.131	123	0.147	276	0.171	476	0.172
	0.6	0.517	135	0.553	283	0.616	456	0.620
	0.5	1.931	136	2.002	272	2.044	456	2.037
	0.4	0.131	135	0.204	243	0.232	380	0.232
	0.3	0.359	130	0.463	217	0.472	356	0.500
	0.2	0.253	134	0.275	204	0.260	321	0.311
	0.1	0.349	133	0.407	171	0.420	239	0.420

6.6. Overall comparison BFLP heuristics

After having discussed all the heuristics on their own, we would now like to compare how they perform, also on the two artificial instances that we created. An overview of how open and close greedy perform on the large instance and the two artificial instances can be found in Table 6.13. We omit the smaller instance here but note that on this smaller instance we achieve results close to what the MIP achieves in less time across all our heuristics. Since this smaller instance is already solved well by the MIP, we are more interested in the other three instances that we discuss here. Open greedy is run with $n_f = |J|$ in this section since recomputing the assignment from scratch every n_f iterations did not lead to any improvement in Section 6.4. We also compare the results of our heuristics to the heuristic developed in Schmitt and Singh (2021). We run this heuristic, which is a local search heuristics, until there are no changes in the objective for 100 iterations to ensure that the algorithm has (most likely) reached its local minimum. Comparing the results of this algorithm to our heuristics by considering the difference to what the MIP achieves after 20,000 seconds, we can see that our heuristics perform better than the algorithm from Schmitt and Singh (2021) at all budgets and on all instances, sometimes significantly so.

In addition, our heuristics perform significantly better than the MIP at lower budgets. Especially for Artificial Instance 1, our largest instance, the heuristics perform significantly better than the MIP at very low budgets. Hence, our heuristics appear to be able to deal better than the MIP solver with very large instances when capacity becomes limited. Additionally, our

heuristics perform only marginally worse than the MIP at higher budgets. By considering the MIP gap in the third column of the table, so the gap between lower and upper bound that the MIP solver reaches after 20,000 seconds, it can be seen that lower budgets are the budgets for which the problem becomes more difficult to solve for the MIP solver.

Considering the objective function value of open and close greedy across the three instances, there is no clear picture of which of these performs better. On the large instance, open greedy performs better, especially once increasing d to 2. On Artificial Instance 1, close greedy performs better at $n_c = 5$. Hence, $n_c = 5$ might be simply not sufficient for open greedy on this instance, as this is a large instance with about 2500 facilities. Some further experiments have shown that increasing n_c to 10 leads to solutions that are significantly closer to the results achieved by $n_c = 50$. For example, on Artificial Instance 1, $n_c = 10$ performs 0.4 percentage points better than $n_c = 5$ regarding its Δ_{MIP} value at a budget of 0.1 and 0.2. Hence, improving n_c all the way to 50 may not be necessary, but a value larger than 5 would be sensible for getting the best performance on this instance. At $n_c = 50$, open greedy generally performs better, surprisingly, especially at higher budgets. However, the open greedy heuristic is slower than the close greedy heuristic at these high budgets.

Finally, on Artificial Instance 2, open greedy performs better than close greedy at lower budgets while close greedy performs better than open greedy at higher budgets. This split between when open and when close greedy performs best is the most expected: fewer facilities need to be closed by close greedy at high budgets, while fewer facilities need to be opened by open greedy at lower budgets. This means there are fewer points where a wrong decision can be made, and additionally the values of δ are less out of date.

Considering the run times of close and open greedy, open greedy is generally slower when comparing "inverse" budgets, e.g. comparing 0.9 of close greedy with 0.1 of open greedy. This is because the procedure that determines a new assignment when a facility is opened also does a small local search, which is not necessary when closing a facility. Especially as d, the depth of this local search, is increased, this is reflected in the run times of open greedy compared to close greedy. By the very nature of the algorithms, it is faster for open greedy to find solutions at lower budgets, while it is faster for close greedy to find solutions at high budgets.

Now, let us discuss the parameter n_c and in the case of open greedy also the parameter d. Across all instances, increasing n_c from 5 to 50 leads to a significant increase in run time but only in a small – or at some budgets no improvement – in the objective function value. In all but one case, the improvement is at most of the order of 0.01 percentage points for close greedy and there also exist a couple budgets for which only considering 5 facilities each iteration performs better. For open greedy, the improvement of increasing n_c from 5 to 50 is generally larger, especially at lower budgets for the two artificial instances. However, most improvements are still in the order of 0.01 percentage points and the largest improvement is 0.5 percentage points. Considering increasing d from 1 to 2 improvements are of a similar order of magnitude as for increasing n_c from 5 to 50. However, the increase in run time when $n_c = 5$ and d increases from 1 to 2 is less extreme than when increasing n_c to 50, which means increasing d to 2 might be more sensible than increasing n_c all the way to 50. Increasing d instead of n_c is sensible, especially in the case of Artificial Instance 2, and also to some extent for the large instance: Since the facilities and users are close together in this instance, an extra depth of local search is more likely to actually lead to an improvement.

Table 6.13: Overview of results of the different BFLP heuristics. All of our heuristics are run with m = greedy_assignment with local_search_reassign and m' = relaxation_rounding with local_search_reassign. The budget factor indicates the proportion of all facilities that should remain open. The third column shows the gap between the solution and the lower bound the MIP solver reaches after 20,000 seconds. For each heuristic, the first column shows the run time. The second column is calculated based on Equation (6.1). For details, see Section 6.6.

			Schmitt-Singh heuristic		**Close greedy** $n_c = 5$		**Close greedy** $n_c = 50$		**Open greedy** $n_c = 5, d = 1$		**Open greedy** $n_c = 5, d = 2$		**Open greedy** $n_c = 50, d = 1$		**Open greedy** $n_c = 50, d = 2$	
Instance	**Budget factor**	**MIP gap after 20k s [%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
Large instance	0.9	1.33	128	-1.34	75	0.00	185	0.00	150	-0.02	229	0.01	1030	0.00	1811	0.00
	0.8	2.36	115	-1.64	84	0.04	306	0.04	136	0.05	210	0.09	901	0.05	1621	0.09
	0.7	3.53	102	-1.81	102	0.13	415	0.13	122	0.14	189	0.19	750	0.14	1425	0.18
	0.6	5.07	90	-1.59	106	0.51	520	0.52	108	0.58	169	0.63	629	0.60	1222	0.63
	0.5	7.49	76	-0.42	117	1.94	618	1.93	95	2.01	148	2.07	491	2.00	1054	2.06
	0.4	6.66	63	-2.28	122	0.11	671	0.13	83	0.16	129	0.26	376	0.20	849	0.26
	0.3	7.42	50	-2.11	137	0.35	724	0.36	72	0.34	108	0.52	271	0.45	634	0.55
	0.2	7.71	36	-2.07	143	0.24	775	0.25	63	0.30	89	0.24	182	0.29	439	0.31
	0.1	7.84	24	-1.40	151	0.39	824	0.35	56	0.37	68	0.46	106	0.40	234	0.47
Artificial Instance 1	0.9	0.28	673	-1.58	343	-0.12	1054	-0.12	803	-0.09	1316	-0.06	5871	-0.10	9685	-0.06
	0.8	0.63	612	-2.30	397	-0.07	1688	-0.07	757	-0.07	1197	-0.04	5038	-0.07	8382	-0.04
	0.7	1.12	567	-3.31	462	-0.04	2310	-0.03	683	-0.03	1077	-0.01	3982	0.01	7300	0.01
	0.6	1.72	489	-4.10	502	0.00	2886	-0.01	605	-0.07	965	-0.03	3260	0.00	6281	0.01
	0.5	3.17	434	-3.62	544	0.75	3419	0.76	532	0.66	862	0.68	2601	0.76	5268	0.76
	0.4	4.46	359	-3.44	587	1.26	3898	1.27	471	1.11	743	1.12	1869	1.26	4220	1.24
	0.3	11.92	315	4.32	610	8.34	4312	8.34	406	8.09	648	8.09	1347	8.33	3268	8.30
	0.2	25.36	215	19.00	657	21.79	4653	21.80	371	21.35	521	21.35	891	21.78	2305	21.76
	0.1	18.08	141	12.30	654	13.96	4902	13.97	314	13.51	410	13.42	530	13.98	1331	13.94
Artificial Instance 2	0.9	0.97	28	-7.45	18	-0.17	21	-0.16	40	-0.39	62	-0.29	167	-0.47	428	-0.29
	0.8	2.75	25	-7.75	17	-0.11	27	-0.11	37	-0.24	60	-0.17	142	-0.22	390	-0.17
	0.7	5.25	22	-8.01	18	-0.17	32	-0.21	39	-0.24	58	-0.06	122	-0.30	371	-0.07
	0.6	8.03	19	-7.71	17	-0.17	38	-0.19	39	-0.27	59	-0.15	114	-0.23	317	-0.13
	0.5	11.44	17	-6.58	20	0.33	64	0.27	38	0.41	57	0.50	104	0.41	295	0.50
	0.4	15.48	14	-3.86	19	1.92	49	1.92	36	1.88	53	2.00	102	1.80	230	2.03
	0.3	16.61	12	-4.70	19	0.31	57	0.32	34	0.12	49	0.45	76	0.17	201	0.49
	0.2	18.26	10	-2.54	26	0.44	55	0.54	33	0.34	40	0.56	52	0.56	152	0.69
	0.1	13.39	8	0.17	32	1.49	58	1.55	23	1.60	25	1.77	35	1.66	74	1.78

