TILBURG UNIVERSITY

Capacitated Maximum Covering Location Problem with Closest Assignment Constraints: MIP Solver and GRASP Heuristic

Case-Study: Stroke Facility Allocation in Timor-Leste

by

Annelies Slot (2058390)

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Business Analytics and Operations Research

Tilburg School of Economics and Management
Tilburg University

Supervised by:
Prof. Dr. Goos Kant (Tilburg University)
Prof. Dr. Ir. Dick den Hertog (Analytics for a Better World)
Prof. Dr. Joaquim Gromicho (Analytics for a Better World)

Date:
December 23, 2024

Abstract

This thesis tackles the Capacitated Maximum Covering Location Problem with Closest Assignment Constraints (CMCLP-CAC), focusing on large-scale instances through the development of exact and heuristic methods. The problem is first formulated and enhanced, then solved using a Mixed Integer Programming (MIP) solver, which achieves optimal solutions for the Timor-Leste dataset comprising 151,765 demand locations and 1,576 potential facilities (3 km grid size). However, the solver faces memory limitations for larger datasets, such as those with 3,539 potential facilities (2 km grid size).

To overcome these limitations, a GRASP heuristic is developed to efficiently handle very large-scale CMCLP-CAC instances. Building on previous work by Fleur Theulen [5], the heuristic incorporates a swap-based local search adapted for maximum covering problems, a sparse matrix implementation, and path relinking to enhance scalability and computational efficiency. This approach is further extended to include capacity constraints and closest assignment rules, introducing additional complexity while maintaining solutions within a 2-hour time frame.

For the Timor-Leste dataset with existing facility locations, the GRASP heuristic achieves a coverage difference of at most 0.096% compared to the MIP solver's optimal solutions. Moreover, when solving larger datasets with 3,539 potential facilities, the heuristic improves coverage by at least 0.331% for capacities of 100,000 citizens by opening 12 additional facilities, with even greater improvements observed for smaller capacities. Notably, the heuristic demonstrates scalability by solving instances with up to 14,147 potential facilities (1 km grid size), showcasing its effectiveness for large-scale CMCLP-CAC problems.

Acknowlegements

I would like to express my sincere gratitude to my supervisors for their invaluable guidance, insightful suggestions, and unwavering enthusiasm throughout this journey. A special thanks to Goos Kant for introducing this interesting research to Tilburg University, for our enriching weekly discussions that shaped the core concepts of this work, and for always listening attentively, and inspiring my ideas with insightful questions. My sincere appreciation also goes to Joaquim Gromicho for his suggestions on more efficient implementations and for sharing his valuable coding insights, which have greatly enhanced my programming skills. Lastly, I am grateful to Dick den Hertog for laying the groundwork through 'Analytics for a Better World' and for always being open to brainstorming new ideas.

Moreover, I want to take a moment to express my appreciation to those closest to me who have supported me not only during this thesis but throughout my entire academic journey at the Department of Econometrics & Operations Research at Tilburg University, from my bachelor's to my master's. To my mom, dad, and Julian, thank you for always listening, believing in me, and giving me the support I needed whenever it mattered most. I am also grateful to my study friends, especially Dominique, for those long hours spent studying at the university, being open to brainstorming, and working tirelessly on assignments together to achieve the best possible results.

Contents

1 Introduction

For more than a century, non-profit organizations have been instrumental in addressing the needs of underserved communities, providing essential services such as healthcare, education, and access to clean water. These organizations often step in where governments encounter challenges, working in close collaboration with local communities and swiftly responding to pressing needs. However, governments bring critical resources to the table, including funding, infrastructure, and regulatory authority, enabling the delivery of services on a much larger scale. Together, their collaboration has the potential to create a long-lasting and sustainable impact, particularly in regions with limited resources or where access to basic services remains difficult.

A prominent example is the Red Cross, established in 1863 by Henry Dunant to provide humanitarian assistance during times of war [21]. Initially focused on emergency medical care, the organization quickly evolved into a global leader in improving healthcare access. It set a benchmark for non-profits in addressing urgent health crises and tackling broader health challenges, ranging from disease prevention to primary care in marginalized communities.

Access to healthcare is especially vital for the treatment of life-threatening conditions such as stroke, heart attack, and traumatic injuries. Stroke, for example, is the leading cause of disability and the second leading cause of death worldwide [9]. Timely treatment significantly reduces the risk of long-term disability, making service distance—the maximum distance between a patient and a healthcare facility to receive timely care—a critical factor in determining adequate healthcare coverage. By providing access within this service distance, non-profit organizations can improve health outcomes and overall quality of life for those who might otherwise be deprived of essential care.

Despite these potential benefits, organizations face significant challenges in expanding healthcare coverage, primarily due to budget limitations that restrict the number of facilities they can establish and maintain. To address these challenges, the Maximum Covering Location Problem (MCLP) provides a framework for optimizing facility placement, aiming to maximize the number of demand locations covered by a limited number of facilities within a specified service distance. Strategic placement of healthcare facilities allows organizations to expand their reach of services, especially in underserved areas where healthcare access directly impacts health outcomes.

The MCLP has been extensively studied in the literature. Researchers such as Antonissen and Theulen [5] have applied exact solvers and heuristics to real-world scenarios, such as the placement of stroke centers in Timor-Leste and Vietnam. However, the classical MCLP assumes that demand is fully covered if a facility is located within the service distance, neglecting practical factors such as facility capacity.

In healthcare, facilities have limited capacities, making it essential to properly manage patient loads to ensure quality service. This capacity constraint often leads to more realistic solutions than classical MCLP models, particularly in healthcare settings. To address this limitation, the Capacitated Maximum Covering Location Problem (CMCLP) integrates capacity constraints, providing a more realistic approach to resource allocation by ensuring that facilities remain within their operational limits.

However, incorporating capacity alone without so-called Closest Assignment Constraints (CAC) can still result in impractical outcomes. Demand locations may be assigned to distant facilities if closer facilities are at capacity, leading to inefficient and often unexplainable outcomes, particularly for those in need. To achieve more explainable and realistic solutions, it is essential to consider both capacity and closest assignment constraints, ensuring that demand locations are served by the nearest available facility.

This leads to the Capacitated Maximum Covering Location Problem with Closest Assignment Constraints (CMCLP-CAC). In this model, the objective is to maximize coverage while respecting budget constraints, service distances, and facility capacities, while ensuring demand is assigned to the nearest available facility. The inclusion of these additional constraints significantly increases the complexity of the problem.

Unlike the simpler MCLP, which uses single-indexed decision variables to indicate whether a facility is open or if a demand location is covered, the CMCLP-CAC employs double-indexed decision variables. These variables explicitly assign each demand location to a specific facility, significantly increasing the complexity of finding optimal solutions and raising the computational requirements necessary to solve the problem effectively.

Model Formulation

Sets

- N: Set of all demand locations.
- M: Set of all potential facility locations.

Parameters

- a_i: Demand at demand location i, $i \in N$.
- d_{ij}: Travel distance from demand location i to potential facility location j, $i \in N$, $j \in M$.
- K: Capacity of each potential facility.
- S: Service distance (the maximum distance within which a facility can serve a demand location).
- p: Maximum number of facilities that can be opened.
- $C_i = \{j \in M \mid d_{ij} \leq S\}$: Set of potential facility locations that can cover demand location i.
- $W_{ij} = \{k \in M \mid d_{ik} \geq d_{ij}\}$: Set of potential facility locations further from demand location i than facility j.

Decision Variables

- x_j: Binary variable, equal to 1 if a facility is established at location j, 0 otherwise ($j \in M$).
- y_{ij}: Binary variable, equal to 1 if demand location i is served by facility j, 0 otherwise ($i \in N$, $j \in M$).

Mathematical Model Formulation

$$\text{Maximize} \quad \sum_{i \in N} \sum_{j \in M} a_i y_{ij} \tag{1}$$

$$\text{Subject to} \quad \sum_{j \in M} x_j \leq p \tag{1}$$

$$y_{ij} = 0 \quad \forall i \in N,\ j \notin C_i \tag{2}$$

$$\sum_{j \in M} y_{ij} \leq 1 \quad \forall i \in N \tag{3}$$

$$\sum_{i \in N} a_i y_{ij} \leq K x_j \quad \forall j \in M \tag{4}$$

$$\sum_{k \in W_{ij}} y_{ik} \leq 1 - x_j \quad \forall i \in N,\ \forall j \in M \tag{5}$$

$$x_j \in \{0, 1\} \quad \forall j \in M \tag{6}$$

$$y_{ij} \in \{0, 1\} \quad \forall i \in N,\ \forall j \in M \tag{7}$$

Explanation of Constraints
The model incorporates the following constraints to ensure realistic and practical outcomes:

- Budget Constraint (1): Limits the number of facilities that can be opened to p, reflecting budget limitations.
- Coverage Constraint (2): Ensures that a demand location i can only be covered by a facility j if it is within the specified service distance S.
- Assignment Constraint (3): Guarantees that each demand location is served by at most one facility.
- Capacity Constraint (4): Imposes capacity limits on each facility, ensuring that no facility serves more demand than its capacity K. If a facility is not opened ($x_j = 0$), then no demand can be allocated to it.
- Closest Assignment Constraint (5): Ensures that each demand location is served by the closest available facility and that no demand is allocated to a facility further than the nearest available one.
- Binary Restrictions (6) and (7): Specify that the facilities are either opened or not, and that demand is either fully assigned or not assigned at all.

This formulation presents the objective and constraints of the CMCLP-CAC model. With the mathematical framework established, the focus now shifts towards addressing the practical challenges involved in solving this problem.

The CMCLP-CAC poses significant challenges due to its complexity, particularly when addressing large-scale problems. This complexity arises from the need to optimize both facility placement and coverage while adhering to capacity and closest assignment constraints. Given these challenges, the primary aim of this thesis is to develop two approaches: a Mixed Integer Programming (MIP) solver and a heuristic method. The MIP solver seeks to find optimal solutions, but is often constrained by problem size and computational resources. In contrast, the heuristic approach is designed to provide near-optimal solutions efficiently, requiring less memory and computation time, making it a more practical solution for larger problem instances.

Moreover, the developed methods are designed to be generalizable and applicable across various real-world scenarios beyond the healthcare sector. Although this study focuses on healthcare facilities, such as stroke centers, the methods can be extended to other areas, including education, clean water, and food centers—all of which require careful consideration of capacity and proximity constraints.

This research is conducted in collaboration with Analytics for a Better World, an organization dedicated to leveraging data and analytics to advance the Sustainable Development Goals (SDGs). By applying advanced analytical techniques to real-world challenges, this partnership aims to create practical tools that foster meaningful societal impact.

The structure of this thesis is as follows. Section 2 reviews the relevant literature on Capacitated Maximum Covering Location Problems and Closest Assignment Constraints. Section 3 discusses the implementation of the MIP solver, highlighting specific adaptations aimed at improving performance. Section 4 describes the heuristic approach, detailing its key phases: construction, local search, and path relinking. In Section 5, the performance of both the MIP solver and the heuristic is analyzed using real-world data from Timor-Leste. Section 7 concludes with a summary of key findings and suggestions for future research. Section 6 highlights the importance and the potential of the research. Finally, the thesis concludes with references and appendices.

2 Literature Review

The Capacitated Maximum Covering Location with Closest Assignment Constraints (CMCLP-CAC) problem, to the best of our knowledge, has not been explicitly addressed in the literature. Therefore, to lay the groundwork for this study, the literature review focuses on two related problems: the Capacitated Maximum Covering Location Problem (CMCLP) and Closest Assignment Constraints (CAC). By examining these problems individually, we try to build a solid foundation to understand and tackle the combined challenges of the CMCLP-CAC.

Literature Review on Capacitated Maximum Covering Location Problem
The Capacitated Maximum Covering Location Problem (CMCLP) was first introduced by Chung, Schilling, and Carbone in 1983. Their work, titled "The Capacitated Maximal Covering Problem: A Heuristic" [10], presented a heuristic to solve the CMCLP using binary assignment variables. However, the computational tests were limited to small instances, involving only 12 demand locations, 30 potential facility locations, and 4 or 8 facilities to be opened.

In 1985, Somogyi and Church expanded on this foundation by introducing several heuristics and an optimal procedure for solving the CMCLP. Their approach allowed for partial assignments and the allocation of multiple facilities at a single site. Documented in "Solving the Capacitated Maximal Covering Location Problem - Draft Manuscript" [36], their contributions significantly enhanced the computational power available for tackling larger problems, enabling the analysis of instances with up to 79 demand locations and the opening of 30 facilities.

Further developments came from Current and Storbeck in 1988 with their work "Capacitated Covering Models" [13]. They reformulated the CMCLP as a Generalized Assignment Problem (GAP), thus allowing the use of heuristics developed for GAP. Although computational results were not presented, they established that CMCLP, which is equivalent to GAP, is also NP-hard.

Pirkul and Schilling made further contributions in 1991 with their work, "The Maximum Covering Location Problem with Capacities on Total Workload" [26]. They proposed two versions of the capacitated MCLP. The first version, similar to previous models, considered only the demand covered within the maximum service distance. The second version included all demands in the workload of a facility, regardless of coverage distance. This addressed the practical need for public facilities to serve all within their jurisdiction, not just those within a fixed distance. They utilized Lagrangian Relaxation to balance coverage while minimizing travel distance for uncovered demand, presenting computational results for up to 200 demand locations, 30 potential facility locations, and 15 facilities.

In 1996, Haghani refined these models further by incorporating additional constraints in his work, "Capacitated Maximum Covering Location Models: Formulations and Solution Procedures" [20]. He incorporated a minimum demand requirement for each facility and proposed two heuristics: the Greedy Adding Heuristic, which iteratively selects the facility providing the most additional coverage, and the Lagrangian Relaxation Heuristic, which uses Lagrangian multipliers to relax capacity constraints and solve subproblems to obtain bounds. The Greedy Adding Heuristic consistently performed well for small instances, while the Lagrangian Relaxation Heuristic showed performance variations, with gaps between lower and upper bounds ranging from 0% to 25%.

After a period of limited progress, more recent research has shifted towards applying metaheuristic approaches to address the CMCLP. Elkady and Abdelsalam (2016) employed Particle Swarm Optimization (PSO) to tackle CMCLP [15]. While PSO demonstrated some effectiveness for medium-sized problems, it was still constrained by scalability, particularly for instances with fewer than 1,000 demand locations and facilities.

Recent works by Atta, Mahapatra, and Mukhopadhyay [6], Vatsa and Jayaswal [38], and Aider, Dey, and Hifi [4] have introduced new complexities to the CMCLP, such as fuzzy coverage areas and server uncertainty. However, these studies have focused mainly on these additional complexities rather than advancing the handling of large-scale capacitated maximum covering location problems [37].

Literature Review on Closest Assignment Constraints

Closest Assignment Constraints (CAC) are critical in models that require clients to be assigned to their nearest facility, ensuring proximity-based allocation. Over the years, numerous CAC models have been proposed in the domain of Integer Programming for location-allocation problems, with new formulations often introduced without reference to existing inequalities.

The concept of CAC in an Integer Programming framework was first introduced by Rojeski and ReVelle in 1970 through their work on "Central Facilities Location under an Investment Constraint" [35]. They introduced the following constraints for a budget-constrained median problem:

$$y_{ij} + \sum_{a:d_{ia}<d_{ij}} x_a \geq x_j \qquad \forall i, j \in A \qquad (\mathcal{RR})$$

In simple terms, this constraint ensures that if a facility is opened at location j ($x_j = 1$) and no other facilities are closer to demand location i than j ($x_a = 0$ for all a such that $d_{ia} < d_{ij}$), then demand location i must be assigned to facility j. However, this formulation is unsuitable for our problem context because not all demand locations can be covered due to capacity limitations.

In 1975, Wagner and Falkson introduced an alternative set of constraints in their paper "The Optimal Nodal Location of Public Facilities with Price-Sensitive Demand" [39] to enforce the closest assignment in an integer-linear model:

$$\sum_{a:d_{ia}>d_{ij}} y_{ia} + x_j \leq 1 \qquad \forall i, j \in A \qquad (\mathcal{WF})$$

This inequality states that if a facility is opened at location j ($x_j = 1$), then demand location i cannot be assigned to any facility further away than j ($\sum_{a:d_{ia}>d_{ij}} y_{ia} = 0$).

In 1976, Church and Cohon proposed a different approach to enforce the closest assignment through linear constraints in their work "Multiobjective Location Analysis of Regional Energy Facility Siting Problems" [11]:

$$\sum_{a:d_{ia}\leq d_{ij}} y_{ia} \geq x_j \qquad \forall i, j \in A \qquad (\mathcal{CC})$$

This formulation ensures that if a facility is opened at location j ($x_j = 1$), then demand location i is allocated to at least one facility a, provided that a is within the distance threshold ($d_{ia} \leq d_{ij}$). However, this constraint does not apply to our situation because, given the capacity limitations, opening a facility does not ensure that a demand location will be entirely served by that facility or any closer facilities.

In 1987, Dobson and Karmarkar introduced a more granular CAC in their paper "Competitive Location on a Network" [14]:

$$y_{ij} + x_a \leq 1 \qquad \forall i, j, a \in A : d_{ia} < d_{ij} \qquad (\mathcal{DK})$$

This constraint implies that if a facility is opened at location a ($x_a = 1$), demand location i cannot be assigned to any facility further away than a. Although similar to Wagner and Falkson's inequality, this version is disaggregated, resulting in cubic complexity.

In 2007, Cánovas et al. proposed an enhancement to the CAC formulations in their work "A Strengthened Formulation for the Simple Plant Location Problem with Order" [8]. They extended the inequalities to achieve a tighter formulation by introducing the following constraint:

$$\sum_{a:d_{ia}>d_{ij}} y_{ia} + \sum_{a:d_{ia}\leq d_{ij}, d_{ka}>d_{kj}} y_{ka} + x_j \leq 1 \qquad \forall i,j,k \in A \qquad (\mathcal{CGLM})$$

While computational studies demonstrated the effectiveness of this approach on small problem instances, the increased cubic complexity could become a drawback as the problem grows.

To better understand the functioning of this constraint, refer to Figure 1. In this figure, demand and facility locations are represented as points in a two-dimensional space, with distances measured using Euclidean metrics. The dashed circles around demand locations k and i illustrate the distances d_{kj} and d_{ij}, respectively.

If $x_j = 1$, meaning facility j is open, the closest assignment constraint ensures that demand locations i and k cannot be assigned to any facility further away than j (i.e., $y_{ia} = y_{ka} = 0$ for any facility a that is more distant than j). Conversely, if $x_j = 0$, meaning facility j is not open, demand location k may be assigned to a facility further than facility j (i.e., $y_{ka} = 1$), provided that this facility is closer to demand location i than facility j ($d_{ia} \leq d_{ij}$). In such a situation, assigning demand location i to a facility more distant than facility j becomes invalid. Alternatively, if demand location i is assigned to a facility beyond facility j ($y_{ia} = 1$), then demand location k cannot be assigned to a facility closer to demand location i than facility j.

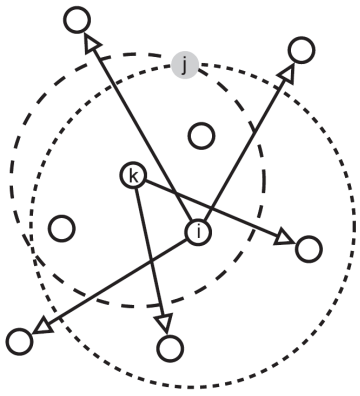


Figure 1: Illustration of how the *CGLM*-constraints work [16].

Table 1 summarizes the CAC formulations presented in the literature. Although many CAC formulations have been introduced, a comprehensive evaluation of all these formulations using our specific dataset falls beyond the scope of this thesis.

Year	**Name**	**Constraints**	**Cardinality**
1970	$\mathcal{RR}$	$y_{ij} + \sum_{a:d_{ia}<d_{ij}} x_a \geq x_j \, \forall i,j \in A$	$\mathcal{O}(n^2)$
1975	$\mathcal{WF}$	$\sum_{a:d_{ia}>d_{ij}} y_{ia} + x_j \leq 1 \, \forall i,j \in A$	$\mathcal{O}(n^2)$
1976	$\mathcal{CC}$	$\sum_{a:d_{ia}\leq d_{ij}} y_{ia} \geq x_j \, \forall i,j \in A$	$\mathcal{O}(n^2)$
1987	$\mathcal{DK}$	$y_{ij} + x_a \leq 1 \, \forall i,j,a \in A : d_{ia} < d_{ij}$	$\mathcal{O}(n^3)$
2007	$\mathcal{CGLM}$	$\sum_{a:d_{ia}>d_{ij}} y_{ia} + \sum_{a:d_{ia}\leq d_{ij}, d_{ka}>d_{kj}} y_{ka} + x_j \leq 1 \, \forall i,j,k \in A$	$\mathcal{O}(n^3)$
2007	$\mathcal{BLMN}$	$p \sum_{a:d_{ia}>d_{ij}} y_{ia} \leq \sum_{a:d_{ia}>d_{ij}} x_a \, \forall i,j \in A$	$\mathcal{O}(n^2)$
2009	$\mathcal{BDTW}$	$\sum_{a\in A} d_{ia} y_{ia} + (M - d_{ij}) x_j \leq M \, \forall i,j \in A$	$\mathcal{O}(n^2)$
2011	$\mathcal{M}'$	$\lvert\theta_{ij}\rvert \sum_{a:d_{ia}\geq d_{ij}} y_{ia} + \sum_{a:d_{ia}<d_{ij}} x_a \leq \lvert\theta_{ij}\rvert \, \forall i,j \in A$	$\mathcal{O}(n^2)$
2011	$\mathcal{M}$	$q_{ij} \sum_{a:d_{ia}\geq d_{ij}} y_{ia} + \sum_{a:d_{ia}<d_{ij}} x_a \leq q_{ij} \, \forall i,j \in A$	$\mathcal{O}(n^2)$

Table 1: Different CAC from the literature, summarized ([16])

Selected approach

The approach selected for the Mixed Integer Programming (MIP) solver involves selecting which closest assignment constraint to implement. Initially, a constraint with low cardinality and good interpretability is prioritized to avoid introducing unnecessary complexity, as the goal is to extend both the MIP solver and heuristic to handle large-scale problems. For this reason, the constraint developed by Wagner and Falkson [39] will be implemented, as it aligns with other constraints and offers both interpretability and low cardinality. However, the constraints proposed by Cánovas et al. [8] have shown efficiency in smaller instances, despite their higher cardinality. As such, this research will explore both sets of constraints in the MIP solver and, after testing, the most effective constraint will be chosen for further use.

The approach selected for the heuristic in this research differs somewhat from the existing literature. Due to the limited progress in solving large instances of the Capacitated Maximum Covering Location Problem, this work explores an alternative method for tackling such large-scale problems. A significant breakthrough has been made in the Maximum Covering Location Problem, where Theulen introduced a heuristic method that integrates the Greedy Randomized Adaptive Search Procedure with path-relinking [5]. This heuristic was successfully applied to instances as large as 300,000 demand locations by 350,000 potential facility locations, representing a significant leap from the smaller instances typically addressed in earlier studies. The goal of this research is to adapt the heuristic to incorporate capacity and closest assignment constraints, aiming to ensure its effectiveness in handling the Capacitated Maximum Covering Location with Closest Assignment Constraints and its performance in large-scale applications.

A brief literature review of previous studies that applied the Greedy Randomized Adaptive Search Procedure (GRASP) with path-relinking to the Maximal Covering Location Problem (MCLP) is provided below.

In 1998, Resende published a foundational paper on applying GRASP to the Maximal Covering Location Problem [33]. GRASP is an iterative approach, in which each iteration consists of a construction phase, followed by a local search phase [1]. During the construction phase, an adaptive greedy algorithm is used to build a solution. Specifically, the greedy function calculates the additional number of demand locations that would be covered if a particular facility were selected. The facilities are then ranked based on this metric, and one is randomly chosen from the top $\alpha\%$ of potential facility locations. After selecting a facility, the greedy metric is updated to account for the coverage that has already been achieved, and this process continues until p facilities are chosen. Following the construction phase, a local search is performed to refine the solution until no better solutions can be found in the current neighborhood, resulting in a local optimum. Remarkably, for its time, the GRASP approach was capable of handling large-scale instances of the MCLP, including problems involving 10,000 demand locations and 1,000 potential facility locations.

Máximo et al. introduced a major improvement in the scalability of the location problem in 2017 [23]. Their method expanded on Resende's work and addressed a significant shortcoming of GRASP: the lack of iteration interdependence. In GRASP, valuable information gathered during the construction and local search phases is typically discarded after each iteration, leading to independent cycles. Máximo et al. addressed this limitation by implementing a mechanism that retains and reuses information, thereby creating interdependent iterations. Specifically, they used an artificial neural network to guide the construction of solutions in subsequent iterations. Each solution, once refined through local search, was used to train the neural network, which, in turn, improved the quality of future solutions. This adaptive memory acted as a learning mechanism, allowing the algorithm to improve over time. The authors named their approach Intelligent-Guided Adaptive Search (IGAS) and demonstrated that it could consistently find solutions as good as or better than those found by GRASP, but in less time.

In 2019, Máximo et al. expanded their research further by incorporating path-relinking into both IGAS and GRASP [22]. Path-relinking is a technique that explores paths connecting two local optima and is typically executed at the end of each GRASP iteration. This method introduces an extra layer of interdependence between iterations, but in a different manner from the adaptive memory used in IGAS. While path-relinking enhances the solution by exploring alternative routes after the local search phase, the adaptive memory in IGAS influences the initial solution construction. Máximo et al. tested the extended versions of IGAS and GRASP on the same benchmark instances used in their previous study. The results showed that path-relinking significantly improved both heuristics. The improvement was more pronounced in GRASP, which initially lacked inter-iteration memory, compared to IGAS, which already benefited from neural network-based learning.

In both studies, Máximo et al. tested instances involving up to 8,000 demand locations and 8,000 potential facility locations, demonstrating the robustness and scalability of their proposed methods.

In 2022, Theulen applied a GRASP heuristic to a large-scale MCLP focused on the allocation of stroke treatment centers in Vietnam [5]. The problem involved 300,000 demand locations and 350,000 potential facility locations, far exceeding the scale of previous studies. To address this complexity, Theulen adapted the swap-based local search procedure from the p-median problem to the MCLP setting and utilized a sparse matrix structure to mitigate memory constraints. The heuristic combined randomized greedy construction, local search, and back- and forward path-relinking as intensification strategies. The approach provided high-quality solutions, achieving a deviation of just 0.025% from the optimum on test instances, while practical solutions for the largest cases were obtained within two hours. This work underscores the effectiveness of GRASP in solving large-scale real-world MCLP applications.

3 Implementation of the MIP Solver

This section is divided into two main parts. First, we introduce the key components and notation required to implement the Capacitated Maximum Covering Location Problem with Closest Assignment Constraint (CMCLP-CAC), establishing a foundation for both the MIP solver and heuristic approaches. Subsequently, we propose refinements and efficiency improvements to the initial CMCLP-CAC formulation, focusing on improving computational performance, particularly in cases involving a large number of demand locations and potential facility locations.

3.1 Key Elements and Notation

The CMCLP-CAC formulation incorporates multiple sets and parameters, as well as additional variables necessary for efficient implementation. These elements will be explained in more detail through a small illustrative example, which highlights the roles of each set, variable, and parameter, offering a clear foundation for understanding. This example will be revisited throughout the thesis to demonstrate the heuristic in detail and to facilitate the explanation of more complex concepts as they are developed.

The sets used in the model include the demand locations, denoted by N, and the potential facility locations, denoted by M. The set N represents the demand locations indexed from 0 to $|N| - 1$, where $|N|$ indicates the total number of demand locations. Likewise, M represents the potential facility locations, indexed from 0 to $|M| - 1$, where $|M|$ is the total number of potential facility locations.

