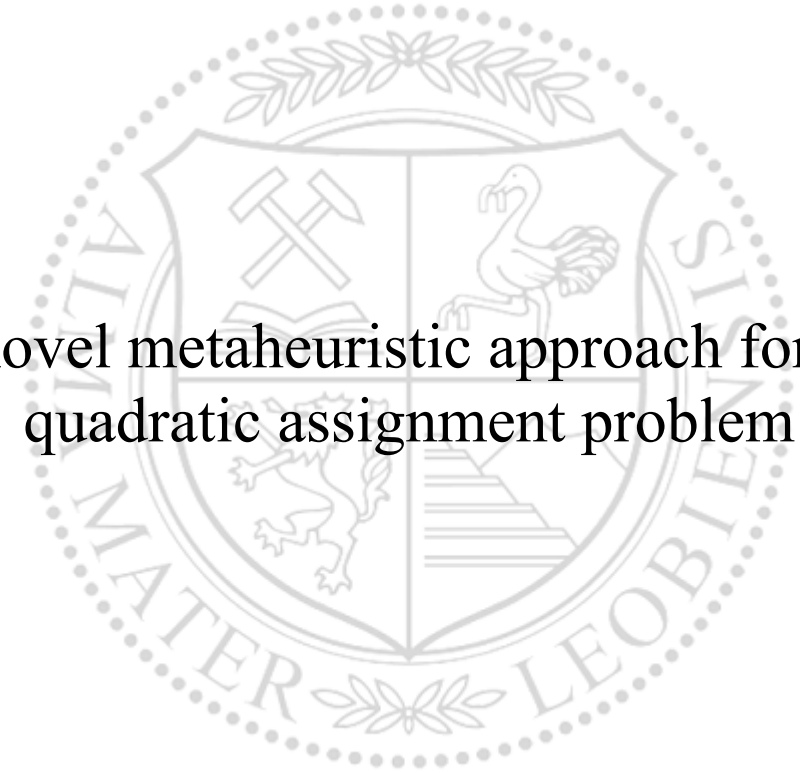
Chair of Applied Mathematics

Master's Thesis



A novel metaheuristic approach for the quadratic assignment problem

Stefan Christopher Nehl, BSc

September 2023

.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Ich erkläre, dass ich die Richtlinien des Senats der Montanuniversität Leoben zu "Gute wissenschaftliche Praxis" gelesen, verstanden und befolgt habe.

Weiters erkläre ich, dass die elektronische und gedruckte Version der eingereichten wissenschaftlichen Abschlussarbeit formal und inhaltlich identisch sind.

Datum 08.09.2023

Unterschrift Verfasser/in
Stefan Christopher Nehl

Contents

List of Figures

List of Tables

Algorithmenverzeichnis

Listings

Acknowledgements

Writing this thesis was not accomplished without any help. First, I would like to thank my supervisor, Stanek Rostislav. He guided me through the process of writing this thesis and supported me in finding the literature and structuring it. Furthermore, he provided me with regular feedback on the thesis and implementation, which helped me be more detailed and precise. Next, I would like to thank the family Elliott for proofreading this thesis. Finally, a big thanks to my friends and family, who supported me through the writing process.

Abstrakt

Das 1957 von Koopmans und Beckmann eingeführte *Quadratische Zuordnungsproblem* (QAP) stellt eine der zentralen Herausforderungen in der kombinatorischen Optimierung dar und dient als mathematisches Rückgrat für zahlreiche Anwendungen. Als *NP-hard* eingestuft, hat es sich als schwer erwiesen, optimale Lösungen für Instanzen größer als 20, zu finden.
Überdies zeigt der traditionelle Algorithmus *Branch and Bound* seine Ineffizienz für dieses Problem, primär aufgrund des Fehlens von guten Schranken für das QAP. Viele schwierige Probleme der kombinatorischen Optimierung, darunter auch das QAP, haben in den vergangenen Jahrzehnten zur Einführung von *Metaheuristiken* geführt. Zu diesen zählt auch *Scatter Search*, aus der Familie der evolutionären Algorithmen. *Metaheuristiken* dienen als übergreifender Rahmen, der in erster Linie dazu entwickelt wurde, den Beschränkungen von Verbesserungsmethoden entgegenzuwirken. Dies geschieht, indem sie suboptimale Züge ermöglichen, um lokale Extrema zu überwinden. Insbesondere die flexible Struktur von *Scatter Search* ermöglicht die Untersuchung verschiedener Strategien während seiner Ausführung. Die Grundform von *Scatter Search* enthält eine *Diversification Method*, eine *Improvement Method*, eine *Reference Set Update Method*, eine *Subset Generation Method* und eine *Solution Combination Method*.
In dieser Arbeit wurden verschiedene Algorithmen für die in *Scatter Search* verwendeten Methoden analysiert. Weiterhin wurde eine Implementierung von *Path Relinking* als zusätzliche *Solution Generation Method* implementiert. Die Idee hinter *Path Relinking* ist es, Trajektorien zu erkunden, die qualitativ hochwertige Lösungen miteinander verbinden.
Die Implementierung erfolgte in C# und folgte der *Scatter Search*-Architektur, die von Nebro A. et al. 2008 vorgestellt wurde. Zusätzlich zur Implementierung des Frameworks wurden auch parallelisierte Algorithmen implementiert.
Zur Vorauswahl der Kombinationen verschiedener Algorithmen, und um die Algorithmen zu vergleichen und die Effizienz einer parallelen Berechnung zu prüfen, wurden Benchmarks durchgeführt. Das Framework *Scatter Search* wurde über mehrere Instanzen unterschiedlicher Größen getestet. Die Testreihe umfasste zehn Instanzen mit bekannten optimalen Zielfunktionswerten und weitere zehn Instanzen, für die nur eine untere Grenze bekannt ist. Die ersten zehn Instanzen wurden für die endgültige Auswahl des Algorithmus verwendet und halfen bei der Einstellung der Parameter. Jeder Test wurde mit einer vorgegebenen Laufzeit von zehn Minuten durchgeführt. Im ersten Test lieferte der Algorithmus *Scatter Search* Ergebnisse, welche zwischen 0 und 35 % schlechter als der optimale Zielfunktionswert waren. In einigen Fällen waren die Ergebnisse jedoch um mehr als 100 % schlechter als der optimale Zielfunktionswert. Daher wurden einige Verbesserungen am *Scatter Search*-Rahmen vorgenommen. Eine davon war eine dynamische Anpassung der Größe der Referenzmenge, wenn nach der *Kombinationsmethode* keine guten Permutationen gefunden wurden. Dadurch verbesserten sich die Ergebnisse derjenigen Instanzen, die zuvor schlecht abgeschnitten hatten. Nach der Verbesserung wurden die beiden besten Kombinationen von Algorithmen ausgewählt und eine weitere Parameterabstimmung vorgenommen.
Nach dem Testen jeder Instanz mit dem *Scatter Search* Framework lagen die Ergebnisse innerhalb eines Zeitrahmens von zehn Minuten zwischen 0 und 30 %, mit Ausnahmen, über dem Optimum oder der unteren Grenze. Die Modularität des Frameworks erlaubt jedoch eine Kombination verschiedener Algorithmen, was eine Anpassung an spezifische Probleme ermöglicht.

Stichwörter: *Quadratisches Zuordnungsproblem*; QAP; *NP-schwer*; *Metaheuristik*; *Scatter Search*; *Path Relinking*; *Evolutionärer Algorithmus*

Abstract

Introduced by Koopmans and Beckmann in 1957, the *Quadratic Assignment Problem* (QAP) represents one of the pivotal challenges in combinatorial optimization, serving as the mathematical backbone for numerous applications. Classified as *NP-hard*, finding optimal solutions for instances exceeding size 20 has proven elusive.

Furthermore, the traditional algorithm *Branch and Bound* demonstrates its inefficiency for this problem, primarily due to the absence of effective bounding strategies for the QAP. Many difficult problems of the combinatorial optimization, the QAP included, led to the introduction of *Meta-heuristic* over the last few decades. Among them, is *Scatter Search* from the family of evolutionary algorithms. *Meta-heuristics* serve as overarching frameworks designed primarily to counter the limitations of improvement methods, permitting sub-optimal moves to surpass local peaks. Notably, the flexible structure of *Scatter Search* enables the exploration of various strategies during its execution. Its basic form of *Scatter Search* contains a *Diversification Method*, an *Improvement Method*, a *Reference Set Update Method*, a *Subset Generation Method* and a *Solution Combination Method*.

This thesis analysed different algorithms for the methods used in *Scatter Search*. Moreover, an implementation of *Path Relinking* as an additional *Solution Generation Method* was implemented. The idea behind *Path Relinking* is to explore trajectories which connect high-quality solutions. The implementation was done in C# and followed the *Scatter Search* architecture introduced by Nebro A. et al. in 2008. In addition to the implementation of the framework, parallelized algorithms were also implemented.

For pre-selecting the combinations of different algorithms, benchmarks were performed to compare the algorithms and verify the efficiency of a parallel computation. The *Scatter Search* framework underwent testing across multiple instances of varying sizes. The test batch comprised 10 instances with known optimal objective values and another 10 for which only a lower bound is known. The initial 10 instances were used for the final algorithm selection and assisted with parameter tuning. Every test was executed with a designated runtime of 10 minutes. In the first test, the *Scatter Search* algorithm delivered results which were between 0 and 35% worse than the optimal objective value. However, some instances had results over 100% percent worse than the optimal objective value. Therefore, some improvements were made to the *Scatter Search* framework. One was a dynamic adjustment to the reference set size if no good permutations were found after the *Combination Method*. This improved the results of those instances which performed poorly before. After the improvement, the best two combinations of algorithms were chosen and further parameter tuning was performed.

After testing every instance with the *Scatter Search* framework, the results were between 0 and 30%, with exceptions, above the optimum or lower bound, respectively, within a time frame of 10 minutes. However, the modularity of the framework allows a combination of various algorithms, which allows an adjustment to specific problems.

Keywords: *Quadratic Assignment Problem*; QAP; *NP-hard*; *Meta-heuristic*; *Scatter Search*; *Path Relinking*; *Evolutionary algorithm*

1 Introduction

Current Situation

First introduced in 1957 by Koopmans and Beckmann [8, p. 53-76], is the quadratic assignment problem is used to model and describe a high number of real-life problems. However, despite the investigation of different solving approaches, the quadratic assignment problem is considered impossible to solve for larger instances than 20 to optimality. [3, p. ix] Due to the difficulty to solve large instances of the quadratic assignment problem, a complete-enumeration approach, for example the *Branch and Bound*, are only used for small instances. [3, p. 73] Therefore, heuristics and meta-heuristics, especially population algorithms like the *Genetic Algorithm* proved to be efficient for solving large instances. This thesis investigates another population algorithm named *Scatter Search*.

Target and Research Question

Bitte hier etwas feedback für die Formulierung der Forschungsfrage. Idee wäre:

1. What does an implementation of *Scatter Search* for the quadratic assignment problem look like?

2. How does the *Scatter Search* framework perform with different implementations for the steps for small, medium and large instances?

Procedure

This thesis will start with the definition of the standard terms for the quadratic assignment problem. After that, it will describe the term effectiveness in the context of computation, time complexity and explain the issues with finding an optimal solution. Moreover, the thesis will enumerate heuristics, meta-heuristics and describe the *Scatter Search* framework in detail. In chapter 4 a detailed description of the implementation in C# will follow. Finally, section 6 will describe the setup and enumerate and interpret the results of the implementation.

Structure

The thesis is divided into four primary sections. It begins with an introduction to the quadratic assignment problem, proceeds with an overview of the Scatter Search methodology and its implementation, and concludes with a presentation of the achieved results through the implementation of the *Scatter Search* framework.

2 Quadratic Assignment Problem

This chapter will focus mainly on the quadratic assignment problem itself. An introduction to the quadratic assignment problem is given along with some historical background, followed by an explanation for its continued relevance and various examples. Furthermore, it describes how to find the optimal solution, identify the challenge of finding an optimal solution and explain a different way of solving this combination optimization problem.

2.1 Explanation

The quadratic assignment problem (QAP) was first introduced by Koopmans and Beckmann in 1957 [8, p. 53-76]. The QAP was used to model a plant location problem and, since then, has been the target of the research. It's considered a classical combinatorial optimization problem and since it's inception, has been used in a wide variety of different issues, such as placement problems, scheduling, manufacturing and many more. According to Çela et al.[3, p. xi], there are three different reasons why the quadratic assignment problem is still attractive as a mathematical model. First, a high and increasing number of real-life problems which can be modelled as a quadratic assignment problem. Second, a high number of well-known combinatorial optimization problems can be formulated as a quadratic assignment problem. Finally, the QAP is NP-hard to solve, NP-hard to approximate, not traceable and is generally considered impossible to solve with instances larger than 20 to optimize within reasonable time limits. [3, p. xi] A QAP can be formulated with a problem statement. For this statement, consider the set *{1, 2, ..., n}* and two n times n matrices $A = (a_{ij})$, $B = (b_{ij})$. Where n indicates the size of the QAP. With these parameters, the quadratic assignment problem with the coefficient matrices A and B ($QAP(A, B)$), can be written as the following cost function:

$$\min_{\pi \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{\pi(i)\pi(j)} b_{ij} \tag{1}$$

Where S_n is a set of permutations with the values *{1, 2, ..., n}*. Therefore, the QAP itself, asks for the permutation $\pi \in S_n$ which minimizes the formulation above. The result of the sum in the formulation depends on the matrices A, B and on the permutation π. Consequently, to indicate those dependencies, the formulation can be denoted as:

$$Z(A, B, \pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{\pi(i)\pi(j)} b_{ij} \tag{2}$$

The formulation $Z(A, B, \pi)$ displays the objective function of $QAP(A, B)$ and a permutation π_0 which minimizes the objective function over S_n is called an optimal solution with the optimal value. The cost function also gives the problem the "quadratic" term, because of the formulation as an integer program with a quadratic cost function.[3, p. 2] The statement in (1) is called the Koopmans-Beckmann QAP. [8, p. 53-76] There is also a more general problem formulation from Lawler. However, this thesis will focus on the Koopmans-Beckmann QAP. Moreover, according to Çela et al., results from the Koopmans-Beckmann QAP can also be extended to the more general QAP from Lawler. [3, p. 3]

2.2 Applications

Koopmans and Beckmann introduced the QAP in 1957. Both considered it as a mathematical model for assigning a set of economic activities to a set of locations. Moreover, it was the first occurrence in the context of a facility location problem. [8, p. 53-76] However, other application would be in the area of scheduling, wiring problems in electronics, design of control panels, sports, statistical data analysis, parallel and distributed computing, balancing of turbine runners and computer manufacturing. [3, p. 3-4]

Facility location

The QAP can be used in the optimization of a facility location. In this example, n is the number of facilities which are assigned to n locations. $A = (a_{ij})$ determines the flow of materials from facility i to facility j and $B = (b_{ij})$ is the matrix with distances between the facility on location i and the facility on location j. The problem value or costs are given with $a_{\pi(i)\pi(j)} \cdot b_{ij}$ for the facility $\pi(i)$ at location i and facility $\pi(j)$ at location j. The assignment of all facilities to the possible locations are represented in the permutation $\pi \in S_n$ and the total costs of an assignment are equal to $Z(A, B, \pi)$. This model can also be reused and renamed with several other applications like a hospital or campus planning [3, p. 4]

Field Programmable Gate Array

Field Programmable Gate Arrays (FPGA) require connections between the different logic blocks on a silicon chip. These logic blocks are used to implement functions like equations, multiplexers or memory elements. The way how the modules are connected in an FPGA is the first step of the configuration. Those ways or routes can be described in a routing matrix $A = (a_{ij})$ which gives the number of connections between modules i and j. The next step is the assignment of each module i to a logic block $\pi(i)$ on the chip. The length of the different links between the modules and block affects the signal propagation delay. The problem can be classified as a quadratic assignment problem due to the objective of minimizing the total propagation time. [17, p. 52]

Configuration of a keypad

Entering text on a keypad with the digits from 0 to 9 requires assigning the 26 letters of the alphabet plus space to the numbers. A typical configuration can be found on cell phones with a dedicated keypad. In order to reduce the overall typing time in a specific language, let's assume that pressing a key takes one time unit and moving from one key to another takes two time units. For example, the frequency of occurrence of the symbol j after the symbol i can be represented as a_{ij} in a typical text and the time between the press of a key in position u and position v in the matrix b_{uv}. The minimization of the problem can be denoted as a QAP. Figure 1 displays the example. (a) indicates a typical phone keyboard and (b) the optimized keyboard. That applies also to modern digital keyboard layout on mobile phones or physical keyboards for PCs.

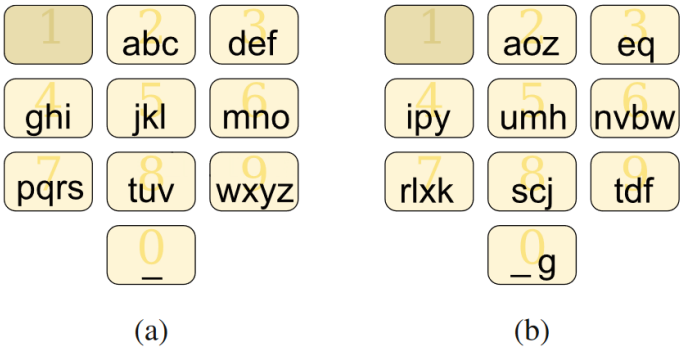


Figure 1: Standard cellular phone keyboard and keyboard optimized for the French language.(a) Standard keyboard. (b) Optimized keyboard. Source: [9, p. 12]

In addition to the typical QAP problems, many other NP-hard problems can be transformed in the QAP. Some of them are, linear ordering, travelling salesman, graph bipartition and others. However, modelling those problems as a QAP does not lead to the most efficient solving method. [17, p. 52-53]

2.3 Time Complexity

For the examples listed in Section 2.2 it would be interesting to find an efficient algorithm to solve them. In those cases, efficient means, a fast algorithm. This time constraint depends on the instance size and complexity of the problem. For example, the number of facilities and locations determine the size of the instance and, furthermore, the time to solve the problem. This time complexity function describes the approximately largest time requirements of an algorithm for a possible input length. The time complexity function can be for example, linear, quadratic, cubic and more. However, there is a distinction between polynomial algorithms and exponential algorithms.