Let us now briefly comment on how the BFLP local search performs when starting it on the results given in Table 6.13. The results of the local search, which is run for $l = 100$ iterations with $d = 1$, $n_c = 5$, can be found in Table 6.14. We only show the results for these values of local search parameters, as increasing the parameters any further makes very little difference to the results. As previously observed, the improvements local search can make are small – most of them are of order 0.01 percentage points, with some improvements being of order 0.1 and sometimes even no improvement being made by the local search. The question now is whether it is better to increase n_c in the original open / close greedy algorithm, or to instead run a local search. For close greedy, the algorithm with $n_c = 5$ and then a local search performs better in almost all cases than simply running close greedy with $n_c = 50$. For open greedy, the picture is less clear due to open greedy performing worse at $n_c = 5$ compared to $n_c = 50$ on Artificial Instance 1 at lower budgets. Local search is not able to improve upon this enough to close this gap.

To conclude, if one is only interested in finding the best possible result no matter the run time, increasing n_c and d and running a local search afterwards will generally lead to the best results. However, if one looks to balance the results achieved with the run time, using either open or close greedy (depending on the budget, with open greedy preferred for lower budgets) at low n_c and d and then running a local search for a few iterations will get very close to what running these algorithms with higher parameters achieves but in significantly less time. The improvements made by increasing the parameters and running local search are generally very small, while they have a significant impact on the run time.

Table 6.14: Overview of results of the different BFLP heuristics after running the BFLP local search for $l = 100$ iterations with $d = 1$ and $n_c = 5$. All of these are run with m = `greedy_assignment` with `local_search_reassign` and m' =`relaxation_rounding` with `local_search_reassign`. The run times are only the run times for running the local search. For details, see Section 6.6.

		Close greedy $n_c = 5$ start		**Close greedy $n_c = 50$ start**		**Open greedy $n_c = 5, d = 1$ start**		**Open greedy $n_c = 5, d = 2$ start**		**Open greedy $n_c = 50, d = 1$ start**		**Open greedy $n_c = 50, d = 2$ start**	
Instance	**Budget factor**	**Run time [s]**	Δ_{MIP} [%]	**Run time [s]**	Δ_{MIP} [%]	**Run time [s]**	Δ_{MIP} [%]	**Run time [s]**	Δ_{MIP} [%]	**Run time [s]**	Δ_{MIP} [%]	**Run time [s]**	Δ_{MIP} [%]
Large instance	0.9	75	0.02	92	0.01	72	0.01	82	0.02	71	0.01	82	0.01
	0.8	76	0.07	71	0.07	71	0.09	74	0.10	86	0.09	82	0.10
	0.7	68	0.15	103	0.15	69	0.16	79	0.19	69	0.16	84	0.18
	0.6	73	0.56	106	0.59	69	0.62	78	0.64	79	0.63	81	0.66
	0.5	111	2.00	105	2.00	64	2.03	67	2.07	75	2.03	86	2.06
	0.4	112	0.20	103	0.19	61	0.20	72	0.26	76	0.20	87	0.26
	0.3	77	0.43	99	0.43	57	0.43	68	0.53	67	0.47	61	0.56
	0.2	88	0.28	96	0.27	54	0.32	83	0.28	71	0.31	67	0.31
	0.1	109	0.42	91	0.40	48	0.41	52	0.46	49	0.43	62	0.47
Artificial Instance 1	0.9	298	-0.06	534	-0.05	497	-0.05	532	-0.06	317	-0.07	482	-0.06
	0.8	300	-0.04	542	-0.03	515	-0.03	527	-0.04	316	-0.04	492	-0.03
	0.7	295	0.02	531	0.03	453	0.03	506	0.03	311	0.04	492	0.04
	0.6	288	0.03	532	0.02	662	0.01	310	0.03	306	0.03	486	0.04
	0.5	284	0.78	465	0.77	667	0.73	300	0.75	293	0.79	491	0.79
	0.4	272	1.28	441	1.29	617	1.19	284	1.22	276	1.28	489	1.28
	0.3	257	8.36	425	8.36	454	8.22	268	8.22	261	8.36	491	8.35
	0.2	242	21.80	411	21.82	473	21.54	249	21.59	379	21.80	495	21.82
	0.1	256	13.98	471	13.98	436	13.79	257	13.77	384	13.99	520	13.98
Artificial Instance 2	0.9	22	-0.13	25	-0.12	22	-0.24	24	-0.25	28	-0.29	31	-0.25
	0.8	24	-0.09	25	-0.09	30	-0.18	36	-0.13	30	-0.18	26	-0.13
	0.7	23	-0.05	24	-0.09	34	-0.14	28	-0.01	38	-0.15	26	-0.01
	0.6	24	-0.10	23	-0.09	22	-0.23	29	-0.11	35	-0.14	24	-0.10
	0.5	23	0.39	23	0.33	25	0.47	22	0.50	22	0.48	34	0.50
	0.4	22	2.00	24	2.00	24	2.01	27	2.06	35	1.95	34	2.08
	0.3	22	0.31	22	0.32	27	0.31	22	0.52	36	0.41	24	0.55
	0.2	22	0.55	22	0.54	47	0.62	29	0.71	26	0.68	23	0.77
	0.1	18	1.67	18	1.65	23	1.73	30	1.81	21	1.70	24	1.78

6.7. Summary of results and recommendations

We now summarise our findings and give recommendations regarding which algorithms are best to use and with which parameters. Firstly, let us discuss the `BUAP` heuristics. For balancing finding a good solution and having very low run time, we recommend using `greedy_assignment` with `local_search_reassign`. Using the additional local search, which is very fast, improves the solution achieved by the very basic `greedy_assignment`. If run time is a slightly smaller concern, but the aim is still finding a good solution quicker than the `MIP`, our recommendation is to run `relaxation_rounding` with `local_search_reassign`. The choice of the parameter n_r here depends on the specific instance, in particular for larger and "denser" instances higher n_r than the 50 we used here may be necessary. `local_search_swap` performed worse than `local_search_reassign`, both regarding the improvements it makes to the objective function value and its run time. Hence, we would not recommend using it over `local_search_reassign`.