In the small example, the sets are defined as follows:

$$N = \{0, 1, 2, 3, 4, 5, 6, 7\}, \qquad M = \{0, 1, 2, 3\}.$$

Here, the set N consists of eight demand locations, indexed from 0 to 7, and the set M consists of four potential facility locations, indexed from 0 to 3.

Next, the parameters used in the model are defined. Each demand location has an associated demand, represented by an array a, where each element reflects the population at a specific demand location. This structure allows for efficient lookup of demand information for each demand location.

In the small example, every demand location has an identical demand value:

$$a = [1, 1, 1, 1, 1, 1, 1, 1].$$

For practical implementation in the MIP solver, this array is converted into a dictionary, using the demand location index as the key and the corresponding demand as the value. This conversion is purely for technical convenience, without altering the underlying functionality.

In the small example, this conversion results in:

$$a = \{0 : 1, 1 : 1, 2 : 1, 3 : 1, 4 : 1, 5 : 1, 6 : 1, 7 : 1\}.$$

The distance between demand locations and potential facilities plays a crucial role in the model. A facility is eligible to serve a demand location if the distance between them is less than or equal to a predefined service distance, denoted as S.

The simplest way to represent this relationship is through the matrix B, where rows correspond to potential facility locations, and columns represent demand locations. In this matrix, a value of 1 indicates a feasible facility-demand pair.

In the small example, there are four potential facility locations and eight demand locations, yielding a B-matrix of size 4×8:

$$B = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

Here, facility 0 can serve demand locations 0 and 1, while facility 3 can serve demand locations 5, 6, and 7. This representation helps to determine which facilities can serve which demand locations based on the service distance S.

To provide additional clarity on the small example, Figure 2 presents a visualization of the scenario. In this figure, the triangles denote the potential facility locations, the orange dots represent demand locations, and the circles surrounding the triangles indicate the service distance for each potential facility location.

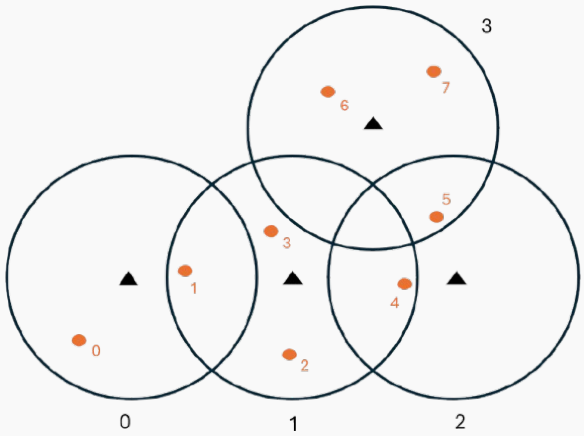


Figure 2: Vizualization of the small example

However, this type of matrix does not indicate which potential facility is closest to a particular demand location - an important detail for applying the closest assignment constraints. To capture this information, the actual distance corresponding to each feasible pair must also be recorded. Consequently, this approach can become memory-intensive due to the additional data storage requirements.

To optimize storage and improve efficiency, a sparse matrix representation is used, focusing only on the relevant data. A dictionary *col* is defined where each key represents a demand location, and the corresponding value is a list of facility indices that are located within the service distance of the corresponding demand location. The facilities in the list are ordered by proximity, with the closest facility appearing first.

In the small example, the dictionary *col* is as follows:

$$col = \{0 : [0], 1 : [0, 1], 2 : [1], 3 : [1], 4 : [2, 1], 5 : [2, 3], 6 : [3], 7 : [3]\}.$$

Here, demand location 0 has only facility 0 within its service distance, while demand location 2 has only facility 1 within its service distance. Demand location 1 has both facilities 0 and 1 within its service distance, with facility 0 being the nearest. Similarly, demand location 4 has facilities 1 and 2 within its service distance, with facility 2 being the closest option.

Similarly, the dictionary *row* is introduced, where each key represents a facility index, and the corresponding value is a list of demand locations that fall within the service distance of that facility. These lists do not need to be sorted by distance.

In the small example, the resulting *row* is as follows:

$$row = \{0 : [0, 1], 1 : [1, 2, 3, 4], 2 : [4, 5], 3 : [5, 6, 7]\}.$$

This representation indicates that facility 0 can serve demand locations 0 and 1, while facility 1 can serve demand locations 1, 2, 3, and 4.

As mentioned above, when determining the optimal set of open facilities, the lists in *row* do not need to be sorted by proximity. During a post-optimization or refinement phase, prioritizing closer demand locations may be more convenient if a facility's total capacity is exceeded. However, whether or not these lists are sorted, the objective value of the original optimization problem remains unchanged.

In addition to distance constraints, each facility is subject to a capacity constraint. The capacity, denoted by K, is an integer representing the maximum number of individuals that a facility can serve.

In the small example, the capacity (K) is equal to 3.

Furthermore, due to budgetary and resource limitations, the number of facilities that can be opened is restricted. To account for this, p is defined as the maximum number of facilities that may be opened, where p is an integer.

3.2 Model Refinements and Efficiency Improvements

This subsection introduces several key concepts aimed at improving efficiency, particularly in scenarios involving a large number of demand locations and facility locations. Detailed results on the effectiveness of these concepts, specifically for the Timor-Leste dataset, are discussed in Section 5.

Reducing Decision Variables
In the basic formulation, an allocation decision variable y is introduced for every possible combination of demand and potential facility locations. Additionally, a constraint is included to ensure that if the distance between a demand and the facility location exceeds the service distance, the corresponding allocation variable is set to 0. However, these variables are redundant and do not add value to the model.

To address this, allocation decision variables are introduced only for demand and facility location pairs where the distance is within or equal to the service distance. In other words, the decision variables are defined exclusively for the combinations included in the previously defined dictionary *row* (or *col*).

In the small example, the original formulation required 32 decision variables (8 demand locations $\times$ 4 facility locations). In contrast, the new approach reduces this number to only 11 decision variables.

This demonstrates a significant improvement even for a small example, and for larger real-world problems, the reduction in the number of variables will be even more pronounced.

With this change, the constraint '$y_{ij} = 0 \quad \forall i \in N,\ j \notin C_i$' can be eliminated, as the redundant variables have already been excluded from the model.

With this new formulation, the mathematical model is now presented as follows. This approach not only reduces the number of decision variables but also minimizes the number of constraints:

$$
\begin{aligned}
\text{Maximize} \quad & \sum_{i \in col} \sum_{j \in col[i]} a_i y_{ij} \\
\text{Subject to} \quad & \sum_{j \in row} x_j \leq p \\
& \sum_{j \in col[i]} y_{ij} \leq 1 && \forall i \in col \\
& \sum_{i \in row[j]} a_i y_{ij} \leq K x_j && \forall j \in row \\
& \sum_{k \in S_{ij}} y_{ik} \leq 1 - x_j && \forall i \in col, \forall j \in col[i] \\
& x_j \in \{0,1\} && \forall j \in row \\
& y_{ij} \in \{0,1\} && \forall i \in col, \forall j \in col[i]
\end{aligned}
$$

This refined formulation reduces the number of decision variables and constraints, enhancing the efficiency of the model. By excluding redundant variables, the computational complexity is reduced, making the model more scalable for larger real-world problems.

Converting Allocation Decision Variables y from Binary to Fractional
In the literature, the Capacitated Maximum Covering Location Problem is commonly modeled using binary allocation decision variables y, reflecting many real-world scenarios where individuals cannot be divided. However, in practice, data are often imperfect, and the actual demand at specific locations may differ from the estimates used in the model. Since the solver operates on approximate data, allowing allocation decision variables to take fractional values can still produce practical and realistic solutions.

Allowing y to take fractional values transforms the model from an Integer Program (IP) into a Mixed Integer Program (MIP), reducing computational complexity and simplifying the problem for the solver. MIPs are generally more efficient to solve than IPs because they utilize a continuous decision space rather than being confined to integers. This adjustment improves the branch-and-bound process by reducing the number of branches required, effectively shrinking the search space compared to strictly binary decisions. Consequently, the solver can reach convergence faster, which is particularly advantageous for large-scale, complex optimization problems [7] [19].

Therefore, the last constraint in the model is updated as follows:

$$y_{ij} \geq 0, \qquad \forall i \in col, \forall j \in col[i].$$

There is no need for an additional constraint to limit the allocation decision variables to a maximum of 1 since the existing constraint '$\sum_{j \in col[i]} y_{ij} \leq 1 \quad \forall i \in col$' already ensures that the total of decision variables across all potential facility locations for each demand location does not exceed 1.

A small remark: Converting the allocation to fractional does not imply that a demand location can be split between two facilities. The closest assignment constraints ensure that each demand location can only be assigned to the nearest open facility.

Preprosessing Index Sets
As discussed in 'Reducing Decision Variables', the decision variable y should only be defined for each demand-facility pair represented in the dictionary row. A simple but inefficient way to achieve this is by iterating through the dictionary keys and adding a decision variable y for each demand-facility pair individually using the $addVar$ function. This approach leads to substantial repeated computational overhead, especially for large datasets, as each variable is created in a separate operation. Similar inefficiencies occur when defining the objective function and implementing constraints, such as the closest assignment constraint.

A more efficient approach is to preprocess these demand-facility pairs in advance and add them collectively using the $addVars$ function. This significantly reduces the number of function calls, takes advantage of batch processing, and greatly improves computational efficiency. This method not only makes the code more concise but also enhances scalability, making it much more suitable for scenarios involving a large number of demand locations and potential facility locations.

For example, the indices can be preprocessed by defining *idx_y = [(i, j) for i in col for j in col[i]]*. Subsequently, decision variables can be added in bulk using *y = m.addVars(idx_y, vtype = gp.grb.continuous, name = "y")*. This approach enables batch processing, which is more efficient.

This preprocessing strategy is similarly applied to the decision variable x and the objective function, ensuring a streamlined and consistent approach across all components of the model.

Defining Weak and Strong Versions of the Model
The solver's efficiency can often be enhanced by refining the model formulation, particularly by strengthening the constraints. While it may initially seem that an optimization problem should involve fewer constraints to be efficient, adding specific, well-targeted constraints can introduce 'stronger cuts' that significantly improve performance. These cuts help to reduce the solution space more effectively, thereby speeding up the solution process [27]. An example of such a targeted constraint is provided below, which strengthens the capacity constraints in the model:

$$y_{ij} \leq x_j \qquad \forall j \in col, \quad \forall i \in col[i].$$

This constraint ensures that a demand location can only be assigned to a facility if that facility is open. By incorporating these additional constraints, the resulting formulation is referred to as the 'strong' version of the model.

When introducing the strong version of the model, the original model without the strengthened constraint will be referred to as the 'weak' version.

The effectiveness of the strong model compared to the weak model depends on the trade-off between adding extra constraints and the benefits of having stronger cuts in the optimization process. In the strong model, these additional constraints need to be loaded in, which can increase the computational burden. Therefore, for smaller models, the weak version may perform better, as the time needed to load and manage these extra constraints might not be justified by the gains in runtime efficiency.

Comparing Closest Assignment Constraints
As discussed in the literature review, multiple closest assignment constraints are available. To determine the most suitable option for this dataset, two versions will be implemented: the basic Wagner and Falkson version [39], and the enhanced Cánovas version [8].

The enhanced version of Cánovas is known to provide stronger cuts, as indicated in the literature. However, it also comes with significantly higher computational complexity compared to the basic version, which can become challenging when applied to large-scale datasets. Specifically, the enhanced version introduces additional constraints, which increase the problem size and computational demands. Moreover, it requires the introduction of an additional set, $G_{ikj} = \{a \in col[i] \cap col[k] \mid d_{ia} \leq d_{ij}, d_{ka} > d_{kj}\}$.

This results in increased memory usage and more complex calculations, potentially causing slower convergence and longer processing times, particularly when dealing with larger datasets.

Lazy Constraints on Closest Assignment Constraints
When developing the optimization model, a significant portion of the constraints involve the closest assignment constraints. These constraints must be added for every combination of demand locations and potential facility locations within or equal to the service distance. This process is time-consuming and can often result in redundant constraints that diminish the solver's efficiency.

To address this, lazy constraints can be utilized. Initially, the model is solved without incorporating the closest assignment constraints, often resulting in a simplified problem that is easier to solve [25]. Once a solution is obtained, the solver checks for any violations of the closest assignment constraints. Only the violated constraints are then added to the model—specifically in the relevant branches during the Branch and Bound process—and the problem is resolved. This iterative approach continues until all closest assignment constraints are satisfied, thereby improving solver performance by focusing solely on the necessary constraints.

This approach is particularly effective when only a relatively small number of closest assignment constraints are binding. However, if the majority of the constraints are binding, the iterative process may not offer significant reductions in computation time, and including all constraints from the beginning might be more efficient.

4 Implementation of the Heuristic

As outlined in the selected approach within the literature review, the heuristic used in this thesis is a variant of GRASP with path-relinking, as proposed by Theulen [5]. It has been adapted to incorporate both capacity constraints and the closest assignment constraints. This section presents a detailed explanation of the heuristic, outlining its key phases: the construction phase, local search (using either first or best improvement), and path-relinking.

Before diving into the details, here is a brief overview of the phases:

- *Construction Phase*: This phase involves building an initial feasible solution by iteratively selecting and opening facilities using a randomized greedy approach. The goal is to create solutions that provide good coverage while ensuring sufficient diversity for effective optimization in the subsequent phases.

Once the initial solution has been constructed, the next step is to perform a local search. This phase can follow either a first improvement or a best improvement strategy:

- *First Improvement*: This strategy involves making localized changes by examining potential facility swaps within the neighborhood of the currently open facilities. Whenever a swap results in improved coverage, it is executed immediately without performing an exhaustive search.
- *Best Improvement*: This phase involves a more exhaustive search, evaluating all possible facility swaps both within and beyond the neighborhood of the opened facilities. The goal is to identify the swap that delivers the greatest improvement in coverage.

After completing each local search phase, the path relinking phase begins:

- *Path-relinking*: This phase involves exploring pathways between the solution obtained after local search and high-quality solutions in an elite pool, aiming to escape local optima and uncover more promising regions in the solution space.

After each path-relinking phase, the best solutions are saved in an elite pool that holds up to six solutions. A new solution is added if it offers better coverage or if they are sufficiently distinct from those already in the pool. This approach ensures a balance between quality and diversity, keeping the optimization process dynamic and avoiding stagnation.

In the following subsections, we will first introduce the key notation required for the heuristic. Subsequently, a detailed technical explanation will be provided for the implementation of each phase of the heuristic.

4.1 Basics Elements

This section expands on the concepts introduced in Section 3.1, where the key elements and notation were defined. To effectively implement the heuristic and maintain clarity throughout the subsequent steps, we will introduce some additional notation and variables.

The first variable we introduce is the theoretical coverage array, also used by Theulen [5]. In this heuristic, the array is referred to as *theoretical_cov*. It keeps track of the number of facilities located within the service distance of each demand location. Specifically, each index in the array corresponds to a demand location, and the value of that index indicates the number of facilities within the service distance for that specific demand location.

To illustrate this, reconsider the small example, where the dictionary *row* is defined as:

$$row = \{0 : [0, 1], 1 : [1, 2, 3, 4], 2 : [4, 5], 3 : [5, 6, 7]\}.$$

Assuming that facilities 1 and 2 are opened (see Figure 3 for visualization), the resulting *theoretical_cov* array is:

$$theoretical\_cov = [0, 1, 1, 1, 2, 1, 0, 0].$$

In this case, demand locations 1, 2, 3, and 5 each have one facility within their service distance, resulting in a value of 1. Demand location 4 has both facilities 1 and 2 within its service distance, giving it a value of 2. Meanwhile, demand locations 0, 6, and 7 are not within the service distance of any open facility, resulting in values of 0.

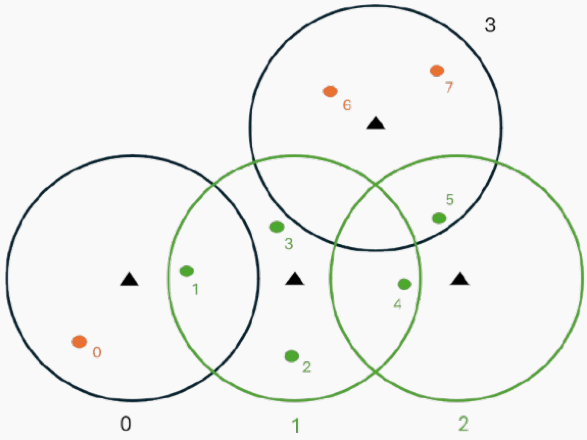


Figure 3: Visualization of the small example with facilities 1 and 2 open

The next variable, denoted as *nearest_opened_fac*, is introduced to identify the closest opened facility for each demand location, which is essential for enforcing the closest assignment constraint. Under this constraint, each demand location can only be assigned to the nearest open facility within its service distance. If no facility is opened within the service distance of a demand location, the corresponding value in *nearest_opened_fac* is set to -1.

The dictionary *col* plays an important role here, recall that each key represents a demand location, and the corresponding value is an ordered list of facilities within its service distance.

Consider the dictionary *col* from the small example:

$$col = \{0 : [0], 1 : [0, 1], 2 : [1], 3 : [1], 4 : [2, 1], 5 : [2, 3], 6 : [3], 7 : [3]\}.$$

Assuming once again that facilities 1 and 2 are opened, the resulting *nearest_opened_fac* array is:

$$nearest\_opened\_fac = [-1, 1, 1, 1, 2, 2, -1, -1].$$

In this case, demand location 4 has a *theoretical_cov* value greater than 1, indicating that multiple facilities are within its service distance. By consulting the *col* dictionary, we find that the nearest facility to demand location 4 is facility 2.

The final variable called *actual_cov*, represents the actual coverage achieved in the model. It accounts for both capacity constraints and the requirement that demand is only allowed to be assigned to the nearest open facility. Essentially, *actual_cov* reflects how demand is truly covered by the open facilities, going beyond the theoretical coverage potential.

To construct *actual_cov*, begin by calculating the total demand assigned to each open facility. This is achieved by adding the demands of all the locations for which that facility is the closest. If the total assigned demand is less than or equal to the facility capacity K, then each demand location served by that facility is assigned a value of one in the *actual_cov* array at the

corresponding indices. If the total demand exceeds the capacity of a facility, only a portion of the demand can be assigned to that facility. In these situations, partial allocations are allowed, which means that some demand locations may only be partially covered. Note that the specific demand locations receiving full or partial coverage can be selected at random, as this randomness does not affect the overall objective value.

With the introduction of *actual_cov*, calculating the total covered demand becomes straightforward:

$$obj = actual\_cov * a.$$

In the small example, with facilities 1 and 2 opened and each having a capacity of 3, the resulting *actual_cov* array is:

$$actual\_cov = [0, 1, 1, 1, 1, 1, 0, 0].$$

In this case, the capacity of each facility is sufficient to serve all demand locations for which it is the nearest, covering a total demand of 5.

To illustrate partial coverage, suppose the capacity is reduced to 2.5 (instead of 3), the resulting *actual_cov* array is:

$$actual\_cov = [0, 1, 1, 0.5, 1, 1, 0, 0].$$

We briefly acknowledge the complexity of the *actual_cov* variable, with more details to follow later. For instance, when a new facility is opened, it can "steal" demand locations from existing facilities. If a demand location is reassigned to the new facility, this may free up capacity at the original facility. If the original facility was previously over capacity, the freed capacity may be subsequently utilized to cover other demand locations that were initially left uncovered due to capacity constraints. These cascading effects require careful consideration throughout the heuristic.

With the key variables introduced and explained through the example, we now proceed to the GRASP heuristic, which relies on these variables for efficient implementation.

4.2 Construction Phase

The construction phase aims to build a feasible solution by iteratively selecting facilities. An effective approach strikes a balance between generating high-quality solutions efficiently and introducing sufficient variation between iterations. If solution quality is poor, the subsequent local search becomes time-consuming, reducing the number of iterations possible within a given timeframe. However, if there is too little variation, the search may repeatedly converge on the same local optima, missing opportunities to explore different areas of the solution space.

Several methods are commonly used in this phase: simple random selection, greedy algorithms, or a hybrid approach that combines random selection followed by refinement using a greedy algorithm [31] [29]. The construction methods are first conceptually introduced in one or two sentences, followed by a detailed analysis of key variables and their associated worst-case complexities.

Although worst-case complexity is considered throughout, it is important to note that dictionaries *row* and *col* are typically much smaller than $|N|$ and $|M|$ due to data sparsity. These dictionaries include only facilities or demand locations within a limited service distance.

To better assess the realistic worst-case complexity of the entire algorithm, we define $k = \max|row|$, representing the maximum number of demand locations that a facility has within its service distance, and $l = \max|col|$, representing the maximum number of potential facility locations that a population point has within its service distance. Due to data sparsity, it generally holds that $k \ll |N|$ and $l \ll |M|$. From this point on, we will denote $|N|$ as n and $|M|$ as m.

In our small example, k equals 4 ($<< 8$) and l equals 2 ($<< 4$).

Although in this small and compact example the difference may not seem substantial, for larger, real-world problems this difference becomes significantly more pronounced.

4.2.1 Random

In this approach, p facilities are randomly selected from the set of all potential facility locations.

Detailed Explanation: Key Variables and Complexity
The selection process of randomly selecting facilities is efficient, with a complexity of $\mathcal{O}(m)$, where m is the total number of potential facilities. Although this step does not directly involve key variables, it is the basis of the subsequent phases of the heuristic. Therefore, after selecting the facilities, the key variables $theoretical\_cov$, $nearest\_opened\_fac$, and $actual\_cov$ need to be updated.

The first step is to update $theoretical\_cov$, which is the most straightforward. For each selected facility, denoted as fac, the demand locations within their service area gain an additional opened facility in their proximity, represented as $theoretical\_cov[row[fac]] += 1$, with a worst-case complexity of $\mathcal{O}(k)$. This operation, applied to each selected facility, results in a worst-case complexity of $\mathcal{O}(kp)$.

The next step is to update $nearest\_opened\_fac$, which involves determining the closest open facility for each demand point (dp) within the service area of the newly opened facility. This means that for each demand location, the intersection of $col[dp]$ and the opened facilities is determined, where the closest facility will be the first element of this intersection as col is sorted. The worst-case complexity of this process is $\mathcal{O}(nlp)$, as up to n demand locations need to be evaluated, and determining the intersection of $col[fac]$ with $opened\_facilities$ has a complexity of $\mathcal{O}(lp)$. However, the complexity of this process can be improved by converting the opened facilities from an array to a set, as searching in a set has complexity $\mathcal{O}(1)$, compared to $\mathcal{O}(p)$ for an array. Due to this improvement, the worst-case complexity of $nearest\_opened\_fac$ reduces to $\mathcal{O}(nl)$.

To enhance efficiency, the search for the nearest facility can be improved. As previously discussed, only the first element of the intersection of col and the opened facilities is of interest. Therefore, in the final code, the search will stop as soon as the first element is found.

Finally, $actual\_cov$ is updated using the function `Update_Actual_Cov`, as described in Algorithm 1. Briefly, this function determines, for each opened facility (fac), which demand locations - those for which this facility is the closest - can be fully or partially covered within the facility's capacity.

Algorithm 1: Update Actual_Cov

```
Function Update_Actual_Cov(fac, nearest_opened_fac, actual_cov, K, a):
    Create a (deep) copy of actual_cov (called actual_cov_copy)                    // O(n)
    Identify (in nearest_opened_fac) the demand locations for which fac is the
     nearest opened facility (called dps)                                          // O(n)
    Set the actual_cov_copy of the demand locations in dps to 0                    // O(k)
    Calculate the cumulative demand of the demand locations in dps                 // O(k)
    Identify which demand locations in dps can be covered by the facility without
     exceeding its capacity, and set actual_cov_copy to 1 for these points (called
     dps_covered)                                                                  // O(k)
    if len(dps_covered) < len(dps) then
        Identify the first demand point in dps that exceeds the capacity
         (called dp)                                                               // O(1)
        Calculate the remaining capacity after covering dps_covered                // O(1)
        if remaining capacity > 0 then
            actual_cov_copy[dp] = remaining capacity/a[dp]                         // O(1)
```

The worst-case time complexity of `Update_Actual_Cov` is $\mathcal{O}(n)$. Since this function must be performed for each of the p opened facilities, the overall worst-case complexity of updating *actual_cov* becomes $\mathcal{O}(np)$.

The complete random selection approach is described in Algorithm 2. Based on this, the worst-case complexity of the random selection approach is derived as $\mathcal{O}(n+m+kp+nl+np)$. Given that $k \ll n$ and $p \geq 1$, this simplifies to $\mathcal{O}(m+n(l+p))$.

Algorithm 2: Construction Phase - Random

```
Function Random(col, row, p, K, a):
    Initialize vectors theoretical_cov and actual_cov with zeros of length |N|   // O(n)
    Initialize vector nearest_opened_fac with -1 of length |N|                   // O(n)
    Randomly select p facilities from the set of all available facilities (called
     opened_facilities)                                                          // O(m)
    Initialize an empty set demand_points                                        // O(1)
    for fac in opened_facilities do                                              // O(p)
        Increase theoretical_cov by one for the demand locations in row[fac]
         and add these to the demand_points set                                  // O(k)
    for dp in demand_points do                                                   // O(n)
        Identify the first facility in col[dp] that is in opened_facilities and update
         nearest_opened_fac[dp] accordingly                                      // O(l)
    for fac in opened_facilities do                                              // O(p)
        actual_cov =
         Update_Actual_Cov( fac, nearest_opened_fac, actual_cov, K, a)           // O(n)
```

4.2.2 Greedy (greedy)

In this approach, the first facility is selected based on the highest additional coverage it provides. After each selection, the additional coverage is recalculated for the remaining facilities and this process continues until p facilities are chosen.

Detailed Explanation: Key Variables and Complexity
As mentioned above, facilities are selected on the basis of their additional coverage contribution when added to the solution. This increase, referred to as the gain in coverage, is determined by constructing the updated *actual_cov* for the new scenario in which the facility is opened. This process involves the following steps: (i) identifying which demand locations already have an open facility within their neighborhood and which do not (ii) updating *nearest_opened_fac* to reflect the new nearest open facility for locations (iii) adjusting *actual_cov* for facilities that serve as the nearest option for a different set of demand locations, including the newly added facility.

First, the demand locations within the service distance of the new facility should be categorized into two groups: those with an open facility already in their neighborhood and those without. This classification can be made by examining the theoretical coverage of each demand location. For demand locations with zero theoretical coverage, the new facility automatically becomes their nearest open facility. For demand locations with theoretical coverage greater than zero, it is necessary to evaluate the proximity of the current nearest open facility with the newly introduced facility. This evaluation is performed using the function `Update_Nearest_Opened_Fac`, which identifies the closer facility.

The `Update_Nearest_Opened_Fac` function operates as follows (outlined in Algorithm 3): for each demand point, it evaluates the indices of the originally nearest facility and the newly introduced facility in *col*. The facility with the smaller index is selected as the updated closest open facility, as *col* is sorted. Furthermore, facilities that lose demand locations for which they were previously the closest are recorded in *facilities_changed*. This information will be used later when updating *actual_cov*.

Algorithm 3: Update Nearest_Opened_Fac

```
1 Function Update_Nearest_Opened_Fac(new_fac, col, nearest_opened_fac,
    demand_points):
2     Create a (deep) copy of nearest_opened_fac (called
        nearest_opened_fac_copy)                                  // O(n)
3     Initialize a set facilities_changed                         // O(1)
4     for dp in demand_points do                                  // O(t)
5         Identify the current nearest facility in nearest_opened_fac_copy (called
            cur_fac)                                              // O(1)
6         Find the indices of the cur_fac and new_fac in col[dp], stored as
            cur_fac_index and new_fac_index respectively          // O(l)
7         if new_fac_index < cur_fac_index then
8             Update nearest_opened_fac_copy and add cur_fac to
                facilities_changed                                // O(1)
```

The function `Update_Nearest_Opened_Fac` has a worst-case time complexity of $\mathcal{O}(tl+n)$, where t represents the number of demand locations provided as input.