An algorithm is considered a polynomial time algorithm if its time complexity function, denoted as $O(\pi(n))$, satisfies the condition that $|f(n)|$ is bounded by $c*|g(n)|$ for all values $n \geq 0$, where $f(n)$ represents the time complexity of the algorithm and $g(n)$ represents a polynomial function.

If an algorithm has a time complexity function $O(\pi(n))$ that can be expressed as a polynomial function, it can be classified as a polynomial time algorithm. Conversely, any other algorithm that cannot solve the problem within a time complexity function that is a polynomial is referred to as an exponential time algorithm.

The distinction between those two complexity functions is important, because of the significant time growth with the instance size is the case of an exponential complexity. Table 1 shows this time increase.

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001	.00002	.00003	.00004	.00005	.00006
n^2	.0001	.0004	.0009	.0016	.0025	.0036
n^3	.001	.008	.027	.064	.125	.216
n^5	.1	3.2	24.3	1.7 [min]	5.2 [min]	13.0 [min]
2^n	.001	1.0	17.9 [min]	12.7 [days]	35.7 [years]	366 [centuries]
3^n	.059	58 [min]	6.5 [years]	3855 [centuries]	$2 \cdot 10^8$ [centuries]	$1.3 \cdot 10^13$ [centuries]

Table 1: Comparison of several polynomial and exponential time complexity functions. (If not explicit stated, the time is in seconds) Adapted from: [6, p. 4-8]

Moreover, the improvements with faster compute times are not improving the calculation time at the same rate. For example, a time complexity with n and a 1000 faster computation time is 1000 times faster. However, with the complexity of 2^n, the 1000 faster computation time, adds only 10 to the size of the largest problem. Table 2 displays this example.

Time complexity function	With present computer	With computers 100 times faster	With computers 1000 times faster
n	N_1	100 N_1	1000 N_1
n^2	N_2	10 N_2	31.6 N_2
n^3	N_3	4.64 N_3	10 N_3
n^5	N_4	2.5 N_4	3.98 N_4
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

Table 2: Effect of improved technology on several polynomial and exponential time algorithms. Adapted from: [6, p. 4-8]

These examples, are the reason polynomial times for algorithms are more desirable than exponential times. [6, p. 4-8] According to Garey et al. [6, p. 8] "there is a wide agreement that a problem is not being 'well-solved' until a polynomial time algorithm is known for it".
Taillard gives a definition for nondeterministic polynomial problems. [17, p. 21]

> *'Informally, the class NP (standing for nondeterministic polynomial) of languages includes all the problems for which we can verify in polynomial time that a given solution produces the answer "yes." For a problem to be part of this class, the requirements are looser than for the class P. Indeed, it is not required to be able to find*

a solution in polynomial time but only to be able to verify the correctness of a given solution in polynomial time. Practically, this class contains intractable problems, for which we are not aware of a polynomial time solving algorithm.'

The quadratic assignment problem is NP-Hard. Furthermore, there is no approximation algorithm in polynomial time unless $P = NP$. [14, p. 555–565] According to Tiallard [17, p. 28] "A problem is NP-Hard, if any problem of the of NP can transform into this problem in polynomial time."

2.4 Optimal Algorithms

The NP-hardness of the QAP, leads to the conclusion that only explicit and implicit enumeration-based methods are known for solving the QAP exactly. Furthermore, the branch and bound algorithms where for a long time the most successful enumeration methods for solving the quadratic assignment problem to optimality.[3, p. 27]

Branch and Bound

Branch and Bound was first developed in 1960 by A. Land and G. Doig for mixed and pure ILP problems. In 1965 the additive algorithm for solving binary ILPs was developed by E. Balas. This further development of the algorithm was initially celebrated as a breakthrough. Unfortunately, the computational advantages were not realized. [16, p. 367] The implementation of cutting algorithms was, because of the unfeasible runtime not successful. Furthermore, it has been observed that for instances larger than 20, no optimal solutions have been found within a reasonable amount of time. Additionally, problems of size greater than 15 are generally regarded as challenging to solve. The reason for the low efficiency of *Branch and Bound* algorithms for the QAP is mainly due the lack of efficient bounding approaches for problems of a large size. This absence, confirms implicitly the difficulty of the QAP. However, one of the most studied topics in the QAP is the computation of lower bounds. Despite this effort those lower bounds have not been found yet. Those lower bounds are crucial for the *Branch and Bound* algorithm. There are five main groups of lower bounds for the QAP: *Gilmore-Lawlwer and related lower bounds, eigenvalue related lower bounds, reformulation based bounds, lower bounds based on LP relaxations* and *lower bounds based on semidefinite relaxations.*[3, p. 27-28] However, this is just an enumeration of the known lower bound techniques. Overall, only small instances of the QAP are optimally solved and many problems or applications of the QAP have instances with a larger size. Therefore, polynomial time heuristics, which have not optimal solutions for QAP instances are used for applications witch a larger size.[3, p. 73]

2.5 Heuristics

The QAP is a real-world problem which is markedly difficult to solve with medium and large instances. These difficulties lead to high computational time, which is impractical. Therefore, non-optimal methods with more feasible computation time are used to solve QAP large instances. Those non-optimal algorithms are called heuristics. The usual classification of those algorithms is constructive and local search improvement methods. [18, p. 12]

Constructive

Constructive algorithms start from scratch and gradually add parts of the solution to the empty initial solution. For example, in schedule problems, the algorithm adds additional operations of the production plan to the initial solution. Constructive algorithms are get inferior results than the local search methods. However, the benefit is usually very fast computational time, although some special implementations for complex problems may increase the computational time. [18, p. 12]

Local Search

Local search algorithms starts with an initial solution, which can be generated by a constructive algorithm or algorithms which do not generate the finding of an optimum, and improves the solution. This happens by changing parts of the solution iteratively and generating a new better solution. A drawback of the local search method is that, the algorithm get trapped easily in a local optimum. Therefore, new modern local search algorithms try with meta-strategies to escape local optima and to explore also distant neighbourhoods. [18, p. 13]

Meta-heuristics

Meta-heuristics are dominating the combinatorial optimization methods for the last decades. Those algorithms are combining heuristics with higher level frameworks. Stützle in 1999 gives a definition of meta-heuristics [15, p. 23]:

> *'Metaheuristics are typically high-level strategies which guide an underlying, more problem specific heuristics, to increase their performance. The main goal is to avoid the disadvantages of iterative improvement and, in particular, multiple descent by allowing the local search to escape from local optima. This is achieved by either allowing worsening moves or generating new starting solutions for the local search in a more "intelligent" way than just providing random initial solutions. Many of the methods can be interpreted as introducing a bias such that high quality solutions are produced quickly. This bias can be of various forms and can be cast as descent bias (based on the objective function), memory bias (based on previously made decisions) or experience bias (based on prior performance). Many of the metaheuristic approaches rely on probabilistic decisions made during the search. But, the main difference to pure random search is that in these algorithms randomness is not used blindly but in an intelligent, biased form.'*

Based on the definition from Stützle, meta-heuristics are an improved way to overcome the downfall of traditional heuristics, without increasing the computational time.

There are several kinds of heuristics and meta-heuristics which can be applied to the QAP. However, according to [3, p. 73] there are several main streams of heuristics for the QAP:

1. Construction methods
2. Limited enumeration methods
3. Improvement methods
4. Tabu search algorithms

5. Simulated annealing approach
6. Genetic algorithms
7. Greedy randomized search

The enumerated list does not display all possible heuristics for the QAP. Moreover, those methods are overlapping. Figure 2 displays further methods and the overlapping.

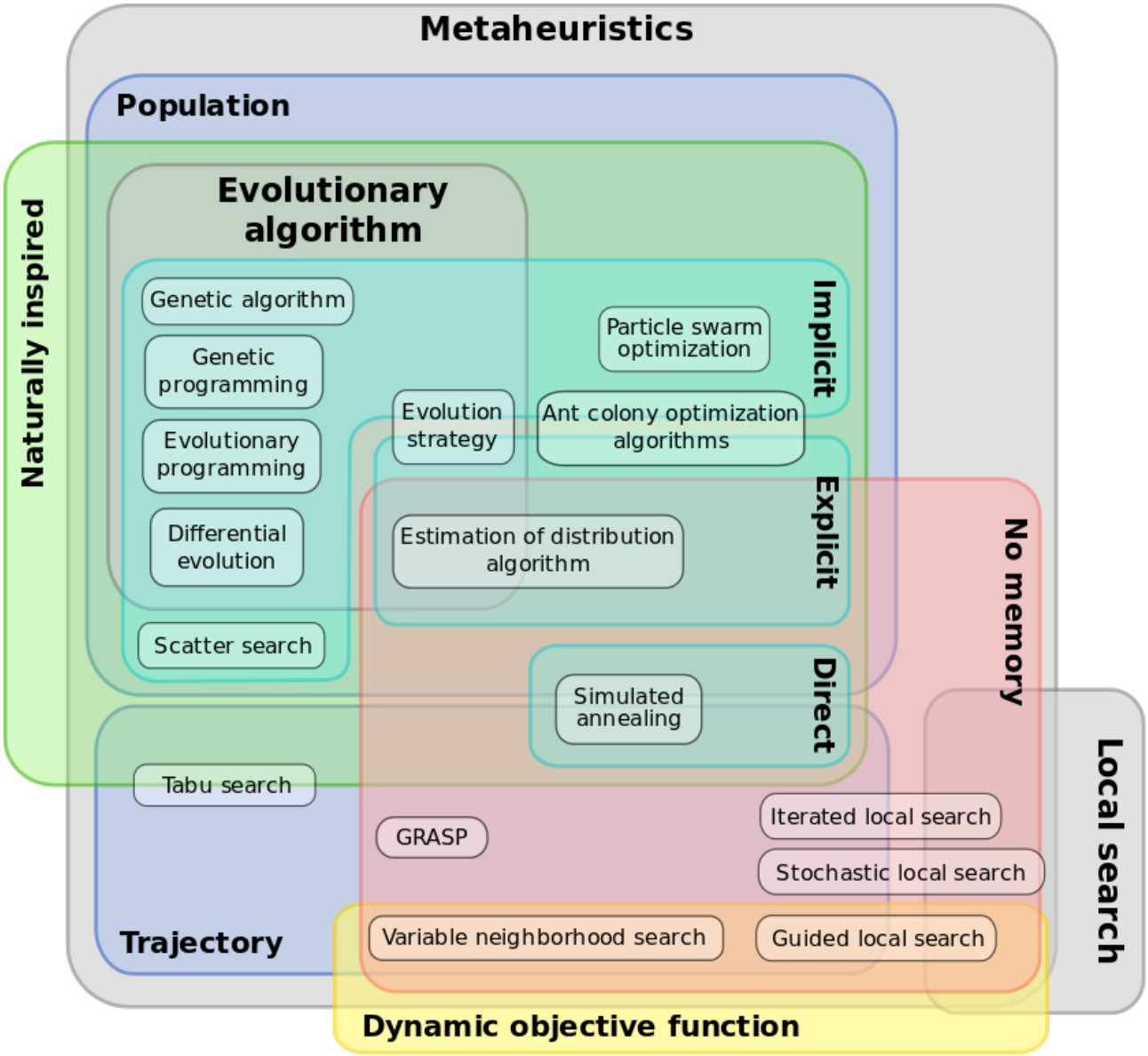


Figure 2: Different classifications of meta-heuristics shown as a Euler Diagram. Source: [5]

However, this thesis will focus on a heuristic introduced 1977 by Glover and is named *Scatter Search*.

3 Scatter Search

This chapter will take a look at a heuristic named *Scatter Search*. It will explain the idea behind it, display the structure of *Scatter Search*, and describe the different modules in the structure. Moreover, this chapter will describe the different methods used in the modules and parts of *Scatter Search*.

Scatter Search was introduced by Glover in 1977 as a heuristic for integer programming. [10, p. 2] *Scatter Search* is in the group of evolutionary algorithms, like the genetic algorithm, and has been successfully applied to hard optimization problems. The difference between *Scatter Search* and other evolutionary algorithms such as the genetic algorithm is, that *Scatter Search* uses systematic methods and strategies for creating new solutions. These methods and strategies for search diversification and intensification have proven effective in optimization problems. [10, p. 1] Adapted from Glover, *Scatter Search* is founded on the following premises [7, p. 6]:

1. A good collection of elite solutions usually contains useful information about the form (or location) of the best solutions.

2. To exploit such information, it is important to provide combinations that can extend beyond the regions spanned by the solutions considered, and to further incorporate heuristic processes to map combined solutions into new points. (This serves to provide both diversity and quality.)

3. Taking account of multiple solutions simultaneously, as a foundation for creating combinations, enhances the opportunity to exploit information contained in the union of elite solutions.

However, the *Scatter Search* procedure itself is not restricted to a single design. Moreover, it's a collection of procedures which adds additional strategies. [7, p. 7]

3.1 Structure

Scatter Search is very flexible because each part of the methodology can be implemented in different ways. The flexible design of the *Scatter Search* methodology gives the possibility to explore different strategies in the implementation. However, adapted from [10, p. 2] there are five well-known methods that build the basic structure of a *Scatter Search* implementation.[10, p. 2]

1. A *Diversification Generation Method* to generate a collection of diverse trial solutions, using an arbitrary trial solution (or seed solution) as an input

2. An *Improvement Method* to transform trial solution into one or more enhanced trial solutions.

3. A *Reference Set Update Method* to build and maintain a reference set consisting of currently the best solutions found with the size b

4. A *Subset Generation Method* to operate on the reference set and produce new subsets of the solutions for creating new combined solutions

5. A *Solution Combination Method* to transform the subsets into new combined solutions

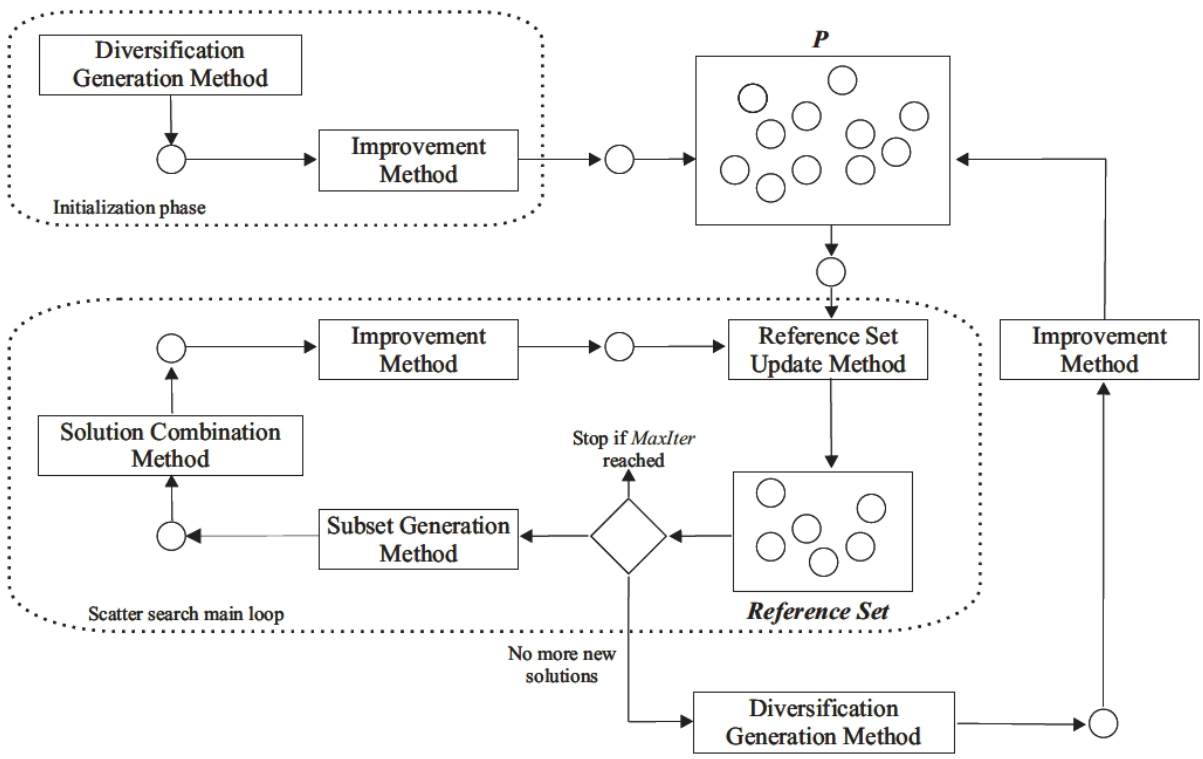


Figure 3: *Scatter Search* Template. [12, p. 12]

Those five methods are combined in the *Scatter Search* template. Figure 3 displays this template. The implementation in figure 3 starts first with the *Diversification Generation Method*, which generates a population with p solutions in the initialization phase. Second, the first improvement of every solution in the population is carried out. This improvement leads to the final population P with the size p. Next, the reference set is generated, the best b solutions of the population are selected, and the reference set, which is ordered starting with the best solutions, is established. In the *Scatter Search* main loop, the implementation starts with generating the subsets, combining those subsets into new solutions, improving the new solutions and updating the reference set. This main loop stops if the maximum number of iterations or a timestamp is reached. If the combination of subsets does not change our reference set, the algorithm uses the diversification method to generate new solutions. As already mentioned, the design of *Scatter Search* is not fixed, therefore, the implementation in other publications can differ. Algorithm 1 displays a basic *Scatter Search* procedure.