We now discuss our `BFLP` heuristics recommendations. The first recommendation when choosing between close greedy and open greedy is to consider the budget B. If this is more than half the facilities, we recommend using close greedy and otherwise we suggest using open greedy in order to decrease the run time. Since there is no clear indicator that one performs better than the other, this allows us to use larger parameters n_c and d while aiming for the same run time. Considering the parameter n_c, relatively low n_c compared to $|J|$ still lead to good solution; however, we do not want the ratio of $\frac{n_c}{|J|}$ to be too small. As we discussed with our largest instance, Artificial Instance 1, increasing n_c from 5 to 10 will lead to some improvements for open greedy. At the same time, for all other instances, increasing n_c from 5 to 50 only leads to very minimal improvement. It therefore appears that we should consider at least approximately 0.5% of all facilities in each iteration, at least for instances that have user to capacity ratios similar to these instances. As a minimum, we would recommend $n_c \geq 5$, as our results in Table 6.6 show that very small n_c will worsen the results and not lead to any big time savings because making the final assignment takes a significant amount of the total run time of the heuristic. If the instance is not too large and run time is less of a concern, larger values of n_c can be used, but the improvements made by this will most likely be minimal.

Regarding the parameter d, we suggest either using 1 or 2. The choice made here depends on the instance: For instances where users have more facilities that they have a significantly large preference for, so a more "dense" instance, we suggest using $d = 2$ as it leads to more improvements than increasing n_c on our instances. Recomputing the assignment from scratch every n_f iterations did not improve the algorithm, so $n_f = |J|$ should be chosen.

Finally, the `BFLP` local search can be run with similar parameters as used for open greedy and close greedy. About 100 iterations appear to be sufficient on our instances to achieve most of the improvement that the local search gives us, but again this could be increased if this is necessary for more difficult instances.

Overall, our results show that on easier instances or at easier (higher) budgets our `BFLP` heuristics only perform marginally worse than the `MIP` but in significantly less time. When the `MIP` starts to struggle, which is the case on larger instances and when the capacity becomes more limited, our heuristics are able to outperform what the `MIP` achieves in 20,000 seconds but in significantly less time.

7

Conclusion

We now summarise the findings of this piece of work. We started by showing that the BFLP and the BUAP are $\mathcal{NP}$-hard. Because of this and since the MIP solver struggles to solve large instances with limited capacity, we developed heuristics for these two problems. The heuristics for the BUAP were developed since they were required as a subroutine for the heuristics of the BFLP, but are also of interest in their own right. The `relaxation_rounding` heuristic we developed has been shown to outperform the greedy algorithm developed in Schmitt and Singh (2021). Additionally, a local search that reassigns users has been shown to work significantly better than a local search that swaps the assignment of users.

For the BFLP, we developed two constructive algorithms, close greedy and open greedy, and one local search algorithm. These heuristics are based on the ideas from the heuristics for the CFLP developed in Jacobsen (1983), but some adjustments needed to be made to adapt them to our problem. When developing our heuristics, there were two crucial decisions made that significantly improved the performance of the algorithms. The first important observation is that it is sufficient to adapt an assignment instead of completely recomputing it. In particular, if a single facility is closed (or opened) it is sufficient to start with the original assignment and simple reassign users where this is necessary. The second idea that worked very well in practice was to choose the facilities that should be considered to be closed (or opened) based on how good they were to close (or open) in a previous iteration. As instances become larger, it becomes intractable to consider all facilities at each step of the algorithm, which is why focusing the algorithm on the right facilities is crucial. This choice of focusing the algorithm on only a few facilities worked exceedingly well in practice. Even considering only 5 facilities on instances that have more than 1000 facilities performs well.

We showed that our heuristics for the BFLP outperform the heuristic developed in Schmitt and Singh (2021). As the problem becomes difficult to solve for MIP solvers, due to the size of the problem or the instance having limited capacity, our heuristics have been shown to outperform what can be achieved by the MIP solver in 20,000 seconds. Generally, it is sufficient to run the simplest versions of our algorithms to get good results – increasing the parameters only leads to very small improvements while significantly increasing the run time. The local search algorithm developed for the BFLP is only able to marginally improve upon the results achieved by open and close greedy. However, we have shown that the local search algorithm performs well at improving a bad starting solution quickly.

We have two suggestions for further work that could be considered to extend upon this thesis. Firstly, some more investigation into why considering facilities that performed well in previous iterations worked well for open and close greedy could be conducted. For example, one could attempt to prove (or disprove) that $f(S)$ as defined in Proposition 3 is supermodular

on instances with sufficient capacity, i.e. instances with $C_j \geq \sum_{i \in I} W_{ij}, \forall j \in J$. Secondly, the performance of the heuristics on the more general Generalised Quadratic Assignment Problem and the corresponding Facility Location Problem could be considered.

References

Amini, M. M. and Racer, M. (July 1994). "A rigorous computational comparison of alternative solution methods for the Generalized Assignment Problem". In: *Management Science* 40.7, pp. 868–890. DOI: 10.1287/mnsc.40.7.868.

Burkard, R. E., Eranda, Ç., Pardalos, P. M., and Pitsoulis, L. S. (Oct. 1998). "The Quadratic Assignment Problem". In: *Handbook of Combinatorial Optimization*. Springer. Chap. 27, pp. 1713–1809. DOI: 10.1007/978-1-4613-0303-9_27.

Conforti, M., Gérard, C., and Zambelli, G. (Nov. 2014). *Integer Programming*. Springer International Publishing. DOI: 10.1007/978-3-319-11008-0.

Cordeau, J.-F., Gaudioso, M., Laporte, G., and Moccia, L. (Nov. 2006). "A memetic heuristic for the Generalized Quadratic Assignment Problem". In: *INFORMS Journal on Computing* 18.4, pp. 433–443. DOI: 10.1287/ijoc.1040.0128.

Delft High Performance Computing Centre (DHPC) (2022). *DelftBlue Supercomputer (Phase 1)*. https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1.

Ghirardi, M. and Potts, C. (Sept. 2005). "Makespan minimization for scheduling unrelated parallel machines: A recovering beam search approach". In: *European Journal of Operational Research* 165.2, pp. 457–467. DOI: 10.1016/j.ejor.2004.04.015.

Hahn, P. M., Kim, B.-J., Guignard, M., Smith, J. M., and Zhu, Y.-R. (Nov. 2007). "An algorithm for the Generalized Quadratic Assignment Problem". In: *Computational Optimization and Applications* 40.3, pp. 351–372. DOI: 10.1007/s10589-007-9093-1.

Hitchcock, F. L. (Apr. 1941). "The distribution of a product from several sources to numerous localities". In: *Journal of Mathematics and Physics* 20.1–4, pp. 224–230. DOI: 10.1002/sapm1941201224.

Holmberg, K., Rönnqvist, M., and Yuan, D. (Mar. 1999). "An exact algorithm for the Capacitated Facility Location Problems with Single Sourcing". In: *European Journal of Operational Research* 113.3, pp. 544–559. DOI: 10.1016/s0377-2217(98)00008-3.

Jacobsen, S. K. (Mar. 1983). "Heuristics for the Capacitated Plant Location Model". In: *European Journal of Operational Research* 12.3, pp. 253–261. DOI: 10.1016/0377-2217(83)90195-9.