To enhance the efficiency of this function, the search for indices is optimized to stop as soon as the index is located, avoiding a full iteration through $facility\_list$.

For facilities in *facilities_changed*, where demand locations are reassigned to the new facility, this reallocation may free up capacity for other demand locations. As a result, *actual_cov* is recalculated from scratch for these facilities and the newly added facility using the function `Update_Actual_Cov` (outlined in Algorithm 4), producing the updated *actual_cov* for the scenario with the new facility.

Algorithm 4: Updating Key Variables for Adding a New Facility

Function `Gain(`*new_fac, theoretical_cov, actual_cov, nearest_opened_fac, col, row, K, a*`):`
  Create (deep) copies of *theoretical_cov*, *actual_cov* and *nearest_opened_fac* (called *theoretical_cov_copy*, *actual_cov_copy* and *nearest_opened_fac_copy*) // $\mathcal{O}(n)$
  Obtain the demand locations within the service distance of *new_fac* (*dps*) // $\mathcal{O}(1)$
  Divide the demand locations in *dps* into those with an opened facility in their neighbourhood (*dps_update*) and those without (*dps_no_update*) // $\mathcal{O}(k)$
  Set *nearest_opened_fac_copy* for the demand locations in *dps_no_update* to *new_fac* // $\mathcal{O}(1)$
  nearest_opened_fac_copy, facilities_changed = `Update_Nearest_Opened_Facility(`*new_fac, col, nearest_opened_fac_copy, dps_update*`)` // $\mathcal{O}(kl+n)$
  Add *new_fac* to *facilities_changed* // $\mathcal{O}(1)$
  for *fac in facility_changed* **do** // $\mathcal{O}(g)$
    actual_cov_copy = `Update_Actual_Cov(`*fac, nearest_opened_fac_copy, actual_cov_copy, K, a*`)` // $\mathcal{O}(n)$
  theoretical_cov_copy[*row*[*new_fac*]]+ = 1 // $\mathcal{O}(k)$

To determine the gain from introducing a new facility, the coverage in the new scenario is calculated, and the original coverage is subtracted from it. The process of calculating the gain for an unopened facility has a worst-case time complexity of $\mathcal{O}(kl + ng)$, where g represents the maximum length of *facilities_changed*, formally expressed as $g = \max |s_j|$. Here, s_j denotes the set of facilities that share demand locations with a given facility j, including j itself.

Reconsider the small example where facilities 1 and 2 are opened (see Figure 4 for visualization, a duplicate of Figure 3), yielding the following key variables: *theoretical_cov* $= [0, 1, 1, 1, 2, 1, 0, 0]$, *nearest_opened_fac* $= [-1, 1, 1, 1, 2, 2, -1, -1]$, *actual_cov* $= [0, 1, 1, 1, 1, 1, 0, 0]$.

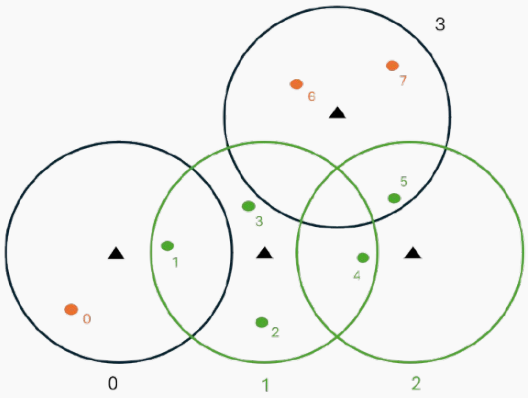


Figure 4: Visualization of the small example with facilities 1 and 2 open

To determine the gain in coverage of facility 3, first identify the demand locations within its service distance, given as $row[3] = [5, 6, 7]$. These demand locations can be divided into two groups:

- with an opened facility in the neighborhood: demand location 5.
- without an opened facility in the neighborhood: demand locations 6 and 7.

For demand locations 6 and 7, $nearest\_opened\_fac$ is updated to facility 3. For demand point 5, the function `Update_Nearest_Opened_Fac` determines the nearest open facility, where $col[5] = [2, 3]$. Facility 2, currently the nearest, is located at index 0 in col, while the newly added facility, facility 3, is at index 1. Therefore, facility 2 remains the nearest open facility for demand point 5.

The function `Update_Nearest_Opened_Fac` does not add any facilities to $facilities\_changed$. Consequently, $actual\_cov$ is updated only for facility 3 using the function `Update_Actual_Cov`. First, determine the demand locations for which facility 3 is the closest: demand locations 6 and 7. Initially, $actual\_cov[6]$ and $actual\_cov[7]$ are set to 0. The total demand for these points is calculated as 2. Since the capacity $K = 3$ is sufficient to cover both points $actual\_cov[6]$ and $actual\_cov[7]$ are updated to 1, resulting in:

$$actual\_cov = [0, 1, 1, 1, 1, 1, 1, 1].$$

The initial coverage without facility 3 was 5, whereas the updated coverage with facility 3 is 7. Therefore, the additional coverage gained by opening facility 3 is 2.

Calculating the gain is performed for each unopened facility (m), resulting in an overall complexity of $\mathcal{O}(m(kl + ng))$. To finalize the selection process, the facility with the highest additional coverage is selected, which has a complexity of $\mathcal{O}(m)$. Therefore, the total worst-case complexity of the selection process is $\mathcal{O}(m(kl + ng))$.

After selecting a facility, all key variables need to be updated:

- $theoretical\_cov$: worst-case complexity of $\mathcal{O}(k)$.
- $nearest\_opened\_fac$: worst-case complexity of $\mathcal{O}(kl + n)$.
- $actual\_cov$: worst-case complexity of $\mathcal{O}(ng)$.

This results in a total worst-case complexity for updating key variables of $\mathcal{O}(kl + ng)$.

When selecting additional facilities, recalculating the gain for all unopened facilities is inefficient, as the gain values for many facilities remain unchanged. Instead, updates are limited to the facilities in the neighborhood of the newly selected facility and those near the affected open facilities. Here, "neighborhood" refers to facilities that share common demand locations. The neighborhood of affected open facilities is considered because the demand locations are shifted in the new scenario to the newly selected facility, potentially altering the gain values of the facilities. To address this, the function `Adjust_To_Update`, outlined in Algorithm 5, is introduced.

Algorithm 5: Adjust To_Update

```
1 Function Adjust_To_Update(select, facilities_changed, col, row):
2     Obtain the demand locations within the service distance of new_fac
        (called dps)                                              // O(1)
3     Initialize a set impacted_fac                               // O(1)
4     for dp in dps do                                          // O(k)
5         Add the facilities within the service distance of dp to impacted_fac   // O(l)
      ⋮
```

Algorithm 5: Adjust To_Update

Function `Adjust_To_Update(`*select, facilities_changed, col, row*`):`
- ⋮
- Identify which facilities of $impacted\_fac$ are in $facilities\_changed$ (called $impacted\_opened\_fac$) // $\mathcal{O}(m)$
- Initialize a set *other_dps* // $\mathcal{O}(1)$
- **for** *fac in impacted_opened_fac* **do** // $\mathcal{O}(p)$
  - Add the demand locations within the service distance of fac to *other_dps* // $\mathcal{O}(k)$
- Initialize a set *additional_impacted_fac* // $\mathcal{O}(1)$
- **for** *dp in other_dps* **do** // $\mathcal{O}(n)$
  - Add the facilities within the service distance of *dp* to *additional_impacted_fac* // $\mathcal{O}(l)$
- Combine the facilities $impacted\_fac$ and $additional\_impacted\_fac$ and filter out any facilities that are already in $opened\_facilities$, stored in *to_update* // $\mathcal{O}(m)$

The function `Adjust_To_Update` Identifying these neighboring facilities has a worst-case complexity of $\mathcal{O}(kl + m + kp + nl) = \mathcal{O}(m + kp + nl)$, given that $k \ll n$.

The complete greedy approach is described in Algorithm 6.

Algorithm 6: Construction Phase - Greedy

Function `Greedy(`*a, K, p, α, col, row*`):`
- Initialize a set $opened\_facilities$ // $\mathcal{O}(1)$
- Initialize vectors *theoretical_cov* and *actual_cov* with zeros of length $|N|$ // $\mathcal{O}(n)$
- Initialize vector *nearest_opened_fac* with -1 of length $|N|$ // $\mathcal{O}(n)$
- Initialize vector *greedy_val* with zeros of length $|M|$ // $\mathcal{O}(m)$
- Initialize vector *to_update* with the potential facility locations indices // $\mathcal{O}(m)$
- **for** $iter = 1, ..., p$ **do** // $\mathcal{O}(p)$
  - **for** *j in to_update* **do** // $\mathcal{O}(m)$
    - *actual_cov_gain* = `Gain(`*j, theoretical_cov, actual_cov, nearest_opened_fac, col, row, a, K*`)` // $\mathcal{O}(kl + ng)$
    - Calculate the gain in coverage by calculating new coverage minus the current coverage and store it at *greedy_val[j]* // $\mathcal{O}(1)$
  - Select the facility (*select*) with the highest additional coverage (in *greedy_val*) // $\mathcal{O}(m)$
  - Update *theoretical_cov, actual_cov, nearest_opened_fac* // $\mathcal{O}(kl + ng)$
  - Add *select* to *opened_facilities* and set *greedy_val[select]* equal to zero // $\mathcal{O}(1)$
  - *to_update* = `Adjust_To_Update(`*select, opened_facilities, col, row*`)` // $\mathcal{O}(m + kp + nl)$

Without leveraging *to_update*, the worst-case complexity is $\mathcal{O}(pm(kl + ng))$. By introducing *to_update*, the maximum length of which is denoted as v (with $v \ll m$), the worst-case complexity becomes $\mathcal{O}(m(kl + ng) + (p - 1)(v(kl + ng) + m + kp + nl))$.

As the expression for complexity involving v (accounting for *to_update*) is complex and overwhelming, the remainder of this thesis will use the simplified expression of complexity without considering *to_update*. However, it will be emphasized whenever relevant that in practice the actual complexity is much lower for larger problems due to the significant reduction in the size of *to_update* as the algorithm progresses.

4.2.3 Randomized Greedy (rgreedy)

The randomized greedy approach is similar to the greedy method, but instead of always selecting the facility with the highest additional coverage, it randomly selects a facility from the top $\alpha\%$ of potential facilities, ranked by their additional coverage.

Detailed Explanation: Key Variables and Complexity
The algorithm for randomized greedy is identical to the greedy algorithm (outlined in Algorithm 6), except that line 11 is modified as follows: Randomly select a facility (*select*) from the top $\alpha\%$ of potential facilities, ranked according to *greedy_val*.

Sorting the values in *greedy_val* results in a worst-case complexity of $\mathcal{O}(m \log m)$. Once sorted, a facility is randomly selected from the top $\alpha\%$ of this list, which has a complexity of $\mathcal{O}(1)$. Therefore, the overall complexity of line 11 is $\mathcal{O}(m \log m)$.

Therefore, the worst-case complexity of the randomized greedy approach is $\mathcal{O}(pm(kl + gn + \log(m)))$. Note that, with the introduction of v, the worst-case complexity is given by: $\mathcal{O}(m(kl + gn + \log(m)) + (p-1)(v(kl + gn) + m\log(m) + kp + nl))$. Note that, assuming $m > e$ (where $e = 2.718$), it follows that $\mathcal{O}(m \log m + m) = \mathcal{O}(m \log m)$.

4.2.4 Proportional Greedy (pgreedy)

Proportional greedy is similar to randomized greedy (and, by extension, to greedy), but instead of randomly choosing, another underlying distribution is used with a bias towards choosing a facility with a higher rank. Let $r(f)$ denote the rank of facility f according to the greedy metric. A linear bias selects facility f with a bias $bias(r) = 1/r$, while an exponential bias selects facility f with $bias(r) = e^{-r}$.

Detailed Explanation: Key Variables and Complexity
This approach is similar to the randomized greedy method. The only difference lies in the selection process: instead of selecting a facility randomly, a bias is applied. For this process, we introduce the function `Apply_Bias`, outlined in Algorithm 7, with a worst-case complexity of $\mathcal{O}(\alpha m)$, where $\alpha < 1$. Here, *choose_from* contains the indices of facilities in the top $\alpha\%$, with higher coverage facilities appearing earlier.

Algorithm 7: Apply Bias

```
1 Function Apply_Bias(choose_from, bias_type):
2     if bias_type is 'lin' then
3         Compute bias_value vector as 1/rank, where rank is an array starting at 1 and
            ending at len(choose_from) + 1                                  // O(αm)
4     else if bias_type is 'exp' then
5         Compute bias_value as e^{-rank}, where rank is an array starting at 1 and ending
            at len(choose_from) + 1                                         // O(αm)
6     Normalize bias_value vector such that the probabilities sum to 1      // O(αm)
7     Randomly select an facility from choose_from using bias_value as the
        probability distribution                                            // O(αm)
```

The algorithm for proportional greedy is identical to greedy algorithm (outlined in Algorithm 6), except that line 11 is modified as follows: Select a facility based on a biased probability from the top $\alpha\%$ of potential facilities, ranked by *greedy_val*. Assuming $m > 2.718 = e$ and given $\alpha < 1$, the complexity of line 11 equals $\mathcal{O}(m \log m + \alpha m) = \mathcal{O}(m \log m)$.

Therefore, the worst-case complexity of the proportional greedy approach is $\mathcal{O}(pm(kl + gn + \log(m)))$, which is identical to the randomized greedy method.

4.2.5 Random plus Greedy (rpg)

In this approach, randomness and greediness are combined: the first $\beta \cdot p$ facilities are chosen randomly, while the remaining facilities are selected in a greedy way, where $\beta \in [0, 1]$.

Detailed Explanation: Key Variables and Complexity
The total worst-case complexity of random plus greedy is $\mathcal{O}(m + n(l + \beta p) + (1 - \beta)pm(kl + ng)$. For more explanation, see 'Detailed Explanation: Key Variables and Complexity' in Sections 4.2.1 and 4.2.2, where the randomized and greedy approaches are discussed.

Note that β is a constant. The worst-case complexity of the expression is primarily determined by the greedy approach, which is represented as $\mathcal{O}(pm(kl + ng))$.

4.2.6 Sample Greedy (sample)

In this approach, at each iteration, a random sample of $q < m$ potential facilities is considered, and the best facility based on the greedy metric is selected. The value of q should be small enough to enhance efficiency while ensuring sufficient variation between different iterations.

Detailed Explanation: Key Variables and Complexity
The sample greedy method is similar to the greedy approach, but the key difference lies in determining for which facilities the gain in coverage is calculated. Instead of evaluating the gain for all unopened facilities, sample greedy evaluates only q facilities, where $q = \theta \cdot p$ and $\theta \in (0, 1]$. These q facilities are randomly selected in each iteration. This reduces the worst-case complexity to $\mathcal{O}(q(kl+ng)+m) = \mathcal{O}(q(kl+ng))$, compared to $\mathcal{O}(m(kl+ng))$, where $q < m$.

The sample greedy algorithm, outlined in Algorithm 8, has a worst-case complexity of $\mathcal{O}(pq(kl + ng))$.

Algorithm 8: Construction Phase - Sample

```
1  Function Sample(a, K, p, θ, col, row):
2      Initialize a set opened_facilities                          // O(1)
3      Initialize vectors theoretical_cov and actual_cov with zeros of length |N|   // O(n)
4      Initialize vector nearest_opened_fac with -1 of length |N|  // O(n)
5      q = int(θ * |M|)                                            // O(1)
6      for iter = 1, ..., p do                                     // O(p)
7          Initialize vector greedy_val with zeros of length |M|   // O(m)
8          Take a random sample of the potential facility indices of length q   // O(m)
9          From the random sample only calculate the gain of unopened
             facilities, stored in random_sample                   // O(q)
           ⋮
```

Algorithm 8: Construction Phase - Sample

```
1  Function Sample(a, K, p, θ, col, row):
       ⋮
6      for iter = 1, ..., p do                                   // O(p)
           ⋮
10         for j in random_sample do                             // O(q)
11             actual_cov_gain = Gain(j, theoretical_cov, actual_cov,
                 nearest_opened_fac, col, row, K, a)           // O(kl + ng)
12             Calculate the gain in coverage by calculating new coverage
                 minus the current coverage and store it at greedy_val[j]   // O(1)
13         Select the facility (select) with the highest greedy_val value   // O(m)
14         Update theoretical_cov, actual_cov, nearest_opened_fac   // O(kl + ng)
15         Add select to opened_facilities                       // O(1)
```

4.2.7 Overview Construction Phases

Table 2 summarizes the worst-case complexities of the various methods proposed to select the facilities during the construction phase. The computational cost of each method varies, depending on how the *gain* values are calculated and evaluated.

Method	Worst-Case Complexity
Random	$\mathcal{O}(m + n(l + p))$
Greedy	$\mathcal{O}(pm(kl + gn))$
Randomized Greedy	$\mathcal{O}(pm(kl + gn + \log(m)))$
Proportional Greedy	$\mathcal{O}(pm(kl + gn + \log(m)))$
Random Plus Greedy	$\mathcal{O}(m + n(l + \beta p) + (1 - \beta)pm(kl + gn))$
Sample	$\mathcal{O}(pq(kl + gn))$

Table 2: Worst-case complexities of various methods for facility selection.

Key observations
The random method is the simplest and fastest, as it avoids exhaustive evaluations by randomly selecting facilities. Its complexity, $\mathcal{O}(m + n(l + p))$, is significantly lower than methods that require detailed gain calculations. However, the quality of the solution is often suboptimal due to the lack of evaluation.

The greedy method evaluates the gain in coverage for all m unopened facilities, leading to a complexity of $\mathcal{O}(pm(kl + gn))$. This comprehensive evaluation ensures high solution quality, but comes with the highest computational cost among these methods.

The randomized and proportional greedy methods enhance the greedy approach by introducing randomization or bias into the selection process. They retain the same base complexity for calculating the gain values as greedy ($\mathcal{O}(pm(kl + gn))$), but the added sorting step increases the complexity to $\mathcal{O}(pm(kl + gn + \log(m)))$. This randomized or biased selection adds variation between solutions while slightly increasing computational effort.

The random plus greedy method combines random selection with the greedy approach, balancing computational efficiency and solution quality. Its complexity, $\mathcal{O}(m + n(l + \beta p) + (1 - \beta)pm(kl + ng))$, grows similarly to the greedy method in the worst case while offering flexibility in the trade-off between random and greedy evaluations through the β parameter.

The sample greedy method evaluates gain in coverage for only q facilities ($q \ll m$) per iteration, rather than all unopened facilities. This reduces the complexity to $\mathcal{O}(pq(kl + gn))$, making it faster than greedy for small θ. The quality of the solution improves with larger θ, but this comes at the cost of reduced variation between solutions.

Note that we have introduced an additional complexity expression involving the *to_update*, which reduces the number of gain calculations. As a result, the difference in complexities between the greedy and sample approaches becomes less pronounced than stated in Table 2.

Furthermore, in the sample greedy method, the advantage of evaluating only facilities within the random sample diminishes somewhat as the number of facilities to be opened increases. While the greedy approach updates values only for the affected facilities, the *sample* method recomputes values from scratch each time, which does not always lead to substantial time savings. However, it is worth mentioning that the *sample* method could be implemented similarly to the greedy approach by maintaining a record of all greedy values and selecting the best facility from a new random sample at each step.

Expectations
The random method is the fastest, as it bypasses gain evaluations entirely, but this comes at the cost of reduced solution quality. The sample method, with $q \ll m$, strikes a balance by being faster than fully greedy approaches while maintaining a reasonable solution quality. On the other hand, greedy, randomized greedy, and proportional greedy methods are slower but achieve higher solution quality due to their exhaustive evaluations, with the quality also depending on the value of α.

4.3 First Improvement Local Search

The first improvement phase focuses on making localized adjustments by evaluating facility swaps based on an initial solution. If a swap results in an increase in coverage, it is executed immediately without conducting a full exhaustive search on other swaps. This approach efficiently enhances coverage while maintaining computational efficiency.

To illustrate the core concept of the first improvement phase, a simplified representation is provided in Algorithm 9, excluding the use of key variables. In this representation, f_l denotes a facility leaving the solution, f_e represents a facility entering the solution, and F is the set of all potential facilities.

Algorithm 9: Simplified Representation of First Improvement Local Search

```
1 Function Local_Search(F, opened_facilities):
2     for f_l ∈ opened_facilities do
3         Compute the loss in coverage (loss_l) by f_l leaving opened_facilities;
4         for f_e ∈ F\opened_facilities do
5             Compute the gain in coverage (gain_e) by f_e entering opened_facilities\{f_l};
6             if gain_e > loss_l then
7                 opened_facilities = opened_facilities ∪ {f_e}\{f_l};
8                 Return to line 2;
```

As the simplified representation illustrates, the local search is primarily based on calculating both losses and gains. The approach for calculating gains in coverage is similar to the method described earlier in the construction phase, and the method for calculating losses in coverage will be explained in the next subsection.

4.3.1 Loss

Calculating the loss of a leaving facility involves constructing *actual_cov* for the new scenario in which the facility is closed. This process involves the following steps: (i) identify the demand locations for which the leaving facility is currently the closest, (ii) if possible, reassign these demand locations to the second-nearest open facility, which becomes the new closest facility once the original facility leaves, and (iii) for the facilities serving new demand locations as their nearest, reconstruct *actual_cov* from scratch to determine which demand locations should receive coverage. If there is unused capacity, this may lead to additional coverage opportunities. To determine the loss from closing a facility, the coverage in the new scenario is calculated and subtracted from the original coverage.

Technical Explanation of Loss Using Key Variables
To begin with, the demand locations for which the leaving facility is the closest open facility can be identified by the indices where $nearest\_opened\_fac$ equals the leaving facility. For each of these demand locations, the corresponding index in $nearest\_opened\_fac$ is updated to -1 (indicating that there is no nearby open facility), and the actual coverage in *actual_cov* is set to zero. At this stage, the reassignment of demand locations has not yet been addressed.

Next, the theoretical coverage in *theoretical_cov* for all demand locations within the service distance of the leaving facility is reduced by one. Reassignment is possible for demand locations that still have a theoretical coverage greater than zero, called *reassign_dps*, indicating that they remain within the service range of other open facilities. To handle the reassignment of these demand locations, the function `Reassign_To_Second_Nearest_Opened_Fac` is introduced.

The function `Reassign_To_Second_Nearest_Opened_Fac`, described in Algorithm 10, operates as follows: for each demand location, it begins by retrieving the facilities within the service distance using *col*. Next, it computes the intersection of these facilities with the currently open facilities and identifies the second-nearest open facility by selecting the second element from the intersection list, as *col* is sorted by proximity. Then, $nearest\_opened\_fac$ is updated accordingly. To facilitate future updates to *actual_cov*, facilities that have become closest to the reassigned demand locations are recorded in $facilities\_changed$.

To improve efficiency, instead of finding all intersections between $col[dp]$ and the open facilities and then selecting the second element, the process can be optimized by halting the search as soon as the intersection list reaches a length of two.

Algorithm 10: Reassignment of Demand Locations to Second Nearest Open Facility

```
1 Function Reassign_To_Second_Nearest_Opened_Facility(opened_facilities, col,
    reassign_dps, nearest_opened_fac):
2     Create a (deep) copy of nearest_opened_fac
        (called nearest_opened_fac_copy)                          // O(n)
3     Initialize a set facilities_changed                         // O(1)
4     for dp in reassign_dps do                                   // O(t)
5         Identify the second-nearest open facility by iterating through facilities in col[dp],
            checking for membership in opened_facilities (called second_fac)   // O(l)
          Update nearest_opened_fac_copy for dp to second_fac     // O(1)
6         Add second_fac to facilities_changed                    // O(1)
```

The function `Reassign_To_Second_Nearest_Opened_Fac` has a worst-case complexity of $\mathcal{O}(tl + n)$, where t represents the number of demand locations provided as input.

For the facilities in $facilities\_changed$, the selection of demand locations to be covered is reevaluated from scratch using the previously introduced function `Update_Actual_Cov`.

All the steps required to construct the key variable $actual\_cov$ for the scenario in which an existing facility is removed are summarized in Algorithm 11.

Algorithm 11: Update Key Variables for Removing an Existing Facility

Function `Loss`(*leaving_fac, opened_facilities, theoretical_cov, actual_cov, nearest_opened_fac, col, row, K, a*)**:**

- Create (deep) copies of *theoretical_cov*, *actual_cov* and *nearest_opened_fac* (called *theoretical_cov_copy*, *actual_cov_copy* and *nearest_opened_fac_copy*) // $\mathcal{O}(n)$
- Identify the demand locations for which the leaving facility is nearest (called *dps*) // $\mathcal{O}(k)$
- For each demand location in *dps*, set *actual_cov_copy* to 0 and *nearest_opened_fac_copy* to -1 // $\mathcal{O}(k)$
- $theoretical\_cov\_copy[row[leaving\_fac]] -= 1$ // $\mathcal{O}(k)$
- Identify which demand locations in *dps* has still *theoretical_cov_copy* greater than 0 (called *reassign_dps*) // $\mathcal{O}(k)$
- $nearest\_opened\_fac\_copy, facilities\_changed =$ `Reassign_To_Second_Nearest_Opened_Facility`(*opened_facilities, col, reassign_dps, nearest_opened_fac_copy*) // $\mathcal{O}(kl + n)$
- **for** *fac in facility_changed* **do** // $\mathcal{O}(g)$
  - *actual_cov_copy* = `Update_Actual_Cov`(*fac, nearest_opened_fac_copy, actual_cov_copy, K, a*) // $\mathcal{O}(n)$

Similarly to calculating the gain, once *actual_cov* is reconstructed for the scenario where the leaving facility is no longer part of the solution, the coverage in this new scenario can be determined. The loss is then calculated as the difference between the original coverage and the coverage in the updated scenario. The process of calculating the loss in coverage for an opened facility has a worst-case complexity of $\mathcal{O}(kl + ng)$.

Reconsider the small example where facilities 1 and 2 are opened (see Figure 6 for visualization, a duplicate of Figure 3), yielding the following key variables: $theoretical\_cov = [0, 1, 1, 1, 2, 1, 0, 0]$, $nearest\_opened\_fac = [-1, 1, 1, 1, 2, 2, -1, -1]$, $actual\_cov = [0, 1, 1, 1, 1, 1, 0, 0]$.

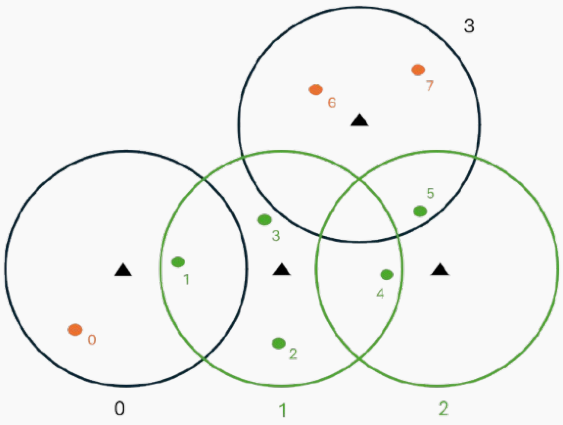


Figure 5: Visualization of the small example with facilities 1 and 2 open

To calculate the loss value of facility 2, start by identifying the demand locations for which facility 2 is the nearest. Here, these are demand locations 4 and 5. The key variables for these demand locations must then be updated.

After the updates, the key variables are as follows: $theoretical\_cov = [0, 1, 1, 1, 1, 0, 0, 0]$, $nearest\_opened\_fac = [-1, 1, 1, 1, -1, -1, -1, -1]$, $actual\_cov = [0, 1, 1, 1, 0, 0, 0, 0]$.