1: Start with $p = \emptyset$ Use the diversification generation method to construct a solution and apply the improvement method. Let x be the resulting solution. If $x \notin P$ then add x to P, otherwise, discard x. Repeat this step until $p = |P|$.
2: Use the reference set update method to build $RefSet = \{x_1, ..., x_b\}$ with the best b solutions in P. Order the solutions in $RefSet$ according to their objective function value such that x_1 is the best solution and x_b the worst. Set $NewSolutions = TRUE$.
3: **while** $NewSolution$ **do**
4: Generate $NewSubsets$ with new solutions using the subset generation method. Set $NewSolutions = FALSE$
5: **while** $NewSubsets \neq \emptyset$ **do**
6: Select the next subset s in $NewSubsets$
7: Apply the solution combination method to s to obtain one or more new trial solutions x.
8: Apply the improvement method to the trial solutions
9: Apply the reference set update method
10: **if** $RefSet$ has changed **then**
11: Make $NewSolutions = TRUE$
12: **end if**
13: **end while**
14: **end while**

Algorithm 1: Basic *Scatter Search* procedure. Source: [10, p. 3]

The implementation can also add additional strategies to the *Scatter Search* procedure, which would lead to more advanced approaches. However, designs of advanced *Scatter Search* algorithms have the goal of improving the performance of the implementation. Nevertheless, this often results in higher complexity and greater difficulties in the fine-tuning of the different methods. [10, p. 4]

3.2 Diversification Method

Diversity control in the population and the reference set is crucial for *Scatter Search* because the algorithm does not allow duplicated solutions in the reference set. However, comparing each index in a solution with each index in every other solution comes at a high computational cost. Therefore, there are different methods to archive the diversity between the solutions. One method is hashing. Hashing is often used to reduce the computational effort. One hashing method is reported by Campos, et. al. [2, p. 397-414]

$$hash(p) = \sum_{i=1}^{m} i\pi(i)^2 \tag{3}$$

Simple *Scatter Search* algorithms check the duplication of the solution. However, they are, in general, not monitoring the diversity of the high-quality solutions in the reference set. To do so, a diversity test can be applied on the set of high-quality solutions in the reference set. This happens after the P set has been created and only updates, removes or add the solutions, in P if it exceeds certain threshold, which depends on the type of the problem. For example, at each step when the algorithm adds the next best solution, we ensure that there is a minimum

distance between the solution. Otherwise, solutions are removed and new one generated. [10, p. 6]

3.3 Improvement Method

The improvement methods improve the solutions in the reference set or in the population. These methods are typically approaches to solve combination optimization problems and usually belong to the class of local search algorithms. A local search procedure is an iterative approach which tries to improve an already feasible solution. The improvement is achieved by replacing the current solution with a better, more feasible solution from the neighbourhood of the current one. The iterative steps are repeated until no improved solution can be found. [3, p. 77] The improvement method is not explicitly needed in a *Scatter Search* structure. However, if the implementation desires high-quality outcomes, the implementation of an improvement method is recommended. [10, p. 4]

Generating a new neighbourhood can be typically done with two different types of exchange methods. The pair-exchange and the cyclic triple-exchange neighbourhood. With the transposition of the pair-exchange neighbourhood of a given solution, or permutation, every permutation can be obtained. The cyclic triple-exchange, works by means of a cyclic exchange of three indices. Using the cyclic exchange approach, every possible permutation of the current solution can also be found. However, according to Çela et al., the cyclic triple-exchange does not lead to considerably better results compared to the pair-exchange. [3, p. 77]

The start of the local search for the improvement, can be done in two different ways. Either, with a fixed predefined order, or with a random order. In addition to the different ways of a local search start, there are three frequently used update rules to update the current feasible solution.[3, p. 78]:

- Method of *First Improvement*
- Method of *Best Improvement*
- *Heider's Method*

First Improvement updates the current solution as soon as a first improving neighbour is found. The *Best Improvement* method, first scans the whole neighbourhood and chooses the best improving neighbour solution. Ding-Zhu et al. defines the Haider's method: [4, p. 281]

> *'Heider's rule starts by scanning the neighborhood of the initial solution in a prespecified cyclic order. The current solution is updated as soon as an improving neighbor solution is found. The scanning of the neighborhood of the new solution starts there where the scanning of the previous one was interrupted (in the prespecified cyclic order).'*

3.4 Reference Set Update Method

The reference set is one of the most important parts of *Scatter Search* If the reference set only contains very similar solutions, *Scatter Search* will not be able to improve the best solution found, even if the combination method is very advanced. It is, therefore, essential to build a good reference set and maintain it throughout the search process.[10, p. 4]
The basic design for the reference set is a pool of the best solutions from the population. Those solutions are combined and if better solutions arise, the reference set is updated. This is called a *static strategy* for updating the reference set.[10, p. 4] The *dynamic update* method updates the reference set as fast as possible. This means, as soon a new solution is provided to the reference set, the combination method is used to generate new solutions.[10, p. 4]

Update Strategies

One of the advantages of the *dynamic update* strategy is, that the reference set contains solutions of better quality and that solutions with an inferior quality are replaced quicker. Furthermore, future combinations are made with already improved solutions from the reference set. However, the *dynamic update* strategy is more challenging than the *static update* strategy and therefore more difficult to implement. Moreover, the dynamic approach can eliminate some potential combinations, which makes the order of the solutions in the reference set more important. In the static counterpart, the order is not relevant because the reference set gets updated after all combinations have been performed. The dynamic update needs some tweaking in the combination orders.[10, p. 4] If there are no new trial solutions for the reference set, a rebuilding is needed.

Reference Set

The idea is to partly rebuild the reference set when the combination and improvement methods do not provide solutions of a better quality than the reference solutions. This rebuild is done with a diversification update that assumes the size of the reference set is $b = b_1 + b_2$, where b_1 for the solutions from x_1 to x_{b_1} and b_2 the solutions from x_{b_1+1} to x_b. The rebuild method deletes the solutions, b_2, from the reference set and uses the diversification method to construct a set P of new solutions. With the criterion of maximizing the diversity, the solutions b_2 are sequentially selected from P. A possible criterion for diversity is to maximize the minimum distance, between the solution values, in the context of the problem being solved. This criterion is a part of the reference set update method. When selecting solution x_{b_1+1} the criterion is applied regarding the solutions $x_1, \ldots, x_{b_1}$, selecting solution x_{b_1+2}, with $x_1, \ldots, x_{b_1+1}$ and so on.[10, p. 4-5]

Reference Set Tiers

One more possibility to handle the reference set and its solutions is to order the reference set by the objective function of a solution. Therefore, if we find a new possible solution for the reference set, we will remove the worst solution and insert the new solution in the correct position, which is given by the objective function value. This approach leads to a converging reference set, which is when there are no possible solutions to add to the reference set, and a diversification procedure needs to be done.
However, there is another possibility to maintain the reference set. Instead of doing the diversification procedure at the end, a proactive injection of diversification is possible. This leads to a splitting of the reference set into b_1, consisting of high-quality solutions, and b_2, consisting of

diverse solutions. This update procedure is called 2-tier design and has the goal of dynamically preserving the diversification in the reference set. This solved the problem, that high-quality solutions tend to be very similar to each other. The 2-tier design updates the reference set in both senses, in being a set of high-quality solutions and highly diverse solutions. Therefore, the set b_1 is ordered by the objective function value and the set b_2 is ordered by the value which defines the diversity.[10, p. 5]
The 2-tier design can be expanded to a 3-tier design. Where the reference set is partitioned in three sets. $RefSet1$ and $RefSet2$ follow the same rules as in the 2-tier approach. For $RefSet3$, we introduce a new variable called $g(x)$, which determines the object value of the best solution ever created from a combination of a solution from $RefSet1$ and any other reference solution. The third reference set is ordered according to $g(x)$. The exact comparison of $g(x)$ depends on the problem itself. For example, for a minimization problem, the order is $g(x_{b_1+b_2+1}) < g(x_{b_1+b_2+2}) < \ldots < g(x_b)$. If a better solution is found, and updating a solution in $RefSet1$ with a higher quality one is necessary, a check for in the $RefSet3$ is also needed.[10, p. 5-6]

Managing the reference set can be quite challenging, and the implementation can be tricky. It's important to find a good balance between the complexity of the implementation and the benefits of solving a problem.

3.5 Subset Generation Method

The combination method in *Scatter Search* is not limited to only combining two solutions. Moreover, the *Scatter Search* algorithm uses subsets of different sizes for combining parts of solutions. However, *Scatter Search* also ensures that a subset is created only once. This differs from the genetic algorithm, where combinations are typically determined by a roulette wheel. However, if the generation method only creates all the subsets of size two, then all subsets of size three, and so on until the method creates the reference set. For a size of b = 10, which is typical, the generation method would create 1013 subsets. This is not effective because this creates a lot of almost identical subsets.[10, p. 6] Therefore, a possibility for the *Scatter Search* algorithm is, to use different subset type generations and switch through those types in every iteration. According to Glover et al. there are 4 different subset types[7, p. 25]:

1. $SubsetType = 1$: all 2-element subsets.
2. $SubsetType = 2$: 3-element subsets derived from the 2-element subsets by augmenting each 2-element subset to include the best solution not in this subset.
3. $SubsetType = 3$: 4-element subsets derived from the 3-element subsets by augmenting each 3-element subset to include the best solutions not in this subset.
4. S$ubsetType = 4$: the subsets consisting of the best i elements, for $i = 5$ to b.

Where b indicates the max size of the reference set.

3.6 Combination Methods

The *Combination Method* uses the subsets and generates new trial solutions for the reference set. The method itself is problem-specific because it is directly related to the solution representation.[9, p. 25] For the implementation, Two different approaches for combining the subsets were chosen. First, an exhausting approach which combines every solution in the subset with each other and generates new solutions and second, a fragmented fill. The fragmented fill algorithm deletes the worst part or a random part from the solutions and fills it with the subsets. This thesis will describe the method in more detail in section 4.

3.7 Path Relinking

Path Relinking was initially a strategy for diversification in tabu search. The strategy was originally suggested by Glover and Laguna and can be considered as an extension of the combination method of *Scatter Search*. The idea is to explore trajectories which connect high-quality solutions. For the exploring, *Path Relinking* starts with one of those high-quality solutions, an initiating solution, which is used to generate a path to other, guiding, solutions. For characterizing the paths, attributes of a solution can be used and added, dropped or otherwise modified by the moves executed. Those attributes can be, for example, the edges and nodes of a graph, sequence positions in a schedule, vectors contained in linear programming basic solutions, values of variables and functions of variables. *Path Relinking* searches for attributes of high-quality solutions, by creating inducements to favour these. For creating the good attribute composition of high-quality solutions in the current solution, *Path Relinking* chooses the best move. The decision uses customary choice criteria, from a set of moves currently available, that integrate a maximum number of the attributes from the guiding solutions.[9, p. 161]. The illustration 4 shows the *Path Relinking* procedure. A represents the start point, start solution. The solid

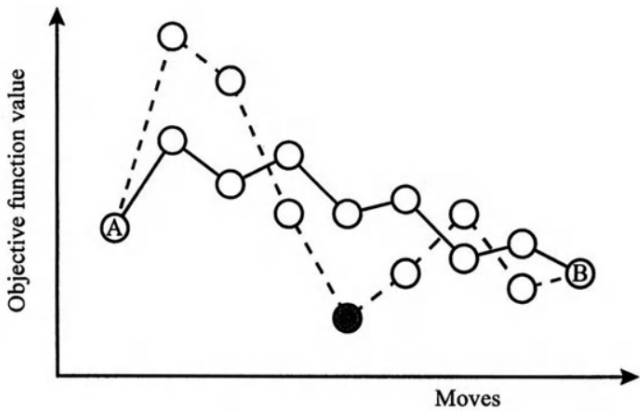


Figure 4: *Path Relinking* illustration. Source: [9, p. 162]

line shows a greedy function towards the solution with the optimal object value B. However, the optimization with the solid line takes more steps to get to the object value B. The relinked path, the dashed line, needs the same amount of steps towards the point B. Moreover, the relinked path was able to reach a better solution, the black filled one, as the greedy algorithm. [9, p. 162].

Figure 5 shows a different example. The starting solution is a permutation with an object function value. The solution gets transformed step by step into the target solution. At each

transformation step, the neighbourhood solution gets calculated and evaluated and the solution with the best value gets selected. [9, p. 162] Similar to the *Subset Generation Method*, *Path*

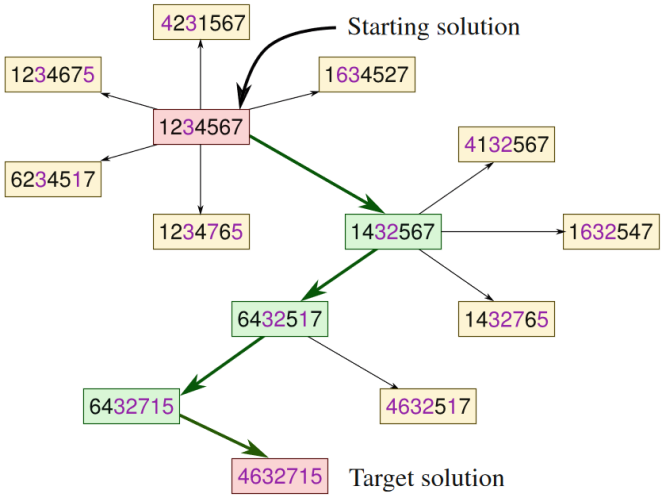


Figure 5: *Path Relinking* illustration. Source: [9, p. 162]

Relinking can use every solution from the reference set to generate the paths. Moreover, for each pair of solutions, two paths can be created with the *Path Relinking* method. One from solution A to B and one from solution B to A. In addition, an improvement method can be applied for the found solutions. However, because of the small differences, usually only one attribute, for every step, it's not efficient to apply the improvement method in every step. [10, p. 9]

4 Implementation

This chapter will describe the implementation of *Scatter Search*. Moreover, it will explain the structure of the program with the architecture, models, interfaces, and classes used in the implementation. Furthermore, it will include some considerations regarding parallel computing, identify the memory constrains and display some benchmark results to consolidate the decisions made.

The 11th version of the programming language C# , was used for the implementation of *Scatter Search*. C# is an object-oriented which is type-safe and has several features to build robust applications. It has a garbage collection, which automatically reclaims memory occupied by unreachable and unused objects, supports nullable, reference and value types and asynchronous operations. The C# program runs on .NET, which is a virtual execution system called common language runtime. In addition, the .NET architecture provides a set of class libraries and functions. [11] For this *Scatter Search* implementation, the .NET version 7.0 was used.

4.1 Program Structure and Architecture

The C# program which was implemented is hosted in a solution. A solution can contain several programs and the programs contain the files with the written code. The solution for the QAP and the *Scatter Search* implementation was split in six different programs:

- Domain
- QAP
- QAPAlgorithms
- QAPBenchmark
- QAPInstanceReader
- QAPTest

Domain

The program *Domain* contains two folders. *Models* and *TestInstances*. The folder *TestInstances* contains the instances for the QAP. These instances were taken from the webpage QAPLIB [1] and used to test the *Scatter Search* implementation The folder *Models* contains the class representation of the QAP. The instances are stored in the *record QAPInstance*. A *record* is an inmutable, not changeable after the first initialization, reference type. This reference type contains the following properties: *string InstanceName, int N, int[,] A* and *int[,] B*. *InstanceName* contains the name of the instance, N the instance size and the two matrices A and B with for example distances and problem value. (This depends on the problem) The *struct InstanceSolution* contains the array *SolutionPermutation*, with the permutation of a solution, *SolutionValue*, the objective value and the property *HashCode* with the hash code of the permutation. In contrast to the record, a *struct* is a value type in C# . In addition to the folders, the program *Domain* also has a static class *InstanceHelpers*. This class has several static helper methods for the *Scatter Search* algorithm and the QAP. For example, the method *GetSolutionValue* calculates the objective value of a given *InstanceSolution*, and the method *GenerateHashCode* generates a hash code for the instance solution. The generation of the hashcode and the calculation of the objective value follows the formula in section 2.

QAP

The program *QAP* contains the files *CSVExport*, which exports test results to a csv file. The file *Program* is the start class and entry point of the solution. *TestSettings* contains the parameters and instances of the used algorithms for a *Scatter Search* run. The settings are a combination of the QAP instance, the population generation method, diversification method, combination method and improvement method. This class also contains the maximum runtime of the algorithm and is the way subsets are generated. *TestInstance* contains the method *StartTest* which start a *Scatter Search* run with the given *TestSettings*. The result, objective value, final permutation, runtime and the number of iterations needed is stored in the *TestResult* class.

QAPAlgorithms

QAPAlgorithms contains the interfaces and the implementation of the different algorithms for the combination, diversification, generation, and improvement. It also contains the single and multithreaded implementation of the different algorithms. This thesis will explain which algorithms were used in subsection 4.3. However, the main class for the *QAPAlgorithms* program is the *ScatterSearchStart* class. At initialization, this class needs the implementation of the methods for population generation, diversification, combination, and improvement. The injection of the methods happens via interfaces. Those interfaces can be found in the folder *Contracts*. The implementation of the needed algorithm needs to implement the correct interface. For example, a new combination algorithm needs to implement the interface *ICombinationMethod*.

The class *ScatterSearchStart* also contains the functions *Solve* which starts the *Scatter Search* algorithm and, after the finalization, returns a tuple with the *InstanceSolution*, the runtime in seconds and the number of iterations. The algorithm finished after a given runtime. This runtime in seconds has to be passed via a parameter in the function *Solve*.

QAPBenchmark

The program *QAPBenchmark* contains the classes and functions for benchmarking different parts of the implementation. This thesis will explain this in further details in section 4.5.

QAPInstanceReader

QAPInstanceReader implements every function for reading test instances. The test instances are stored in different folders. The method *ReadFileAsync* has as parameter the folder and the file name and returns the *QAPInstance* filled with the values in the files.

QAPTest

The program *QAPTest* contains the unit tests for the algorithms. The implementation of *Scatter Search* can be quite difficult and debugging, finding and fixing errors can be very time-consuming. For this reason, almost every part of the program is covered by unit tests. This made finding errors much easier and therefore changes in the code can be carried out without to much effort.