Karp, R. M. (Aug. 1972). "Reducibility among combinatorial problems". In: *Complexity of Computer Computations*, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

Laporte, G., Nickel, S., Francisco, S. d. G., Fernández, E., and Landete, M. (Feb. 2015). "Fixed-Charge Facility Location Problems". In: *Location science*. 1st ed. Springer Cham. Chap. 3, pp. 47–77. DOI: 10.1007/978-3-319-13111-5.

Lee, C.-G. and Ma, Z. (Jan. 2004). "The Generalized Quadratic Assignment Problem". unpublished. URL: https://www.researchgate.net/publication/228575335_The_generalized_quadratic_assignment_problem.

Lenstra, J. K., Shmoys, D. B., and Tardos, É. (Jan. 1990). "Approximation algorithms for scheduling unrelated parallel machines". In: *Mathematical Programming* 46.1-3, pp. 259–271. DOI: 10.1007/bf01585745.

Martello, S. and Toth, P. (Jan. 1987). "Linear assignment problems". In: *Surveys in Combinatorial Optimization*, pp. 259–282. DOI: 10.1016/s0304-0208(08)73238-9.

Mateus, G. R., Resende, M. G., and Silva, R. M. (Sept. 2010). "Grasp with path-relinking for the Generalized Quadratic Assignment Problem". In: *Journal of Heuristics* 17.5, pp. 527–565. DOI: 10.1007/s10732-010-9144-0.

McKendall, A. and Li, C. (Nov. 2016). "A tabu search heuristic for a Generalized Quadratic Assignment Problem". In: *Journal of Industrial and Production Engineering* 34.3, pp. 221–231. DOI: 10.1080/21681015.2016.1253620.

Montes de Oca, M. A., Ner, F., and Cotta, C. (Oct. 2011). "Local search". In: *Handbook of Memetic Algorithms*. Springer-Verlag. Chap. 3, pp. 29–42. DOI: 10.1007/978-3-642-23247-3_3.

Osman, I. H. (Dec. 1995). "Heuristics for the Generalised Assignment Problem: Simulated annealing and tabu search approaches". In: *Operations-Research-Spektrum* 17.4, pp. 211–225. DOI: 10.1007/bf01720977.

Owen, S. H. and Daskin, M. S. (Dec. 1998). "Strategic facility location: A Review". In: *European Journal of Operational Research* 111.3, pp. 423–447. DOI: 10.1016/s0377-2217(98)00186-6.

Özbakir, L., Baykasoğlu, A., and Tapkan, P. (Feb. 2010). "Bees algorithm for Generalized Assignment Problem". In: *Applied Mathematics and Computation* 215.11, pp. 3782–3795. DOI: 10.1016/j.amc.2009.11.018.

Pal, M., Tardos, É., and Wexler, T. (Oct. 2001). "Facility location with nonuniform hard capacities". In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. DOI: 10.1109/sfcs.2001.959907.

Pinedo, M. L. (Feb. 2016). "Scheduling theory, algorithms, and systems". In: *Scheduling theory, algorithms, and systems*. 5th ed. Springer International. Chap. 5, pp. 113–151. DOI: 10.1007/978-3-319-26580-3.

Risanger, S., Singh, B., Morton, D., and Meyers, L. A. (2021). "Selecting pharmacies for COVID-19 testing to ensure access". In: *Health Care Management Science* 24, pp. 330–338. DOI: 10.1007/s10729-020-09538-w.

Schmitt, C. and Singh, B. (Nov. 2021). *Balancing preferential access and fairness with an application to waste management: Mathematical models, optimality conditions, and heuristics*. (Accessed 13 April 2023). URL: https://www.researchgate.net/publication/355984329_Balancing_preferential_access_and_fairness_with_an_application_to_waste_management_mathematical_models_optimality_conditions_and_heuristics.

Schrijver, A. (2003). "Submodular functions and polymatroids". In: *Combinatorial optimization*. Springer. Chap. 44, pp. 766–785.

Shmoys, D. B. and Tardos, É. (Feb. 1993). "An approximation algorithm for the Generalized Assignment Problem". In: *Mathematical Programming* 62.1-3, pp. 461–474. DOI: 10.1007/bf01585178.

Sridharan, R. (Dec. 1995). "The Capacitated Plant Location Problem". In: *European Journal of Operational Research* 87.2, pp. 203–213. DOI: 10.1016/0377-2217(95)00042-o.

Weber, A., Pick, G., and Friedrich, C. J. (1965). *Alfred Weber's theory of the location of industries*. The University of Chicago Press.

Wolfe, P. (July 1959). "The simplex method for quadratic programming". In: *Econometrica* 27.3, pp. 382–398. DOI: 10.2307/1909468.

Zhang, J., Chen, B., and Ye, Y. (May 2005). "A multiexchange local search algorithm for the Capacitated Facility Location Problem". In: *Mathematics of Operations Research* 30.2, pp. 389–403. DOI: 10.1287/moor.1040.0125.

A

Examples for SCUAP and UAP

When considering the SCUAP in Section 3.2, we note that even when the capacity of facilities is sufficient, finding a solution with a certain objective function is $\mathcal{NP}$-hard. Below, we provide two examples that show how intuitive ideas of assigning users lead to suboptimal solutions. We first provide an example for how an intuitively feasible solution that assigns users to their most preferred facility is suboptimal.

Example 4. *Consider an instance of Model (3.1) with* $I = \{1, 2, 3\}$, $J = \{1, 2\}$. *Let* $C_1 = 10, C_2 = 20$. *Let the* $W_{i,j}$ *be defined as follows:*

$$W_{1,1} = 3 \qquad W_{1,2} = 4$$
$$W_{2,1} = 4 \qquad W_{2,2} = 10$$
$$W_{3,1} = 2 \qquad W_{3,2} = 0$$

The feasible solution that assigns each i *to* $j^* = \arg\max_{j \in J} W_{i,j}$ *is* $I_1 = \{3\}$, $I_2 = \{1, 2\}$ *with a corresponding objective function value of* $10(1 - 0.2)^2 + 20(1 - 0.7)^2 = 8.1$. *A better solution is obtained by* $I_1 = \{1, 3\}$, $I_2 = \{2\}$ *with a corresponding objective function value of* $10(1-0.5)^2+20(1-0.5)^2 = 7.5$.

Next, we provide an example for how another intuitive feasible solution that assigns user i to $j^* = \arg\max_{j \in J} \frac{W_{i,j}}{C_j}$ is still suboptimal.

Example 5. *Consider an instance of Model (3.1) with* $I = \{1, 2, 3\}$, $J = \{1, 2\}$. *Let* $C_1 = C_2 = 10$. *Let the* $W_{i,j}$ *be defined as follows:*

$$W_{1,1} = 1 \qquad W_{1,2} = 0$$
$$W_{2,1} = 0 \qquad W_{2,2} = 4$$
$$W_{3,1} = 2 \qquad W_{3,2} = 3$$

The feasible solution that assigns each i *to* $j^* = \arg\max_{j \in J} \frac{W_{i,j}}{C_j}$ *is* $I_1 = \{1\}$, $I_2 = \{2, 3\}$ *with a corresponding objective function value of* $10(1 - 0.1)^2 + 10(1 - 0.7)^2 = 9$. *A better solution is obtained by* $I_1 = \{1, 3\}$, $I_2 = \{2\}$ *with an objective value of* $10(1 - 0.3)^2 + 10(1 - 0.4)^2 = 8.5$.

Example 4 and 5 suggest that a balance between utilisation and access is required. These examples provide a motivation to study the hardness of the problem, as we do in Section 3.2.

The following example provides an illustration of the mapping we use for the construction in Theorem 1.

Example 6. *Consider an instance of the* PP *as follows:* $T = \{t_1, t_2, t_3, t_4\} = \{2, 3, 4, 5\}$. *The corresponding* UAP *instance then has* $I = \{1, 2, 3, 4\}$, $S = \{1, 2\}$, $C_1 = C_2 = 7$ *and* $W_{i,j} = t_i, \forall i \in I, j \in S$.