At this point, it can be observed that demand location 4 still has a theoretical coverage greater than zero, which means that it can be reassigned to the second-nearest open facility. For this demand location, the function `Reassign_To_Second_Nearest` identifies the second-nearest open facility using $col[4] = [2, 1]$. Here, facility 1 is identified as the second-nearest open facility. Consequently, $nearest\_opened\_fac[4]$ is updated, yielding $nearest\_opened\_fac = [-1, 1, 1, 1, 1, -1, -1, -1]$. Since facility 1 gains a new demand point, it is added to $facilities\_changed$.

Next, $actual\_cov$ is updated for facilities in $facilities\_changed$ (only facility 1) using the function `Update_Actual_Cov`. First, determine the demand locations for which facility 1 is the closest: demand locations 1, 2, 3, and 4. Initially, set for these locations $actual\_cov$ to zero. The total demand for these points is calculated as 4. Given a capacity limit $K = 3$, facility 1 can cover only the first three demand locations 1, 2 and 3. As a result, $actual\_cov[4]$ remains zero, resulting in: $actual\_cov = [0, 1, 1, 1, 0, 0, 0, 0]$.

The initial coverage with facility 2 was 5, whereas the updated coverage without facility 2 is 3. Therefore, the loss in coverage by closing facility 2 is 2.

4.3.2 Final refinements

Using the simplified representation in Algorithm 9, combined with the introduction of the methods for calculating *loss* and *gain*, the first improvement can be implemented. The worst-case complexity of the first improvement cannot be precisely defined, as the number of swaps required to reach a local optimum is unknown. However, the worst-case complexity of executing a single swap, when identifying the most profitable swap in the final attempt, is $\mathcal{O}(p(m-p)(kl+gn)^2)$.

To reduce complexity and improve efficiency, several enhancements to the basic first improvement algorithm are introduced.

Initially, $facilities\_opened$ is searched in the same order each time, leading to the repeated consideration of the same swaps. Since only one facility changes in the solution at each step of the local search, it can be more efficient to prioritize exploring new swaps. A simple way to achieve this is by applying a random shuffle to $opened\_facilities$ after each swap, ensuring that the solutions are searched in a different order each time.

An additional enhancement for efficiently considering swaps is the introduction of the no_improv variable. This variable tracks how many times a facility in the open facilities has been searched without finding a profitable swap. After the random shuffle of $opened\_facilities$, the facilities are sorted based on their corresponding no_improv values. This ensures that facilities that showed no improvement earlier are less prioritized.

To further accelerate the algorithm and make the local search feasible for larger problems, the search is restricted to the neighborhood of each facility by allowing swaps only with relatively nearby facilities. This reduces the number of swaps to evaluate, thereby speeding up the local search. However, limiting the number of swaps may result in a less optimal local solution. To define the neighborhood of a leaving facility, the haversine distance is used. This is a measure

of the great-circle distance between two points on the Earth's surface, calculated using their latitude and longitude. Only facilities within a specified *border* distance (in kilometers) are considered as *candidates*. These facilities are then sorted based on their proximity to the leaving facility.

To avoid recalculating the neighborhood within a specified *border* distance for each leaving facility, the neighborhood list is stored in a dictionary called *candidates_dict*. For the facilities in $opened\_facilities$, these neighborhoods are constructed at the start of the local search. When a new facility enters $opened\_facilities$, its corresponding neighborhood is added to *candidates_dict*. This approach saves time by reusing previously computed results. It is important to note that the *candidate* list includes both opened and unopened facilities, so an additional step is required to filter and retain only the unopened facilities.

Limiting the number of candidate facilities and preprossessing *candidates_dict* reduces the worst-case complexity of a swap to $\mathcal{O}(pz(kl+gn)^2) + plog(p))$, where $z = \max r_j$, with r_j representing the number of facilities within the border for facility j ($r_j \leq z \ll m - p < m$). If a swap is found in the final iterations, there is an additional complexity of $\mathcal{O}(m + z \log z)$ for constructing *candidates_dict* for the entering facility. The complexity m arises from checking the distance to each facility and selecting those within the specified *border*, while $z \log z$ accounts for sorting the facilities that fall within the border.

The final improvement to the local search algorithm focuses on optimizing the loss calculation, particularly within the function `Reassign_To_Second_Nearest_Opened_Fac`. This function determines the second-nearest open facility for each demand point previously served by the leaving facility by finding the intersection of *col* and $opened\_facilities$. However, many facilities in $opened\_facilities$ are irrelevant, as they lie outside the service distance of the demand point.

To enhance efficiency, only open facilities with overlapping demand locations are considered, reducing the set of facilities to evaluate. Similar to *candidates_dict*, at the start of the local search, overlapping facilities for each facility in $opened\_facilities$ are identified and stored in a dictionary called $near\_fac\_dict$. When a new facility is added to $opened\_facilities$, its corresponding neighborhood is calculated and added to $near\_fac\_dict$. Since the facility list includes both open and unopened facilities, an additional step is required to filter and retain only the open facilities.

In the worst case, the length of $near\_fac\_dict$ equals kl, as a facility can have at most k demand locations within its service distance, and each demand location can be associated with up to l facilities within their service distance.

The whole the first improvement algorithm, including the key variables and the final refinements, is outlined in Algorithm 12.

Algorithm 12: First Improvement Local Search

```
1  Function First_Improvement(opened_facilities, theoretical_cov, actual_cov,
     nearest_opened_fac, col, row, K, a, facilities_data, border):
2      Create (deep) copies of theoretical_cov, actual_cov and nearest_opened_fac
         (called theoretical_cov_copy, actual_cov_copy and nearest_opened_fac_copy)
3      finished = False                                                  // O(1)
4      Initialize a vector count_no_improv of zeros of length |M|        // O(m)
5      Initialize sets candidates_dict and near_fac_dict                 // O(1)
6      for fac in opened_facilities do                                   // O(p)
7          Construct near_fac_dict and candidates_dict for fac           // O(kl + m + zlogz)
8      while finished == False do
9          flag = False                                                  // O(1)
10         Random shuffle opened_facilities, then sort it in ascending order based on
             count_no_improv                                             // O(plogp)
11         for i in opened_facilities do                                 // O(p)
12             Identify which of the facilities in near_fac_dict[i] are open (called
                 opened_fac_nearby)                                      // O(kl)
13             theoretical_cov_i, actual_cov_i, nearest_opened_fac_i = Loss(i,
                 opened_fac_nearby, theoretical_cov_copy, actual_cov_copy,
                 nearest_opened_fac_copy, col, row, K, a)                // O(kl + ng)
14             Calculate the loss in coverage by calculating the current coverage minus the
                 new coverage                                            // O(1)
15             Identify which of the facilities in candidates_dict[i] are not open (called
                 notopened_candidates)                                   // O(z)
16             for j in notopened_candidates do                          // O(z)
17                 theoretical_cov_j, actual_cov_j, nearest_opened_fac_j =
                     Gain(j, theoretical_cov_i, actual_cov_i, nearest_opened_fac_i, col,
                     row, K, a)                                          // O(kl + ng)
18                 Calculate the gain in coverage by calculating new coverage minus the
                     current coverage (with facility i)                  // O(1)
19             if gain > loss then
20                 flag = True                                           // O(1)
21                 Set theoretical_cov_copy to theoretical_cov_j         // O(1)
22                 Set actual_cov_copy to actual_cov_j                   // O(1)
23                 Set nearest_opened_fac_copy to nearest_opened_fac_j   // O(1)
24                 Replace i with j in opened_facilities                 // O(p)
25                 if j not in nearest_fac_dict then
26                     Conduct nearest_fac_dict and candidates_dict for the entering
                         facility j                                      // O(kl + m + zlogz)
27                 if flag == True then
28                     break
29             if flag == False then
30                 count_no_improv[i] += 1                               // O(1)
31             else
32                 break
33         if flag == False then
34             finished = True                                           // O(1)
```

4.4 Best Improvement Local Search

In the best improvement local search, swaps are selected through an exhaustive evaluation of all possible facility swaps, within and beyond the neighborhood of the open facilities. The aim is to identify the swap that yields the largest improvement in coverage. This process is repeated until no further profitable swaps can be found.

The best improvement local search can be implemented with only small adjustments to the first improvement method. In the best improvement approach, all potential swaps are evaluated, eliminating the need for a random shuffle of *opened_facilities* and the *no_improv* variable. Furthermore, this method evaluates all possible swaps, not just those within predefined borders, making the *candidates_dict* variable unnecessary. Instead of executing a profitable swap immediately, the algorithm identifies the best swap and applies it only after evaluating all possible options. Other elements of the algorithm's structure remain consistent with the first improvement method.

This simple approach, which evaluates all possible swaps in each iteration (where one swap is identified and executed), results in significant redundant calculations, both within and across iterations. Executing a single swap with this approach has a worst-case complexity of $\mathcal{O}(p(m-p)(kl+gn)^2)$.

Within a single iteration, for instance, the loss of a leaving facility and the gain of an entering facility are calculated together. The gain is recalculated if another leaving facility is considered with the same entering facility. If the two leaving facilities do not share any demand locations or open facilities with the entering facility, these gains will be identical, making the recalculation unnecessary.

Across iterations, after a swap is executed, all gains and losses are recalculated using the simple approach. However, this recalculation is redundant for facilities where the swapped facility, whether entering or leaving, is not within their neighborhood, as the value remains unchanged.

The advanced approach is introduced to address and eliminate these repetitive computations, streamlining the process. In short, the advanced approach calculates and stores the objective change for all possible swaps. Once the best swap is identified, it is executed and the gain and loss are calculated for the affected facilities.

The advanced approach for the best improvement local search is based on the work of Resende and Werneck on the p-median problem [28]. The adaptation of their method to the maximal covering location problem (MCLP) was later developed by Theulen [5]. However, extending this approach to a capacitated version of the MCLP, where assignments are restricted to the closest open facility, presents additional challenges. The following sections will outline the implementation of the advanced approach tailored to address these complexities in our specific problem.

4.4.1 Constructing *loss*, *gain* and *extra*

The advanced approach is built around three key data structures: *loss*, *gain*, and *extra*. These structures store precomputed values to minimize redundant calculations. The *loss* vector records the decrease in coverage that would occur if a facility currently included in the solution is removed. The *gain* vector tracks the increase in coverage that would result from the addition of a new facility, currently not part of the solution.

The calculations for the loss and gain are similar to those used in the first improvement local search. These calculations construct the new scenario in *actual_cov*, taking into account both capacity constraints and the requirement to assign to the closest facility. This approach effectively assesses the individual impact of adding or removing a facility. However, while the vectors *loss* and *gain* provide valuable insights into the effects of changes in a single facility, they do not capture the combined impact of performing a swap.

For example, in the current solution, a leaving facility uniquely serves a specific demand location, giving it a theoretical coverage value of one. If the leaving facility is removed and its capacity is not fully utilized, this demand location contributes to the loss. If an entering facility has this demand location within its service area, it is assigned a theoretical coverage value of two during the gain calculation. However, if the leaving facility remains the closest open facility, this demand location does not contribute to the gain. When the swap is executed, the entering facility becomes the nearest open facility for this demand location. If it has remaining capacity, it increases coverage. However, this effect is not captured when evaluating loss and gain independently.

Reconsider the small example where facilities 1 and 2 are opened (see Figure 6 for visualization, a duplicate of Figure 3), yielding the following key variables: $theoretical\_cov = [0, 1, 1, 1, 2, 1, 0, 0]$, $nearest\_opened\_fac = [-1, 1, 1, 1, 2, 2, -1, -1]$, $actual\_cov = [0, 1, 1, 1, 1, 1, 0, 0]$.

In Section 4.3.1, the loss associated with facility 2 is determined to be 2. This is because demand location 5 had facility 2 as the only open facility within its service distance, contributing to the calculated loss. Similarly, in Section 4.2.1, the gain associated with facility 3 is also calculated to be 2. However, since facility 2 remained in the solution, it was still the closest open facility to demand location 5 and therefore this demand location did not change during the gain calculations.

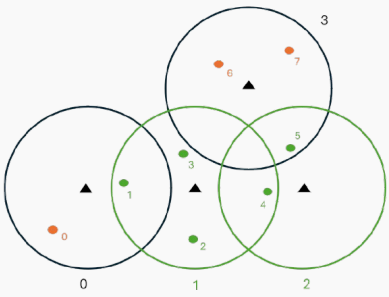


Figure 6: Visualization of the small example with facilities 1 and 2 open

When facilities 1 and 3 (see Figure 7 for visualization) are swapped, facility 3 becomes the closest open facility to demand location 5 and still has available capacity. As a result, this demand location should be accounted for in the *extra* term.

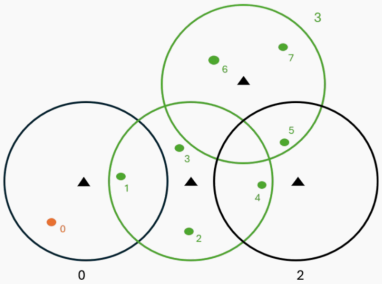


Figure 7: Visualization of the small example with facilities 1 and 3 open

In the new scenario where facilities 1 and 3 are open, the key variables are updated as follows: $theoretical\_cov = [0, 1, 1, 1, 1, 1, 1, 1]$, $nearest\_opened\_fac = [-1, 1, 1, 1, 1, 3, 3, 3]$, $actual\_cov = [0, 1, 1, 1, 0, 1, 1, 1]$.

Using only loss and gain, the expected objective value would be calculated as 5 (original objective) + 2 (gain) − 2 (loss), resulting in 5. However, the actual objective value is 6, revealing an additional effect of + 1 that is not captured by loss and gain alone.

Another example arises when a leaving facility serves a specific demand location with a theoretical coverage value greater than 1. If the leaving facility is removed from the current solution, this demand location is reassigned to another open facility. If the other facility has available capacity, this demand location does not contribute to the loss. Consider an entering facility with this demand location within its service distance. If this location still regards the leaving facility as its closest open facility, it is excluded from the gain calculation. However, after executing the swap, if the entering facility becomes the closest open facility to this demand location but lacks sufficient capacity to accommodate it, the demand is still lost. This effect is not captured when evaluating loss and gain in isolation.

To illustrate this, the small example needs to be adjusted slightly with three changes (see Figure 8 for visualization): demand location 2 is removed, $col[3]$ is updated from $[1] \rightarrow [2, 3, 1]$, and $col[5]$ is updated from $[2, 3] \rightarrow [3]$. These adjustments result in the following col and row:

$$row = \{0 : [0, 1], 1 : [1, 3, 4], 2 : [3, 4], 3 : [3, 5, 6, 7]\}.$$

$$col = \{0 : [0], 1 : [0, 1], 3 : [2, 3, 1], 4 : [2, 1], 5 : [3], 6 : [3], 7 : [3]\}.$$

When facilities 1 and 2 are opened, the key variables are as follows: $theoretical\_cov = [0, 1, \times, 2, 2, 0, 0, 0]$, $nearest\_opened\_fac = [-1, 1, \times, 2, 2, -1, -1, -1]$, $actual\_cov = [0, 1, \times, 1, 1, 0, 0, 0]$. This configuration achieves an objective value of 3.

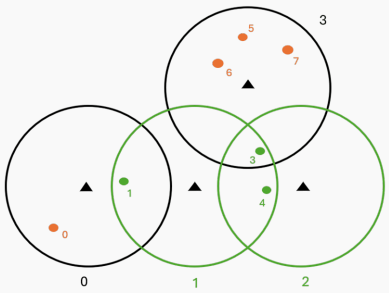


Figure 8: Visualization of the changed small example with facilities 1 and 2 open

When calculating the loss in coverage associated with facility 2, the result is 0. This is because demand locations 3 and 4 can be reassigned to facility 1, which has sufficient remaining capacity to accommodate them. Similarly, when calculating the gain for facility 3, the result is 3, as facility 3 becomes the closest open facility for demand locations 5, 6, and 7, which were previously uncovered. Note that demand location 3 remains covered by facility 2 during this calculation.

However, when the swap is executed, demand location 3 now has facility 3 as its closest open facility, but facility 3 does not have sufficient remaining capacity to cover it. Initially, demand location 3 was not considered a loss because it could be reassigned to facility 1. After the swap, due to facility 3 exceeding its capacity, this demand location must now be included as a loss. Consequently, this demand should be taken into account in the *extra* term.

In the new scenario where facilities 1 and 3 are open, the key variables are updated as follows: $theoretical\_cov = [0, 1, \times, 2, 1, 1, 1, 1]$, $nearest\_opened\_fac = [-1, 1, \times, 3, 1, 3, 3, 3]$, $actual\_cov = [0, 1, \times, 0, 1, 1, 1, 1]$.

Using only loss and gain, the expected objective value would be calculated as 3 (original objective) + 3 (gain) - 0 (loss), resulting in 6. However, the actual objective value is 5, revealing an additional effect of -1 that is not captured by loss and gain alone.

The previous examples focus on demand locations shared between the leaving and entering facilities, but this concept extends beyond shared demand locations. The gain and loss values may also fail to accurately represent a swap when the leaving and entering facilities share a common open facility.

For example, when calculating the loss for a leaving facility, it may occur that a shared open facility (also connected to the entering facility) is already operating at full capacity. In this situation, demand locations for which the leaving facility was the closest and the shared facility was the second nearest will be counted as a loss, as the shared facility cannot accommodate additional demand. However, when the entering facility is introduced, it might take over some of the demand from the shared facility, freeing up capacity. Consequently, after the swap is executed, the demand initially counted as loss can now be covered by the shared facility and is no longer lost. This effect, once again, is not captured when evaluating loss and gain in isolation.

Reconsider the original small example where facilities 1 and 2 were opened (see Figure 6 for visualization), yielding the following key variables: $theoretical\_cov = [0, 1, 1, 1, 2, 1, 0, 0]$, $nearest\_opened\_fac = [-1, 1, 1, 1, 2, 2, -1, -1]$, $actual\_cov = [0, 1, 1, 1, 1, 1, 0, 0]$.

In Section 4.3.1, it is calculated that the loss associated with facility 2 equals 2. In this case, demand location 4 will be reallocated to facility 1, but since facility 1 has no remaining capacity, it contributes to the calculated loss.

Calculating the gain for facility 0 yields the following key variables: $theoretical\_cov = [1, 2, 1, 1, 2, 1, 0, 0]$, $nearest\_opened\_fac = [0, 0, 1, 1, 2, 2, -1, -1]$, $actual\_cov = [1, 1, 1, 1, 1, 1, 0, 0]$. The gain in coverage associated with facility 0 equals 1. Here, demand location 1 is reassigned to facility 0, freeing up capacity in facility 1.

After swapping the facilities (see Figure 9 for visualization), facility 1 now has available capacity, allowing it to cover demand location 4. Consequently, this demand should be included in the *extra* term.

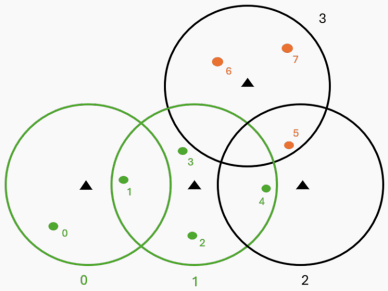


Figure 9: Visualization of the small example with facilities 0 and 1 open

In the new scenario where facilities 0 and 1 are open, the key variables are updated as follows: $theoretical\_cov = [1, 2, 1, 1, 1, 0, 0, 0]$, $nearest\_opened\_fac = [0, 0, 1, 1, 1, -1, -1, -1]$, $actual\_cov = [1, 1, 1, 1, 1, 0, 0, 0]$.

Using only loss and gain, the expected objective would be calculated as 5 (original objective) + 1 (gain) - 2 (loss), resulting in 4. However, the actual objective value is 5, revealing an additional effect of +1 that is not captured by loss and gain alone.

To capture this interaction, a third data structure, *extra*, is introduced. The value of *extra* can be negative, zero, or positive, as shown in the small examples. However, calculating this value is complex due to the capacity constraint and the closest assignment constraints, which require the construction of *actual_cov* as if the swap were executed.

To achieve this efficiently, key variables from the gain calculation are used. Since the leaving facility is part of the gain solution, the following steps are performed: All demand locations for which the leaving facility is the closest are assigned an *actual_cov* value of zero. These demand locations are then reassigned, if possible, to the second nearest open facility in the original solution, including the entering facility. For open facilities that acquire new demand locations where they become the closest, *actual_cov* is reconstructed from scratch.

To enhance efficiency in *extra* calculations, demand locations for which the leaving facility is the nearest and have a theoretical coverage of one in the solution without the entering facility but increase to a theoretical coverage of two in the solution with the entering facility can be directly assigned to the entering facility. These locations do not need to be processed through the function that determines the second nearest facility, as it is clear that the entering facility becomes their new nearest facility.

To simulate the update of all key variables as if a swap were executed, the *Extra* function is introduced, as detailed in Algorithm 14 (see Appendix). The worst-case complexity of this function is $\mathcal{O}(kl + gn)$.

After reconstructing the swap, the *extra* value can be calculated. The *extra* value for a leaving and entering facility pair is determined by evaluating the total coverage in the newly constructed scenario after the swap, subtracting the gain of the entering facility, and adding the loss of the leaving facility.

The extra values computed are stored in a matrix of size $p \times (m - p)$, providing a more efficient and sparse representation compared to a full $m \times m$ matrix. In this sparse matrix, rows correspond to open facilities and columns to unopened facilities. A complete $m \times m$ matrix would consist mainly of zeros, making it computationally inefficient. To further reduce redundant calculations, the *extra* term is calculated only for pairs of open and unopened facilities that share demand locations or have a common open facility.

To obtain specific values for a leaving facility in the *loss* vector, an entering facility in the *gain* vector, or the combined extra term in the *extra* matrix for a leaving and entering facility pair, the order of the facilities is crucial. The order of the facilities in the solution (*opened_facilities*) is preserved, while the variable *candidates* captures the order of potential entering facilities.

For example, if the leaving facility 10 has an index of 3 in *opened_facilities*, and the entering facility 12 has an index of 8 in *candidates*, the corresponding loss value is the third element of the *loss* vector, and the gain value is the eighth element of the *gain* vector. The extra value for this pair can be found in the *extra* matrix at row 3 and column 8.

After calculating the loss, gain, and extra values, the profit formula is used to identify the swap that yields the highest profit. This profit is computed as the gain value of the entering facility, minus the loss value of the leaving facility, plus the interaction term extra between the two facilities. If the resulting profit is positive, the swap is performed. If no swap results in a

positive profit, it indicates that the current solution has reached a local optimum, and the best improvement process is complete.

$$profit(f_l, f_e) = gain(f_e) - loss(f_l) + extra(f_l, f_e)$$

4.4.2 Updating *loss*, *gain* and *extra*

After executing a swap, the essential data structures must be updated. Recomputing everything from scratch would be inefficient, as most values remain unchanged between iterations. Instead, only the affected values are updated, while the rest of the calculations are retained, significantly improving efficiency. This approach is analogous to the construction phase of the greedy method, where only portions of the gain were recalculated.

Before updating the data structures, it is important to note that they are implemented sparsely. As a result, the index of the facility leaving the solution in the *loss* vector is replaced by the index of the entering facility. The same approach is applied to the *gain* vector and the *extra* matrix. These values are set to zero before updating, as they now correspond to a different facility than the original.

The question now is which values are effected in the *loss*, *gain*, and *extra* structures. To begin, consider the facilities that share demand locations with the leaving and/or entering facilities. For example, if the leaving facility (denoted as *fac_l*) leaves the solution, it affects the gain of unopened facilities and the loss of open facilities in the neighborhood.

(i) For an unopened facility, if it were to enter the solution, it could gain additional demand locations for which it would now be the nearest facility (where *fac_l* was previously closer). If these demand locations fit within the facility's capacity, this would increase its gain value.

(ii) For an opened facility, if it were to leave the solution, demand locations that are now exclusively covered by this facility would be lost (where *fac_l* was previously the second nearest facility). This would result in an increase in the loss value.

When an entering facility (denoted as *fac_e*) joins the solution, the gain and loss values for nearby facilities are likely to decrease, following reasoning similar to that of the construction phase. Therefore, at a minimum, the gain, loss and extra values of facilities that share demand locations with both the entering and leaving facilities must be updated.

Additionally, both the entering and leaving facilities have an extra interaction term with facilities of the opposite type - the entering facility has an extra term with unopened facilities, while the leaving facility has an extra term with opened facilities. In summary, the gain, loss, and extra values should be updated for all facilities that share demand locations with either the entering or leaving facility, including the facilities themselves.

As discussed previously, the gain, loss, and extra terms extend beyond simply sharing demand locations. When an opened facility (referred to as *sfac*) is linked to the entering or leaving facility, additional updates are required. This is because the capacity usage of *sfac* is likely to change - it may need to accommodate new demand locations if a leaving facility exits the solution, or it may lose demand locations if an entering facility joins the solution. Consequently, the impact on the opened and unopened facilities that share demand locations with *sfac* will differ. Therefore, these data structures must be updated for facilities linked through an open facility.

In summary, all data structures should be updated for facilities that share demand locations with the entering, leaving, or *sfac* facility. To identify these specific facilities, the function `Adjust_To_Update_BI` is introduced and outlined in Algorithm 13.

Algorithm 13: Identify the Facilities That Require Data Structure Updates

```
1  Function Adjust_To_Update_BI(in, out, col, row, opened_facilities):
2      Identify the demand locations within the service distance of the facilities in and out
         (called dps)                                                    // O(k)
3      Initialize a set impacted_fac                                     // O(1)
4      for dp in dps do                                                  // O(n)
5          Add the facilities within the service distance of dp to impacted_fac   // O(l)
6      Identify which of the facilities in impacted_fac_set are open (called
         impacted_opened_fac)                                          // O(m)
7      Initialize a set other_dps                                        // O(1)
8      for fac in impacted_opened_fac do                                 // O(p)
9          Add the demand locations within the service distance of fac to
             other_dps                                                 // O(k)
10     Initialize a set additional_impacted_fac                          // O(1)
11     for dp in other_dps do                                            // O(n)
12         Add the facilities within the service distance of dp to
             additional_impacted_fac                                   // O(l)
13     to_update = impacted_fac.union(additional_impacted_fac)           // O(m)
14     Initialize sets to_update_loss and to_update_gain                 // O(1)
15     for fac in to_update do                                           // O(m)
16         if fac in opened_facilities then
17             Add fac to to_update_loss                                 // O(1)
18         else
19             Add fac to to_update_gain                                 // O(1)
```

The worst-case complexity of the function `Adjust_To_Update_BI` is $\mathcal{O}(nl + m + pk)$, where *dps* and *other_dps* can contain at most n points, and *impacted_opened_fac* includes at most m facilities.

The complete advanced best-improvement local search approach is detailed in Algorithm 15 (see Appendix). Performing a single swap using the advanced method has a worst-case complexity of $\mathcal{O}(p(m-p)(kl+gn))$, representing a significant improvement over the simple approach.

4.5 Path Relinking

The general idea of path relinking is to find an improved solution by exploring the solution space between the local search solution and other promising solutions. Path relinking is therefore a version of an intensification strategy, focusing on researching a promising part of the solution space more intensively.

After the completion of the local search, a promising solution is randomly selected from a so-called pool of elite solutions to serve as the *guiding solution*, while the solution found in the local search serves as the *initial solution*. The initial solution is gradually transformed into the guiding solution by swapping facilities that occur only in the guiding solution with those that occur only in the initial solution, and the objective value is evaluated for each intermediate solu-

tion. This process is repeated until the initial solution becomes identical to the guiding solution. The solution with the highest objective during this process is selected as the final result of the path relinking. This solution will also be considered as a candidate for the pool of elite solutions.

A candidate is allowed to enter the pool of elite solutions if it satisfies one of the following decision rules:

1. The maximum capacity of the pool has not been reached yet.
2. The candidate solution is the best solution found so far, it will replace the solution in the pool with the lowest objective value.
3. The candidate solution is sufficiently different from all existing elite solutions (e.g., differing in 20% of the facilities), and it has a better objective value than the elite solution with the worst objective value. The candidate replaces its most similar solution in the subset of elite solutions that are of no better quality than the candidate.
4. The candidate solution is not sufficiently different from at least one of the existing elite solutions, but it has a better objective value than one of these similar solutions. The candidate solution replaces the worst similar solution in the pool.