4.2 Scatter Search Architecture

The *Scatter Search* implementation followed the in the subsection 3.1 and figure 3 explained implementation. For the reference update function the static approached was used. The static approach was used because of the easier implementation with different improvement, combination, and diversification methods. *Path Relinking* and *Scatter Search* outweighs the speed-up update of the reference set function. The static approach also enables an easier implementation of a parallel update function for the reference set. The main part of the implementation is the *ScatterSearch* Class itself. It combines the different population, diversification, combination, improvement, and solution generation methods to solve a given QAP instance. The combination of the classes for the different algorithms is done via interface and composition via injection over the constructor of the *ScatterSearch* class. Figure 6 shows the class with the different interfaces for the population, diversification, combination, improvement, and solution generation method.

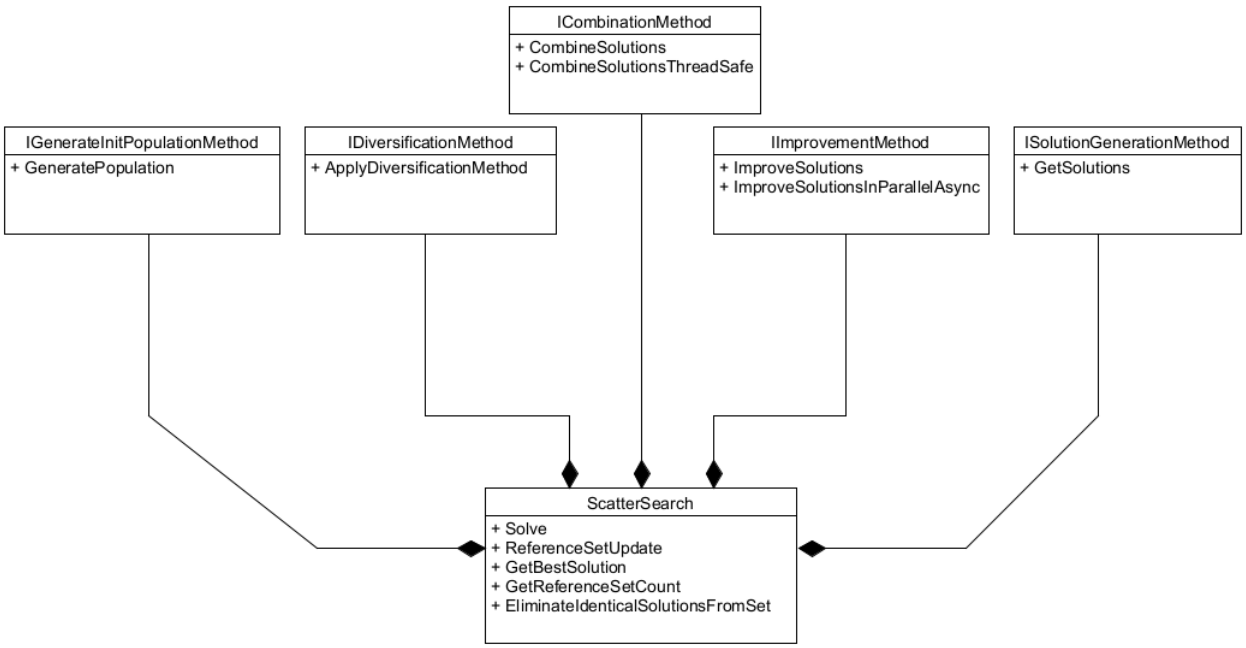


Figure 6: *Scatter Search* class with the interfaces for the interfaces for the different algorithms.

The created *ScatterSearch* class has the method *Solve* which applies the *Scatter Search* algorithm with the via the constructor injected methods. The *Solve* method takes an object from the *QAPInstance* class, the max runtime in seconds and wherever the progress should be displayed in the console. However, the last parameter is set to false as default and is only for debugging reasons. The figure 7 displays the steps of the *Solve* method.

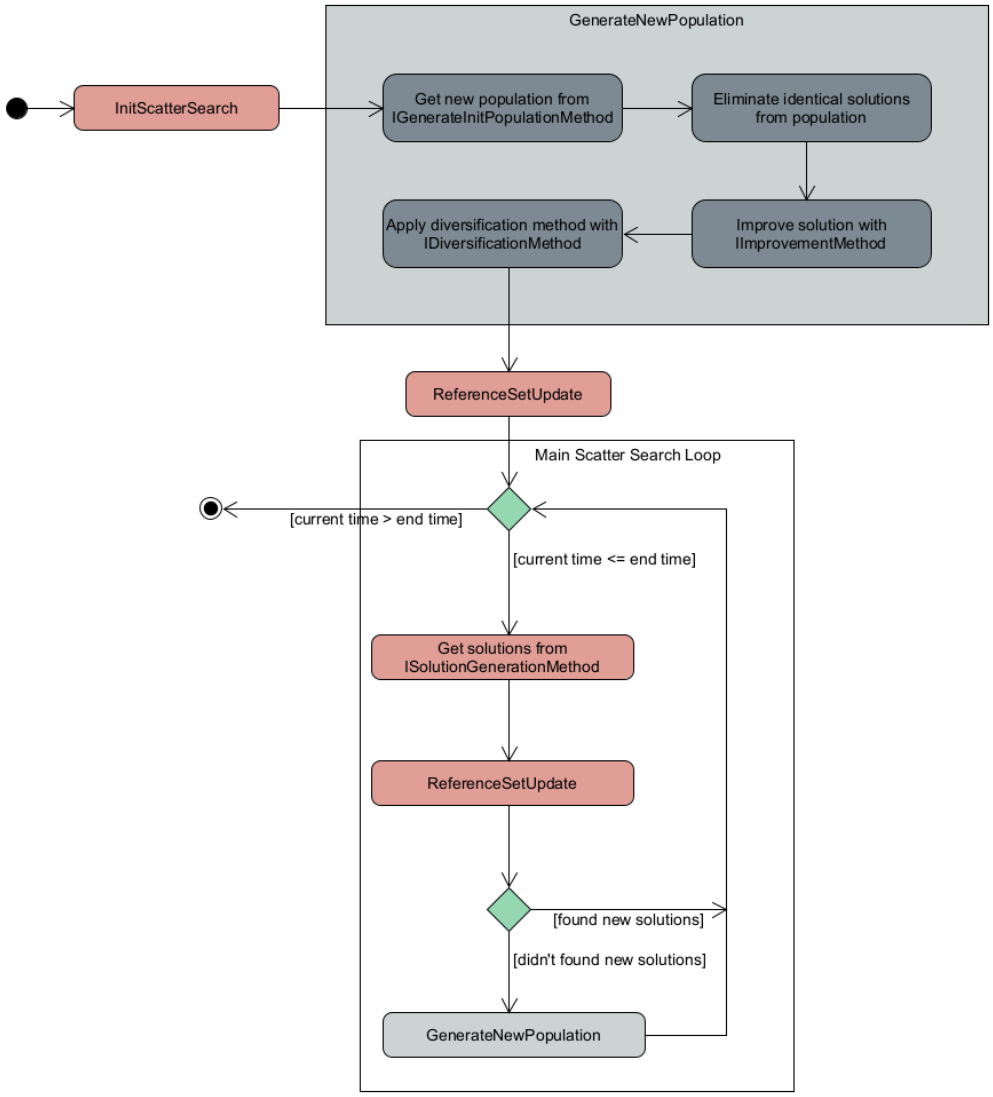


Figure 7: Activity diagram for the *Solve* method.

The method starts with the sub-method *InitScatterSearch*. This method initializes the needed variables for the *Scatter Search* algorithm. This includes the start and end time, the iteration count and initializing the different methods for the population generation, diversification, combination, improvement and solution generation. Next, the *GenerateNewPopulation* sub-method is called, which generates a new population, applies an elimination of duplicated solutions, improves the solutions in the population set and applies a diversification method. After the *GenerateNewPopulation* sub-method, the *ReferenceSetUpdate* method gets called for every solution in the population set. The sub-method *ReferenceSetUpdate* updates the reference set and sorts it after the objective value, where the index zero is the best solution with the lowest objective value. Next, the main *Scatter Search* loop starts. The loop starts with comparing the current time with the end time. If the current time is higher than the end time, the loop stops

and the solve method ends. If the current time is less than the end time, the *Scatter Search* main loop continues. First, with generating new solutions with the *ISolutionGenerationMethod* and updating the reference set with the *ReferenceSetUpdate* method. Second, the main loop checks, if new solutions were added to the reference set after the *ReferenceSetUpdate* . If yes, the loop jumps back to the start. However, if not, the *GenerateNewPopulation* method is called and then the loop starts again.

4.3 Algorithms

As explained in section 3.1, interfaces were used for implementing the different algorithms for the combination, diversification, improvement, population generation and solution generation, and they were injected into the main *Scatter Search* class. Most of those algorithms have different implementations:

- Combination Methods:

  DeletionPartsOfTheFirstSolutionAndFillWithPartsOfTheOtherSolution

  ExhaustingPairwiseCombination

- Diversification Methods:

  HashCodeDiversification

  HashCodeThreePartDiversification

- Improvement Methods:

  ParallelImprovedLocalSearchBestImprovement

  ParallelImprovedLocalSearchFirstImprovement

  ImprovedLocalSearchBestImprovement

  ImprovedLocalSearchFirstImprovement

  LocalSearchBestImprovement

  LocalSearchFirstImprovement

- Initial Population Generation Methods:

  RandomGeneratedPopulation

  StepWisePopulationGeneration

  ParallelRandomGeneratedPopulation

- Solution Generation Methods:

  ParallelPathRelinking

  ParallelPathRelinkingSubSetGenerationCombined

  ParallelSubSetGeneration

  PathRelinking

  PathRelinkingSubSetGenerationCombined

  SubSetGeneration

Combination Methods

Two different combination methods where implemented. The *DeletionPartsOfTheFirstSolutionAndFillWithPartsOfTheOtherSolution* and the *ExhaustingPairwiseCombination*. Both use a base class, the *CombinationBase* for shared functions. For example, the method *WereSolutionsAlreadyCombined* in the *CombinationBase* stores already combined solutions and checks if the composition was already combined. Furthermore, the *WereSolutionsAlreadyCombined* is also available as a thread safe method which allows to check the solutions in parallel. For comparison of the solutions a new hash code was generated and stored in a hash set and in a concurrent dictionary for the thread save version. A hash set is a collection which contains no duplicated elements and provides a high-performance set of operations.[11] The hash code was calculated from the following formulas:

$$hash(s) = \sum_{i=1}^{m} s(i)_h * (i) \tag{4}$$

Which was used for combinations where the order of the solutions is necessary. This is the case for the *DeletionPartsOfTheFirstSolutionAndFillWithPartsOfTheOtherSolution* algorithm.

$$hash(s) = \sum_{i=1}^{m} s(i)_h \tag{5}$$

To calculate the hashcode without taking into consideration the order of the solutions. In the above formulas, s indicates the set of solutions for the combination and $s(i)_h$ the hash code of the solution in the set s at the position i. Because of the hash code and the hash set, a check if the solution combination has already been used is possible in $O(1)$. Both calculation methods have the possibility to generate a non unique hashcode. However, the probability of loosing possible combinations is very low. Moreover, the possibility that a better solutions can be found within the lost solutions is negligible.

The *DeletionPartsOfTheFirstSolutionAndFillWithPartsOfTheOtherSolution* method, deletes parts of a solution and fills the deleted part with a part of an alternative solution.
The percentage of how much of the solution should be deleted is passed via the constructor. The part which should be deleted can be chosen at random, or the part with the worst object value is chosen.
The *ExhaustingPairwiseCombination* extracts every pair of a solution and tries to combine those pairs in every possible way. This algorithm has two parameters. One is for the step size of the pairs. This parameter indicates how many indices the pointer is moved after a pair has been selected. For example, if the step size is 1, the algorithm would find for the permutation $(0, 1, 2, 3)$ the pairs $(0, 1), (1, 2), (2, 3), (3, 0)$ and for the step size of two, $(0, 1)$ and $(2, 3)$. This parameter influences how many new solutions are found with the combination method. The other parameter, *maxNumberOfPairs* is a limit for the maximum number of pairs used for the combination.

Diversification Methods

For the diversification of the reference set, the *HashCodeDiversification* algorithm was implemented. This algorithm works by calculating the average possible hash code with the minimum and maximum hash code of a permutation. This calculation takes place in the *InitMethod* and the average hash code does not change during a solve process. If the diversification is applied to a set, the algorithm calculates the hash code of the set with the following formula:

$$hash(s) = \sum_{i=1}^{m} s(i)_h, \tag{6}$$

In formula 6, s indicates the set of solutions for the combination and $s(i)_h$ the hash code of the solution at the position i in the set s. The result is then divided by the number of solutions in the set. This results in an average hash code for the set. After the calculation of the average hash code of the set, the algorithm removes the worst part, which is an adjustable parameter, of the reverence set and fills it with solutions with a hash code which is as far away from the average hash code as possible.

In addition, a second diversification method was implemented which increases the hash code diversity of the reference set. The *HashCodeThreePartDiversification* also deletes the worst part of the reference set and fills the reference set with three different types of solutions. The first part has a hash code near the best solution of the reference set. The solutions in the second and third part includes solutions which are as far away as possible from the best solution of the reference set. The solutions in the second part have a hash code lower, in the third part higher than the best solution in the reference set.

The source for new solutions to fill the reference set for both diversification methods is the population. This population is newly generated each time before applying the diversification method.

Improvement Methods

For the improvement methods, the local search algorithms with first and best improvement were implemented. The algorithm checks the neighbourhoods from the solution by swapping pairs in the solution. The first improvement algorithm stops at the first improvement it can find, the best improvement iterates over all swaps and uses the best improvement. In addition, an improved version was included where the objective value is calculated in a more efficient way. In the default version, the object value was calculated with formula 1. This formula takes $O(n^2)$ to calculate the objective value. However, for the implementation of the improved version the method uses the known indexes of a swap and takes only $O(n)$ for calculating the objective value. The algorithm 2 displays the implementation.

1: Get I, p, i and j.
2: $d = 0$
3:
4: **for** $i = 0$; $i < p.length$; $i{+}{+}$ **do**
5: $\quad d = d - I.A[i,k] * I.B[p[i],p[k]]$
6: $\quad d = d - I.A[j,k] * I.B[p[j],p[k]]$
7:
8: $\quad d = d + I.A[i,k] * I.B[p[j],p[k]]$
9: $\quad d = d + I.A[j,k] * I.B[p[i],p[k]]$
10: **end for**
11:
12: $d = d + I.A[i,j] * I.B[p[i],p[j]]] * 2$
13:
14: *return* $d * 2$

Algorithm 2: Improved implementation of the object value calculation.

The variable I contain the 2-dimensional arrays A and B which are needed for the calculation of the new objective value. p is the permutation. The variable i indicates the first index and j the second index for the swap. First, the algorithm introduces the variable d which stores the delta of the objective value before and after the swap of the values on the indices i and j. Second, the $For - loop$ gets initialized to loop from zero to $p.length$. Next, delta gets reduced by the actual values of the permutation. Fourth, delta gets increased by the values after the swap. After the loop, d gets increased by doubled amount of the value for i and j. Last, the variable d gets doubled and returned. This thesis will demonstrate the performance improvements from the improved method of calculation of the solution value in section 4.5.

In addition to the improved version, the improvement algorithms include a parallel calculation for the improved algorithms.

Initial Population Generation Methods

The initial generation methods include two different algorithms. First the *RandomGeneratedPopulation* and second the *StepWisePopulationGeneration*. The *RandomGeneratedPopulation* generates several solutions for the population. This is achieved by generating a list with the size n. The size n is the size of a permutation. The list is filled with the numbers from zero to $n-1$. Next, a random number r is generated from the numbers from zero to $n-1$. The number on the index of the generated number r is removed from the list, and the next random number r from zero to the new size $n-1$ is generated. This continues until the list is empty, and the permutation arrives at the solutions. However, this algorithm does not check for duplicated permutations and therefore this check needs to be carried out after the generation. The *StepWisePopulationGeneration* algorithm fills the first permutation p_1 with the values from 0 to the size of the permutation n. The next population p_2 moves the start point to the right and starts with 1 to n and ends with 0. For example, with a permutation size 4, the permutations are $p_1 = (0, 1, 2, 3)$ and $p_2 = (1, 2, 3, 0)$. The step of the shift can be set via the parameter $nrOfIndexesToMovePerIteration$. However, for a large population size and a small permutation size, the algorithm produces very similar permutations and duplicated solutions. Therefore, it was not used in the further implementation.

Solution Generation Methods

The solution generation methods are the algorithms for the subset generation and *Path Relinking*. The algorithms, *SubSetGeneration, PathRelinking, PathRelinkingSubSetGenerationCombined* and their parallel versions were implemented. The *SubSetGeneration* algorithms follows the implementation given in section 3.5. The procedure implements the 4 different generation types and cycles through those types. Moreover, the implementation added 2 parameters which can be set in the constructor. First, the parameter *typeCount* which indicates the start of the cycling and second, the parameter *subSetGenerationMethodType*. This parameter is an enum, has the values *Cycle* and *UseOnlyOne* and indicates two different procedures of the algorithm. One where the algorithm cycles through the subset generation types and one where the algorithm stays at the type set in the parameter *typeCount*. In addition to the two parameters, the constructor has also the improvement and combination method as necessary parameters. Similarly, the implementation of the *SubSetGeneration* follows the explanation of *Path Relinking* given in section 3.7. The algorithm *PathRelinking* generates two different paths between all solution pairs of the reference set. One with the starting solution s_1 and guiding solution s_2 and one with the starting solutions s_2 and guiding solutions s_1. For the path generation, the algorithm adds one attribute of the guiding solution to the starting solution until they have the same hash code. The algorithm also implements the improvement method. However, according to Martí et al. [10] the improvement method should not be applied in every iteration. Therefore, the constructor of *PathRelinking* algorithm also supports the parameter *improveEveryNSolutions* which indicates when the improvement method should be applied. The *PathRelinkingSubSetGenerationCombined* algorithm combines *Scatter Search* and *Path Relinking* to generate even more possible solutions.