Given a YES *instance of the* PP *by* $L = \{1, 4\}$, *the corresponding* UAP *solution is then* $I_1 = L = \{1, 4\}$ *and* $I_2 = \{1, 2, 3, 4\} \setminus \{1, 4\} = \{2, 3\}$. *This is a feasible solution to the* UAP *since both facilities have 7 people assigned to them, which is within their capacity of 7. Conversely, a* YES *instance (this is the only feasible solution) to the* UAP *is to assign users* $1, 4$ *to the first facility and* $2, 3$ *to the second, i.e.* $I_1 = \{1, 4\}$ *and* $I_2 = \{2, 3\}$ *since then both facilities have 7 people assigned to them. The corresponding solution to the* PP *is* $L = \{1, 4\}$. *We have* $\sum_{i \in L} t_i = 5 + 2 = 7 = 3 + 4 = \sum_{i \in \{1,2,3,4\} \setminus L} t_i$.

Similar to Example 6, the following example provides an illustration of the mapping we use for the construction in Theorem 2.

Example 7. *Consider an instance of the* PP *as follows:* $T = \{t_1, t_2, t_3, t_4\} = \{2, 3, 4, 5\}$. *The corresponding* SCUAP *instance then has* $I = \{1, 2, 3, 4\}$, $S = \{1, 2\}$, $C_1 = C_2 = 14$, $W_{i,j} = t_i, \forall i \in I, j \in S$, *and* $M = 7$.

Given a YES *instance of the* PP *by* $L = \{1, 4\}$, *the corresponding* SCUAP *solution is then* $I_1 = L = \{1, 4\}$ *and* $I_2 = \{1, 2, 3, 4\} \setminus \{1, 4\} = \{2, 3\}$. *This is a* YES *instance of the* SCUAP *since* $\sum_{i \in I} W_{i,1} = 7 \leq 14 = C_1$, $\sum_{i \in I} W_{i,2} = 7 \leq 14 = C_2$ *and the objective function value is* $14(1 - \frac{7}{14})^2 + 14(1 - \frac{7}{14})^2 = 7 \leq M$. *Conversely, a* YES *instance to the* SCUAP *(there is only one optimal solution) is to assign users* $\{1, 4\}$ *to the first facility and* $\{2, 3\}$ *to the second, giving an optimal objective function value of* $7 = M$; *i.e.,* $I_1 = \{1, 4\}$ *and* $I_2 = \{2, 3\}$. *The corresponding* YES *instance to the* PP *is given by* $L = \{1, 4\}$, *since we have* $\sum_{k \in L} t_k = 5 + 2 = 7 = 3 + 4 = \sum_{k \in \{1,2,3,4\} \setminus L} t_k$.

B

BUAP local search algorithm subroutines and algorithms pseudocode

In this appendix, we provide pseudocodes for the two local search algorithms, Algorithm 12 and Algorithm 14, which we discuss in Section 4.3. We further provide two subroutines, Algorithm 13 and 15, that we use within these algorithms, respectively. Both these algorithms seek to improve a given feasible solution for Model (3.1).

Algorithm 12 The `local_search_reassign` algorithm

Input: an instance of Model (3.1); an assignment x; a cut-off in preference P (default 0.2); a time limit t for the `while` loop; the negation of the minimum improvement l for the `while` loop (default -10); the `reassignment_is_better` subroutine.

Output: status f = either `feasible` or `infeasible` for the given inputs; if `feasible`, returns an assignment x for the input instance with corresponding utilisation u and objective function value $\overline{z}$.

1: $R_j \leftarrow C_j - \sum_{j \in I} U_i P_{ij} x_{ij}$, $\forall j \in S$.
2: $A_j = \{i \in I : x_{ij} = 1\}$, $\forall j \in S$; $B_i \leftarrow \{j \in S : P_{ij} \geq P\}$, $\forall i \in I$; $\gamma \leftarrow l - 1$.
3: **if** $\exists j \in S$ s.t. $R_j < 0$, return `infeasible`.
4: **while** $\gamma < l$ and time since start of loop $< t$ **do**
5: $\quad \gamma \leftarrow 0$.
6: $\quad$ **for** $j' \in S$ **do**
7: $\quad\quad$ **if** $|A_{j'}| \leq 1$, break. Continue with next iteration of outer `for` loop (Line 6).
8: $\quad\quad$ **for** $i \in A_{j'}$ **do**
9: $\quad\quad\quad$ **for** $j'' \in B_i$ **do**
10: $\quad\quad\quad\quad$ $[b, c] \leftarrow$ `if_reassignment_better`$(i, j', j'', R_{j'}, R_{j''})$.
11: $\quad\quad\quad\quad$ **if** $b =$ `True`
12: $\quad\quad\quad\quad\quad$ $x_{ij'} \leftarrow 0$; $x_{ij''} \leftarrow 1$; $A_{j'} \leftarrow A_{j'} \setminus \{i\}$; $A_{j''} \leftarrow A_{j''} \cup \{i\}$.
13: $\quad\quad\quad\quad\quad$ $R_{j'} \leftarrow R_{j'} + U_i P_{ij'}$; $R_{j''} \leftarrow R_{j''} - U_i P_{ij''}$; $\gamma \leftarrow \gamma + c$.
14: $\quad\quad\quad\quad\quad$ break. Continue with next iteration of inner `for` loop (Line 8).
15: return $f \leftarrow$ `feasible`; x; $u_j \leftarrow \frac{\sum_{i \in I_j} W_{ij} x_{ij}}{C_j}$, $\forall j \in J$; $\overline{z} \leftarrow \sum_{j \in J} C_j (1 - u_j)^2$.

We first discuss our `local_search_reassign` algorithm that we summarise in Algorithm 12.

We initialise with a given feasible solution, and compute the remaining capacity R_j for all facilities, the set of users A_j assigned to each facility j, and the facilities B_i that user i has at least preference P for (Lines 1-2). If the input assignment is infeasible – measured by a facility used in excess of its capacity – the algorithm immediately terminates, returning `infeasible` (Line 3). If not, we continue to the main while loop; this terminates either when the improvement in the objective function value in the previous iterations is small or if a time limit is exceeded (Line 4). We choose a time limit here instead of terminating after a certain number of iterations since this algorithm is generally very quick to terminate on our good starting solutions generated by `relaxation_rounding` or `greedy_assignment`. Hence, terminating when a time limit is reached is a fail-safe – we practically never reach this time limit, but it is there in case of a very unusual starting solution. Hence, we avoid introducing another parameter that needs careful tuning to ensure it is set correctly.

Algorithm 13 The `if_reassignment_better` subroutine

Input: user i to reassign; facility j' that user i is currently assigned to; facility j'' that user i is considered to be reassigned to; remaining capacities of both facilities $R_{j'}, R_{j''}$.

Output: either `True` if the reassignment improves the objective function value, and the corresponding improvement; `False` otherwise.

1: **if** $U_i P_{ij''} > R_{j''}$
2: return `False`.
3: $obj' \leftarrow C_{j'} \left(\frac{R_{j'}}{C_{j'}}\right)^2 + C_{j''} \left(\frac{R_{j''}}{C_{j''}}\right)^2$.
4: $obj'' \leftarrow C_{j'} \left(\frac{R_{j'}+U_i P_{ij'}}{C_{j'}}\right)^2 + C_{j''} \left(\frac{R_{j''}-U_i P_{ij''}}{C_{j''}}\right)^2$.
5: **if** $obj'' < obj'$
6: return `True`; $obj'' - obj'$.
7: **else**
8: return `False`.