These decision rules are introduced to ensure that the pool of elite solutions remains valuable by including both high-quality and diverse solutions. Resende and Ribeiro [32] specified the first three rules as standard selection criteria in GRASP for candidate solutions, with the third rule specifically aimed at maintaining diversity in the elite pool. Theulen [5] formulated the fourth rule to allow the inclusion of candidate solutions with better objective values than similar solutions already in the pool. By replacing one of these similar solutions, the diversity of the pool remains unaffected.

The standard path-relinking approach has several variations. The described path-relinking corresponds to the *forward path-relinking*, where the GRASP solution serves as the initial solution and the elite solution as the guiding solution. Conversely, in *backward path-relinking*, these roles are reversed. This reversal allows for a more thorough exploration of the solution space around the elite solution, whereas forward path-relinking emphasizes the solution space originating from the GRASP solution. Experiments conducted by Resende and Ribeiro [30] demonstrated that backward path-relinking produced better results than the standard forward approach.

An extension to these methods is the *back and forward path-relinking* strategy, where a backward path-relinking is first applied, followed by forward path-relinking [17]. By definition, this approach identifies solutions that are at least as good as those found using either backward or forward path-relinking individually. However, it requires roughly double the computational time compared to either method alone.

To mitigate this computational overhead, *mixed path-relinking* was proposed as a more efficient alternative, initially suggested by Glover [18]. This approach retains the benefits of exploring both neighborhoods, similar to back-and-forward path-relinking, but achieves it with a computational effort comparable to forward or backward path-relinking alone. Mixed path-relinking alternates the roles of the initial and guiding solutions at each step. This alternation creates two paths: one progressing from the initial solution to the guiding solution, and the other from the guiding solution to the updated initial solution. These paths converge in the middle, effectively linking the original initial and the guiding solution, while allowing a comprehensive exploration of both neighborhoods. Ribeiro and Rosseti [34] implemented and tested this approach, demonstrating that mixed path-relinking not only matched the computational efficiency of the forward strategy but also outperformed forward, backward, and back-and-forward path-relinking in identifying optimal solutions.

All described variations of path-relinking will be evaluated in Section 5 using the Timor-Leste dataset.

Implementation of Path-Relinking
Path-relinking can be easily implemented once the simple and advanced best-improvement local search methods are in place, as the underlying mechanics are quite similar. However, there are two key differences between these approaches:

- Stopping criterion: In path-relinking, the best swap is always accepted, even if it results in a decrease in the objective value, and the process continues until the initial solution becomes identical to the guiding solution. This contrasts with the best-improvement local search, which stops when no profitable swaps remain, thereby reaching a local optimum.
- Candidate moves: Path-relinking permits only those swaps in which the leaving facility is not part of the guiding solution, and the entering facility is part of the guiding solution. By selecting and executing the best move from this constrained set, the initial solution incrementally moves closer to the guiding solution. This contrasts with the best-improvement local search, which allows all swaps, considering any move that improves the objective value.

There are both a simple and an advanced approach for the best improvement local search, implying that the path-relinking process can also be implemented in two corresponding variants. By applying the described adjustments, the simple best-improvement approach easily transforms into the simple path-relinking process.

The advanced path-relinking process is more complex and requires additional adjustments.

First, introduce the *candidate* vector, which is initialized with all facilities present in the guiding solution but absent from the initial solution. Similarly, create the *leaving* vector, which includes all facilities present in the initial solution but absent from the guiding solution.

At the start of the algorithm, the data structures *loss*, *gain*, *extra*, and $near\_fac\_dict$ are initialized and calculated only for the facilities in the *candidate* and *leaving* vectors. If the number of swaps required is w, the *loss* and *gain* vectors will each have a length of w, and the *extra* matrix will have a size of $w \times w$. As before, the order of facilities in *leaving* and *candidate* is preserved to maintain consistency.

Deleting facilities directly from *leaving* and *candidate* after a swap complicates indexing. Instead, boolean vectors of the same size as *leaving* and *candidate* are used to track deletions. When a facility from *leaving* is removed from the solution, its corresponding boolean value is set to `False`. This ensures that accessing '*leaving*[boolean]' provides the correct subset of facilities as if the element were physically deleted. The same logic applies to the facilities in *candidate*. These boolean vectors are also used for the data structures *loss*, *gain*, and *extra*, enabling efficient calculation of the profit and identification of the best new swaps.

Furthermore, the $Adjust\_To\_Update$ procedure in Algorithm 13 requires adjustments. Previously, the facilities were added to to_update_loss if they were in $opened\_facilities$. Now, facilities are added to to_update_loss only if they are in *leaving*. Similarly, for to_update_gain, facilities are added only if they are in *candidate*. These refinements ensure the correct updating process tailored for advanced path-relinking.

5 Application and Results

5.1 Data

This thesis examines the optimization of the placements of health facilities in Timor-Leste, employing methodologies that can be generalized to address other critical infrastructure needs, such as the location of wells or schools, or applied to different regions. Given that data often differ from reality and continually improve, especially in developing countries, the model is designed to be highly adaptable. This allows for the integration of updated data as they become available, ensuring that the results remain relevant and reflective of current realities.

The Democratic Republic of Timor-Leste, a developing country located between Indonesia and Australia, gained independence in 2002 [24]. With a population of 1.36 million, 70% of whom reside in rural areas, access to healthcare remains a significant challenge [40]. Rural regions face stark underdevelopment in healthcare infrastructure, while the national hospital in Dili struggles with overcrowded maternity wards, limited bed capacity, and insufficient resources [12]. These issues extend to critical services such as stroke care, contributing to preventable mortality, with 13.09% of all deaths attributed to stroke [9].

To address these challenges, optimization techniques, such as the capacitated maximum covering location problem offer an effective framework for improving the placement of health centers. These approaches enhance healthcare access and enable equitable allocation of resources.

5.1.1 Data Collection and Processing

This subsection details the data collection and processing steps used, including geographic boundaries, healthcare facilities, population distribution, and potential facility locations.

Geospatial Administrative Boundaries
The administrative boundaries of Timor-Leste are obtained from the Global Administrative Areas (GADM) database, which provides high-resolution shapefiles. This study focuses on first-level administrative regions, such as districts such as Baucau and Dili. By combining these first-level administrative regions, it is possible to form the boundaries of Timor-Leste as needed, with the flexibility to exclude certain areas for separate optimization.

Healthcare Facility Locations
Data for healthcare facilities, including hospitals and clinics, are obtained from OpenStreetMap (OSM) using the Overpass API. This dataset provides the exact coordinates (latitude and longitude) of healthcare facilities in Timor-Leste, enabling geographic analysis of their proximity to demand locations.

When solving the model in a real-world scenario, existing facilities are considered fixed resources, which naturally influence the optimal solution instead of approaching the model from scratch.

Population Distribution Data
Population distribution data comes from the 2020 Facebook Population Data, offering detailed population estimates at a granular level throughout Timor-Leste. These data are essential for visualizing demand locations and mapping population density within each administrative region. Each data point includes geographic coordinates and corresponding population sizes, which helps to analyze healthcare access based on population coverage.

Potential Facility Locations (Grid-Based Approach)
A grid-based approach is used to identify potential locations for new healthcare facilities. The region of Timor-Leste is divided into equally spaced cells using a spatial grid. Each cell represents a possible facility location, and these locations are then evaluated based on their proximity to nearby populations and their potential to improve healthcare access.

For the Timor-Leste data, three separate problems are analyzed based on the spacing of potential facility locations: the grid points are placed at intervals of 2 km, 3 km, or 4 km. Each spacing represents a distinct problem, offering different levels of spatial detail for the analysis.

5.1.2 Technical Aspects of Distance Calculations

Rather than relying on straight-line (Euclidean) distances, our analysis incorporates road network data to calculate actual travel distances between healthcare facilities and demand locations. This road network data comes from *OpenStreetMap*, offering detailed information on roads and travel routes within Timor-Leste.

Once road information is available, the *pyrosm* library is used to construct a network from the geographic data. The *pyrosm* library provides an efficient method to process *OpenStreetMap* data into a network that can be used for further analysis. Roads are converted into a network structure comprising nodes (intersections) and edges (connections). This data is then used to create a network graph, specifically using the *Pandana* framework, which is well suited to perform route calculations and determine the accessibility of specific locations.

Before distances could be calculated, the nearest nodes for both demand locations and potential facilities had to be determined. This ensures that each population and facility point is linked to the nearest node in the network. This step is crucial because distance calculations within the network take place between nodes. By connecting the population and facility locations to specific nodes, distances within the network model can be accurately determined. Additionally, the Haversine distance is used to determine the distance between the original locations (population and facility locations) and their respective nearest nodes.

Subsequently, the road distance between different nodes can be calculated using the shortest path calculation method. This function employs algorithms such as Dijkstra to determine the minimum travel distance, taking into account the actual structure of the road network.

To determine the actual distance between the demand locations and the facility locations, the calculation involves summing the following: the distance from the demand location to its nearest node, the distance from the facility location to its nearest node, and the distance between the respective nodes. The result is a road distance matrix, where each entry represents the shortest travel distance between a healthcare facility and a demand point, reflecting real-world conditions and existing road infrastructure.

5.1.3 Output

When applying the explained techniques to obtain the data from Timor-Leste, the following analyses can be performed.

For this case, the following areas are excluded from the analysis:

- Oecusse (or Oecusse-Ambeno): This is the isolated part of Timor-Leste, located on the western side of the island. It is surrounded by Indonesian territory.
- Atauro: The island is located to the north of the main part of Timor-Leste.

These areas can be optimized individually, as the service distances of the facilities within these areas do not overlap with those of the main area and, therefore, do not share demand locations.

For the rest of Timor-Leste, according to Facebook data, there are approximately 1,236,452 citizens living in these regions. These citizens are grouped into 151,765 demand locations, each with a specific demand. In addition, 128 hospitals, clinics, and other health centers have been identified.

When implementing the distance calculations, there are 452,907 nodes that represent possible demand and facility locations, and 914,500 edges available to determine the shortest paths between them.

Table 3 summarizes the data characteristics for different grid sizes and service distances. The parameter $|M|$ represents the number of potential facility locations, determined by the size of the grid, excluding existing facilities. After constructing the key variables *row* and *col* (based on the service distance), the number of facilities within the service area of each demand location, and vice versa, can be calculated. The averages of these values are represented by μ_{col} and μ_{row}, respectively. Additionally, the previously defined parameters $k = \max |row|$ and $l = \max |col|$ are included in the table. The sparsity metric, also presented, indicates the percentage of feasible facility-demand pairs relative to the total possible combinations.

Grid	$\|M\|$	Service Distance	μ_{col}	μ_{row}	k	l	Sparsity
4 km	881	5 km	1.6	246.6	4	13,621	0.152%
		10 km	4.6	802.2	12	20,329	0.515%
3 km	1576	5 km	2.6	254.3	7	11,831	0.156%
		10 km	8.4	809.3	22	20,130	0.521%
2 km	3539	5 km	5.5	250.5	15	14,919	0.154%
		10 km	18.7	813.1	45	20,394	0.523%

Table 3: Characteristics Data Timor-Leste

As previously mentioned, in real-world scenarios, the number of potential facility locations capable of covering a demand location, and vice versa, is significantly smaller than the total available. For example, in this case, there are 151,765 demand locations (n), and the 4 km grid contains 881 potential facility locations (m). Within a 5 km service distance, $l = 13,621$ (a small fraction of 151,765) and $k = 4$ (considerably less than 881).

In this thesis, unless otherwise stated, the capacity parameter K is calculated as the total demand divided by the number of open facilities, multiplied by 1.5 (and rounded). Table 4 summarizes the relationship between the number of open facilities (p) and the corresponding capacity.

p	80	110	128	140	170	200
K	23183	16867	14499	13248	10917	9273

Table 4: Capacities for various p values

Choosing this capacity may sometimes lead to counterintuitive results, as increasing p does not always result in higher coverage. However, it is important to note that any capacity value can be freely selected.

This thesis focuses on evaluating the performance of the heuristic, particularly its ability to solve the model for smaller grid sizes. The analysis is carried out separately for each p value, avoiding comparisons with different p values. This approach ensures that the dependency of the capacity on p does not affect the validity of the analysis or conclusions.

Real-life scenario
In the analysis of the real-life scenario in Timor-Leste, the model accounts for existing facilities by including their locations in the set of potential facility locations. This adjustment adds 128 existing facilities to the model, which are required to be open.

5.2 Results MIP Solver

This section evaluates the MIP solver using the Timor-Leste dataset. The analysis begins with an assessment of the model refinements and efficiency improvements introduced in Section 3.2, applied to the 4 km grid, constructing the facilities from scratch. This subsection focuses on runtime performance and the identification of the most efficient model for implementation. In the subsequent subsection, after selecting the optimal model, the solver calculates the optimal coverage for each value of p and service distance. This analysis is performed for both the constructed-from-scratch model and the real-life scenario on the 4 km grid, as well as for the 3 km grid.

Note that since the capacity depends on the number of facilities opened, solving the model for larger p values may sometimes simplify the problem, potentially reducing runtime.

5.2.1 Evaluation Model Refinement and Efficiency Improvements

The evaluation of the model refinements and efficiency improvements begins with the most basic version of the model, as outlined in the introduction. The refinements are then incrementally incorporated, and each modification is retained if it demonstrates an improvement.

Before starting the analysis, we want to note that the log files are consistently structured with the same columns throughout. This means some columns may contain identical values or may not be particularly relevant for certain enhancements. However, this uniformity is maintained to ensure consistency.

Reducing Decision Variables
The first enhancement of the solver focuses both on improving computational efficiency and addressing the feasibility of running the model on large-scale data. This is particularly crucial for the Timor-Leste dataset, which includes 151,765 demand locations and 881 potential facility locations with a 4 km grid. In the original model, the decision variables (y) were defined for every possible combination of the demand and facility locations. Additionally, variables were set to zero if the distance between a demand location and a facility exceeded the service distance threshold. However, this approach resulted in a model that was impractical to run on standard hardware. For example, the number of decision variables was $881 + 881 \times 151,765$, leading to nearly 134 million decision variables. This creates significant memory and CPU constraints, making the model intractable on a laptop with 16 GB of RAM.

To overcome these limitations, the model was refined to introduce decision variables only for feasible pairs of demand and facility locations, significantly reducing the size of the problem. For example, with a 5 km service distance, only 0.152% of the original demand-facility pairs are feasible, while for a 10 km service distance, this increases slightly to 0.515%. These percentages align with the sparsity discussed earlier. This adjustment not only reduces the number of decision variables but also decreases the number of constraints, resulting in a much more manageable model.

For a 5 km service distance, this enhancement allows the model to find optimal solutions within a short time. For example, with $p = 140$, the solution time improved from being unsolvable within two hours to finding an optimal solution in just 115 seconds, a significant improvement. The runtime results for the 5 km service distance are represented by the pink line labeled 'Bi-

nary' in Figure 10a. However, for a 10 km service distance, this enhancement alone remains insufficient to find an optimal solution within two hours.

Examining the logfile generated by the Gurobi solver for a 5 km service distance and 4 km grid size reveals that the model starts with 0 continuous variables and 204,179 integer variables, all binary. The optimization involves 333,080 rows, 204,179 columns, and 697,343 non-zero elements. Gurobi begins with a presolve phase, followed by the Branch-and-Bound and Simplex methods to find the optimal solution.

Presolve is a critical step in the gurobi solver, involving routines that eliminate redundant information and strengthen the model's formulation, thereby enhancing solution efficiency. Here, the focus is on root presolve, which is performed before solving the first linear programming relaxation [2, 3].

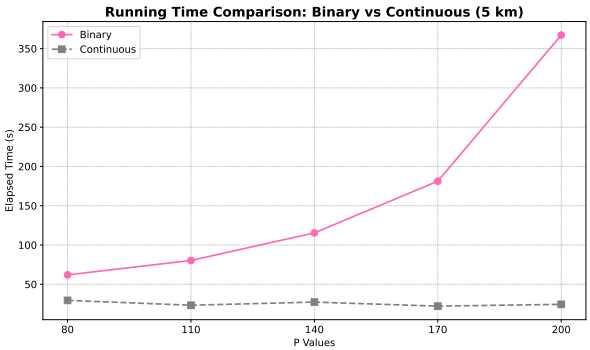
Table 5 summarizes the presolve phase results, including the number of nodes explored during Branch-and-Bound and the Simplex iterations required to reach the optimal solution.

p	Rows	Columns	Nonzeros	Continuous Var.	Integer Var.	Explored Nodes	Simplex Iterations
80	139036	140163	439276	0	140163 (139222 binary)	1	66555
110	139036	140163	439276	0	140163 (139222 binary)	1	85223
140	139016	140143	439216	0	140143 (139202 binary)	1	119744
170	139016	140143	439216	0	140143 (139202 binary)	529	127905
200	139016	140143	439216	0	140143 (139202 binary)	1405	280282

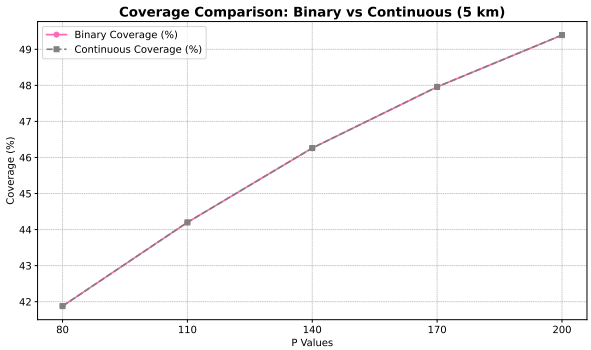
Table 5: Logfile of the binary model with a 5 km service distance.

Converting Allocation Decision Variables y from Binary to Fractional
Relaxing the allocation decision variables y from binary to fractional significantly improves computational efficiency for both the 5 km and 10 km service distances. This relaxation reduces the complexity of the optimization process, resulting in substantial runtime improvements.

Figure 10a illustrates the runtime improvements for the 5 km service distance across various p values. The results clearly show the efficiency gains achieved through the relaxation of the model. In addition, Figure 10b compares the percentage coverage achieved by binary and continuous models across the same p values within the 5 km grid. Notably, the coverage results for the binary and continuous models are closely aligned.



(a) Running times for binary and continuous models.



(b) Percentage coverage achieved by binary and continuous models.

Figure 10: Comparison between the binary and continuous model with a 5 km service distance.

These findings emphasize the practicality of model relaxation as a strategy to enhance computational performance. By allowing fractional allocation, the method effectively reduces runtime while retaining comparable coverage outcomes, underscoring its value in large-scale applications.

The enhancements are even more significant for the 10 km service distance. Previously, certain instances were computationally intractable. However, with this relaxation, they can now achieve an optimal solution within a practical time frame, such as two hours. The runtimes are depicted by the grey line labeled 'Basic Model' in Figure 11b.

Further insights into the computational improvements are provided by the Gurobi solver logfile. For a 5 km service distance, the model initially has the same number of rows, columns, and non-zero elements as the binary case. However, in this relaxed instance, it includes 203,361 continuous variables alongside 818 binary integer variables, greatly enhancing the root presolve phase compared to the fully binary model.

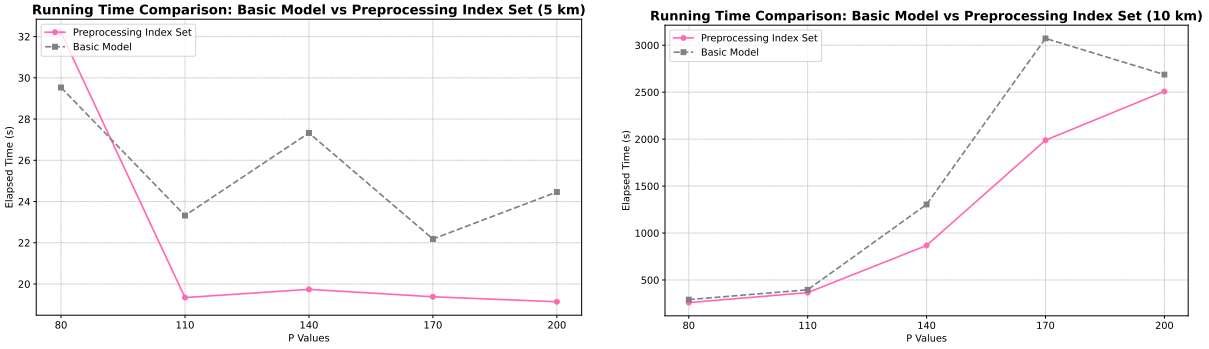
As summarized in Table 6, the root presolve process significantly reduces the size of the model. The number of continuous variables decreases from 203,361 to just 4,869, while corresponding reductions occur in the rows and non-zero elements. This drastic reduction in problem size substantially increases computational efficiency, minimizing the number of simplex iterations required and eliminating the need for a branch-and-bound phase for $p = 170$ and 200.

p	Rows	Columns	Nonzeros	Continuous Var.	Integer Var.	Explored Nodes	Simplex Iterations
80	5067	5695	16353	4869	826 (826 binary)	1	3844
110	5064	5692	16344	4866	826 (826 binary)	1	4039
140	5064	5692	16344	4866	826 (826 binary)	1	3898
170	5061	5689	16335	4863	826 (826 binary)	1	3815
200	5061	5689	16335	4863	826 (826 binary)	1	3523

Table 6: Logfile of the continuous model with a 5 km service distance.

Preprosessing Index Sets
Preprocessing the indices for the decision variables significantly enhances runtime, with its impact becoming more pronounced as the problem size grows.

For a 5 km service distance, the effect is less evident, as the preprocessing runtime is not consistently faster across all p values, with the runtime being slower for $p = 80$. However, for the 10 km service distance, preprocessing consistently reduces runtime compared to the basic model without preprocessing, as shown in Figures 11a and 11b.



(a) Results for 5 km service distance.

(b) Results for 10 km service distance.

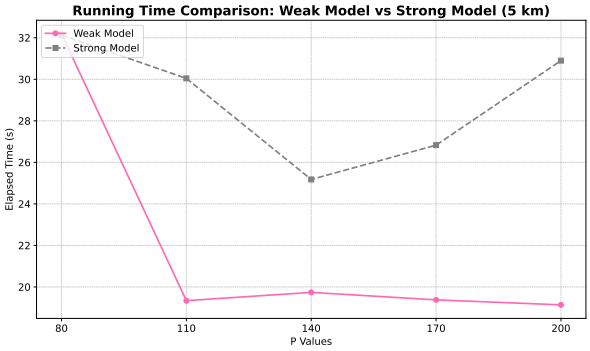
Figure 11: Comparison of the running times between the basic and preprosessing model.

To determine whether preprocessing enhances the basic model, the same analysis was performed on the 3 km grid with a 5 km service distance. In this case, preprocessing resulted in faster solver performance across all p values. Given the aim of applying the solver to larger models, preprocessing the indices of the decision variables will be integrated into the basic model.

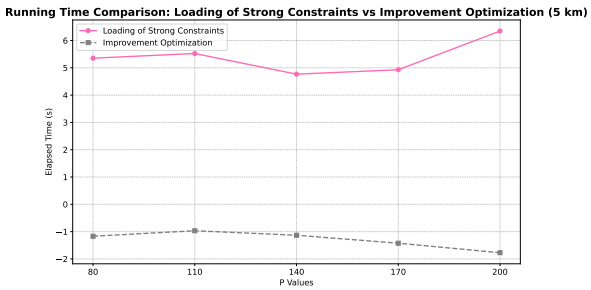
It is worth noting that the logfile here is similar to Table 6. Preprocessing the index set does not change the structure of the model but improves runtime by streamlining the loading of variables and constraints.

Defining Weak and Strong Versions of the Model
As discussed previously, the performance of the strong model compared to the weak model depends significantly on the size of the model. This observation is confirmed by our dataset.

For the 5 km grid, representing a smaller model, the weak model outperforms the strong model. Here, the additional constraints in the strong model do not reduce runtime in the optimization as intended. Instead, the overhead of loading and processing these constraints leads to an increase in overall runtime, as illustrated in Figures 12a and 12b.



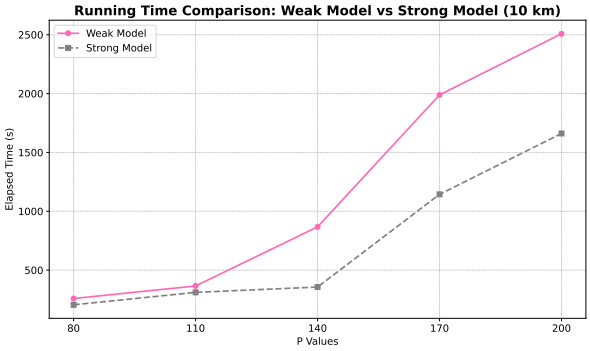
(a) Comparison running times



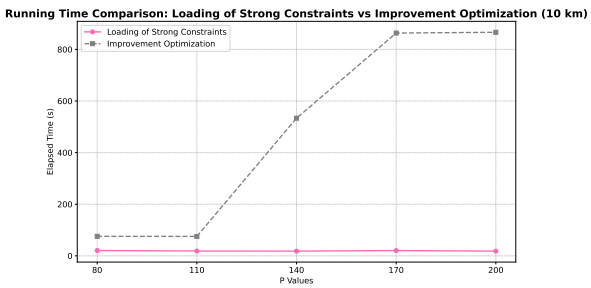
(b) Running time of loading the strong constraints and the improvement in optimization

Figure 12: Comparison between the weak and strong model with a 5 km service distance.

However, for the 10 km grid, the strong model emerges as the better option. As p increases, optimization benefits from a considerable reduction in runtime that more than compensates for the extra time required to load the strong constraints. These results are shown in Figures 13a and 13b.



(a) Comparison running times



(b) Running time of loading the strong constraints and the improvement in optimization

Figure 13: Comparison between the weak and strong model with a 10 km service distance.

To evaluate whether the weak or strong model demonstrates greater effectiveness, the same analysis was applied to the 3 km grid with a 5 km service distance. In this case, the weak model still performs consistently faster. Based on these findings, the weak model is preferable for the 5 km service distance, while the strong model demonstrates advantages for the 10 km service distance.

The Gurobi solver logfile provides additional insight into the differences between the two models and explains why the weak model outperforms the strong model for the 5 km service distance. For the strong model, Gurobi initializes a problem with 536,441 rows, 204,179 columns, and

1,104,065 nonzero elements, comprising 203,361 continuous variables and 818 binary integer variables. The additional constraints in the strong model result in a significant increase in the number of rows, thereby raising the model's complexity compared to the weak model.

Although presolve effectively reduces the size of both models, as shown in Table 7, the strong model retains a higher number of rows due to its additional constraints. Additionally, the decrease in continuous variables within the strong model is almost entirely offset by a corresponding increase in integer variables. This shift results in a comparable number of simplex iterations, or slightly more. As a result, the strong model exhibits longer optimization times, as depicted in Figure 12b.