4.4 Parallel Implementations

Several algorithms of the *Scatter Search* implementation can compute their results in parallel. However, generating tasks for different threads allocates additional memory which needs to be released by the garbage collector after the thread terminates. Therefore, additional tasks can potentially slower an algorithm. Consequently, this implementation of *Scatter Search* uses a parallel computation for algorithms which needs to be applied to many solutions or algorithms that have to create numerous solutions. Therefore, a parallel implementation was employed for the improvement methods and initial population methods and solution generation methods. Moreover, the implementation of the combination methods is thread safe and can be used in parallel. However, for the diversification methods no parallel implementation was considered in this thesis. The parallel implementation was tested and benchmarked. The results are displayed in section4.5.

Improvement methods

The improvement methods include the parallel implementation of the improved algorithms of the local search algorithms for first and best improvement. The parallel computation was not implemented for the improvement methods themselves. Moreover, the improvement methods improve the solutions in parallel. The implementation of the parallel algorithms was done via a task list. For every solution a task was created, started and added to a list. When every task is finished the method is complete. However, the algorithm only starts the parallel computation if the list of solutions is greater than five. The code in listing 1 displays the implementation.

```
public new void ImproveSolutions(List<InstanceSolution>
    instanceSolutions)
{
    if (instanceSolutions.Count <= 5)
    {
        base.ImproveSolutions(instanceSolutions);
        return;
    }

    var taskList = new List<Task>();
    for (int i = 0; i < instanceSolutions.Count; i++)
    {
        var i1 = i;
        var newTask = Task.Factory.StartNew(() =>
            instanceSolutions[i1] = ImproveSolution(
            instanceSolutions[i1]));
        taskList.Add(newTask);
    }
    Task.WhenAll(taskList).Wait();
}
```

Listing 1: Algorithm of the parallel best improvement algorithm

A similar approach was used for the implementation of the initial population generation methods. However, to store the solutions a concurrent bag was used. The concurrent bag is a thread safe collection in C# . Listing 2 displays the code.

```
        var taskList = new List<Task>();
        _newSolutions.Clear();

        for(int i = 0; i < populationSize; i++)
        {
            var task = Task.Factory.StartNew(() =>
            {
                _newSolutions.Add(GenerateSolutionThreadSafe());
            });
            taskList.Add(task);
        }

        Task.WhenAll(taskList).Wait();
        return _newSolutions.ToList();
    }

    private InstanceSolution GenerateSolutionThreadSafe()
```

Listing 2: Algorithm for the parallel initial population generation method.

The implementation for the solution generation method follows the same scheme as the implementation for the initial population generation methods

4.5 Benchmarks

The multitude of methods for generating, combining, and optimizing solutions, as well as the multitude of parameters, can be exhausting when implementing *Scatter Search*. In order to eliminate inefficient algorithms and generate a preselection of test methods, benchmarks were created.
For generating the benchmarks, the library BenchmarkDotNet was used. This library is easy to use and generates reliable and precise benchmark results.[13] A benchmark can be created with the code displayed in listing 3.

```
[MemoryDiagnoser]
public class TestBenchmark
{
    private ImprovedLocalSearchFirstImprovement
        _improvedLocalSearchFirstImprovement;
    private InstanceSolution _instanceSolution;

    [GlobalSetup]
    public void SetUp()
    {
        _improvedLocalSearchFirstImprovement = new
            ImprovedLocalSearchFirstImprovement();
        _instanceSolution = new InstanceSolution();
    }

    [Benchmark]
    public void Benchmark()
    {
        _improvedLocalSearchFirstImprovement.ImproveSolution(
            _instanceSolution);
    }
}
```

Listing 3: Small example benchmark.

And run with the following code:

```
    BenchmarkRunner.Run<TestBenchmark>();
```

Benchmarks for the initial population generation, the improvement method, instance helpers and the solution generation methods were created. For the benchmarks, three different instances of different size were used. Those instances are, *chr*12*a.dat*, *chr*25*a.dat* and *tai*256*c.dat*. The sizes of the instances are 12, 25 and 256. Moreover, not only the algorithms where tested with different solution sizes, number of solutions generated and the number of calls were benchmarked. The number of calls was tested with 10, 100 and 200 calls for a particular setting. Those benchmarks should give an indication for good heuristics in *Scatter Search*. The benchmarks were run on a pc with an AMD 5800X3D processor with 8 cores, 16 threads and a maximum CPU clock from 4.5 GHz. However, in multicore scenarios the CPU only clocks to 4.2 GHz. In addition to the CPU, 32 GB of memory were used. No other tasks except for the benchmarks were performed on this pc.

Generate Initial Population Benchmark

The generation of the initial population is important for the start of the *Scatter Search* methods. Furthermore, if the *Scatter Search* algorithm cannot update the reference set, a generation method creates a new population. However, the call of the generation method is far more seldom used than the improvement one. Therefore, it's more influential that the generation method generates a diverse population than the runtime of the method itself. Figure 8 displays the time in microseconds of the different generation methods. The figure shows the increased performance of the parallel random generation method with larger solutions.

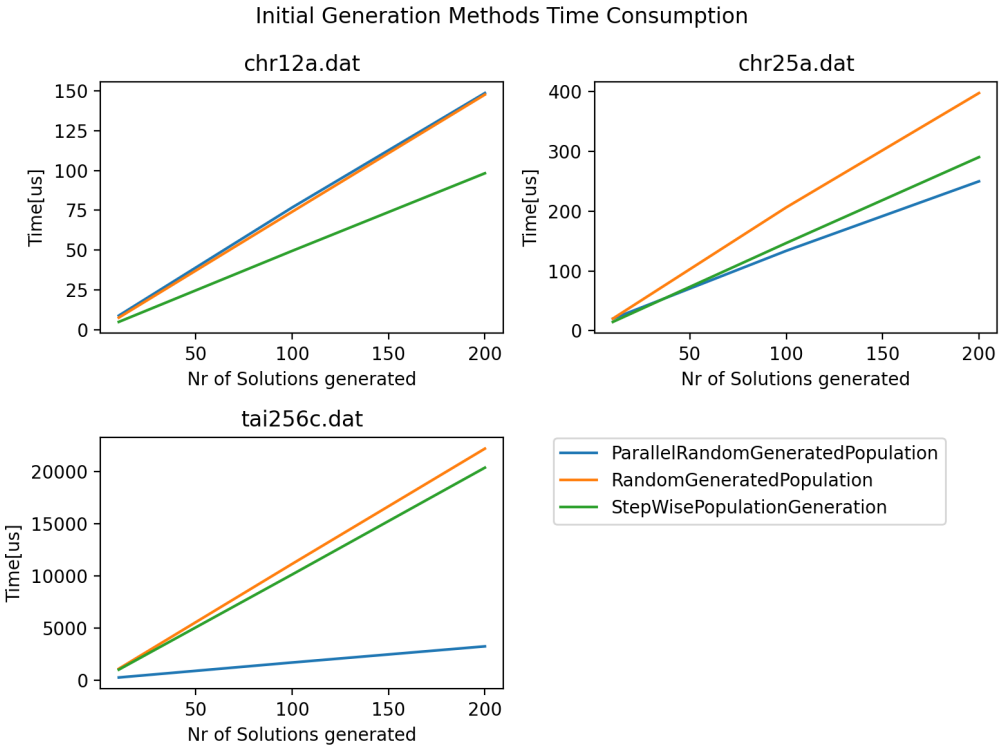


Figure 8: Runtime in microseconds for the population generation methods.

Figure 9 compares the memory consumption of the population generation methods. It shows the increased memory consumption of parallel implementations. However, as mentioned, the performance of the population generation method is not as critical. Moreover, generates the StepWisePopulationGeneration method, does not generate sufficiently diverse solutions for the population. Therefore, the focus was kept on the random generation methods for the *Scatter Search* implementation.

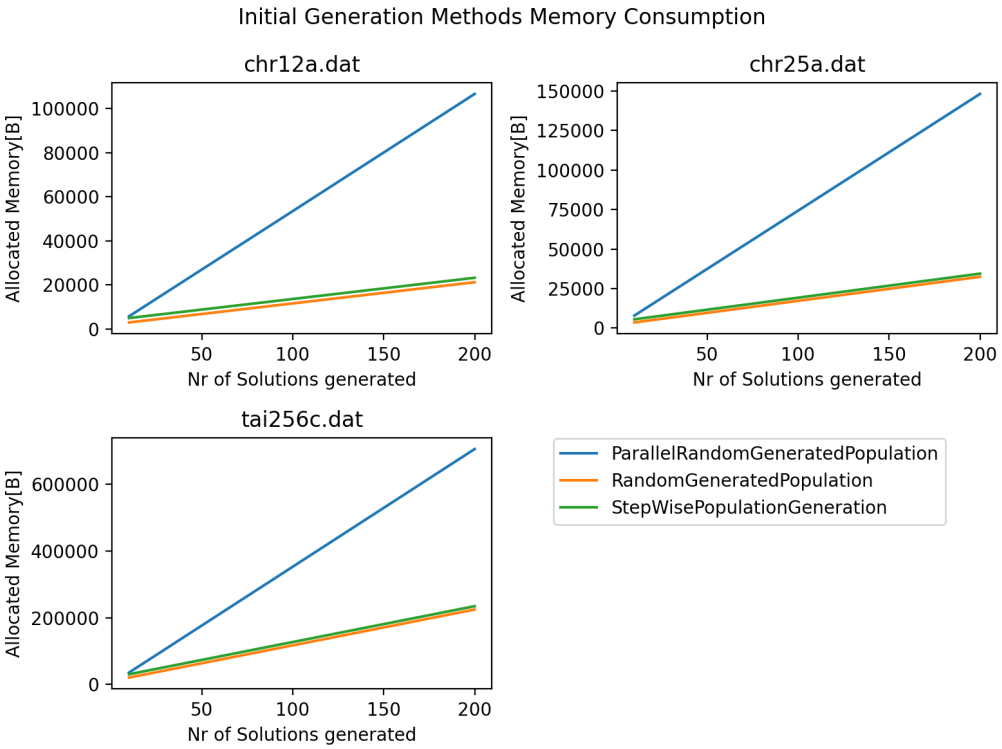


Figure 9: Memory consumption in bytes for the population generation methods.[1]

Improvement Method Benchmark

The improvement methods are called in for every iteration of the newly generated solutions. Therefore, the improvement methods have a high impact on the performance of *Scatter Search*. Hence, a benchmark was created for all the different implementations of the improvement methods. The benchmarks include the *Improved Local Search Best Improvement, Improved Local Search First Improvement, Local Search Best Improvement, Local Search First Improvement* and the parallel implementations of the *Improved Local Search Best Improvement* and *Improved Local Search First Improvement*. Figure 11 plots the results of the benchmarks. It shows an increased performance with the improved versions of the improvement methods. This increased performance is the result of a more efficient calculation of the solution value after a swap. This computation of the solution value has a runtime of $O(n)$ instead of the regular implementation with $O(n^2)$. Therefore, the improved algorithm demonstrates a much lower runtime. Moreover, figure 11 and 13 display the additional effort for the parallel computation. This additional effort is a result of the state machine initialization for the computing on different threads. Hence, the time benefit of the parallel computation is not present. Overall, the results of figure 10 and 12 lead to the conclusion, that the most efficient ways for improving the solutions are the faster algorithms for first and best improvement. This result also be clearly seen in figure 11 and 13 which display the increased memory consumption for the parallel implementations.

[1]Small transformation of the lines to prevent overlapping in the figure

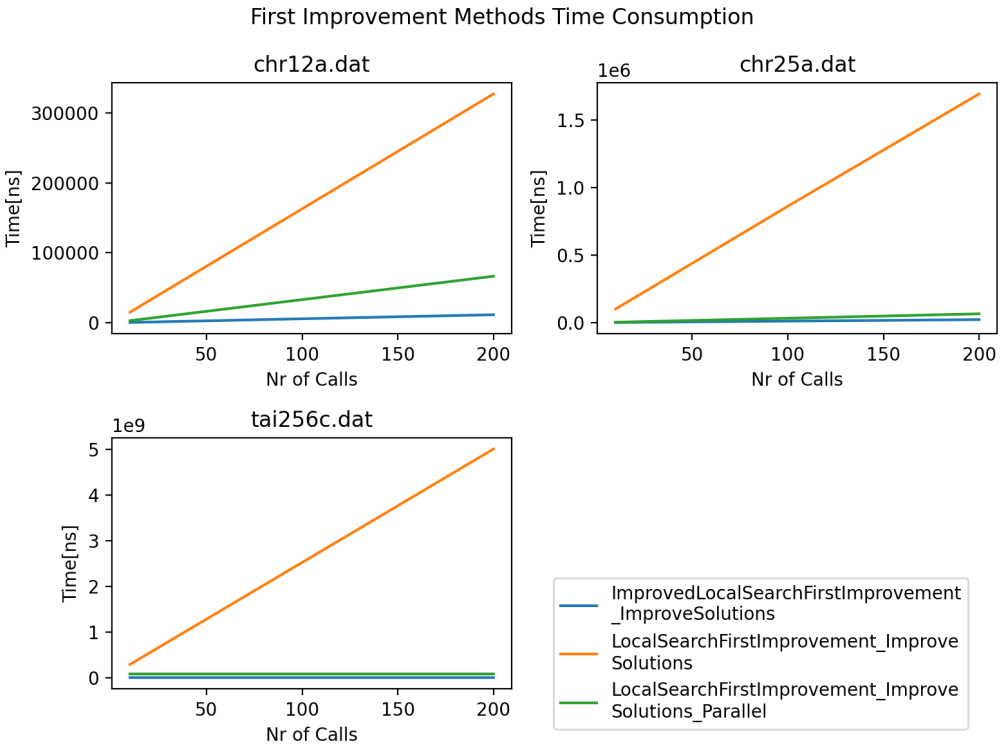


Figure 10: Runtime in nanoseconds for the first improvement methods.[1]

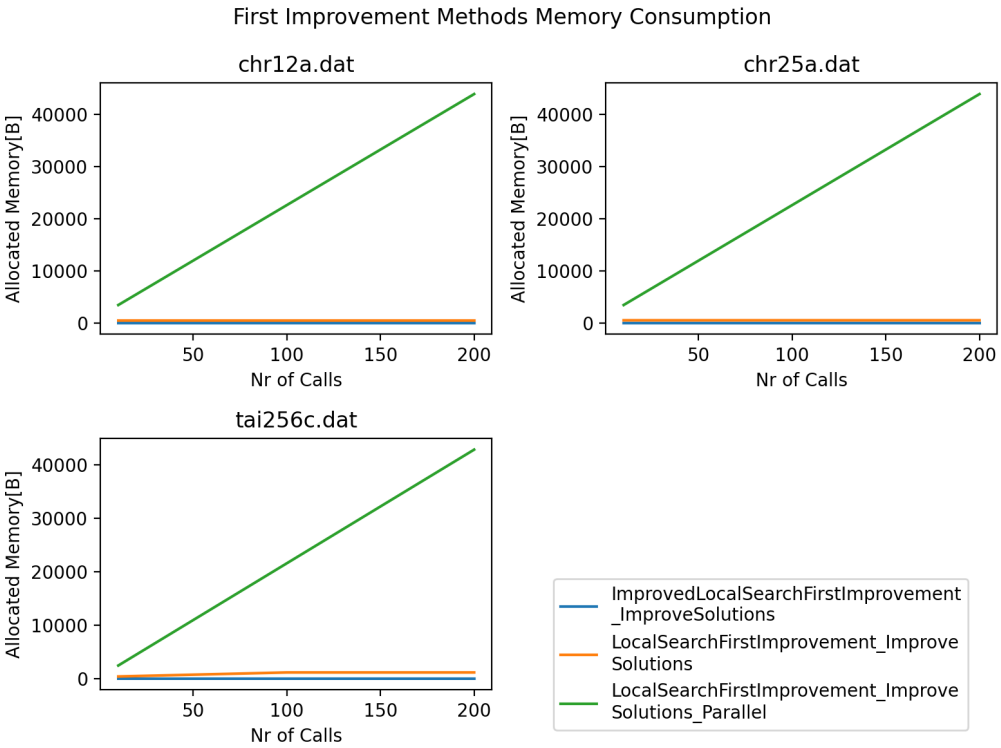


Figure 11: Memory consumption in bytes for the first improvement methods.[1]

[1]Small transformation of the lines to prevent overlapping in the figure

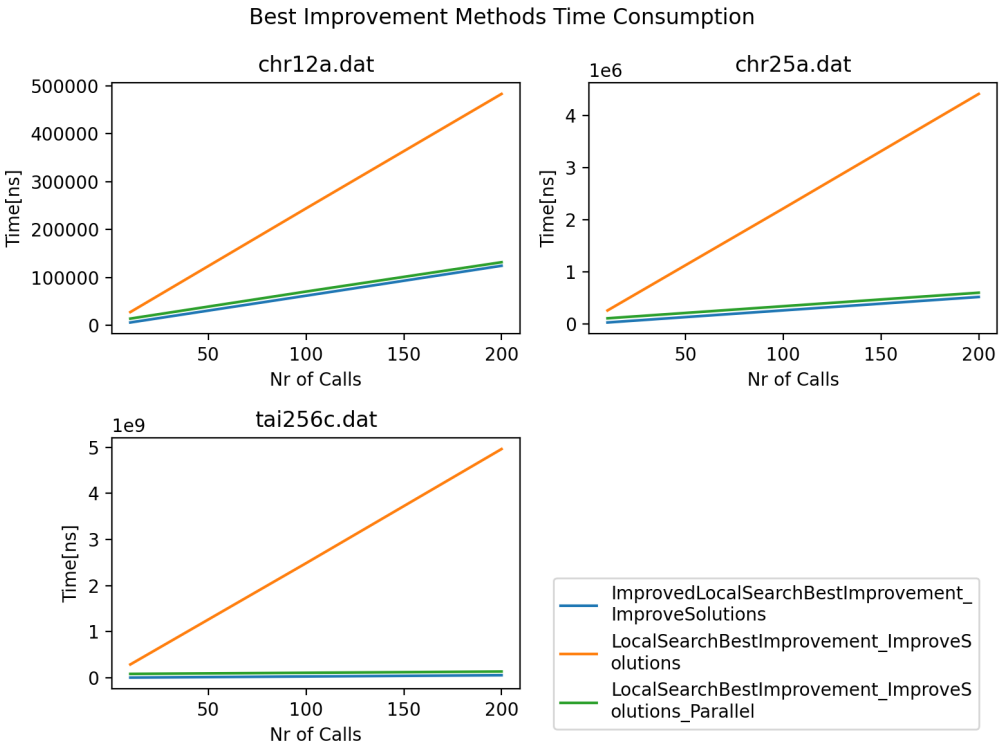


Figure 12: Runtime in nanoseconds for the best improvement methods.[1]