We consider reassignments of users only to facilities that are open. Consider a user i assigned to facility j' that we seek to reassign to facility j'' (where the preference of i to j'' is at least P). We then use the subroutine in Algorithm 13 to determine if the reassignment is expected to perform better (Line 10). If so, we update all the parameters: the new assignment x, the new sets A_j, the new remaining capacities R_j, and the change in the objective function γ (Lines 11-13). Note that this change will be a negative number since we calculate the change in objective by subtracting the previous objective from the improved objective. Hence, the termination condition of the while loop is $\gamma < l$ and not $\gamma > l$. We continue this procedure until one of the termination criteria is met. The algorithm then returns the best assignment, with the corresponding utilisation and objective function value (Line 15).

Next, we discuss the local search swap algorithm that we summarise in Algorithm 14. This relies on a subroutine we present in Algorithm 15. The idea here is similar to the above-mentioned reassignment algorithm, with the difference that we swap two users instead of reassigning them. To this end, there are two major changes in Algorithm 14 from Algorithm 12. First, we have a second `for` loop to include the user i'' assigned to j''; here, j'' is also the facility we consider swapping a user from with j' (Line 9). Secondly, if we do identify a better assignment via swapping, we continue with the next iteration of the outer `for` loop rather than the inner `for` loop as in `local_search_reassign` (Line 15). We do so since we now have an additional user added to the set $A_{j'}$, unlike in the `local_search_reassign` algorithm.

Algorithm 14 The `local_search_swap` algorithm

Input: an instance of Model (3.1); an assignment x; a cut-off in preference P (default 0.2); a time limit t for the `while` loop; the negation of the minimum improvement l for the `while` loop (default -10); the `swap_is_better` subroutine.

Output: status f = either `feasible` or `infeasible` for the given inputs; if `feasible`, returns an assignment x for the input instance with corresponding utilisation u and objective function value $\overline{z}$.

1: $R_j \leftarrow C_j - \sum_{j \in I} U_i P_{ij} x_{ij},\ \forall j \in S$.
2: $A_j \leftarrow \{i \in I : x_{ij} = 1\},\ \forall j \in J$; $B_i \leftarrow \{j \in S : P_{ij} \geq P\},\ \forall i \in I$; $\gamma \leftarrow l - 1$.
3: **if** $\exists j \in S$ s.t. $R_j < 0$, return `infeasible`.
4: **while** $\gamma < l$ and time since start of loop $< t$ **do**
5: $\quad \gamma \leftarrow 0$.
6: $\quad$ **for** $j' \in S$ **do**
7: $\quad\quad$ **for** $i' \in A_{j'}$ **do**
8: $\quad\quad\quad$ **for** $j'' \in B_{i'}$ **do**
9: $\quad\quad\quad\quad$ **for** $i'' \in A_{j''}$ **do**
10: $\quad\quad\quad\quad\quad [b, N', N'', c] \leftarrow$ `if_swap_better`$(i', i'', j', j'', R_{j'}, R_{j''})$.
11: $\quad\quad\quad\quad\quad$ **if** $b =$ `True`
12: $\quad\quad\quad\quad\quad\quad x_{i'j'} \leftarrow 0$; $x_{i'j''} \leftarrow 1$; $x_{i''j'} \leftarrow 1$; $x_{i''j''} \leftarrow 0$.
13: $\quad\quad\quad\quad\quad\quad A_{j'} \leftarrow A_{j'} \setminus \{i'\} \cup \{i''\}$; $A_{j''} \leftarrow A_{j''} \setminus \{i''\} \cup \{i'\}$.
14: $\quad\quad\quad\quad\quad\quad R_{j'} \leftarrow N'$; $R_{j''} \leftarrow N''$; $\gamma \leftarrow \gamma + c$.
15: $\quad\quad\quad\quad\quad\quad$ break. Continue with next iteration of outer `for` loop (Line 6).
16: return $f \leftarrow$ `feasible`; x; $u_j \leftarrow \frac{\sum_{i \in I_j} W_{ij} x_{ij}}{C_j},\ \forall j \in S$; $\overline{z} \leftarrow \sum_{j \in J} C_j (1 - u_j)^2$.

Algorithm 15 The `if_swap_better` subroutine

Input: users i' and i'' to swap; facilities j' and j'' that the users are currently assigned to, respectively; remaining capacities of both facilities $R_{j'}, R_{j''}$.

Output: either `True` if the swapping improves the objective function value, the corresponding improvement, and updated remaining capacities of both facilities; `False` otherwise.

1: $N_{j'} \leftarrow R_{j'} + U_{i'} P_{i'j'} - U_{i''} P_{i''j'}$; $N_{j''} \leftarrow R_{j''} + U_{i''} P_{i''j''} - U_{i'} P_{i'j''}$.
2: **if** $N_{j'} < 0$ or $N_{j''} < 0$
3: $\quad$ return `False`.
4: $obj' \leftarrow C_{j'} \left(\frac{R_{j'}}{C_{j'}}\right)^2 + C_{j''} \left(\frac{R_{j''}}{C_{j''}}\right)^2$.
5: $obj'' \leftarrow C_{j'} \left(\frac{N_{j'}}{C_{j'}}\right)^2 + C_{j''} \left(\frac{N_{j''}}{C_{j''}}\right)^2$.
6: **if** $obj'' < obj'$
7: $\quad$ return `True`; $obj'' - obj'$; $N_{j'}$; $N_{j''}$.
8: **else**
9: $\quad$ return `False`.

C

Helper functions BFLP local search

In Algorithm 11, we call some functions that we only discuss now since they are easily summarised in words but would only distract from the main ideas of the local search algorithm if they were to be included in Algorithm 11.

Algorithm 16 is a procedure used in Algorithm 11 to initialise the parameter δ. Hence, for each facility $j' \in J$, it computes what the change in the objective function would be if that facility were to be closed / opened if it currently is open / closed.

Algorithm 16 The `initialise_change` procedure

Input: an instance of Model (2.2); a feasible solution to the model (assignment x, the set of open facilities S, the objective function value $\overline{z}$); scalar depth d for Algorithm 9.

Output: the change in objective function δ observed by each facility when it is opened / closed based on the current assignment.

Initialise: $\delta_j \leftarrow \infty$, $\forall j \in J$.
for $j' \in S$ **do**
    $[f', x', u', z'] \leftarrow$ `greedy_reassign`(I, S, j', x).
    if $f' =$ `feasible`
        $\delta_j \leftarrow z' - \overline{z}$.
for $j' \in J \setminus S$ **do**
    $[f', x', u', z'] \leftarrow$ `greedy_reassign_open`(I, S, j', x, d).
    if $f' =$ `feasible`
        $\delta_j \leftarrow z' - \overline{z}$.
return δ.

The other procedure, Algorithm 17, used in the local search algorithm, determines how the next facility to consider is chosen from J'. Here, we want to choose a facility that according to δ is good to consider, i.e. has small δ. Since δ is based on closing the facility if it is open and opening if it is closed, it would not make sense to simply choose the facility with the smallest δ: in that case as long as there are still closed facilities in J', one of those would be chosen since opening a facility cannot increase the objective. This is because in `greedy_reassign_open`, we only accept changes that improve the objective. Instead, in Algorithm 17 we first make a random choice, based on how many open facilities there are out of the total number of facilities, of whether to consider an open or closed facility. Then, we consider the best open / closed facility in the list of facilities we want to consider, J', based on it having the smallest δ. In the (unlikely) case that multiple facilities have the minimum δ, we choose one of these randomly. The extra

conditions seen in Line 2 deal with the case that J' contains no open or no closed facilities, in which case the best facility from the non-empty set is chosen regardless of the random choice made of whether to consider an open or a closed facility.