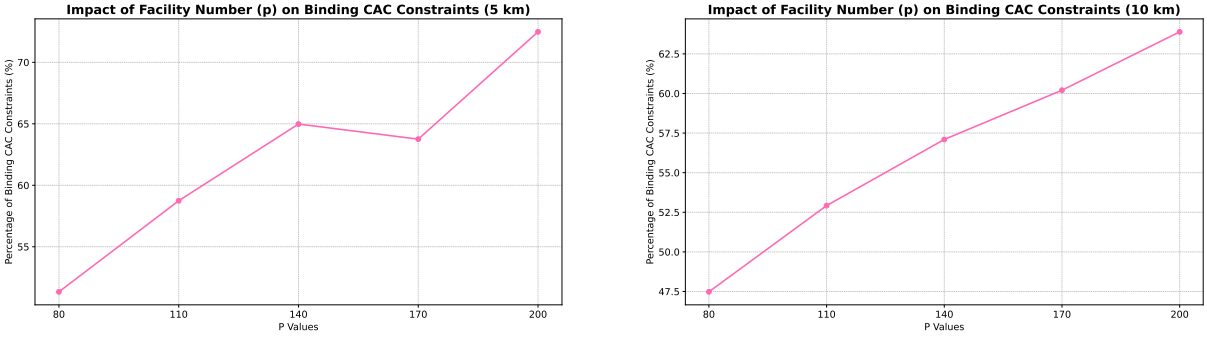
p	Rows	Columns	Nonzeros	Continuous Var.	Integer Var.	Explored Nodes	Simplex Iterations
80	9139	5497	22083	3357	2140 (2140 binary)	1	4064
110	9139	5497	22083	3357	2140 (2140 binary)	1	3941
140	9139	5497	22083	3357	2140 (2140 binary)	1	3827
170	9139	5497	22083	3357	2140 (2140 binary)	1	3822
200	9139	5497	22083	3357	2140 (2140 binary)	1	4840

Table 7: Logfile of the strong model with a service distance of 5 km.

When reviewing the logfiles for the 10 km service distance, specifically for $p = 200$, several key observations emerge. The strong model initially starts with a significantly larger problem size, ranging from 838,012 to 1,526,633 rows and 3,588,884 to 4,965,526 non-zeros. Although presolve reduces the size of both models, the strong model remains slightly larger. Despite this, the solver performance shows a clear advantage for the strong model. While the weak model explores 26,622 nodes and requires 5,512,909 simplex iterations, the strong model explores only 15,736 nodes with 2,609,267 simplex iterations, demonstrating a more efficient solving process.

Comparing Closest Assignment Constraints
In the basic model, the Wagner and Falkson approach to the closest assignment constraints is implemented by preprocessing the set S_{ij} into a dictionary, which is then used during constraint construction. The enhanced version of Cánovas introduces an additional set $G_{ikj} = \{a \in col[i] \cap col[k] \mid d_{ia} \leq d_{ij}, d_{ka>kj}\}$. However, preprocessing this set leads to a memory error in the solver, even for a 4 km grid with a service distance of 5. This outcome aligns with the expected increase in the memory requirements for the enhanced constraints. Consequently, for the Timor-Leste data and the selected parameters, this constraint cannot be applied.

Lazy Constraints on Closest Assignment Constraints
Lazy constraints are most effective when the model has a low proportion of binding constraints. To assess this, the basic model is analyzed at both service distances, as illustrated in Figure 14.

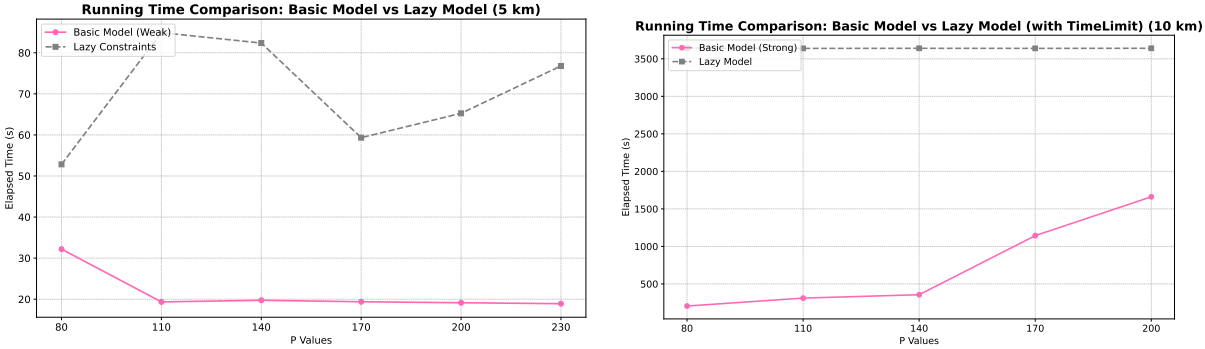


(a) Results for 5 km service distance.

(b) Results for 10 km service distance.

Figure 14: Percentage of binding CAC constraints.

Figures 14a and 14b indicate that the percentage of binding constraints in this model is quite high. Consequently, lazy constraints are unlikely to enhance the running time. This conclusion is confirmed by the results obtained from running the lazy model for both service distances. For the 10 km service distance, a one-hour time limit was imposed.



(a) Results for 5 km service distance.

(b) Results for 10 km service distance.

Figure 15: Comparison of the running times between the basic and lazy model.

The lack of improvement in running time when using lazy constraints in this model can be best understood by examining the log file for the 5 km service distance, as detailed in Table 8. To begin, it is essential to clarify the role of lazy constraints in optimization.

Lazy constraints are excluded from the presolve process and become active only during the Branch-and-Bound procedure. During this phase, a solution is considered feasible if it satisfies the closest assignment constraints. If any of these constraints are violated, the specific violated constraints are added to the corresponding branch of the search tree.

In this problem, the exclusion of closest assignment constraints significantly undermines the effectiveness of presolve. Without these constraints, the presolve process is unable to effectively simplify the model, leading to a substantially larger problem size. Specifically, the problem size grows from 9,139 rows to 64,677 rows, with corresponding increases in the number of columns and nonzero elements. The variable count also rises drastically, from 5,497 (3,357 + 2,140) variables to 204,179 (203,361 + 818) variables. As a result, the problem remains much larger after presolve compared to scenarios where the closest assignment constraints are included.

This inefficiency is not confined to the presolve process but also extends into the optimization phase. With closest assignment constraints, the model completes optimization without requiring Branch-and-Bound, resolving the problem with approximately 4,000 simplex iterations. In contrast, when lazy constraints are employed (for $p = 110$), the Branch-and-Bound procedure explores 154 nodes, necessitating a significantly higher number of simplex iterations.

p	Rows	Columns	Nonzeros	Continuous Var.	Integer Var.	Explored Nodes	Simplex Iterations
80	64677	204179	343316	203361	818 (818 binary)	34	57594
110	64677	204179	343316	203361	818 (818 binary)	154	99051
140	64677	204179	343316	203361	818 (818 binary)	127	123918
170	64677	204179	343316	203361	818 (818 binary)	1	93861
200	64677	204179	343316	203361	818 (818 binary)	13	95908

Table 8: Logfile for the lazy constraint model with a service distance of 5 km.

5.2.2 Results of the MIP Solver for Timor-Leste

In this subsection, the optimal placement of facility locations in Timor-Leste is analyzed using the finalized Mixed-Integer Programming (MIP) solver. The solver integrates several enhancements, including the reduction of decision variables, conversion of allocation decision variables y from binary to fractional, preprocessing of the index set, and considering both weak (5 km service distance) and strong (10 km service distance) models. Two distinct approaches are explored:

- From Scratch Optimization: Assumes no existing facilities, letting the model freely select optimal locations, offering an idealized solution.
- Real-Life Scenario Optimization: Considers existing facilities in Timor-Leste as fixed, optimizing within practical constraints.

By combining these two approaches, the study aims to understand both the ideal configurations for facility placement and how those might be adapted to real-world constraints.

All results are presented as percentage coverage. Any reference to an increase in coverage refers to the absolute increase.

From Scratch Optimization
This analysis considers a scenario without preexisting facilities, allowing the identification of optimal locations without constraints imposed by existing infrastructure.

For the Timor-Leste dataset, optimal solutions have been analyzed using a 4 km grid resolution. This resolution allows the model to compute optimal solutions for all parameters, service distance and p, within a two-hour timeframe. For a finer resolution of 3 km, the optimal values are calculated using an optimality cap. In the largest scenario, with a service distance of 10 km and $p = 200$, an optimality gap of 1% was achieved in approximately two hours. This optimality cap is consistently applied across all 3 km scenarios to maintain uniformity in the approach.

It is worth mentioning that the 1% optimality gap is related to the objective value rather than directly to the percentage of coverage.

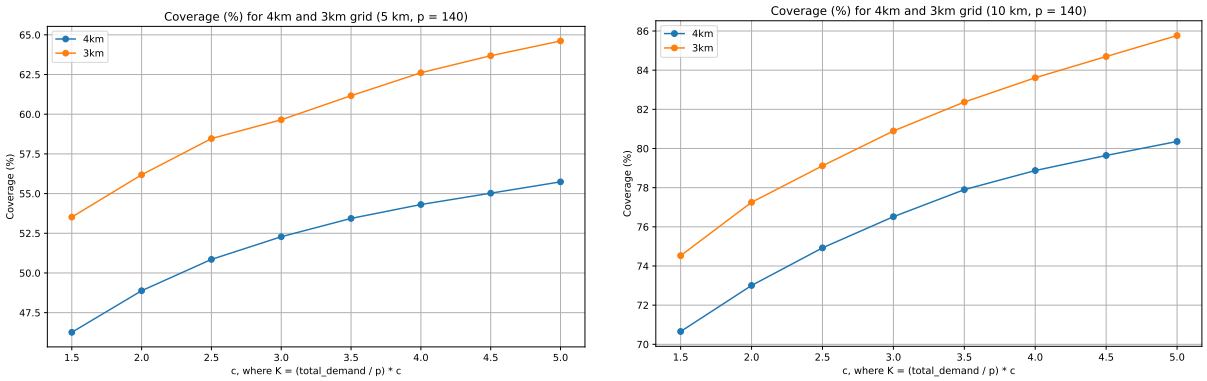
		grid-size = 4km		grid-size = 3km	
S	p	Opt_4	Time_4 (s)	Opt_3^*	Time_3 (s)
5	80	41.877	32.2	48.302	58.2
	110	44.199	19.3	51.094	49.2
	140	46.261	19.7	53.393	53.5
	170	47.957	19.4	55.491	58.9
	200	49.394	19.1	57.806	53.7
10	80	65.399	205.6	69.733	1348.8
	110	68.603	311.1	72.915	2478.9
	140	70.654	356.3	74.544	6326.2
	170	71.404	1143.9	74.965	4701.3
	200	71.273	1661.0	75.110	7282.4

Table 9: Optimal coverage of the From Scratch Optimization.

The results show that, for the 3 km grid, the coverage increases by at least 3.561% and at most 8.412%. This improvement can be attributed to two key factors. First, facilities can be placed more strategically, such as closer to the center of densely populated areas. Second, the smaller grid allows facilities to be placed closer to each other, which is especially beneficial in cities where demand locations are concentrated. In contrast, with a 4 km grid, many demand locations remain uncovered due to capacity limitations, as facilities are placed further apart. This

demonstrates the importance of having multiple facilities to effectively serve densely populated areas, rather than relying on a single facility (a common outcome when capacity constraints are ignored). Notably, this effect is observed not only at low capacity levels, but also at higher capacities.

In short, the impact of increasing capacity to exceed total demand divided by the number of facilities opened, scaled by a factor $c > 1.5$, will be highlighted. The results, presented in Figure 16 for $p = 140$, demonstrate a trend that is consistent between all the tested p values. Additionally, it is important to emphasize that the 3 km grid solution is achieved with an optimality gap of 1%. This implies that the observed differences could be slightly underestimated.



(a) Results for 5 km service distance.

(b) Results for 10 km service distance.

Figure 16: Comparison of coverage between 4 km and 3 km grids for different capacities.

However, when the capacity becomes extremely large, approaching an uncapacitated model, a smaller grid might achieve higher coverage. This could occur, for instance, if the 4 km grid happens to have a point precisely at the center of a major city.

An additional observation is that the smaller grid becomes more effective as p increases, due to the corresponding reduction in the capacity of the facility. This allows facilities to be placed closer to each other on the smaller grid.

Real-life Scenario Optimization

For the real-life scenario, only the p values of 128 (representing coverage with existing facilities), 140, 170, and 200 are considered. The corresponding results are shown in Table 10. Here, the model for the 3 km is solved optimally as this is possible for all possible p-values in approximately two hours, with a slight margin over.

		grid-size = 4km		grid-size = 3km	
S	p	Opt_4	Time_4 (s)	Opt_3	Time_3 (s)
5	128	41.524	-	41.524	-
	140	49.589	42.9	50.477	65.5
	170	54.042	42.4	57.119	73.1
	200	56.307	40.9	60.247	82.2
10	128	57.311	-	57.311	-
	140	66.385	140.7	66.738	429.4
	170	73.311	174.0	74.339	3553.2
	200	74.766	186.0	76.471	8984.4

Table 10: Optimal coverage of the Real-Life Scenario Optimization.

For a 3 km grid, the coverage increases by at least 0.353% when considering the existing situation, and this difference becomes larger when the solver allows placing more new facilities, particularly for $p = 170$ and $p = 200$.

This increase is noticeably smaller compared to the 'From Scratch Optimization'. This can be explained by the fact that, for instance, when $p = 140$, the heuristic in the 'From Scratch Optimization' can select 140 facilities. In contrast, in the real-life scenario, $p = 140$ corresponds to opening only 12 additional facilities, meaning only these 12 facilities can benefit from the smaller grid size.

5.3 Results Heuristic

Up to this point, Section 4 has detailed all aspects and potential configurations of the heuristic. The optimal approach for each phase depends on the specific dataset and the number of GRASP iterations. To provide further insights into each phase, a separate analysis will be performed for each phase using the Timor-Leste dataset. The results will be presented for two scenarios, both starting from scratch: (i) a grid size of 4 km with a service distance of 10 and $p = 140$, and (ii) a grid size of 2 km with a service distance of 5 and $p = 80$.

To determine the final heuristic, including the optimal parameters for the construction phase, the local search strategy, and the path-relinking configuration, a comprehensive analysis will be conducted using the 3 and 4 km grid.

5.3.1 Construction Phase

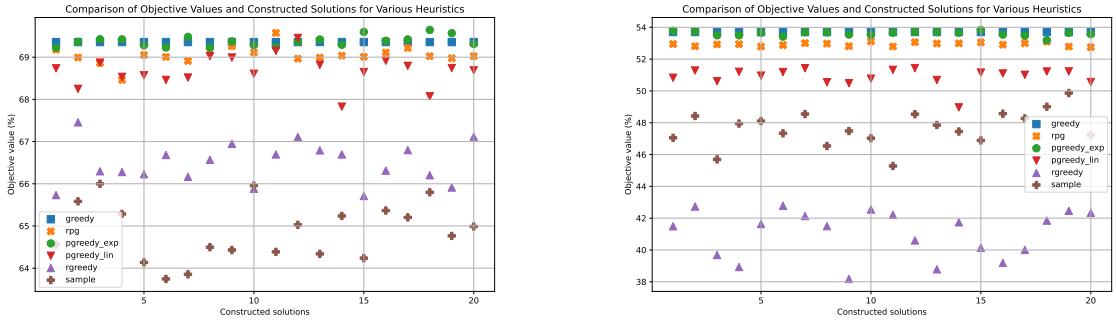
An effective construction phase strikes a balance between generating high-quality solutions, maintaining computational efficiency, and ensuring sufficient variation across iterations. Adequate variation prevents the local search from repeatedly converging to the same local optimum, thereby enhancing the effectiveness of path-relinking between distinct optima. Additionally, high-quality initial solutions improve the efficiency of the local search by enabling faster convergence to a local optimum. The quality and diversity of solutions produced during the construction phase depend on the methods and parameters employed.

In Figure 17, the impact of different construction methods on the objective value is illustrated for the two scenarios. Table 11 provides the corresponding maximum, average and standard deviation of the objective values, along with the average overlap between two solutions (μ_Λ) and the average time required to construct the solutions. For the *pgreedyexp*, *pgreedylin*, and *rgreedy* methods, $\alpha = 0.05$ was applied. In the case of *rpg*, both α and β were set to 0.05, while the sample-based method used $q = 0.02 \times |M|$.

The *random* approach is excluded from testing because it has been observed that fully random solutions result in insufficient quality, making the local search process overly time-consuming.

	4 km grid, $S = 10$, $p = 140$					2 km grid, $S = 5$, $p = 80$				
	max_obj	μ_{obj}	σ_{obj}	μ_Λ	μ_{time}	max_obj	μ_{obj}	σ_{obj}	μ_Λ	μ_{time}
greedy	69.36	69.36	0.0	140	23.5	53.7	53.7	0.0	80	21.7
pgreedy_exp	69.65	69.38	0.12	121	21.9	53.83	53.6	0.14	67	22.0
rpg	69.57	69.05	0.2	120	22.5	53.12	52.93	0.11	75	22.4
pgreedy_lin	69.45	68.68	0.36	83	23.1	51.43	50.9	0.53	30	21.2
rgreedy	67.46	66.48	0.47	59	23.0	42.78	41.05	1.44	16	19.5
sample	66.0	64.87	0.66	52	10.8	49.87	47.66	1.07	23	15.5

Table 11: Heuristic Comparison for Different Parameters



(a) Results of scenario 1 (4 km grid, $S = 10$, $p = 140$).

(b) Results of scenario 2 (2 km grid, $S = 5$, $p = 80$).

Figure 17: Analysis of construction phase methods and objective values across generated solutions

As discussed in Section 4.2, the *sample* method was anticipated to run faster than fully greedy approaches due to $q \ll m$, aligning with its lower computational complexity. Similarly, we expected the *greedy*, *randomized greedy*, and *proportional greedy* methods to provide higher solution quality at the cost of increased computation, due to their exhaustive evaluations. After implementing and performing these approaches on our two scenarios, our observations align largely with these expectations.

Detailed observations on the behavior of each method in the Timor-Leste dataset for the two scenarios are outlined below.

The *greedy* approach performs well in terms of objective value, achieving high average objective values across the constructed solutions. However, it does not attain the highest maximum objective, suggesting that consistently selecting the facility with the highest additional coverage does not necessarily result in the highest overall coverage. Moreover, the lack of variation in the selected facilities reduces the diversity of solutions, as the greedy approach constructs the same solution each time it is applied. Consequently, during the local search and path-relinking phases, this approach often explores the same local optimum, significantly limiting its ability to identify other promising regions in the solution space.

Other methods introduce more variation, which can be quantified in terms of the standard deviation of the objective value (σ_{obj}) and the diversity of the selected facilities (μ_λ). For the scenarios analyzed, the greater diversity in the selected facilities is correlated with higher variation in objective values. However, this variation often comes at a cost in mean coverage (μ_{obj}).

The *pgreedy_exp* and *pgreedy_lin* methods are probabilistic variants of the greedy approach, introducing randomness into the selection process.

The *pgreedy_exp* method closely resembles the greedy approach, as the facility with the highest additional coverage has a significantly higher probability of being selected. However, the inclusion of randomness allows facilities with lower coverage to occasionally be chosen. This approach achieves a slightly improved maximum objective compared to the greedy method and, in Scenario 1, even results in a higher average objective. This improvement is accompanied by a minor variation in the objective values (σ_{obj}) and moderate diversity in the solutions.

The *pgreedy_lin* method introduces greater variation, resulting in slightly lower maximum and mean coverage compared to the exponential bias approach. However, the increased variation facilitates more effective local search and path-relinking phases, potentially leading to better

overall solutions in the final iterations.

The *randomized plus greedy* approach performs well in terms of the maximum and average objective values. However, after the initial random selection, the greedy phase often selects the same facilities, resulting in limited variation between solutions. This lack of diversity reduces its effectiveness in later phases, such as path-relinking. Note that the value of β was set quite low, leading to a behavior that was more similar to the *greedy* approach.

The *rgreedy* method introduces even more variation than the previously discussed approaches. Its performance regarding the objective value is scenario-dependent. For the 4 km grid, the added variation does not significantly impact the objective. However, for the 2 km grid, the increased variation leads to a substantial decrease in mean coverage, which reduces it by 12.65%.

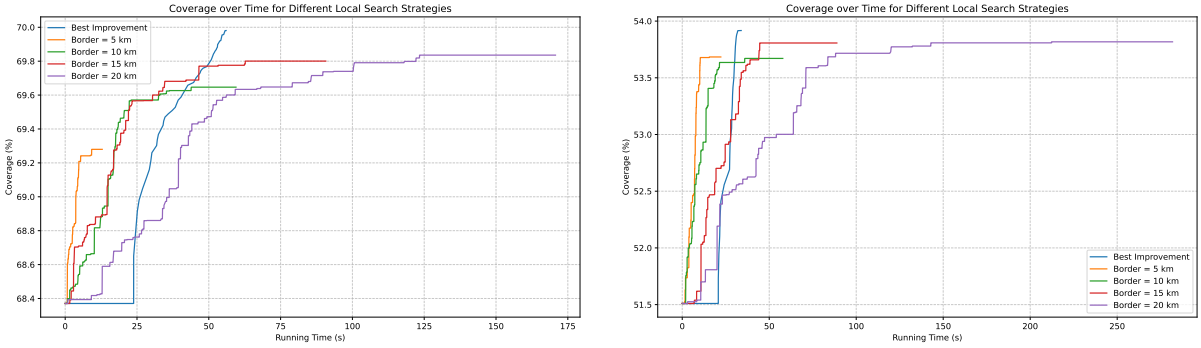
The *sample* approach also demonstrates significant variation in solutions, similar to *rgreedy*. However, the reduction in objective value is smaller, with only a decrease of 6.04% in mean coverage. Notably, the *sample* method requires less computational time than other approaches. This is because it calculates greedy values only for the facilities within the random sample, rather than for all facilities.

In conclusion, the *sample* and *pgreedy_lin* methods strike a promising balance between variation and solution quality, depending on parameter choices. The *sample* method offers greater variation, while *pgreedy_lin* achieves higher objective values. These characteristics suggest that both methods are well suited for integration into the GRASP algorithm.

5.3.2 Local Search

After completing the construction phase, the local search phase begins. During this phase, two strategies can be applied: best improvement or first improvement. When using the first improvement strategy, an additional decision must be made regarding the boundary condition, specifically the distance to neighboring facilities within which a swap is considered.

To assess the impact of different local search strategies on the quality of local optima and the time required to reach them, the progression of coverage over time is visualized for both scenarios in Figure 18. For the first improvement strategy, boundary distances of 5, 10, 15, and 20 km are evaluated. To ensure a fair comparison, each local search strategy starts with the same initial solution generated by $pgreedy_{lin}$ with $\alpha = 0.05$.



(a) Results of scenario 1 (4 km grid, $S = 10$, $p = 140$).

(b) Results of scenario 2 (2 km grid, $S = 5$, $p = 80$).

Figure 18: Coverage improvement over time for local search strategies

For the best improvement strategy, the initialization cost (before any swaps) is clearly visible in the figures. This phase lays the groundwork, after which the swapping phase begins, showing a smoother and steady progression. This demonstrates that swaps are efficiently identified and executed, resulting in consistent increases in coverage values.

For the first improvement strategy, varying boundary conditions (5 km, 10 km, 15 km, and 20 km) have a significant impact on both convergence time and final coverage quality:

- *Smaller Boundaries (e.g., 5 km)*: With smaller boundaries, swap decisions are limited to nearby neighbors, reducing the number of facilities to evaluate and speeding up the process. However, this often leads to lower-quality final coverage, as improvements remain more localized. For a 5 km boundary, the local optimum is reached faster than the initialization phase of the best improvement approach. In the first test scenario, the objective value is noticeably lower compared to larger boundaries and the best improvement strategy. Interestingly, in the second scenario, the performance with a 5 km boundary surpasses that of the 10 km boundary, showing no notable decrease in the objective value. This highlights that larger boundaries do not always guarantee better or comparable outcomes.

- *Larger Boundaries (e.g., 20 km)*: A larger boundary condition enables broader exploration of potential improvements, potentially leading to better overall coverage. However, this comes at the cost of higher computation time, as more neighboring facilities must be evaluated before making a decision. In both scenarios, the improvement in objective value between the 15 km and 20 km boundaries is minimal, while the computational cost increases substantially. The 20 km boundary requires excessive time without delivering a significant improvement in the objective value, making it inefficient.

When comparing the 10 km and 15 km boundaries, there is a noticeable improvement in both scenarios, highlighting a balance between solution quality and computational time.

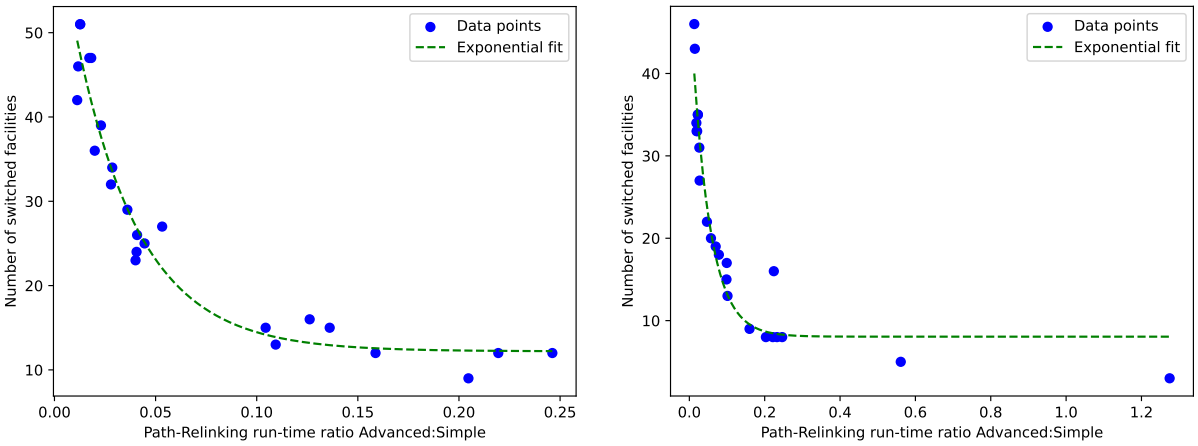
Based on these observations, further analysis will focus on the performance of the local search with boundaries of 5 km, 10 km, and 15 km.

5.3.3 Path Relinking

After completing the local search, the path-relinking phase begins. In this phase, the algorithm explores the path between the local optimum identified during the local search and one of the elite solutions.

Section 4.5 describes both a basic and an advanced approach to path relinking. According to Theulen [5], the basic method can outperform the advanced approach when only a small number of swaps are required, as the initialization cost of the advanced method can outweigh the benefits for such cases.

However, as shown in Figures 19a and 19b, tests conducted across various scenarios (4 km and 2 km grids with different p values) reveal that the advanced method consistently outperforms the basic approach in terms of running time. This difference is primarily due to the high computational cost of calculating the gain and loss metrics in the basic method. The basic approach is only slightly faster when very few swaps are needed. Based on these results, the advanced method will be used for the remainder of the analyses.



(a) Results of scenario 1 (4 km grid, $S = 10$, $p = 140$).

(b) Results of scenario 2 (2 km grid, $S = 5$, $p = 80$).

Figure 19: Relation between number of switched facilities and path-relinking runtime ratio

Several variations of path-relinking strategies are introduced: forward, backward, forward-and-backward, and mixed. To assess the impact of each strategy, a small experiment is conducted on both test scenarios by running GRASP with path-relinking four times, each consisting of 10 iterations. Each iteration includes a construction phase using *pgreedy_lin* with $\alpha = 0.05$, followed by a local search phase using the *best_improvement* strategy, and finishing with a path-relinking phase. During each iteration, all four path-relinking strategies are applied and their performance is recorded in terms of objective value and running time.

The choice to run four GRASP instances of 10 iterations each was based on the knowledge that the final model would likely be restricted to 10 iterations. Running a single GRASP instance with 40 iterations could create a significantly higher quality elite pool than what would realistically be achievable in the final model, making it unsuitable for fair comparison.

The performance of path-relinking strategies in terms of objective value is evaluated using two scoring methods: (i) the highest objective value earns three points, the second highest receives two points, and the third highest earns one point; (ii) only the highest objective value earns one point, while all others receive none. In both methods, ties are resolved by awarding the same number of points to all strategies with the same objective value.

The results of this experiment, including the scoring methods and the mean running times, are shown in Tables 12 and 13.