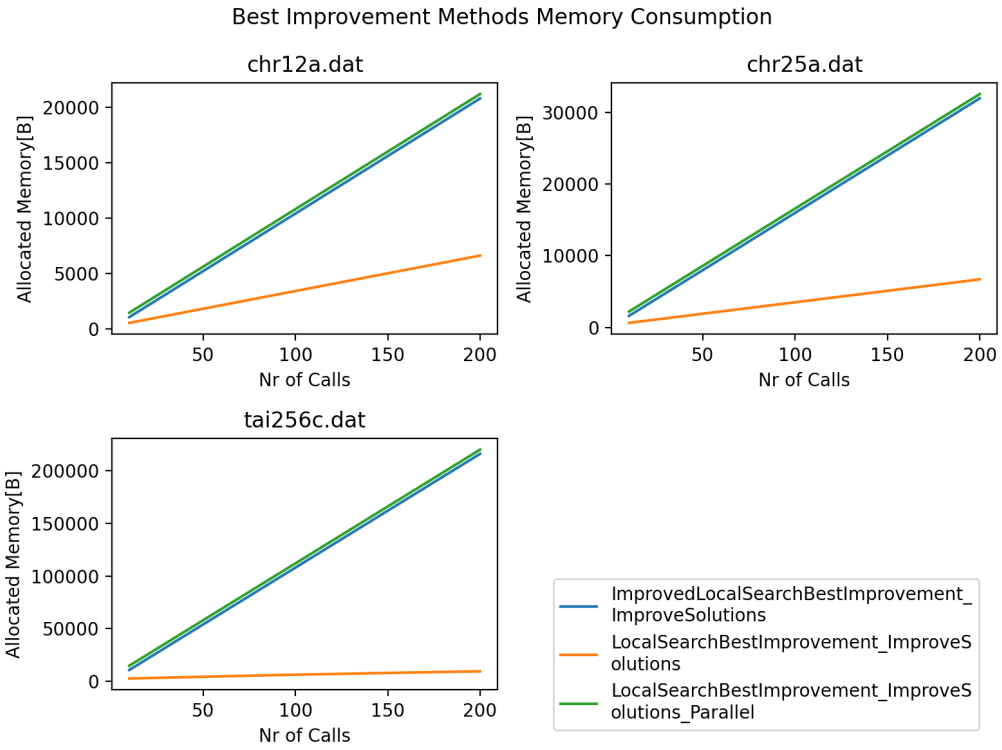


Figure 13: Memory consumption in bytes for the best improvement methods.[1]

[1]Small transformation of the lines to prevent overlapping in the figure

Solution Generation Benchmark

The solution generation methods are used for combining the solutions in the population, using those to generate new solutions. Three different approaches were implemented. First, the subset generation algorithm, second, *Path Relinking* and last, a combined method with subset generation and *Path Relinking*. The methods used are not only different in their runtime and memory consumption. Moreover, they differ in their affectivity to generate new solutions for the *Scatter Search* algorithm. Therefore, before analysing the runtime and memory consumption, it is necessary to consider the number of new solutions generated from a given population. For the analysis, three different instances were used which generated a population of ten solutions. After the creation, the three solution generation algorithms were used for generating new solutions. However, for combination of solutions the exhausting pairwise method was used and for the improvement method the local search, the best improvement method. Table 3 shows the results of the test.

Method	Instance Name	New Solutions Generated
Subset generation	chr12a.dat	290
Path Relinking	chr12a.dat	449
Combined	chr12a.dat	645
Subset generation	chr25a.dat	992
Path Relinking	chr25a.dat	1606
Combined	chr25a.dat	2468
Subset generation	tai256c.dat	4091
Path Relinking	tai256c.dat	22306
Combined	tai256c.dat	26384

Table 3: Number of new solutions generated.

The results in table 3 indicate that the subset generation method is not the most efficient way to generate new solutions. However, the *Path Relinking* combined with the subset generation method creates the highest number of new distinct solutions. Regarding the runtime of the algorithms, the more complex implementation of *Path Relinking* results in a higher computation time than the subset generation method. However, the parallel implementation reduces the computation time significantly. Plot number 14 gives the results.

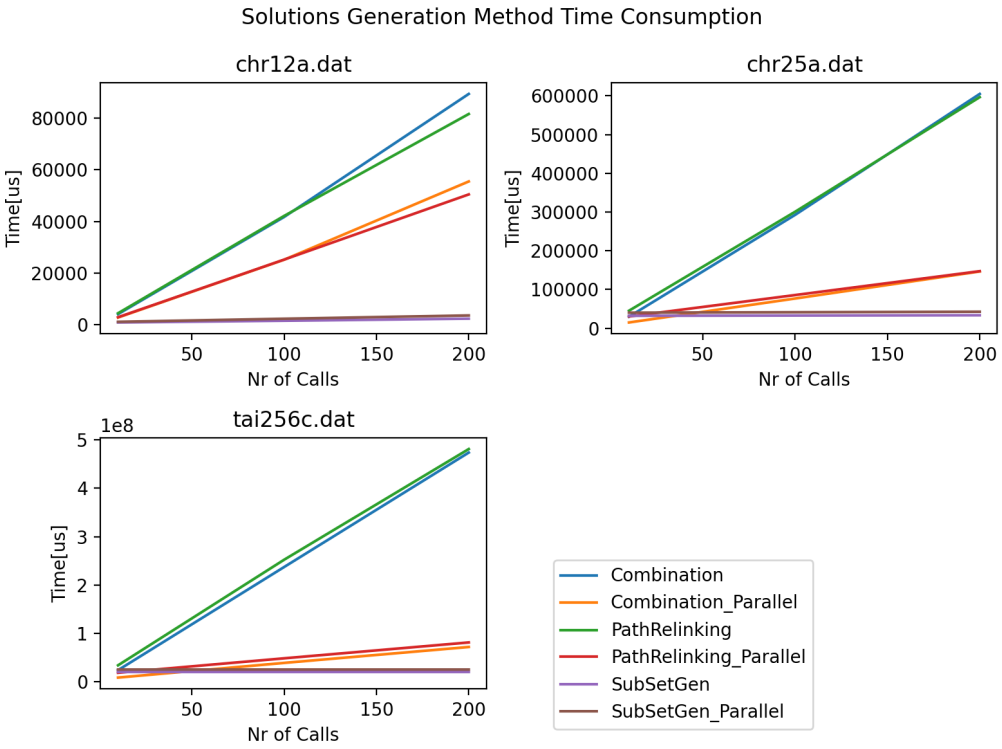


Figure 14: Runtime in microseconds for the solution generation methods.[1]

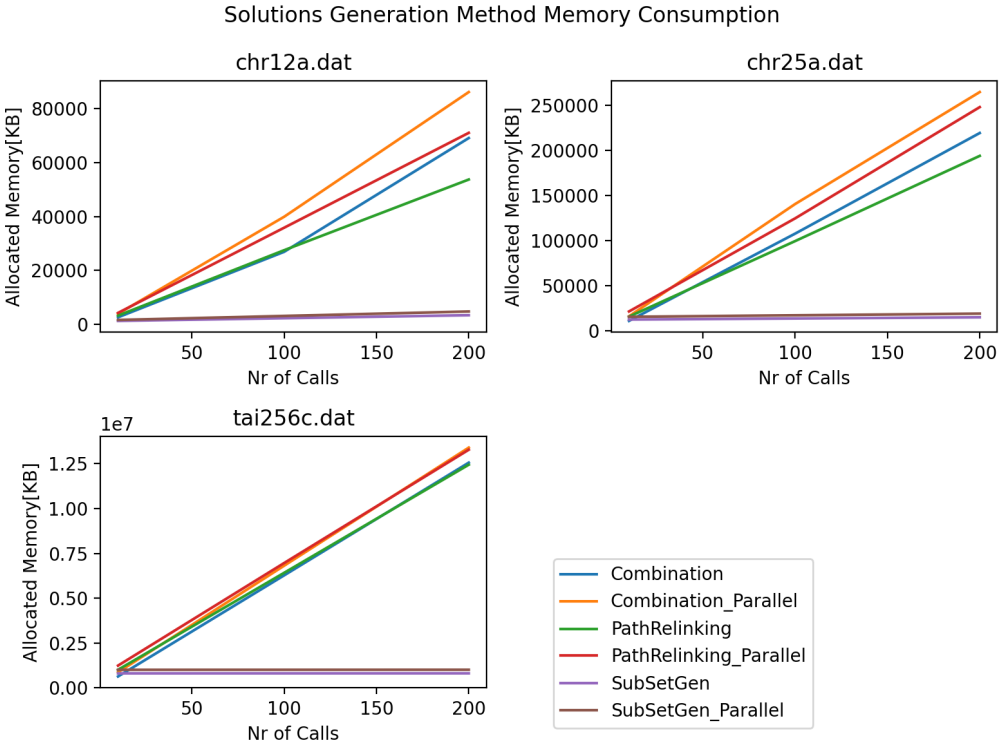


Figure 15: Memory consumption in kilobytes for the solution generation methods.[1]

[1]Small transformation of the lines to prevent overlapping in the figure

On the other hand, the memory consumption does not differ between the implemented algorithms. Figure 15 displays an increasing memory consumption overtime. However, this is due to the hash table which stores the already found solutions. In conclusion to the results of the benchmarks, a preselection of the algorithms was created. The investigation of the *Scatter Search* implementation will focus on this selection. Table 4 lists the selected algorithms.

Generation Method	*ParallelRandomGeneratedPopulation*
Combination Methods	*DeletionPartsOfTheFirstSolution-AndFillWithPartsOfTheOtherSolutions* *ExhaustingPairwiseCombination*
Diversification Method	*HashCodeThreePartDiversification*
Improvement Method	*ParallelImprovedLocalSearchBestImprovement* *ParallelImprovedLocalSearchFirstImprovement*
Solution Generation Method	*ParallelPathRelinkingSubSetGenerationCombined*

Table 4: Preselection of the used algorithms.

5 Analysis

This chapter will describe the test and analysis procedure for the implementation of *Scatter Search*. Furthermore, it will also describe the test settings, display and explain the results of the pre-test.
By utilizing the algorithms that have been pre-selected in chapter 4.5, four distinct settings for the tests can be defined:

- Shared algorithms:

  ParallelRandomGeneratedPopulation

  HashCodeThreePartDiversification

  ParallelPathRelinkingSubSetGenerationCombined

- Test Setting 1:

  DeletionPartsOfTheFirstSolutionAndFillWithPartsOfTheOtherSolution

  ParallelImprovedLocalSearchBestImprovement

- Test Setting 2:

  ExhaustingPairwiseCombination

  ParallelImprovedLocalSearchBestImprovement

- Test Setting 3:

  DeletionPartsOfTheFirstSolutionAndFillWithPartsOfTheOtherSolution

  ParallelImprovedLocalSearchFirstImprovement

- Test Setting 4:

  ExhaustingPairwiseCombination

  ParallelImprovedLocalSearchFirstImprovement

Those test settings were applied to the following instances.

- *chr12a.dat*
- *chr15b.dat*
- *chr25a.dat*
- *esc16b.dat*
- *esc32c.dat*
- *esc128.dat*
- *nug24.dat*
- *nug30.dat*
- *sko64.dat*
- *tai256c.dat*

Those instances were selected due to their different type and size. The testing of *Scatter Search* can be time-consuming. For example, the runtime for one pre-test is 400 minutes. The runtime for one test setting is 600 seconds, for four test settings and 10 different instances yields a total of 400 minutes for the entire test. For the final test, the number of repetitions will be increased to five, which will further increase the runtime. Therefore, the pre-test settings were created to investigate different behaviours when applying *Scatter Search* to different instances. In addition to the combination of algorithms for the *Scatter Search* framework, two additional parameters were also selected. The number of repetitions, which indicates how often a test should repeat and the runtime in seconds. The settings for both the preliminary and final tests are presented in Table 5.

Pre Test Settings	runtime[s]: 600 no of repetitions: 1 reference set size: 20 populations size: 100 algorithms used: all 4 settings
Final Test Settings	runtime[s]: 600 no of repetitions: 5 reference set size: 20 populations size: 100 algorithms used: best 2 settings

Table 5: Test Settings.

At the outset, the reference set was set to a size of 20 and the population to a size of 100. According to Laguna, is a size of the population, five times the size of the reference set, used in most *Scatter Search* applications [9, p. 25].

5.1 Pretests

The first pre-test was performed with all four combinations of algorithms and applied to every instance. This first test demonstrates how *Scatter Search* performs with instances of different size and structure. Table 6 displays the outcome of the first pre-test. The column *Instance Name* represents the name of the instance tested, N the size, and *Difference* the difference between the found optimum and the lower bound. The difference is calculated with the following formula:

$$Difference[\%] = ((foundOptimum/knownOptimum) - 1) * 100 \tag{7}$$

During the final tests with five repetitions, the geometric mean was calculated.

$$Difference[\%] = (\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} - 1) * 100 = (\sqrt[n]{x_1 x_2 \cdots x_n} - 1) * 100 \tag{8}$$

Where x_n is the optimum determined from one repetition. The last column, *Test Setting* represents the different test settings as indicated at the beginning of chapter 5.

Instance Name	N	Difference [%]	Test Setting
chr12a.dat	12	11.22	1
chr12a.dat	12	6.7	2
chr12a.dat	12	11.22	3
chr12a.dat	12	6.7	4
chr15b.dat	15	11.86	1
chr15b.dat	15	17.07	2
chr15b.dat	15	45.86	3
chr15b.dat	15	24.76	4
chr25a.dat	25	76.13	1
chr25a.dat	25	72.08	2
chr25a.dat	25	97.47	3
chr25a.dat	25	101.63	4
esc16b.dat	16	0.0	1
esc16b.dat	16	0.0	2
esc16b.dat	16	0.0	3
esc16b.dat	16	0.0	4
esc32c.dat	32	0.0	1
esc32c.dat	32	0.0	2
esc32c.dat	32	4.05	3
esc32c.dat	32	5.61	4
esc128.dat	128	137.5	1
esc128.dat	128	125.0	2
esc128.dat	128	118.75	3
esc128.dat	128	171.88	4
nug24.dat	24	6.31	1
nug24.dat	24	4.99	2
nug24.dat	24	9.52	3
nug24.dat	24	7.8	4
nug30.dat	30	6.11	1
nug30.dat	30	6.73	2
nug30.dat	30	7.77	3
nug30.dat	30	8.65	4
sko64.dat	64	15.85	1
sko64.dat	64	13.37	2
sko64.dat	64	17.26	3
sko64.dat	64	14.73	4
tai256c.dat	256	8.47	1
tai256c.dat	256	5.33	2
tai256c.dat	256	6.97	3
tai256c.dat	256	3.4	4

Table 6: Result of the first pre-test.

From Table 6 it can be seen that the majority of instances exhibit a variance of 0 - 15% between the found and known optimum. Nonetheless, there are two instances which perform worse, namely *chr25a.dat* and *esc128.dat*, and which necessitate further investigation.

5.2 Investigation of specific Solutions

The instances *chr25a.dat* and *esc128.dat* needed more investigation because of the poorer performance of *Scatter Search*. As often in the *QAP* it could be due to the early development of a local minimum. Therefore, the reference set size was increased to improve the possibility to leave the local minimum. This investigation was done with 10-times repetition of the tests and increasing the reference set by 10 in every repetition. However, with an increased reference set, also the population size was increased to keep the same ratio of five between the reference set and the population. The results of this test are plotted in Figures 16 and 17.

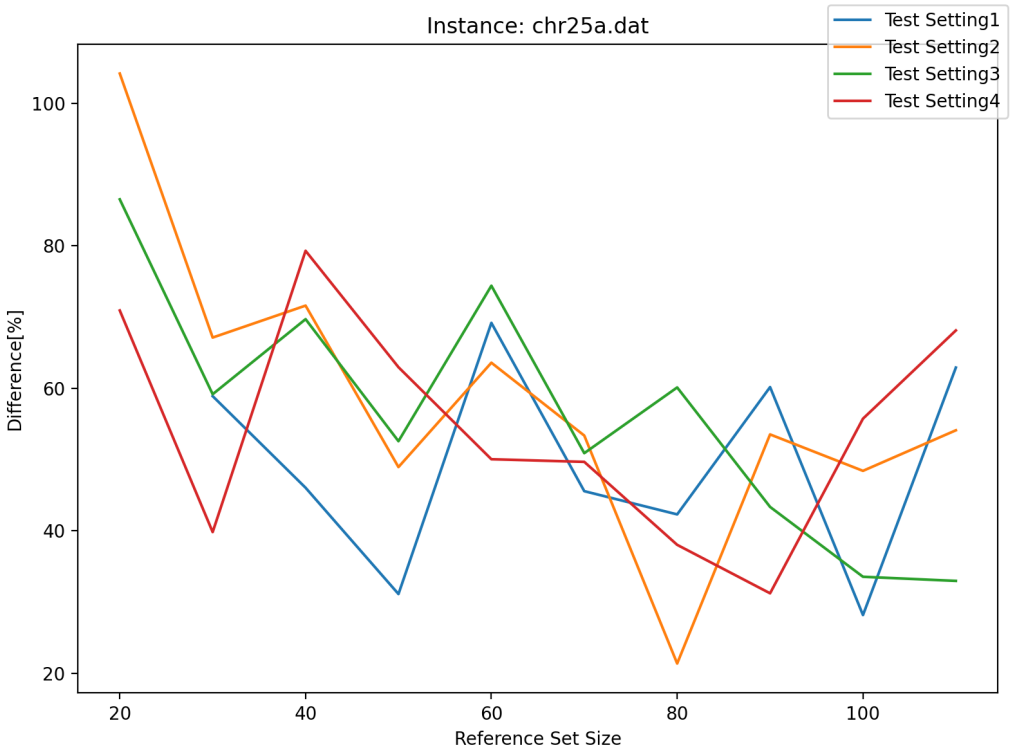


Figure 16: Increased Reference Set Size for *chr25a.dat*

Both results indicate, that an increased reference set size would reduce the difference between the found and known optimum. Moreover, the correlation between reference set size and decreased difference is independent from the algorithm combination which was used. Nonetheless, to ensure that the correlated parameter was not the population, a rerun of the test was conducted solely with an increased population and not the reference set size. Figures 18 and 19 plot the results and display that the solution quality does not correlate with the population size if increased independently to the size of the reference set. The missing correlation can be explained by the different roles the reference set and the population in *Scatter Search*. Where the population is solely utilized for the generation of novel solutions that cater for diversity, the reference set is also utilized to generate novel solutions by utilizing the existing solutions. Moreover, a bigger reference set increases the chance that there are more diverse solutions in the set.