Algorithm 17 The `choose_fac_based_on_change` procedure

Input: the set of facilities J; the set of facilities that still need to be considered $J' \subseteq J$; the set of open facilities $S \subseteq J$; the change in objective for each facility δ.

Output: the chosen facility j'.

1: $r \leftarrow$ `random_choice` $([0, 1])$; $j' \leftarrow$ "$None$".

2: **if** $\left(r < \frac{|S|}{|J|} \text{ and } |J' \cap S| > 0\right)$ or $|J' \cap (J \setminus S)| = 0$

3: $\quad j' \leftarrow$ `random_choice` $\left(\arg\min_{j \in J' \cap S}\{\delta_j\}\right)$.

4: **else**

5: $\quad j' \leftarrow$ `random_choice` $\left(\arg\min_{j \in J' \cap (J \setminus S)}\{\delta_j\}\right)$.

6: return j'.

D

Data generation

We give the pseudocode of how the data for Artificial Instance 1 and 2 was generated in Algorithm 18. Artificial Instance 1 was generated with $n = 5000$, $r = 0.5$, $lon_{min} = 5.8663153$, $lon_{max} = 15.0419319$, $lat_{min} = 47.2701114$, $lat_{max} = 55.099161$. These coordinates approximately encapsulate Germany. Artificial Instance 2 was generated with $n = 1500$, $r = 0.3$, $lon_{min} = 9.0$, $lon_{max} = 11.5$, $lat_{min} = 47.3$, $lat_{max} = 49.0$. This is approximately a quarter of the size of Bavaria.

Algorithm 18 Data generation

Input: number of users, n; probability to open a facility, r; coordinates boundaries from which to choose the locations, $lon_{min}, lon_{max}, lat_{min}, lat_{max}$; function $gauss(a, b)$ to return a random number from the Gaussian distribution with mean a and standard deviation b; function $uniform(a, b)$ to return a number from the range $[a, b]$ uniformly at random; function $uniformInt(a, b)$ to return an integer from the range $[a, b]$ uniformly at random inclusive of a and b; function $\hat{P}(a, b)$ from Schmitt and Singh (2021) where a is the distance in kilometres and b is an indicator of whether the user is rural (0) or urban (1); function $distance(lon_1, lat_1, lon_2, lat_2)$ to return distance in kilometres between the two coordinates.

Output: a data class for Model 2.1, i.e. sets I and J; C_j for $j \in J$; U_i for $i \in I$; P_{ij} for $i \in I$, $j \in J$.

$I \leftarrow \{0, 1, \ldots, n-1\}$; $J \leftarrow \emptyset$; $C_j \leftarrow 0$, $\forall j \in I$; $U_i \leftarrow 0$, $\forall i \in I$; $LU_i \leftarrow [0, 0]$, $\forall i \in I$; $LF_j \leftarrow [0, 0]$, $\forall j \in I$; $T_i \leftarrow 0$, $\forall i \in I$.
for $i = 0, 1, \ldots, n-1$ **do**
    $U_i = uniformInt(0, 20000)$; $LU_i \leftarrow [uniform(lon_{min}, lon_{max}), uniform(lat_{min}, lat_{max})]$.
    $T_i \leftarrow uniformInt(0, 1)$.
    if $uniform(0, 1) < r$
        $J \leftarrow J \cup \{i\}$; $C_j \leftarrow uniformInt(0, 80000)$.
        $LF_j = [LU_{i0} + gauss(0, 0.01), LU_{i1} + gauss(0, 0.01)]$.
for $i \in I$ **do**
    for $j \in J$ **do**
        $P_{ij} = \hat{P}(distance(LU_{i0}, LU_{i1}, LF_{j0}, LF_{j1}), T_i)$
return $I; J; C, U; P$.

E

Additional results close greedy

We give some additional results for the basic close greedy algorithm, Algorithm 6, which is discussed in Section 5.1.1, and whose results are discussed in Section 6.3.1.

Table E.1: Results of close greedy algorithm, Algorithm 6, using m = `greedy_assignment` throughout. The final assignment is done with m' = `relaxation_rounding` with `local_search_reassign` in both cases. This is with $n_c = 20$ and on the larger instance. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the gap between the solution and the lower bound the `MIP` solver reaches after 20,000 seconds. The run time, the difference to the `MIP` after 20,000 seconds and at the same time, calculated using Equation (6.1), are displayed. For details, see Section 6.3.1.

Budget factor	**`MIP` gap after 20k s [%]**	**Run time [s]**	Δ_{MIP} **[%]**	Δ_S **[%]**
0.9	1.33	1770	-2.58	-2.58
0.8	2.36	3392	-6.69	-6.69
0.7	3.53	4856	-11.66	-11.66
0.6	5.07	6053	-17.50	-17.45
0.5	7.49	7193	-22.42	-22.32
0.4	6.66	8176	-31.62	-31.62
0.3	7.42	9038	-34.70	-34.70
0.2	7.71	9827	-29.51	-28.12
0.1	7.84	10507	-16.41	-15.02

F

Additional results open greedy

We give some additional results for the open greedy algorithm, Algorithm 10, which is discussed in Section 5.2, and whose results are discussed in Section 6.4.

Table F.1: Results of open greedy algorithm, Algorithm 10, with m = `greedy_assignment` with `local_search_reassign`, m' = `relaxation_rounding` with `local_search_reassign`. This is with $n_c = 50$ and $n_f = |J|$ on the smaller instance. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the gap between the solution and the lower bound the `MIP` solver reaches after 20,000 seconds. For each depth d, the first column shows the run time. The second column is calculated based on Equation (6.1). For details, see Section 6.4.

		$d = 1$		$d = 2$		$d = 3$	
Budget factor	**`MIP` gap after 20k s [%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
0.9	0.57	16	0.00	32	0.00	40	0.00
0.8	1.41	15	0.00	30	0.02	38	0.00
0.7	2.19	13	0.06	26	0.00	34	0.00
0.6	3.05	12	-0.01	23	0.00	31	0.00
0.5	3.53	10	-0.01	20	0.00	26	0.00
0.4	3.93	8	0.00	16	0.00	24	0.00
0.3	4.12	7	-0.07	14	-0.02	18	-0.01
0.2	4.64	5	0.00	9	0.02	13	0.02
0.1	4.18	3	-0.03	5	-0.03	7	0.00

Table F.2: Results of open greedy algorithm, Algorithm 10, with m = `greedy_assignment` with `local_search_reassign`, m' = `relaxation_rounding` with `local_search_reassign`. This is with $n_c = 5$ and $n_f = |J|$ on the larger instance. The budget factor indicates the proportion of all facilities that should remain open. The second column shows the gap between the solution and the lower bound the `MIP` solver reaches after 20,000 seconds. For each depth d, the first column shows the running time. The second column is calculated based on Equation (6.1). For details, see Section 6.4.

		$d = 1$		$d = 2$		$d = 3$	
Budget factor	**`MIP` gap after 20k s [%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
0.9	1.33	150	-0.02	229	0.01	286	0.00
0.8	2.36	136	0.05	210	0.09	266	0.09
0.7	3.53	122	0.14	189	0.19	245	0.18
0.6	5.07	108	0.58	169	0.63	222	0.64
0.5	7.49	95	2.01	148	2.07	199	2.07
0.4	6.66	83	0.16	129	0.26	175	0.25
0.3	7.42	72	0.34	108	0.52	150	0.50
0.2	7.71	63	0.30	89	0.24	122	0.30
0.1	7.84	56	0.37	68	0.46	89	0.44

G

Additional results local search

Table G.1: Results of `BFLP` local search algorithm, Algorithm 11. The starting solution is from Algorithm 8 with m = `greedy_assignment` with `local_search_reassign`, m' = `relaxation_rounding` with `local_search_reassign`, $n_c = 5$ or $n_c = 50$ on the larger instance. Local search is run with $n_c = 5$, $d = 1$ and m' = `relaxation_rounding` with `local_search_reassign`. The budget factor indicates the proportion of all facilities that should remain open. The third column shows the difference to the `MIP` after 20k seconds before the local search is run, so of the starting solution, calculated based on Equation (6.1). For each number of iterations l, the first column shows the run time. The second column is calculated based on Equation (6.1). For details, see Section 6.5.