Method		Forward	Backward	Forward and Backward	Mixed$_f$	Mixed$_b$
(i)	Points	39	89	110	60	68
(ii)	Points	11	28	34	13	16
	Time (s)	5.48	5.48	10.96	7.41	7.02

Table 12: Performance of path relinking methods on a 4 km Grid with $S = 10$ and $p = 140$

Method		Forward	Backward	Forward and Backward	Mixed$_f$	Mixed$_b$
(i)	Points	76	90	119	76	77
(ii)	Points	24	28	39	21	21
	Time (s)	2.92	2.73	5.65	3.52	3.53

Table 13: Performance of path relinking methods on a 2 km Grid with $S = 5$ and $p = 80$

Tables 12 and 13 show that the backward approach is generally preferred over the forward approach, consistent with the findings of Resende and Ribeiro [30]. Although the forward-and-backward path relinking requires approximately twice the computational time, it significantly outperforms both forward and backward methods in terms of objective value, as anticipated.

However, the experimental results for the mixed path relinking strategy differ from the findings of Ribeiro and Rosseti [34], which suggested that mixed path relinking should outperform forward, backward, and forward-and-backward methods. This study did not observe such an advantage. Instead, the backward and forward-and-backward strategies demonstrated better performance compared to the mixed path relinking approach.

Comparing variations of the mixed strategy that begin with forward (Mixed_f) or backward (Mixed_b) relinking reveals a slight preference to start with backward relinking.

To ensure clarity and simplicity in the overall implementation of the algorithm, the forward-and-backward path relinking strategy will be adopted. This method consistently achieved the best performance, balancing high objective values with fewer relinking operations. Additionally, within the constraints of a two-hour time frame and a limited number of GRASP iterations, the number of relinking operations remains manageable. In these scenarios, forward-and-backward path relinking accounted for approximately 11% and 9% of the total computational time, respectively.

5.3.4 Analysis of the Whole Heuristic

After analyzing each phase of the heuristic independently, the next step is to integrate these phases through heuristic tuning. This involves selecting strategies and parameters to achieve optimal performance for the Timor-Leste case on the 3 km and 4 km grids. The performance of various configurations is evaluated across multiple scenarios, varying the service distance ($S = 5$ and 10 km) and the number of facilities opened ($p = \{80, 110, 140, 170, 200\}$). This results in 10 test cases used to compare different configurations of construction methods, local search strategies, and path-relinking techniques on each grid.

For the construction methods, the *sample* method is tested with $\theta \in \{0.02, 0.05, 0.10\}$, while other construction methods involving α are tested with $\alpha \in \{0.05, 0.10, 0.20\}$. The *rps* method additionally includes $\beta \in \{0.05, 0.10, 0.20\}$. Local search strategies include configurations for the first improvement with boundaries ($border \in \{5, 10, 15\}$) or the best improvement. For the path-relinking phase, a forward-and-backward strategy is applied.

To ensure fairness in comparison between grid sizes, each case consists of a fixed number of GRASP iterations rather than a time limit. Using a time-based criterion could lead to inconsistencies, as 50 iterations on a 4 km grid may not be comparable to completing only 5 iterations on a 2 km grid. However, to reflect performance expectations on the 2 km grid, a time limit equivalent to eight standard GRASP iterations is imposed. This ensures most 2 km grid cases complete in approximately two hours, which is considered an acceptable runtime even for the largest cases.

For each test case (a specific combination of p and S), several key performance indicators (KPIs) are tracked. These include the best objective value found during the iterations, the best objective value achieved after the first four iterations, and two time-to-target (TTT) values. A target is defined at 0.2% below the optimum, representing an acceptable deviation, while a stricter target is set at 0.05%, representing a negligible difference. These KPIs are referred to as $TTT_{0.2}$ and $TTT_{0.05}$, respectively. For ranking purposes, the configuration with the best score for each KPI is awarded 5 points, the second-best receives 4 points, and so on. The total score for all cases determines the best-performing configurations.

The results will be presented in tables showing the top five configurations for each KPI. The table labels indicate the settings used. For example, '*lin*' refers to the '*pgreedy_lin*' construction method, while '*exp*' refers to the '*pgreedy_exp*' construction phase. The type of construction method is followed by the randomness parameter value. The local search strategy is identified as '*BI*' for best improvement or by the boundary value for the first improvement. Finally, the total score is included, enabling comparisons between methods.

The analysis begins with the 4 km grid, using the different configurations. To examine whether there is an optimal choice of the construction method and its parameters regardless of the local search strategy, the KPI scores are first analyzed separately for the best improvement and first improvement strategies (with varying boundaries). Table 14 presents the results for the best improvement strategy on the 4 km grid.

	Best	**Half-time**	**TTT**$_{0.2}$	**TTT**$_{0.05}$
1st	sample, 0.05, BI: 39	rgreedy, 0.1, BI: 37	greedy, BI: 37	greedy, BI: 18
2nd	rgreedy, 0.1, BI: 37	sample, 0.05, BI: 34	lin, 0.1, BI: 13	rpg, (0.1, 0.05), BI: 11
3rd	lin, 0.2. BI: 35	lin, 0.2. BI: 34	lin, 0.05, BI: 12	rpg, (0.05, 0.05), BI: 11
4th	lin, 0.05, BI: 33	lin, 0.05, BI: 34	rpg, (0.05, 0.05), BI: 12	rpg, (0.2, 0.05), BI: 7
5th	rgreedy, 0.05, BI: 33	lin, 0.1, BI: 33	rgreedy, 0.1, BI: 11	rpg, (0.2, 0.2), BI: 7

Table 14: Comparison of construction phase performance using best improvement strategy

The results show that the scores for both the best and the half-time objectives are consistently high in all configurations. However, a notable observation is that entirely different configurations of construction phases perform well for time-to-target metrics. What explains this discrepancy?

For the 5 km service distance, all configurations demonstrate strong performance, as many configurations reach the optimal solution or come very close to it. Consequently, the high scores for the best and halftime objectives reflect this overall success, rather than a meaningful differentiation among configurations. Meanwhile, the time-to-target values can be somewhat misleading in this context, as they primarily highlight which configuration completes a single GRASP iteration the fastest, rather than providing deeper insights into performance differences. As a result, no single configuration–parameter combination consistently appears in all KPI categories.

To provide a more representative assessment, we focus on a more challenging problem by analyzing the KPI results exclusively for the 10 km service distance on the 4 km grid. This scenario offers a better benchmark to evualate the performance of each configuration. The results are presented in Table 15.

	Best	**Halftime**	**TTT**$_{0.2}$	**TTT**$_{0.05}$
1st	**sample, 0.05, BI: 14**	**sample, 0.05, BI: 14**	lin, 0.1, BI: 13	**rgreedy, 0.1, BI: 5**
2nd	**lin, 0.05, BI: 13**	**lin, 0.05, BI: 14**	**lin, 0.05, BI: 12**	**lin, 0.2, BI: 5**
3rd	**rgreedy, 0.1, BI: 12**	**rgreedy, 0.1, BI: 12**	**rgreedy, 0.1, BI: 11**	rpg, (0.2, 0.2), BI: 4
4th	**lin, 0.2, BI: 10**	**lin, 0.2, BI: 9**	**sample, 0.05, BI: 9**	**lin, 0.05, BI: 3**
5th	rgreedy, 0.05, BI: 8	rgreedy, 0.2, BI: 8	**lin, 0.2, BI: 6**	**sample, 0.05, BI: 2**

Table 15: Comparison of construction phase performance using best improvement strategy for only 10 km service distance

From this analysis, it can be concluded that no single construction phase performs best across all KPIs. The best performing configurations include the *sample* method with a θ-value of 0.05, *pgreedy-lin* with α-values of 0.05 or 0.2, and *rgreedy* with an α-value of 0.1. Although one configuration may slightly outperform others in objective values, another may excel in the time-to-target KPIs.

Next, we turn to the results for the first improvement local search on the 4 km grid. Given the knowledge that the 5 km service distance is relatively straightforward for the heuristic, often finding solutions close to optimal within the first GRASP iterations, we focus only on the more challenging 10 km service distance. The relevant KPI results are presented in Table 16.

	Best	**Halftime**	**TTT**$_{0.2}$	**TTT**$_{0.05}$
1st	**lin, 0.05, 10: 10**	**lin, 0.05, 10: 10**	lin, 0.05, 15: 9	rpg, (0.05, 0.2), 10: 5
2nd	lin, 0.1, 10: 9	lin, 0.05, 15: 8	**lin, 0.05, 10: 8**	sample, 0.02, 15: 4
3rd	sample, 0.1, 15: 7	sample, 0.1, 10: 8	sample, 0.1, 5: 8	exp, 0.1, 5: 3
4th	sample, 0.1, 10: 7	lin, 0.1, 15: 8	sample, 0.1, 15: 5	lin, 0.2, 10: 2
5th	lin, 0.1, 15: 7	sample, 0.1, 15: 8	lin, 0.2, 10: 5	**lin, 0.05, 10: 1**

Table 16: Comparison of construction phase performance using first improvement strategy for only 10 km service distance

It can be concluded that one configuration performs reasonably well across all KPIs: *pgreedy-lin* with an α-value of 0.05 and a border of 10 km. However, this configuration does not clearly outperform the others as its total score of points is not significantly higher. When looking at the table more broadly (without focusing on specific parameters), both *pgreedy-lin* and *sample* show strong overall performance.

Based on Tables 15 and 16, it cannot be concluded that a single combination of construction method and parameter consistently outperforms others across all KPIs, regardless of the local search strategy. To address this, the KPIs have been re-analyzed by combining the results from both best and first improvement strategies. This means that the points for each configuration are calculated by considering its performance in both local search approaches. The consolidated findings are presented in Table 17, which focuses solely on the 10 km service distance for the 4 km grid.

	Best	**Halftime**	**TTT**$_{0.2}$	**TTT**$_{0.05}$
1st	**rgreedy, 0.1, BI: 10**	**rgreedy, 0.1, BI: 12**	**lin, 0.05, BI: 11**	**rgreedy, 0.1, BI: 5**
2nd	lin, 0.05, 10: 8	sample, 0.05, BI: 10	**rgreedy, 0.1, BI: 10**	**lin, 0.2, BI: 5**
3rd	**lin, 0.2, BI: 8**	**lin, 0.05, BI: 10**	lin, 0.1, BI: 10	rpg, (0.2, 0.2), BI: 4
4th	lin, 0.1 10: 7	lin, 0.1, BI: 8	sample, 0.05, BI: 7	rpg, (0.05, 0.2), 10: 3
5th	**lin, 0.05, BI: 7**	**lin, 0.2, BI: 7**	**lin, 0.2, BI: 5**	**lin, 0.05, BI: 2**

Table 17: Comparison of construction phase performance using both local search strategies

It can be observed that *rgreedy* with an α-value of 0.1 demonstrates strong performance across all KPIs. However, these conclusions should be tempered by noting that the analysis was conducted on the 4 km grid. Although *rgreedy* performed well in the construction phase analysis, presented in Section 5.3.1, for the 4 km grid in terms of objective values, it showed the worst overall performance during the construction phase on the 2 km grid.

Furthermore, it can be observed that *pgreedy-lin*, across all α-values, performs well in the KPIs regardless of the local search strategy. Specifically, for α-values of 0.2 and 0.05 combined with the best improvement strategy, it appears consistently in the top 5 for all KPIs.

The combined local search KPI analysis for the 4km grid suggests that the best improvement strategy is preferable to achieve strong performance across all KPIs. As shown in Table 15, the best improvement results highlight that *sample*, *pgreedy-lin*, and *rgreedy* all deliver strong performance, with no single method clearly outperforming the others. To gain deeper insights, further KPI analysis for these three construction phases, using the best improvement strategy, will be conducted on the 3 km grid.

The analysis on the 3 km grid will follow the same KPI-based methodology used for the 4 km grid, with a slight adjustment. Since the optimal solution is unknown for the 3 km grid, the time-to-target (TTT) metric will instead measure the time required to reach the objective value obtained by the MIP solver with a 1% optimality gap. Due to the larger problem size of the 3 km grid, it is expected to better represent the challenges and behavior observed in the 2 km grid compared to the 4 km grid. The results for both service distances are presented in Table 18.

	Best	**Halftime**	**TTT**
1st	**lin, 0.2, BI: 28**	**lin, 0.2, BI: 29**	**lin, 0.1, BI: 24**
2nd	rgreedy, 0.1, BI: 25	sample, 0.1, BI: 27	lin, 0.05, BI: 19
3rd	**lin, 0.1, BI: 21**	**lin, 0.1, BI: 19**	sample, 0.02, BI: 19
4th	sample, 0.1, BI: 20	lin, 0.05, BI: 16	**lin, 0.2, BI: 17**
5th	sample, 0.05, BI: 18	rgreedy, 0.2, BI: 16	rgreedy, 0.05, BI: 7

Table 18: Comparison of construction phase performance using best improvement strategy on the 3 km grid

Based on these results, *pgreedy-lin* with an α-value of 0.2 clearly outperforms other configurations on the objective KPIs and also performs reasonably well on the time-to-target metric for the 3 km grid. Note that the 5 km service distance is included here, as the problem on the 3 km grid is slightly more complex for the heuristic than the 4 km grid.

Combined with the findings that this configuration, paired with the best improvement strategy, also demonstrated strong performance in the KPI analysis on the 4 km grid (see Table 17), we can conclude that "*lin, 0.2, BI*" is the most optimal choice for the heuristic on both the 3 km and 4 km grids. This means that *pgreedy-lin* will be used as the construction phase with the parameter α set to 0.2, and the best improvement strategy will be applied for the local search.

5.3.5 Results of the Tuned Heuristic for Timor-Leste

After identifying the optimal heuristic configuration, it can be applied to the three grid sizes - 4 km, 3 km, and 2 km - across various p values to evaluate the resulting coverage. This analysis is conducted for both approaches: "From Scratch Optimization" and "Real-Life Scenario Optimization."

All results are presented as percentage coverage. Any reference to an increase in coverage refers to the absolute increase.

From Scratch Optimization
We start with the results for the "From Scratch Optimization", as presented in Table 19. For each grid size, the table includes a BI column, representing the objective achieved by the heuristic without path-relinking, and a GRASP column showing the result with path-relinking.

		grid-size = 4 km			grid-size = 3 km			grid-size = 2 km		
S	p	BI_4	$GRASP_4$	$Time_4$ (s)	BI_3	$GRASP_3$	$Time_3$ (s)	BI_2	$GRASP_2$	$Time_2$ (s)
5	80	41.877	41.877	45.6	48.425	48.425	103.3	54.203	54.254	308.8
	110	44.199	44.199	51.6	51.355	51.372	127.7	57.804	57.973	395.7
	140	46.261	46.261	57.4	53.834	53.835	170.6	61.246	61.418	447.2
	170	47.954	47.957	64.1	55.723	55.753	156.3	63.682	63.784	556.4
	200	49.387	49.394	70.7	57.829	57.853	168.3	65.637	65.771	609.3
10	80	65.303	65.393	302.8	69.490	69.537	868.9	74.120	74.558	2659.9
	110	68.492	68.523	376.6	72.457	72.650	1099.5	78.736	78.862	3909.5
	140	70.245	70.603	510.6	74.309	74.511	1509.5	80.484	80.742	5333.2
	170	71.165	71.300	611.6	74.755	74.987	1775.7	80.853	81.115	6782.3
	200	71.084	71.207	674.5	74.599	74.920	2214.3	80.868	81.068	8121.9

Table 19: Results of the tuned heuristic for the From Scratch Optimization

This table illustrates the impact of incorporating path-relinking into the heuristic across different grid sizes, highlighting its role in enhancing percentage coverage. On the 4 km grid, the coverage improves by up to 0.358%, while on the 3 km grid, the maximum increase is 0.321%. On the 2 km grid, path relinking achieves the highest improvement, with coverage increasing by up to 0.438%. These results indicate that path relinking consistently enhances the performance of the heuristic, leading to better solutions across all grid sizes.

In addition to the effect of path relinking, the influence of grid size itself on the heuristic outcomes can also be analyzed. When comparing the results for the 3 km grid with those for the 2 km grid, it becomes clear that using a smaller grid size leads to increased coverage. For example, at a 5 km service distance with $p = 170$, the percentage coverage improves by 8.031% when switching from the 3 km grid to the 2 km grid. This improvement is not limited to a single case: On average, the transition from the 3 km grid to the 2 km grid results in an overall mean percentage coverage increase of 6.570%.

These findings underline two key insights for the "From Scratch Optimization". First, path relinking plays a vital role in improving the performance of the heuristic across all grid sizes. Second, applying the heuristic on a smaller grid size contributes significantly to better coverage outcomes.

Real-Life Scenario Optimization
The results of the "Real-Life Scenario Optimization" are presented in Table 20. Here, the 128 facilities already opened represent the current situation in Timor-Leste and are incorporated into the grid points. Consequently, reducing the grid size for $p = 128$ does not lead to any additional coverage improvement.

		grid-size = 4 km			grid-size = 3 km			grid-size = 2 km		
S	p	BI_4	$GRASP_4$	$Time_4$ (s)	BI_3	$GRASP_3$	$Time_3$ (s)	BI_2	$GRASP_2$	$Time_2$ (s)
5	128	41.524	41.524	-	41.524	41.524	-	41.524	41.524	-
	140	49.589	49.589	61.1	50.477	50.477	71.3	52.578	52.578	219.5
	170	54.042	54.042	65.8	57.119	57.119	110.3	61.540	61.542	281.9
	200	56.307	56.307	74.5	60.236	60.247	137.1	65.503	65.548	361.3
10	128	57.311	57.311	-	57.311	57.311	-	57.311	57.311	-
	140	66.385	66.385	142.3	66.718	66.718	258.0	68.107	68.107	819.8
	170	73.311	73.311	328.5	74.306	74.306	593.3	78.519	78.553	2096.7
	200	74.724	74.752	345.9	76.311	76.375	901.3	81.952	82.001	2913.6

Table 20: Results of the tuned heuristic for the Real Life Scenario Optimization

The effect of adding path relinking to the heuristic for the different grid sizes is less significant, with an increase in percentage coverage of at most 0.064%.

When comparing the results for the 3 km grid with those for the 2 km grid, it becomes evident that the use of a smaller grid size leads to improved coverage. For example, at a 10 km service distance with $p = 200$, the coverage increases by 5.626% when switching from the 3 km grid to the 2 km grid. This improvement is not limited to a single case: On average, the transition from the 3 km grid to the 2 km grid results in an overall mean percentage coverage increase of 3.848%.

Similarly to the results observed with the MIP solver, the increases in both the impact of path-relinking and the percentage improvement from using a smaller grid are significantly smaller compared to the "From Scratch" scenario. This can be attributed to the real-life scenario, where the heuristic selects fewer (additional) facilities. As a result, fewer facilities benefit from the advantages of path-relinking and finer grid size.

5.4 Comparison Between the Results of the MIP Solver and the Tuned Heuristic

In the previous subsections, we identified the best-performing MIP solver and the tuned heuristic. However, how does the performance of the heuristic compare to the optimal solution generated by the MIP solver?

From Scratch Optimization
The percentage coverage results for both the MIP solver and the heuristic of the "From Scratch Optimization" are presented in Table 21.

		Grid Size = 4 km		Grid Size = 3 km		Grid Size = 2 km
S	p	Opt_4	GRASP_4	Opt_3^*	GRASP_3	GRASP_2
5	80	41.877	41.877	48.302	48.425	54.254
	110	44.199	44.199	51.094	51.372	57.973
	140	46.261	46.261	53.393	53.835	61.418
	170	47.957	47.957	55.491	55.753	63.784
	200	49.394	49.394	57.806	57.853	65.771
10	80	65.399	65.393	69.733	69.537	74.558
	110	68.603	68.523	72.915	72.650	78.862
	140	70.654	70.603	74.544	74.511	80.742
	170	71.404	71.300	74.965	74.987	81.115
	200	71.273	71.207	75.110	74.920	81.068

Table 21: Comparison of coverage obtained between the MIP solver and the tuned heuristic for the From Scratch Optimization

For the 4 km grid, the largest difference between the optimal and the heuristic percentage coverage is 0.104%, with a mean difference of 0.031%. For the 3 km grid, the optimal values were obtained with a 1% optimality gap. In this case, the heuristic outperforms the MIP solver in all scenarios with a service distance of 5 km. For the 10 km service distance, the largest difference between the percentage coverage of the optimal solution and the heuristic solution is 0.265%.

Although this difference is notable, the ability of the heuristic to solve the problem on a 2 km grid, something not achievable with the MIP solver due to memory limitations, results in significantly higher coverage compared to the 3 km grid with a 1% optimality gap or even if the solution were solved to optimality.

In addition to coverage performance, we will also briefly highlight the running time performance of the MIP solver and the heuristic in Table 22.

		Grid Size = 4 km		Grid Size = 3 km	
S	p	Time_{opt}	Time_{GRASP}	Time_{opt}	Time_{GRASP}
5	80	32.3	45.6	58.2	103.3
	110	19.3	51.6	49.2	127.7
	140	19.7	57.4	53.5	170.6
	170	19.4	64.1	58.9	156.3
	200	19.1	70.7	53.7	168.3
10	80	205.6	302.8	1348.8	868.9
	110	311.1	376.6	2478.9	1099.5
	140	356.3	510.6	6326.2	1509.5
	170	1143.9	611.6	4701.3	1775.7
	200	1661.0	674.5	7282.4	2214.3

Table 22: Comparison in running time between the MIP solver and the tuned heuristic for the From Scratch Optimization

The table highlights key insights: The MIP solver is consistently faster than the heuristic for a 5 km service distance, regardless of grid size. However, for a 10 km service distance and a 3 km grid, the heuristic outperforms the MIP solver across all p values, demonstrating its efficiency for larger problem instances. On the 4 km grid, the MIP solver is faster for smaller p values (80, 110, and 140), but the heuristic becomes faster when $p \geq 170$. This shows that as the problem size increases, whether through more selected facilities or finer grids, the heuristic becomes more computationally favorable.

Real-Life Scenario Optimization
The percentage coverage results for both the MIP solver and the heuristic of the "Real-Life Scenario Optimization" are presented in Table 23.

		Grid Size = 4 km		Grid Size = 3 km		Grid Size = 2 km
S	p	Opt_4	GRASP_4	Opt_3	GRASP_3	GRASP_2
5	128	41.524	41.524	41.524	41.524	41.524
	140	49.589	49.589	50.477	50.477	52.578
	170	54.042	54.042	57.119	57.119	61.542
	200	56.307	56.307	60.247	60.247	65.548
10	128	57.311	57.311	57.311	57.311	57.311
	140	66.385	66.385	66.738	66.718	68.107
	170	73.311	73.311	74.339	74.306	78.553
	200	74.766	74.752	76.471	76.375	82.001

Table 23: Comparison coverage obtained between the MIP solver and the tuned heuristic for the Real-Life Scenario Optimization

For the 4 km grid, the heuristic performs exceptionally well, achieving the optimal value in all test cases. On the 3 km grid, the heuristic also reaches the optimal solution for the 5 km service distance. However, for the 10 km service distance, the largest difference between the percentage coverage of the optimal solution and the GRASP solution is 0.096. This difference is minimal, and considering the results across all test cases, it can be concluded that the heuristic performs very well for the "Real-Life Scenario Optimization."

Furthermore, the heuristic enables the model to be solved for a smaller grid size, which leads to a significant improvement in percentage coverage. When comparing the optimal coverage on the 3 km grid with the heuristic results on the 2 km grid, the percentage coverage increases by at least 1.369.

6 Relevance and Future Potential of the Research

6.1 Relevance of Incorporating Capacity and Closest Assignment Constraints

As discussed in the literature review, this specific model has not been explicitly documented in existing studies. However, simply accepting this and solving the model without incorporating capacity and closest assignment constraints can lead to impractical solutions. This subsection highlights the importance of explicitly including both capacity constraints and closest assignment conditions in the solver.

To evaluate the impact of excluding these constraints, despite their relevance in real-world scenarios, the following approach is applied. First, the model is solved optimally as a Maximum Covering Location Problem (MCLP) without accounting for capacity or closest assignment constraints. After determining the optimal facility locations, the coverage is recalculated by imposing these constraints. This adjusted coverage is then compared to the results generated by the heuristic introduced in this thesis, where both constraints are incorporated directly into the optimization process.

This study is conducted using the Timor-Leste dataset with a 2 km grid for potential facility locations, where $K = (\text{total demand}/p) * c$.

In the tables presented, the column Opt_{MCLP} represents the coverage achieved by solving the MCLP optimally without accounting for capacity or closest assignment constraints. The columns $c = 1.5$ and $c = 5$ display the recalculated coverage after applying these constraints to the MCLP solution. In contrast, the columns $GRASP_{1.5}$ and $GRASP_5$ present the coverage obtained directly from the heuristic, where the capacity and closest assignment constraints are incorporated into the optimization process.

From Scratch Optimization
The results of the "From Scratch Optimization" are shown in Table 24.

		grid-size = 2 km				
S	p	Opt_{MCLP}	c = 1.5	$\text{GRASP}_{c=1.5}$	c = 5	$\text{GRASP}_{c=5}$
5	80	66.057	46.947	53.254	57.212	65.580
	110	72.186	50.002	57.973	59.932	70.896
	140	77.189	52.610	63.784	63.329	73.448
	170	81.127	54.451	65.771	65.376	75.387
	200	84.406	56.487	65.637	67.331	77.224
10	80	88.537	62.602	74.558	69.295	87.391
	110	93.833	65.015	78.558	73.206	91.761
	140	96.400	64.865	80.742	77.062	92.267
	170	97.662	65.019	81.115	77.052	91.898
	200	98.341	63.393	81.068	76.293	90.770

Table 24: Comparison of optimal MCLP coverage, realistic MCLP coverage, and optimal CMCLP-CAC coverage for the From Scratch Optimization

The table highlights that the coverage achieved by solving the model without capacity and closest assignment constraints is overly optimistic. Even with a capacity where $c = 5$, the difference between the theoretical coverage and the actual coverage in the MCLP model ranges from 8.845% to 22.048%. This discrepancy becomes even more pronounced when the capacity constraint is tighter, as in $c = 1.5$, where the differences range from 19.110% to 34.948%.

The key question then is how much additional coverage can be achieved by incorporating the capacity and closest assignment constraints into the optimization. For a capacity where $c = 1.5$,

solving with these constraints increases coverage by 6.307% to 17.675%; for a capacity where $c = 5$, then the coverage improvements range from 8.368% to 18.555%. These results emphasize that even with higher capacities, integrating capacity and closest assignment constraints remain crucial for the "From Scratch Optimization."

Real-Life Scenario Optimization
The results of the "Real-Life Scenario Optimization" are shown in Table 25

		grid-size = 2 km				
S	p	Opt_{MCLP}	c = 1.5	$\text{GRASP}_{c=1.5}$	c = 5	$\text{GRASP}_{c=5}$
5	128	58.962	41.524	41.524	53.608	53.608
	140	67.759	48.705	52.578	62.003	66.460
	170	75.767	54.132	61.542	67.787	74.769
	200	80.615	56.765	65.548	70.946	79.284
10	128	80.379	57.311	57.311	73.779	73.779
	140	89.277	63.420	68.107	81.445	87.541
	170	95.020	66.767	78.553	84.726	94.024
	200	97.147	65.279	82.001	85.101	96.300

Table 25: Comparison of optimal MCLP coverage, realistic MCLP coverage, and optimal CMCLP-CAC coverage for the Real-Life Scenario Optimization

Similar findings are observed here. The coverage obtained by solving the model without accounting for capacity and closest assignment constraints is overly optimistic. Incorporating these constraints into optimization leads to significant coverage improvements.

For a capacity where $c = 1.5$, coverage increases range from 3.873% to 16.722%. When the capacity is raised to $c = 5$, the improvements remain substantial, with coverage increasing by at least 4.457% and up to 11.199%. These results underscore that even with higher capacities, explicitly including capacity and closest assignment constraints in the optimization is essential for achieving realistic and effective solutions in the "Real-Life Scenario Optimization."