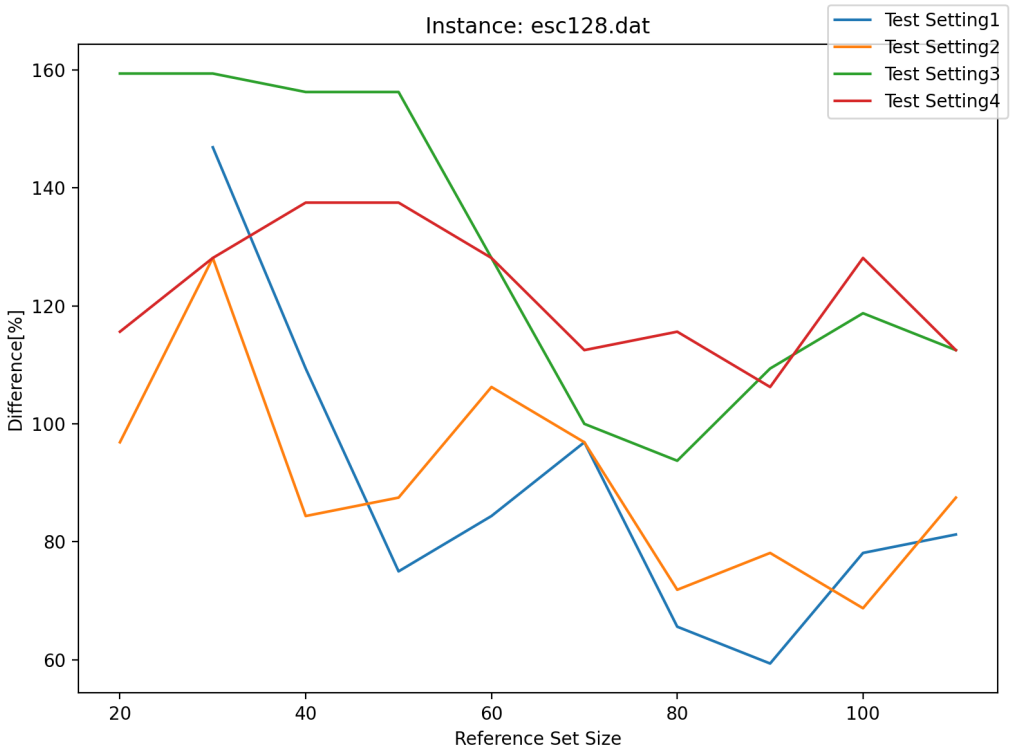


Figure 17: Increased Reference Set Size for *esc128.dat*

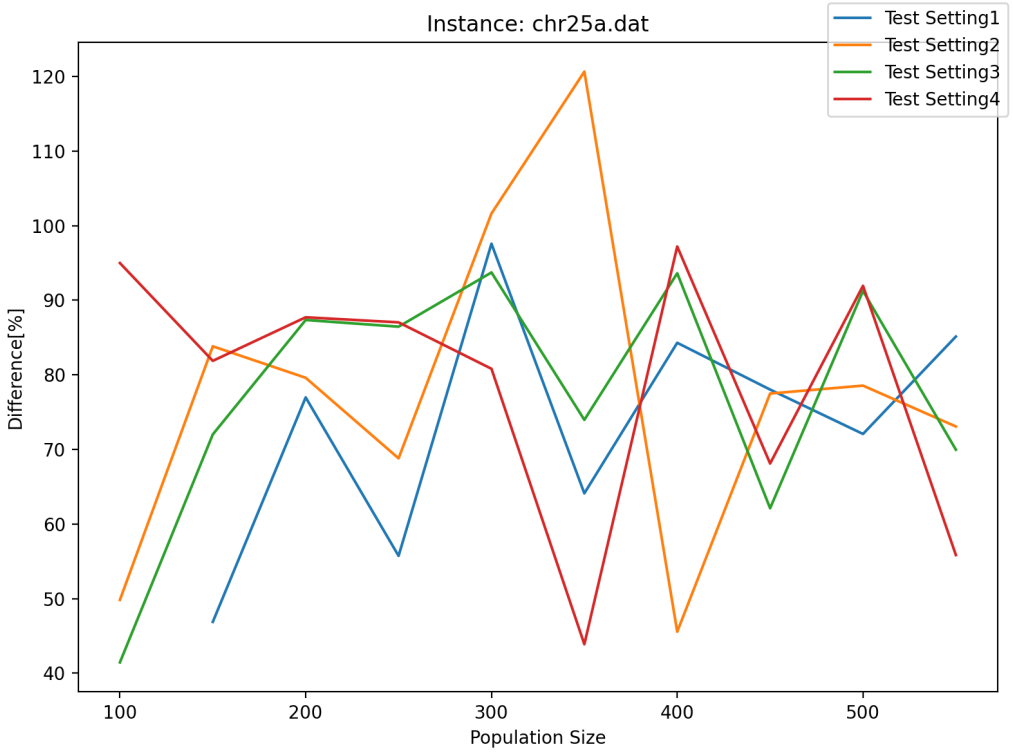


Figure 18: Increased Population Size for *chr25a.dat*

Figure 19: Increased Population Set Size for *esc128.dat*

The results with the adjustment of the reference set led to an implementation of a dynamic adjustment of the reference set size in *Scatter Search*. This dynamic adjustment of the reference set size takes place, if the solution generation methods, *Subset Generation* and *Path relinking* cannot find additional solutions. If additional solutions are not found within ten iterations, the reference set size is increased by ten together with the population size. However, two tests were carried out, one with an unlimited reference set and one with a limit of 200 solutions in the reference set. This measure is intended to avoid the use of additional memory and time. Table 7 displays the results.

Instance Name	N	No Dyn. [%]	Dyn. [%]	Dyn. with Limit[%]	Test Setting
chr12a.dat	12	11.22	0.0	0.0	1
chr12a.dat	12	6.7	3.81	11.22	2
chr12a.dat	12	11.22	6.7	6.7	3
chr12a.dat	12	6.7	6.7	11.22	4
chr15b.dat	15	11.86	21.05	11.86	1
chr15b.dat	15	17.07	13.34	46.98	2
chr15b.dat	15	45.86	42.3	32.44	3
chr15b.dat	15	24.76	27.56	18.05	4
chr25a.dat	25	76.13	87.2	55.69	1
chr25a.dat	25	72.08	51.53	59.69	2
chr25a.dat	25	97.47	55.22	47.1	3
chr25a.dat	25	101.63	70.02	73.45	4
esc16b.dat	16	0.0	0.0	0.0	1
esc16b.dat	16	0.0	0.0	0.0	2
esc16b.dat	16	0.0	0.0	0.0	3
esc16b.dat	16	0.0	0.0	0.0	4
esc32c.dat	32	0.0	0.62	2.8	1
esc32c.dat	32	0.0	1.25	0.0	2
esc32c.dat	32	4.05	5.92	6.54	3
esc32c.dat	32	5.61	1.87	1.25	4
esc128.dat	128	137.5	109.38	109.38	1
esc128.dat	128	125.0	131.25	100.0	2
esc128.dat	128	118.75	150.0	121.88	3
esc128.dat	128	171.88	125.0	140.62	4
nug24.dat	24	6.31	4.42	5.39	1
nug24.dat	24	4.99	5.45	2.75	2
nug24.dat	24	9.52	7.22	7.17	3
nug24.dat	24	7.8	6.48	4.82	4
nug30.dat	30	6.11	5.13	8.23	1
nug30.dat	30	6.73	7.9	5.52	2
nug30.dat	30	7.77	8.46	11.4	3
nug30.dat	30	8.65	8.43	10.16	4
sko64.dat	64	15.85	12.59	15.59	1
sko64.dat	64	13.37	14.75	15.1	2
sko64.dat	64	17.26	17.65	14.79	3
sko64.dat	64	14.73	18.22	16.53	4
tai256c.dat	256	8.47	7.25	8.2	1
tai256c.dat	256	5.33	3.19	3.05	2
tai256c.dat	256	6.97	9.11	7.3	3
tai256c.dat	256	3.4	3.69	3.2	4

Table 7: Result of the dynamic reference set adjustment.

The dynamic adjustment of the reference size does slightly improve the results of the *Scatter Search* algorithm. The geometric mean of the results for all test instances is displayed in table 8.

Method	Geometric Mean [%]
No dynamic adjustment	23.31
Dynamic adjustment	20.88
Dynamic adjustment with limit	20.22

Table 8: Geometric mean of the dynamic reference set adjustment.

The differences between the results are small. However, because of the lower mean the dynamic reference set adjustment with limit was chosen.

5.3 Selection and Optimization

For the final tests, the focus was on the two best test settings. Therefore, a comparison of the results of the different instances was conducted. Figure 20 plots the results of the comparison.

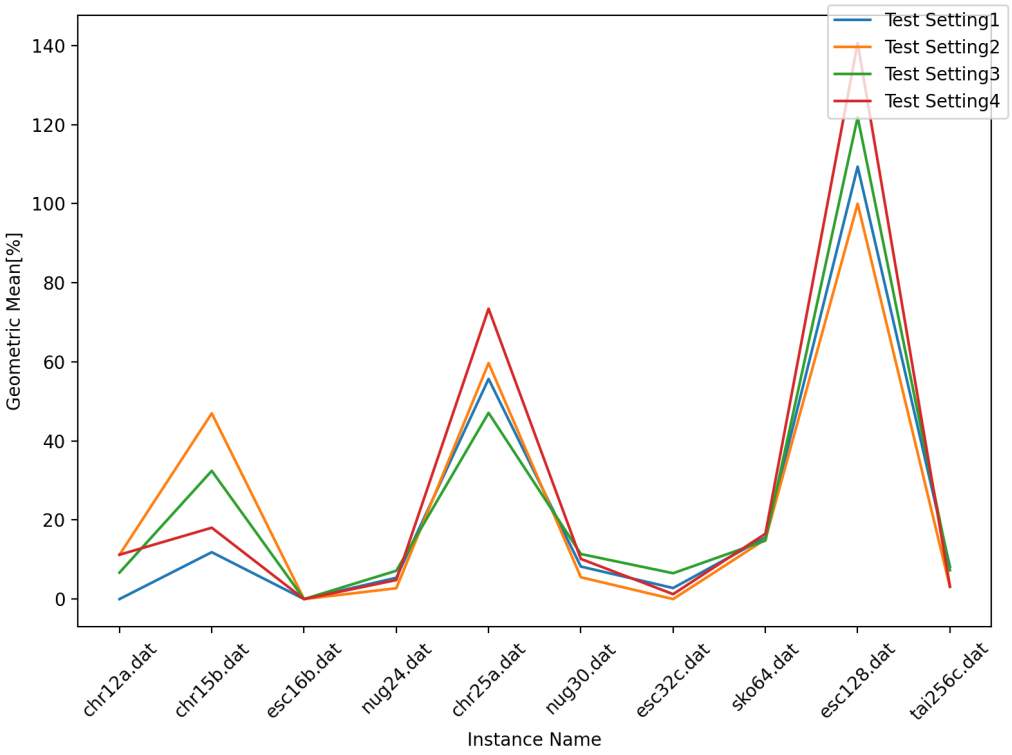


Figure 20: Comparison of different test settings.

The results in figure 20 indicates that *Test Setting 1* and *Test Setting 2* perform slightly better than the other settings. Therefore, the focus will be on the those configurations for further tests. The settings are displayed in table 9. Figure 20 also shows, that there is not a connection between the instance size and the difficulty of solving it. For example, *chr25a.dat*, *esc32c.dat*, *nug24.dat* and *nug30.dat* have a similar size. However, the result after ten minutes computation time differs significantly between those instances.

Final Test Settings	runtime[s]: 600 nr of repetitions: 5 reference set size: 20 populations size: 100 algorithms used: *Test Setting 1, 2*

Table 9: Final Test Settings.

The *Test Setting 1* and *Test Setting 2* use the *ParallelImprovedLocalSearchBestImprovement* as improvement algorithm, *DeletionPartsOfTheFirstSolutionAndFillWithPartsOfTheOtherSolutions* and *ExhaustingPairwiseCombination* as the combination method. The results for the final tests without tuning are displayed in figure 21.

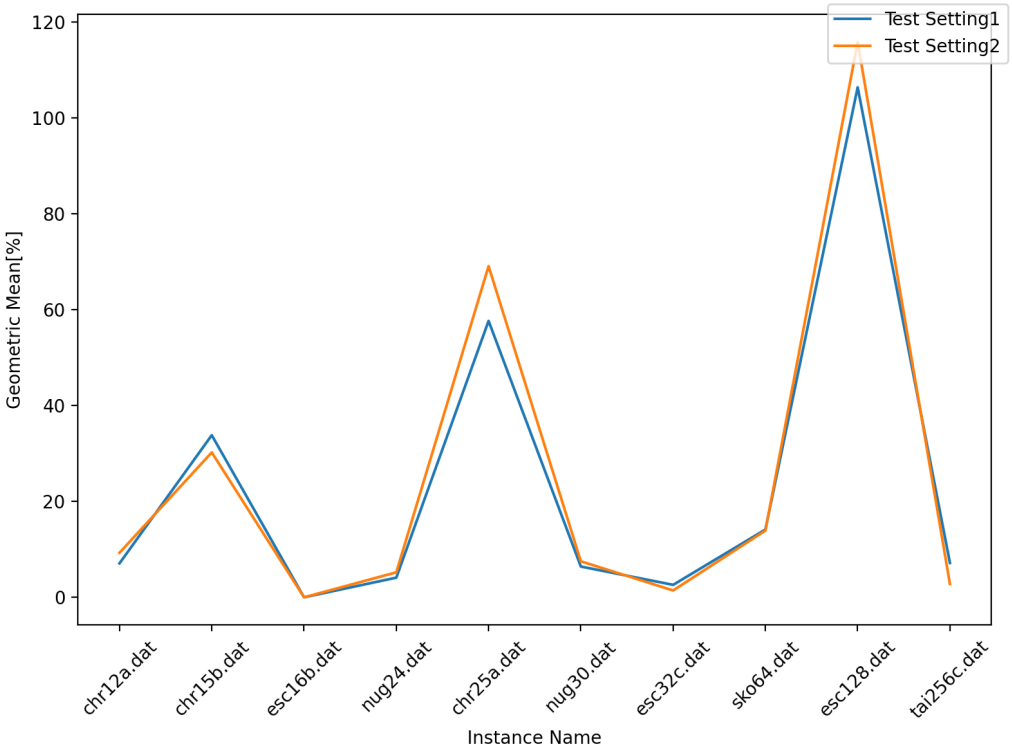


Figure 21: Final results without any adjustment to parameters.

Optimization with the parameters

Both combination methods have additional parameters to enable fine tuning of the algorithm. *DeletionPartsOfTheFirstSolutionAndFillWithPartsOfTheOtherSolutions* has the parameters *percentageOfSolutionToDelete*, configures the percentage of the solution to delete and *deleteWorstPart* which switches between the deletion of the worst part or a random part. *ExhaustingPairwiseCombination* has the parameter *stepSizeForPairs*. This parameter affects the number of pairs for the combination. In addition to the parameters in the test settings, the *Scatter Search* algorithm itself has the possibility to adjust the parameter, *percentageOfSolutionToRemove*, which is the percentage of a solution which should be deleted. Therefore, there are four different parameters to fine tune:

- *percentageOfSolutionToDelete*
- *deleteWorstPart*
- *stepSizeForPairs*
- *percentageOfSolutionToRemove*

The parameter *stepSizeForPairs* will not be investigated further, because increasing the number will only minimize the number of solutions and all the previous tests have been made with the value of one. This value provides the highest output of possible combinations. Therefore, an increase will not be done. Moreover, for further optimization the focus was set on the more problematic solutions *chr15b.dat*, *chr25a.dat*, *esc128.dat* and *tai256c.dat* as control instance. For the optimization, first a reduction of the values for the parameters *percentageOfSolutionToDelete* and *percentageOfSolutionToRemove* was carried out. In a second run these values were increased.

Parameter	Default	Decreased Value	Increased Value
percentageOfSolutionToDelete	0.5	0.2	0.8
percentageOfSolutionToRemove	0.5	0.2	0.8

Table 10: Values for the optimization.