			$l = 20$		$l = 100$		$l = 200$	
Starting instance	**Budget factor**	**Diff. to `MIP` after 20k s before local search [%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
Close greedy with $n_c = 5$	0.9	-0.006	64	0.007	75	0.016	85	0.016
	0.8	0.041	63	0.054	76	0.067	89	0.067
	0.7	0.126	57	0.154	68	0.151	83	0.156
	0.6	0.508	59	0.569	73	0.559	88	0.601
	0.5	1.936	99	2.012	111	2.003	123	2.006
	0.4	0.106	101	0.181	112	0.201	124	0.183
	0.3	0.353	77	0.440	77	0.427	108	0.432
	0.2	0.237	82	0.310	88	0.276	93	0.310
	0.1	0.388	93	0.415	109	0.424	110	0.424
Close greedy with $n_c = 50$	0.9	-0.004	79	0.006	92	0.012	100	0.015
	0.8	0.044	62	0.059	71	0.067	90	0.069
	0.7	0.131	92	0.145	103	0.146	121	0.165
	0.6	0.517	96	0.590	106	0.590	121	0.595
	0.5	1.931	94	2.018	105	1.997	117	1.994
	0.4	0.131	93	0.189	103	0.192	112	0.188
	0.3	0.359	90	0.432	99	0.425	111	0.438
	0.2	0.253	94	0.301	96	0.265	104	0.301
	0.1	0.349	86	0.409	91	0.399	98	0.409

Table G.2: Results of `BFLP` local search algorithm, Algorithm 11. The starting solution is from Algorithm 10 with m = `greedy_assignment` with `local_search_reassign`, m' = `relaxation_rounding` with `local_search_reassign`, $n_c = 5$, $d = 2$ on the smaller instance. The second starting solution is the same but with $n_c = 50$ in the open greedy algorithm, as seen in Table F.1. The local search is run with $l = \infty$ and m' = `relaxation_rounding` with `local_search_reassign`. The budget factor indicates the proportion of all facilities that should remain open. The third column shows the difference to the `MIP` after 20k seconds before the local search is run, so of the starting solution, calculated based on Equation (6.1). For each depth d, the first column shows the run time. The second column is calculated based on Equation (6.1). For details, see Section 6.5.

			$d = 1$				$d = 2$			
			$n_c = 50$		$n_c = \|J\|$		$n_c = 50$		$n_c = \|J\|$	
Starting instance	**Budget factor**	**Δ_{MIP} before local search [%]**	**Run time [s]**	**Δ_{MIP} [%]**	**Run time [s]**	**Δ_{MIP} [%]**	**Run time [s]**	**Δ_{MIP} [%]**	**Run time [s]**	**Δ_{MIP} [%]**
Open greedy with $n_c = 5$, $d = 2$	0.9	-0.002	19	-0.002	18	-0.002	19	-0.002	24	-0.002
	0.8	-0.002	21	-0.002	29	-0.002	35	-0.002	41	-0.002
	0.7	-0.019	21	-0.009	36	-0.009	35	-0.009	58	-0.009
	0.6	-0.018	21	-0.004	41	-0.004	35	-0.004	63	-0.004
	0.5	-0.061	18	0.000	42	0.000	39	0.000	65	0.000
	0.4	-0.139	17	-0.002	35	-0.002	30	0.002	68	0.002
	0.3	-0.049	15	-0.008	29	-0.008	23	-0.008	53	-0.008
	0.2	0.016	13	0.016	23	0.016	21	0.016	44	0.016
	0.1	-0.025	7	-0.025	26	-0.025	11	-0.025	23	-0.025
Open greedy with $n_c = 50$, $d = 2$	0.9	-0.002	14	-0.002	19	-0.002	20	-0.002	25	-0.002
	0.8	-0.002	23	-0.002	28	-0.002	35	-0.002	41	-0.002
	0.7	0.000	22	0.000	37	0.000	36	0.000	57	0.000
	0.6	-0.005	21	-0.005	38	-0.005	34	-0.005	62	-0.005
	0.5	0.000	18	0.000	41	0.000	32	0.000	68	0.000
	0.4	0.002	17	0.002	50	0.002	27	0.002	64	0.002
	0.3	-0.020	38	-0.020	31	-0.020	25	-0.020	56	-0.020
	0.2	0.016	21	0.016	21	0.016	19	0.016	43	0.016
	0.1	-0.025	7	-0.025	12	-0.025	11	-0.025	23	-0.025

Table G.3: Results of `BFLP` local search algorithm, Algorithm 11. The starting solution is from Algorithm 8 with m = `greedy_assignment` with `local_search_reassign`, m' = `relaxation_rounding` with `local_search_reassign`, $n_c = 5$ on the smaller instance, as seen in Table 6.7. The second starting solution is the same but with $n_c = 50$ in the close greedy algorithm. The local search is run with $l = \infty$ and m' = `relaxation_rounding` with `local_search_reassign`. The budget factor indicates the proportion of all facilities that should remain open. The third column shows the difference to the `MIP` after 20k seconds before the local search is run, so of the starting solution, calculated based on Equation (6.1). For each depth d, the first column shows the run time. The second column is calculated based on Equation (6.1). For details, see Section 6.5.

			$d = 1$				$d = 2$			
			$n_c = 50$		$n_c = \lvert J \rvert$		$n_c = 50$		$n_c = \lvert J \rvert$	
Starting instance	**Budget factor**	Δ_{MIP} **before local search [%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**	**Run time [s]**	Δ_{MIP} **[%]**
Close greedy with $n_c = 5$	0.9	-0.008	14	-0.004	17	-0.004	20	-0.004	24	-0.004
	0.8	-0.025	23	-0.012	29	-0.012	41	-0.003	46	-0.003
	0.7	-0.057	26	-0.016	37	-0.016	45	-0.010	58	-0.010
	0.6	-0.094	20	-0.042	46	-0.020	49	-0.020	84	-0.020
	0.5	-0.157	18	-0.036	41	-0.036	44	-0.036	73	-0.036
	0.4	-0.075	18	-0.037	38	-0.037	32	-0.029	70	-0.029
	0.3	-0.258	19	-0.087	31	-0.087	26	-0.020	61	-0.020
	0.2	-0.094	14	-0.077	24	-0.077	26	0.016	57	0.005
	0.1	-0.217	6	-0.025	11	-0.025	10	-0.025	22	-0.025
Close greedy with $n_c = 50$	0.9	-0.008	13	-0.004	17	-0.004	18	-0.004	33	-0.004
	0.8	-0.025	24	-0.012	33	-0.012	60	-0.003	49	-0.003
	0.7	-0.069	30	-0.016	38	-0.016	51	-0.010	61	-0.010
	0.6	-0.101	25	-0.040	43	-0.040	37	-0.020	76	-0.020
	0.5	-0.141	19	-0.036	39	-0.036	33	-0.036	67	-0.036
	0.4	-0.051	16	-0.037	34	-0.037	29	-0.0003	66	-0.0003
	0.3	-0.258	16	-0.087	31	-0.087	29	-0.029	63	-0.020
	0.2	-0.094	14	-0.077	23	-0.077	24	0.016	52	0.005
	0.1	-0.217	6	-0.025	11	-0.025	10	-0.025	22	-0.025