6.2 Improving the Situation in Timor-Leste

We acknowledge that linking capacity with the number of opened facilities may not be the most practical approach to assess how the situation in Timor-Leste can be improved by adding new facilities. This method causes the capacity to decrease as more facilities are introduced. To still highlight the improvements in coverage achieved by adding new facilities, we present Table 26, which reports the results using three fixed capacity values: $K = 20,000$, $K = 50,000$, and $K = 100,000$.

This experiment emphasizes two key aspects. First, as in previous analyses, we report the optimal solutions for the 3 km grid. Additionally, we include the percentage coverage achieved by the heuristic on the 2 km grid, demonstrating its performance on a more detailed grid under fixed capacity conditions. Second, the results allow us to determine the additional coverage that can be achieved by opening more facilities for a given capacity.

		grid-size = 3 km			grid-size = 2 km		
S	p	$Opt_{K=20000}$	$Opt_{K=50000}$	$Opt_{K=100000}$	$GRASP_{K=20000}$	$GRASP_{K=50000}$	$GRASP_{K=100000}$
5	128	44.924	53.984	58.669	44.924	53.984	58.669
	140	56.907	64.996	67.392	59.655	66.997	67.759
	170	65.198	72.844	75.000	70.537	75.417	75.733
	200	69.720	77.286	79.387	75.910	80.406	80.615
10	128	63.043	74.256	79.428	63.043	74.256	79.428
	140	75.432	86.058	88.580	78.180	88.094	88.911
	170	83.897	92.259	94.560	89.681	94.584	94.858
	200	86.698	94.576	96.770	92.950	96.969	97.082

Table 26: Results of percentage coverage obtained for the real-life scenario optimization using fixed capacities

First, we want to emphasize that the heuristic on the 2 km grid consistently outperforms the optimal solutions obtained on the 3 km grid in all three capacities and p parameters. This highlights the clear benefit of applying the heuristic to a finer grid, resulting in improved percentage coverage in every test case.

Moreover, the additional coverage achieved by introducing 12 new facilities highlights this advantage further. For a 5 km service distance and $K = 20,000$, the heuristic on the 2 km grid achieves a coverage increase of 14.731%, compared to 11.983% obtained by optimally solving on the 3 km grid. For $K = 50,000$, the heuristic achieves a coverage of 13.013%, outperforming the optimal 10.982% on the 3 km grid. Even with a relatively large capacity ($K = 100,000$), the heuristic on the 2 km grid achieves 9.090% coverage, exceeding 8.723% from the optimal solution on the 3 km grid.

Similar trends are observed for the 10 km service distance, further demonstrating the effectiveness of the GRASP heuristic on the finer 2 km grid.

In summary, for the 5 km service distance, introducing 12 new facilities achieves a minimum coverage increase of 9.090% with the heuristic, which is 0.367% higher than optimally solving on the 3 km grid. Similarly, for the 10 km service distance, the heuristic achieves a minimum increase of 9.483%, which is 0.331% higher than the optimal solution on the 3 km grid.

6.3 Potential for increasing the number of GRASP iterations

The performance of the heuristic, compared to the optimal solution for the 4 km grid and the 3 km grid solution with a 1% optimality gap, was initially evaluated with 8 iterations. To assess its potential with increased computational effort, the results for 16 iterations in the "From Scratch Optimization" are presented in Table 27.

		Grid Size = 4 km			Grid Size = 3 km			Grid Size = 2 km	
S	p	Opt_4	$GRASP_4$ (8)	$GRASP_4$ (16)	Opt_3^*	$GRASP_3$ (8)	$GRASP_3$ (16)	$GRASP_2$ (8)	$GRASP_2$ (16)
5	80	41.877	41.877	41.877	48.302	48.425	48.425	54.254	54.254
	110	44.199	44.199	44.199	51.094	51.372	51.385	57.973	57.983
	140	46.261	46.261	46.261	53.393	53.835	53.835	61.418	61.424
	170	47.957	47.957	47.957	55.491	55.753	55.753	63.784	63.784
	200	49.394	49.394	49.394	57.806	57.853	57.863	65.771	65.878
10	80	65.399	65.393	65.398	69.733	69.537	69.662	74.558	75.155
	110	68.603	68.523	68.552	72.915	72.650	73.098	78.862	79.066
	140	70.654	70.603	70.603	74.544	74.511	74.511	80.742	80.742
	170	71.404	71.300	71.303	74.965	74.987	75.040	81.115	81.167
	200	71.273	71.207	71.207	75.110	74.920	74.920	81.068	81.145

Table 27: Results 16 GRASP iterations for grid sizes 4 km, 3 km, and 2 km

For the 4 km grid, using 8 iterations, the largest observed difference in percentage coverage is 0.104%, with a mean difference of 0.031%. Increasing to 16 iterations slightly reduces the largest difference to 0.101%, and the mean difference improves to 0.027%, indicating minor progress.

For the 3 km grid, with 8 iterations, the largest difference in percentage coverage compared to the optimal solution with a 1% optimality gap is 0.265%. With 16 iterations, this difference decreases to 0.190%, reflecting a more noticeable improvement.

For the 2 km grid, while direct comparison to the optimal solution is not possible, increasing to 16 iterations enhances the percentage coverage by up to 0.597% compared to 8 iterations, with a mean improvement of 0.105%.

6.4 Potential for performing analysis on finer grid

This thesis evaluates the performance of the heuristic using the Timor-Leste dataset, with grid sizes of 4 km, 3 km, and 2 km, resulting in approximately 3,600 potential facility locations. The heuristic demonstrates strong potential for application to even larger datasets, especially when the algorithm is further optimized for computational efficiency or executed on more powerful computer.

To showcase this potential, the heuristic will also be applied to a 1 km grid, resulting in 14,147 potential facility locations. Furthermore, the approach could be extended to larger datasets, such as Vietnam, which comprises 406,784 demand locations representing more than 97 million inhabitants. For a 10 km grid, this would correspond to 3,555 potential facility locations.

However, due to time constraints in this research, the focus has been on increasing the number of facilities rather than the number of demand locations. This choice significantly increases the computational complexity in each phase. For instance, the complexity of the best-improvement phase is given by:

$$\mathcal{O}(p(m-p)(kl+gn))$$

Here, increasing m generally has a more pronounced effect on overall complexity compared to increasing n, as m directly scales the multiplier $(m-p)$ applied to $kl+gn$.

The same principle applies to the construction phase, where a proportional greedy approach with linear bias is utilized in the tuned heuristic. Its complexity is given by $\mathcal{O}(pm(kl+gn+\log(m)))$. Table 28 presents the results of applying the heuristic on a 1 km grid for the "From Scratch Optimization".

		Grid Size = 1 km		
S	p	BI	GRASP_2 (8)	Time (h)
5	80	64.655	64.967	1.0
	110	70.353	70.526	1.3
	140	73.643	73.776	1.5
	170	76.417	76.586	1.6
	200	77.515	77.678	1.7
10	80	84.478	84.885	9.5
	110	87.527	87.567	12.8
	140	91.546	91.716	15.2
	170	92.568	92.651	19.2
	200	92.875	93.065	21.1

Table 28: Results for Grid Size 1 km

For the 5 km service distance, performing the heuristic on the 1 km grid instead of the 2 km grid improves the percentage coverage by 10.401% to 12.633%. For the 10 km service distance, this improvement ranges between 8.705% and 11.997%.

7 Conclusion and Recommondations

This thesis aimed to address the Capacitated Maximum Covering Location Problem with Closest Assignment Constraints (CMCLP-CAC) for large-scale datasets, a challenge not previously tackled in the literature. Previous studies on the Capacitated Maximum Covering Location Problem (CMCLP) were primarily restricted to datasets with approximately 1,000 demand locations and 1,000 potential facility locations. This research successfully extends the problem to significantly larger scales by developing two solution approaches: a Mixed Integer Programming (MIP) solver and a GRASP-based heuristic method. Although the MIP solver aims for optimal solutions, it is limited by the size of the problem and computational resources. In contrast, the heuristic method provides near-optimal solutions efficiently, making it more practical for large-scale instances.

The CMCLP-CAC is inherently complex, as it involves optimizing both facility placement and the allocation of demand locations to facilities while adhering to capacity and closest assignment constraints. To address this complexity, several enhancements were applied to the MIP solver. This included reducing the number of decision variables, converting allocation decision variables from binary to fractional, preprocessing the index set, and incorporating two service distance models: a weak model (5 km service distance) and a strong model (10 km service distance).

These improvements allow us to solve the Timor-Leste dataset using a 4 km grid in "From Scratch Optimization" test cases, which consist of 151,765 demand locations and 881 potential facility locations, within 30 minutes. For the 3 km grid, featuring 1,576 potential facility locations, we achieved an optimality gap of 1% within two hours. In the "Real-Life Scenario Optimization" test cases, an additional 128 facilities are included in the data for each grid size, and these are forced to be opened. As a result, the heuristic has to choose fewer facilities under the same constraints for the number of facilities allowed to open. In this context, the 4 km and 3 km grids were solved optimally; the 4 km grid required only 3 minutes, while the 3 km grid was solved in approximately two hours. However, for both optimization scenarios, the solver reached its memory limit when applied to the 2 km grid.

To address these computational challenges, we extended and adapted the memory-efficient GRASP heuristic developed by Theulen [5] to incorporate both capacity and closest assignment constraints. By carefully tuning the heuristic, specifically the construction phase, parameters, local search, and path-relinking methods, we achieved results that are closely aligned with the MIP solver [1]. For the 4 km grid in the "From Scratch Optimization" test cases, the percentage coverage obtained by the heuristic differs only 0.104% from the optimal solution. For the 3 km grid, where the MIP solver is solved with a 1% optimality gap, the heuristic performed well with a percentage coverage gap of 0.265% and, in some cases, outperformed the solver solutions. In the "Real-Life Scenario Optimization" test cases, the performance gap for the 4 km and 3 km grids was reduced to just 0.096%, highlighting the practical effectiveness of the heuristic.

The most notable result occurred when solving for smaller grid sizes. Transitioning from the 3 km grid to the 2 km grid consistently improves coverage percentages. For the "From Scratch Optimization" test cases, the percentage coverage increased on average by 6.570%, while in the "Real-Life Scenario Optimization" test cases, the increase on average was 3.838%.

To evaluate potential improvements for the Timor-Leste scenario, we conducted additional optimizations of the real-life scenario using fixed capacities, as the test cases varied capacities based on the number of facilities to open. Even with large fixed capacities, such as 100,000 citizens, solving for a 2 km grid resulted in significant percentage coverage improvements of at least

[1] All results are presented as percentage coverage. Any reference to an increase in coverage refers to the absolute increase.

0.331%. These findings highlight the critical role of fine spatial resolution in enhancing overall coverage.

The findings of this thesis are of significant practical relevance, particularly for real-world facility location and resource allocation problems. Solving the problem without considering capacity and closest assignment constraints often results in overly optimistic coverage estimates that do not reflect realistic outcomes. Incorporating these constraints, however, leads to more practical and accurate solutions, as demonstrated by an improvement in percentage coverage of at least 3.873% in the "Real-Life Scenario Optimization" and for the "From Scratch Optimization" this is even more pronounced.

Furthermore, the results indicate that increasing computational resources and extending time-frames allow for additional GRASP iterations and/or solving the problem with refined grid resolutions, leading to even better outcomes. These enhancements further improve the applicability of the proposed methods (to large-scale problems).

We conclude that the main goal of this thesis has been successfully achieved. Several test instances of the Timor-Leste facility allocation problem were solved within two hours on a standard 16 GB RAM computer, demonstrating that the heuristic is both time and memory efficient. It performs robustly on a wide range of values for K, S, and p, while also having significant potential for further enhancements.

Building on these findings, there are several promising directions for future research. An immediate extension involves incorporating varying capacities for potential facility locations. As more information on feasible capacities becomes available, these can be seamlessly integrated into both the MIP solver and the GRASP heuristic. However, determining whether to open small or large facilities introduces a new level of complexity that warrants further investigation.

Another valuable direction lies in shifting the focus from service distance, such as road distance, to service time. By introducing a maximum allowable service time as a constraint, the model could better address real-world, time-sensitive applications where accessibility in minutes, rather than kilometers, is critical.

Future work could also aim to enhance the efficiency of the developed heuristic. Exploring advanced metaheuristic techniques, such as Tabu Search or similar approaches, may further improve solution quality and computational performance. In addition, refining the heuristic to handle even larger datasets, or reducing grid sizes to increase spatial resolution, could uncover further opportunities to optimize coverage.

These possibilities highlight the flexibility and potential of the methods developed in this work, offering a solid foundation for ongoing advancements and practical applications.

References

[1] Somayeh Abdi et al. “Cost Minimization for Bag-of-Tasks Workflows in a Federation of Clouds”. In: *The Journal of Supercomputing* 74.9 (2018), pp. 2801–2822. DOI: 10.1007/s11227-018-2322-9. URL: https://doi.org/10.1007/s11227-018-2322-9.

[2] Tobias Achterberg and Roland Wunderling. “Mixed integer programming: Analyzing 12 years of progress”. In: *Facets of combinatorial optimization: Festschrift for martin grötschel*. Springer, 2013, pp. 449–481.

[3] Tobias Achterberg et al. “Presolve reductions in mixed integer programming”. In: *INFORMS Journal on Computing* 32.2 (2020), pp. 473–506.

[4] Méziane Aider, Imene Dey, and Mhand Hifi. “A hybrid population-based algorithm for solving the fuzzy capacitated maximal covering location problem”. In: *Computers & Industrial Engineering* 177 (2023), p. 108982.

[5] Joyce Antonissen et al. “Optimizing Geospatial Accessibility to Healthcare Services in Low- and Middle-Income Countries”. 2023.

[6] Soumen Atta, Priya Ranjan Sinha Mahapatra, and Anirban Mukhopadhyay. “Solving a new variant of the capacitated maximal covering location problem with fuzzy coverage area using metaheuristic approaches”. In: *Computers & Industrial Engineering* 170 (2022), p. 108315.

[7] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Belmont, MA: Athena Scientific, 1997. ISBN: 1-886529-19-1.

[8] Lázaro Cánovas et al. “A strengthened formulation for the simple plant location problem with order”. In: *Operations Research Letters* 35.2 (2007), pp. 141–150.

[9] Marcos Carvalho, Levi Anatolia SM Exposto, and Aniceto da Conceição Pacheco. “Critical Determinant of Stroke Risk to Timorese People: Retrospective Analysis”. In: *KESANS: International Journal of Health and Science* 3.12 (2024), pp. 575–586.

[10] CH Chung, DA Schilling, and R Carbone. “The capacitated maximal covering problem: A heuristic”. In: *Proceedings of Fourteenth Annual Pittsburgh Conference on Modeling and Simulation*. Vol. 1983. 1983, pp. 1423–1428.

[11] Richard L Church and Jared L Cohon. *Multiobjective location analysis of regional energy facility siting problems*. Tech. rep. Brookhaven National Lab.(BNL), Upton, NY (United States), 1976.

[12] Sophie Cousins. “Health in Timor-Leste: 20 years of change”. In: *The Lancet* 394.10216 (2019), pp. 2217–2218.

[13] John Richard Current and James Edward Storbeck. “Capacitated covering models”. In: *Environment and planning B: planning and Design* 15.2 (1988), pp. 153–163.

[14] Gregory Dobson and Uday S Karmarkar. “Competitive location on a network”. In: *Operations Research* 35.4 (1987), pp. 565–574.

[15] Sahar K ElKady and Hisham M Abdelsalam. “A modified particle swarm optimization algorithm for solving capacitated maximal covering location problem in healthcare systems”. In: *Applications of intelligent optimization in Biology and Medicine: current trends and open problems* (2016), pp. 117–133.

[16] Inmaculada Espejo, Alfredo Marín, and Antonio M Rodríguez-Chía. “Closest assignment constraints in discrete location problems”. In: *European Journal of Operational Research* 219.1 (2012), pp. 49–58.

[17] Paola Festa and Mauricio GC Resende. “Hybridizations of GRASP with path-relinking”. In: *Hybrid metaheuristics*. Springer, 2013, pp. 135–155.

[18] Fred Glover. "Ejection chains, reference structures and alternating path methods for traveling salesman problems". In: *Discrete Applied Mathematics* 65.1-3 (1996), pp. 223–253.

[19] LLC Gurobi Optimization. *Mixed-Integer Programming (MIP): A Primer on the Basics*. https://www.gurobi.com/resources/mixed-integer-programming-mip-a-primer-on-the-basics/. Accessed: 2024-10-10. 2023.

[20] Ali Haghani. "Capacitated maximum covering location models: Formulations and solution procedures". In: *Journal of advanced transportation* 30.3 (1996), pp. 101–136.

[21] International Committee of the Red Cross. *Our History*. https://www.icrc.org/en/our-history. Accessed: 2024-12-06. 2024.

[22] Vinícius R Máximo and Mariá CV Nascimento. "Intensification, learning and diversification in a hybrid metaheuristic: an efficient unification". In: *Journal of Heuristics* 25 (2019), pp. 539–564.

[23] Vinícius R Máximo, Mariá CV Nascimento, and André CPLF Carvalho. "Intelligent-guided adaptive search for the maximum covering location problem". In: *Computers & Operations Research* 78 (2017), pp. 129–137.

[24] Shyam Paudel et al. "Agroforestry: opportunities and challenges in Timor-Leste". In: *Forests* 13.1 (2022), p. 41.

[25] Robin H. Pearce. "Towards a general formulation of lazy constraints". Retrieved from CORE repository: https://core.ac.uk/display/227721823. PhD thesis. The University of Queensland, 2019.

[26] Hasan Pirkul and David A Schilling. "The maximal covering location problem with capacities on total workload". In: *Management Science* 37.2 (1991), pp. 233–248.

[27] Krzysztof Postek et al. *Hands-on Mathematical Optimization with Python*. Pre-publication version. Cambridge University Press, 2024.

[28] Mauricio G. C. Resende and Renato F. Werneck. "A fast swap-based local search procedure for location problems". In: *Annals of Operations Research* 150 (2007), pp. 205–230. DOI: 10.1007/s10479-006-0154-0. URL: https://link.springer.com/article/10.1007/s10479-006-0154-0.

[29] Mauricio G. C. Resende and Renato F. Werneck. "A Hybrid Heuristic for the p-Median Problem". In: *Journal of Heuristics* 10.1 (Feb. 2004). Submitted in September 2002 and accepted in November 2003 after 1 revision, pp. 59–88. DOI: 10.1023/B:HEUR.0000019986.96257.50. URL: https://doi.org/10.1023/B:HEUR.0000019986.96257.50.

[30] Mauricio G.C. Resende and Celso C. Ribeiro. "A grasp with path-relinking for private virtual circuit routing". In: *Networks: an international journal* 41.2 (2003), pp. 104–114.

[31] Mauricio G.C. Resende and Celso C. Ribeiro. "GRASP: Greedy Randomized Adaptive Search Procedures". In: *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Ed. by Edmund K. Burke and Graham Kendall. 2nd ed. New York: Springer, 2014, pp. 287–312. DOI: 10.1007/978-1-4614-6940-7_11.

[32] Mauricio G.C. Resende and Celso C. Ribeiro. "Greedy randomized adaptive search procedures: Advances, hybridizations, and applications". In: *Handbook of metaheuristics*. Springer, 2010, pp. 283–319.

[33] Mauricio GC Resende. "Computing approximate solutions of the maximum covering problem with GRASP". In: *Journal of Heuristics* 4.2 (1998), pp. 161–177.

[34] Celso C Ribeiro and Isabel Rosseti. "Efficient parallel cooperative implementations of GRASP heuristics". In: *Parallel Computing* 33.1 (2007), pp. 21–35.

[35] Peter Rojeski and Charles ReVelle. "Central facilities location under an investment constraint". In: *Geographical Analysis* 2.4 (1970), pp. 343–360.

[36] Christopher Somogyi and Richard Church. *Solving the Capacitated Maximal Covering Location Problem*. Draft Manuscript. 1985.

[37] V.F. Stienen et al. "A method for huge-scale maximum covering facility location problems with an application to water well placement in West Darfur". Unpublished manuscript. 2024.

[38] Amit Kumar Vatsa and Sachin Jayaswal. "Capacitated multi-period maximal covering location problem with server uncertainty". In: *European Journal of Operational Research* 289.3 (2021), pp. 1107–1126.

[39] JL Wagner and LM Falkson. "The optimal nodal location of public facilities with price-sensitive demand". In: *Geographical Analysis* 7.1 (1975), pp. 69–83.

[40] World Bank. *World Bank Open Data*. `https://data.worldbank.org/`. Accessed: 2024-11-13.

Appendix: Algorithm Extra

Algorithm 14: Update Key Variables for Executing a Swap

Function `Extra(`*fac_l, fac_e, opened_facilities, theoretical_cov_e, actual_cov_e, nearest_opened_fac_e, theoretical_cov, col, K, a, near_fac_dict*`):`

Create (deep) copies of *theoretical_cov_e*, *actual_cov_e* and *nearest_opened_fac_e* (called *theoretical_cov_copy*, *actual_cov_copy* and *nearest_opened_fac_copy*) // $\mathcal{O}(n)$

Identify the demand locations for which the leaving facility is nearest (called *dps*) // $\mathcal{O}(n)$

For each demand point in *dps*, set *actual_cov_copy* to 0 and *nearest_opened_fac_copy* to -1 // $\mathcal{O}(k)$

$theoretical\_cov\_copy[row[fac\_l]] -= 1$ // $\mathcal{O}(k)$

From *dps*, identify which demand locations has still *theoretical_cov_copy* greater than 0 (called *reassign_dps*) // $\mathcal{O}(k)$

Identify, for each demand point in *reassign_dps*, which has *theoretical_cov* equal to 1 (called *simple_dps*) // $\mathcal{O}(k)$

Set *nearest_opened_fac_copy* to *fac_e* for the demand locations in *simple_dps*, remove these demand locations from *reassign_dps* // $\mathcal{O}(k)$

Identify which of the facilities in $near\_fac\_dict[fac\_l]$ are open including *fac_e* (called *opened_fac_nearby*) // $\mathcal{O}(kl)$

nearest_opened_fac_copy, $facilities\_changed =$ `Reassign_To_Second_Nearest_Opened_Facility(`*opened_fac_nearby, col, reassign_dps, nearest_opened_fac_copy*`)` // $\mathcal{O}(kl + n)$

if *simple_dps != []* **then**

  Add fac_e to *facilities_changed* // $\mathcal{O}(1)$

for *fac in facility_changed* **do** // $\mathcal{O}(g)$

  actual_cov_copy = `Update_Actual_Cov(`*fac, nearest_opened_fac_copy, actual_cov_copy, K, a*`)` // $\mathcal{O}(n)$

Appendix: Algorithm Advanced Best Improvement Local Search

Algorithm 15: Advanced Best Improvement Local Search

```
1  Function Advanced_Best_Improvement(opened_facilities, theoretical_cov,
     actual_cov, nearest_opened_fac, col, row, K, a):
2      Create (deep) copies of theoretical_cov, actual_cov and nearest_opened_fac
         (called theoretical_cov_copy, actual_cov_copy and
         nearest_opened_fac_copy)                                   // O(n)
3      Initialize the loss, gain, and extra data structures with zeros, having
         sizes p, m − p, and p × (m − p), respectively              // O((m − p)p)
4      Initialize the set near_fac_dict for all facilities         // O(mkl)
5      Initialize candidates, which are the candidates that are not opened.  // O(m)
6      Construct dictionaries look_up_openfacilities and look_up_candidates,
         which keeps track of the indices of the facilities in opened_facilities or in
         candidates                                                 // O(m)
7      for j in opened_facilities do                               // O(p)
8          index_j = look_up_openfacilities[j]                     // O(1)
9          Identify which of the facilities in near_fac_dict[i] are open (called
             opened_fac_nearby)                                    // O(kl)
10         actual_cov_j = Loss(j, opened_fac_nearby, theoretical_cov_copy,
             actual_cov_copy, nearest_opened_fac_copy, col, row, K, a)   // O(kl + ng)
           Calculate the loss in coverage by calculating the current coverage minus the
             new coverage, and assign it to loss[index_j]          // O(1)
11     for k in candidates do                                      // O(m − p)
12         index_k = look_up_candidates[k]                         // O(1)
13         theoretical_cov_withk, actual_cov_withk, nearest_opened_fac_withk =
             Gain(j, theoretical_cov_copy, actual_cov_copy, nearest_opened_fac_copy,
             col, row, K, a)                                       // O(kl + ng)
14         Calculate the gain in coverage by calculating new coverage minus the current
             coverage, and assign it to gain[index]                // O(1)
15         Identify the open facilities with overlapping demand locations, as well as those
             that share a common open facility (called extra_fac)  // O(pkl)
16         for s in extra_fac do                                   // O(p)
17             index_s = look_up_openfacilities[s]                 // O(1)
18             actual_cov_extra = Extra(s, k, opened_facilities, theoretical_cov_withk,
                 actual_cov_withk, nearest_opened_fac_withk, theoretical_cov_copy, col,
                 K, a, near_fac_dict)                              // O(kl + ng)
19             Calculate the extra[index_s, index_k] term, by the increase in coverage by
                 swapping the facilities - gain[index_k] + loss[index_s]   // O(1)
       ⋮
```

Algorithm 15: Advanced Best Improvement Local Search

```
1  Function Advanced_Best_Improvement(opened_facilities, theoretical_cov,
     actual_cov, nearest_opened_fac, col, row, K, a):
       ⋮
20     improv = True                                                    // O(1)
21     while improv == True do
22       Calculate the profit of each swap and identify the most profitable swap (in_,
           out_)                                                      // O(m(m − p))
23       Identify which of the facilities in near_fac_dict[out_] are open (called
           opened_fac_nearby)                                         // O(kl)
24       Update theoretical_cov_copy, actual_cov_copy,
           nearest_opened_fac_copy                                    // O(kl + ng)
25       Set the loss, gain, and extra values for in_ and out_ to zero, based on their
           indices in the look_up dictionaries                        // O(1)
26       Update opened_facilties and candidates, and their look_up_openfacilities
           and look_up_candidates                                     // O(1)
27       to_update_loss, to_update_gain = Adjust_To_Update_BI(in, out, col, row,
           opened_facilities)                                         // O(nl + m + pk)
28       for j in to_update_loss do                                   // O(p)
29         index_j = look_up_openfacilities[j]                        // O(1)
30         Identify which of the facilities in near_fac_dict[i] are open (called
             opened_fac_nearby)                                       // O(kl)
31         actual_cov_j = Loss(j, opened_fac_nearby, theoretical_cov_copy,
             actual_cov_copy, nearest_opened_fac_copy, col, row, K, a) // O(kl + ng)
32         Calculate the loss in coverage by calculating the current coverage minus the
             new coverage, and assign it to loss[index_j]             // O(1)
33       for k in to_update_gain do                                   // O(m − p)
34         index_k = look_up_candidates[k]                            // O(1)
35         theoretical_cov_withk, actual_cov_withk, nearest_opened_fac_withk =
             Gain(j, theoretical_cov_copy, actual_cov_copy, nearest_opened_fac_copy,
             col, row, K, a)                                          // O(kl + ng)
36         Calculate the gain in coverage by calculating new coverage minus the current
             coverage, and assign it to gain[index]                   // O(1)
37         Identify the open facilities with overlapping demand locations, as well as
             those that share a common open facility (called extra_fac) // O(pkl)
38         for s in extra_fac do                                      // O(p)
39           index_s = look_up_openfacilities[s]                      // O(1)
40           actual_cov_extra = Extra(s, k, opened_facilities, theoretical_cov_withk,
               actual_cov_withk, nearest_opened_fac_withk, theoretical_cov_copy,
               col, K, a, near_fac_dict)                              // O(kl + ng)
41           Calculate the extra[index_s, index_k] term, by the increase in coverage
               by swapping the facilities - gain[index_k] + loss[index_s] // O(1)
```