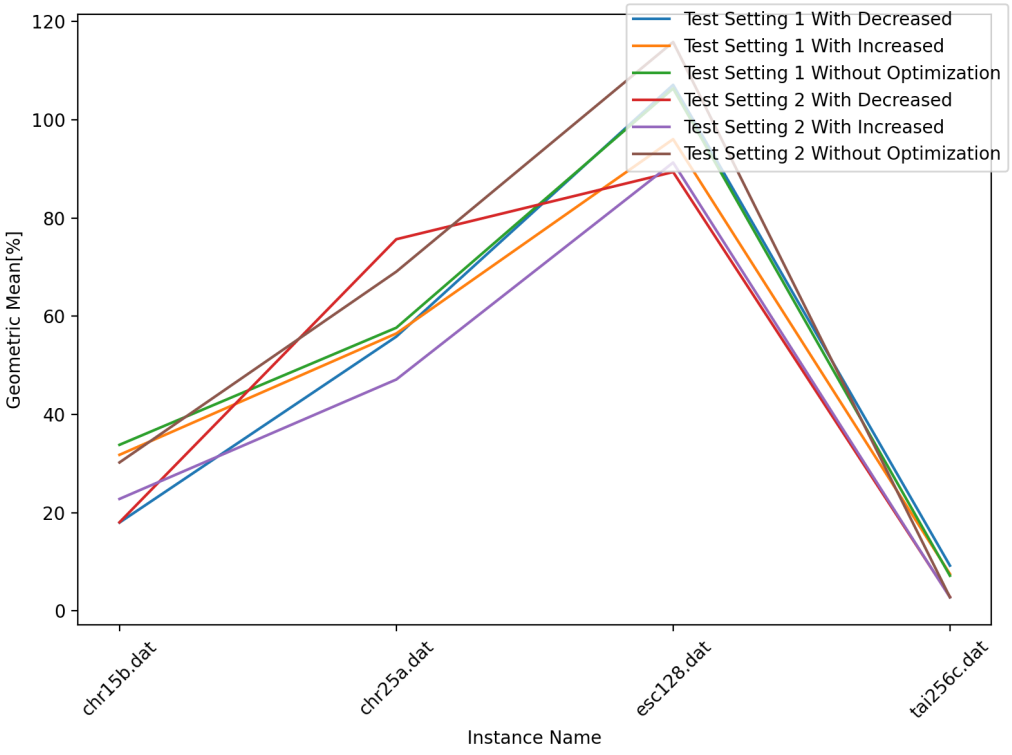


Figure 22: Results adjusted parameters.

Figure 22 displays the mean of the results. Moreover, the figure indicates a slight improvement with the increased values for the parameters. Therefore, every further test was carried out with these optimized parameters.

6 Results and conclusion

Chapter 6 initially enumerates the parameter settings and the selection of algorithms applied with the Scatter Search framework. Furthermore, the outcomes of the final assessments are presented.

6.1 General Results

The final results were achieved using test settings 1 and 2. In addition, a dynamic reference set adjustment with a limit was implemented. Moreover, the two parameters, *percentageOfSolutionToDelete* and *percentageOfSolutionToRemove*, were set to the value of 80%. The final settings are displayed in Table 11.

Shared	Population Generation Diversification Solution Generation	*ParallelRandomGeneratedPopulation* *HashCodeThreePartDiversification* *ParallelPathRelinkingSubSetGeneration-Combined*
Test Setting 1	Solution Combination Solution Improvement	*DeletionPartsOfTheFirstSolutionAnd-FillWithPartsOfTheOtherSolution* *ParallelImprovedLocalSearchBest-Improvement*
Test Setting 2	Solution Combination Solution Improvement	*ExhaustingPairwiseCombination* *ParallelImprovedLocalSearchBest-Improvement*
Parameter Values	Reference Set Size Population Size Dynamic Reference Set *percentageOfSolutionToDelete* *percentageOfSolutionToRemove* Runtime in seconds:	20 100 True, Limit: 200 80% 80% 600

Table 11: Final parameter values and algorithms.

Table 12 and Figure 23 display the results of instances with known solutions. The results indicate that the implementation with test setting 2, therefore with the *ExhaustingPairwiseCombination* and *ParallelImprovedLocalSearchBestImprovement*, performs slightly better than the implementation with the other algorithms.

The last test was done with instances where the optimal objective value is not known. For those instances, the lower bound of the solution for the difference calculation was taken. Furthermore, the gap between the lower bound and the current best objective value found was added. In addition, the heuristic which found the lower bound was also added. Table 13 and figure 24 illustrate the result. Both test settings perform very similarly. Furthermore, at a runtime of 600 seconds, no implementation was able to get lower the currently found gap.

Instance Name	Test Setting	Geometric Mean
chr12a.dat	1	6.81
chr12a.dat	2	5.53
chr15b.dat	1	32.21
chr15b.dat	2	13.97
chr25a.dat	1	52.46
chr25a.dat	2	47.11
esc128.dat	1	114.46
esc128.dat	2	93.02
esc16b.dat	1	0.0
esc16b.dat	2	0.0
esc32c.dat	1	1.36
esc32c.dat	2	0.74
nug24.dat	1	4.09
nug24.dat	2	4.41
nug30.dat	1	5.97
nug30.dat	2	5.93
sko64.dat	1	12.92
sko64.dat	2	14.87
tai256c.dat	1	7.98
tai256c.dat	2	2.82

Table 12: Final results of instances with known objective value.

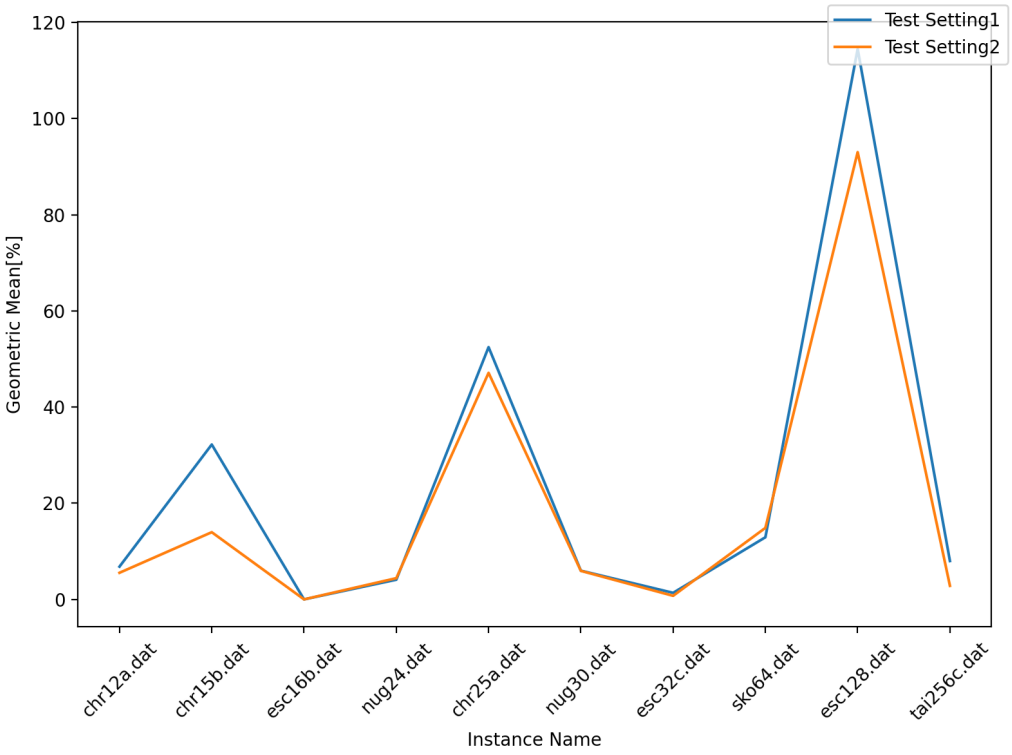


Figure 23: Final results of instances with known objective value.

Instance Name	Test Setting	Geometric Mean [%]	Gap [%]	Heuristic
sko100c.dat	1	14.2	5.73	GEN
sko100c.dat	2	14.52	5.73	GEN
sko42.dat	1	13.22	5.56	Ro-TS
sko42.dat	2	13.09	5.56	Ro-TS
sko64.dat	1	12.67	5.70	Ro-TS
sko64.dat	2	13.97	5.70	Ro-TS
sko90.dat	1	14.77	6.10	Ro-TS
sko90.dat	2	14.08	6.10	Ro-TS
tai100a.dat	1	43.51	25.10	Re-TS
tai100a.dat	2	42.96	25.10	Re-TS
tai30a.dat	1	24.71	15.90	Ro-TS
tai30a.dat	2	26.15	15.90	Ro-TS
tai50a.dat	1	37.22	22.00	GEN
tai50a.dat	2	36.95	22.00	GEN
tho150.dat	1	18.73	6.30	SIM-3
tho150.dat	2	18.51	6.30	SIM-3
tho40.dat	1	22.48	10.94	SIM-2
tho40.dat	2	22.69	10.94	SIM-2
wil100.dat	1	7.63	3.35	GEN
wil100.dat	2	7.74	3.35	GEN

Table 13: Final results of instances with unknown objective value. (GEN=genetic hybrids, Ro-TS=robust tabu search, Re-TS=reactive tabu search, SIM-3, SIM-2=simulated annealing)[1]

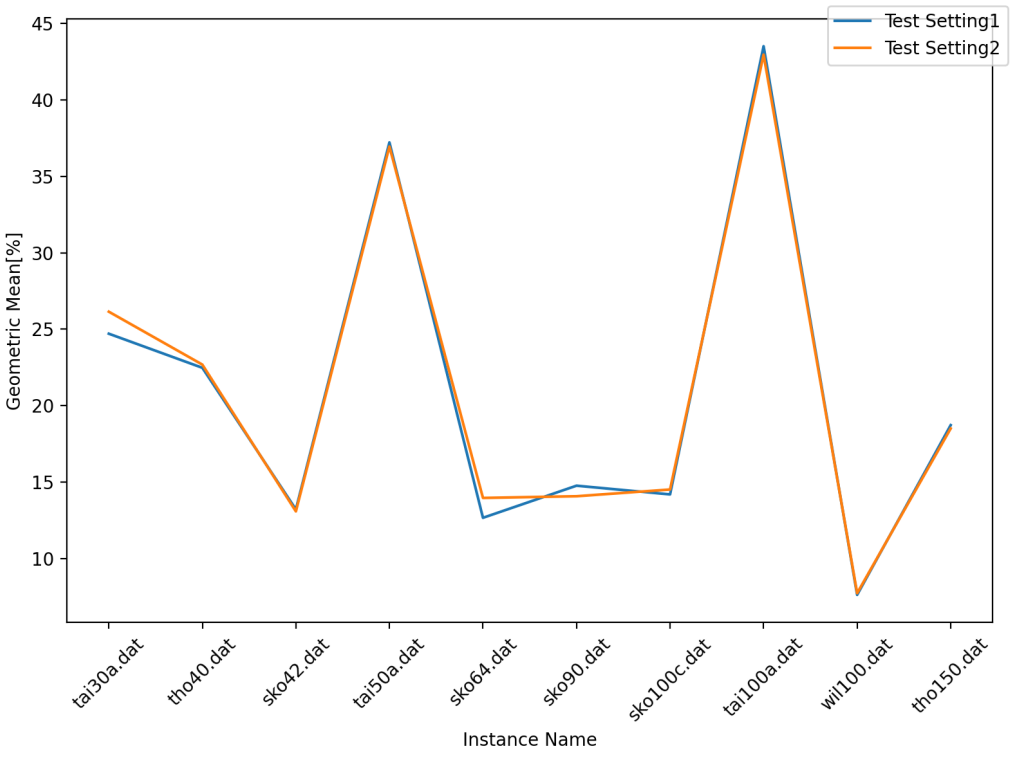


Figure 24: Final results of instances with unknown objective value.

6.2 Conclusion

The Scatter Search framework provides an approach to get good results for medium and large instances in a reasonable time limit. Furthermore, the framework allows to combine different algorithms and parameters to tune the meta heuristic. The subset generation and path relinking methods generate a large number of possible permutations for an instance, which allows a good solution to be found. However, there is a high risk of getting stuck in local minima with those permutations because they all are derived from the permutations in the current reference set. Consequently, a highly diverse reference set is essential for the Scatter Search framework. Tests with an increase in the reference set size, and hence an increase in the diversity of the reference set, show an improvement of the objective value. This led to the implementation of a dynamic reference set adjustment. With the first adjustment to the algorithm done, the algorithms *DeletionPartsOfTheFirstSolutionAndFillWithPartsOfOtherSolutions*, *ExhaustingPairwiseCombination* and *ParallelImprovedLocalSearchBestImprovment*, which are used in the *Scatter Search* framework, tend to perform slightly better than the other implemented algorithms. Moreover, by means of an increase in the removed solutions in the *HashCodeThreePartDiversification* and an increase in the parts to delete of a permutation in the *DeletionPartsOfTheFirstSolutionAndFillWithPartsOfOtherSolutions* algorithms further improvements could be observed.

In total, 20 instances derived from *QAPLib*, varying in size and method of generation, were subjected to testing. The results are between 5 and 30% within a time frame of 10 minutes. However, there are solutions which perform significantly worse than that. Therefore, I believe a specified diversity algorithm for a specific class of instance can further improve the results of the Scatter Search framework. Moreover, it may be possible to achieve a diverse reference set with modern machine learning algorithms for a specific problem. This approach could be used for further research. The Scatter Search framework provides a platform to connect those methods easily. This modularity of the framework allows the easy combination of different algorithms. Nevertheless, with the increasing number of complicated algorithms and parameters, the implementation and the tuning become increasingly complex.

Appendix

https://github.com/StefNehl/QAP

Literatur

[1] BURKARD, R. E. ; ÇELA, E. ; KARISCH, S.E ; RENDL F.: *QAPLIB - A Quadratic Assignment Problem Library.* https://www.opt.math.tugraz.at/qaplib/. Version: 02.2002

[2] Campos, V. ; Glover, F. ; Laguna, M. ; MartÌ, R.: An Experimental Evaluation of a Scatter Search for the Linear Ordering Problem: Journal of Global Optimization. 21 ((2001)), Nr. 4, S. 397–414

[3] Çela, Eranda: *Combinatorial Optimization.* Bd. 1: *The Quadratic Assignment Problem: Theory and Algorithms.* 1998 http://GW2JH3XR2C.search.serialssolutions.com/?sid=sersol&SS_jc=TC0001298202&title=The%20Quadratic%20Assignment%20Problem%20Theory%20and%20Algorithms. – ISBN 9781475727876

[4] Ding-Zhu, Du ; Pardalos, P. M.: *Handbook of combinatorial optimization.* Boston and London : Kluwer Academic, 1998. – ISBN 0792350197

[5] Dréo, Johann ; Candan, Caner: *Different classifications of metaheuristics shown as a Euler Diagram.* https://en.m.wikipedia.org/wiki/File:Metaheuristics_classification.svg. Version: 28.08.2011

[6] Garey, Michael R. ; Johnson, David S.: *Computers and intractability: A guide to the theory of NP-completeness.* San Francisco : W. H. Freeman, 1979 (A Series of books in the mathematical sciences). – ISBN 0716710447

[7] Glover, Fred: A template for scatter search and path relinking. Version: 1998. http://dx.doi.org/10.1007/BFb0026589. In: Goos, G. (Hrsg.) ; Hartmanis, J. (Hrsg.) ; van Leeuwen, J. (Hrsg.) ; Hao, Jin-Kao (Hrsg.) ; Lutton, Evelyne (Hrsg.) ; Ronald, Edmund (Hrsg.) ; Schoenauer, Marc (Hrsg.) ; Snyers, Dominique (Hrsg.): *Artificial Evolution* Bd. 1363. Berlin, Heidelberg : Springer Berlin Heidelberg, 1998. – DOI 10.1007/BFb0026589. – ISBN 978–3–540–64169–8, S. 1–51

[8] Koopmans, Tjalling ; Beckmann, Martin J.: *Assignment Problems and the Location of Economic Activities: Cowles Foundation Discussion Papers.* https://EconPapers.repec.org/RePEc:cwl:cwldpp:4

[9] Laguna, Manuel ; MartÌ, Rafael: *Scatter Search: Methodology and Implementations in C.* Boston, MA : Springer US, 2003. http://dx.doi.org/10.1007/978-1-4615-0337-8. http://dx.doi.org/10.1007/978-1-4615-0337-8. – ISBN 978–1–4615–0337–8

[10] Martí, Rafael ; Laguna, Manuel ; Glover, Fred: Principles of Scatter Search. (2006)

[11] Microsoft: *Mircosoft C# Tutorial.* https://learn.microsoft.com/en-gb/dotnet/csharp. Version: 13.02.2023

[12] Nebro, Antonio ; Luna, Francisco ; Alba, Enrique ; Dorronsoro, Bernabe ; Durillo, Juan ; Beham, Andreas: AbYSS: Adapting Scatter Search to Multiobjective Optimization. In: *Evolutionary Computation, IEEE Transactions on* 12 (2008), S. 439–457. http://dx.doi.org/10.1109/TEVC.2007.913109. – DOI 10.1109/TEVC.2007.913109

[13] .NET Foundation and contributors: *BenchmarkDotNet.* https://benchmarkdotnet.org

[14] Sahni, Sartaj ; Gonzalez, Teofilo: P-Complete Approximation Problems. In: *Journal of the ACM* 23 (1976), Nr. 3, S. 555–565. http://dx.doi.org/10.1145/321958.321975. – DOI 10.1145/321958.321975. – ISSN 0004–5411

[15] Stützle, Thomas: *Local Search Algorithms for Combinatorial Problems: Analysis, Improvements and New Applications*. Darmstadt, TU Darmstadt, Diss., 1998

[16] Taha, Hamdy A.: *Operations Research: An Introduction*. Global edition. Harlow : Pearson, 2018. – ISBN 978–1–292–16554–7

[17] Taillard, Éric D.: *Design of Heuristic Algorithms for Hard Optimization: With Python Codes for the Travelling Salesman Problem*. 1st edition 2023. Cham : Springer International Publishing, 2023 (Graduate Texts in Operations Research). http://dx.doi.org/10.1007/978-3-031-13714-3. http://dx.doi.org/10.1007/978-3-031-13714-3. – ISBN 978–3–031–13714–3

[18] Xhafa, Fatos ; Abraham, Ajith: *Studies in computational intelligence, 1860-949X*. Bd. v. 128: *Metaheuristics for scheduling in industrial and manufacturing applications*. Berlin : Springer, 2008 http://www.springer.com/gb/BLDSS. – ISBN 978–3–540–78984–0