UNIVERSITY OF
EASTERN FINLAND

Analysis of rebuild local search operator for TSP

Jimi Tuononen

Tietojenkäsittelytieteen koulutusohjelma

Itä-Suomen yliopisto

Luonnontieteiden ja metsätieteiden tiedekunta

Tietojenkäsittelytieteen laitos / tietojenkäsittelytiede

14.4.2022

University of Eastern Finland, Faculty of Forestry and Natural Sciences

School of Computing

Tuononen, Jimi P.: Analysis of rebuild local search operators for TSP

Thesis, 55 pages

Supervisor: Prof. Pasi Fränti

April 2022

Abstract

The traveling salesman problem is an intriguing NP-hard problem in the field of computer science. The challenging nature of this problem has required a significant amount of research so that better results can be obtained with acceptable efficiency. TSP is an important problem to study because of its many real-world application areas. Many solving strategies and exact solvers exist for the optimization task. The most common and in many cases the best approach, local search, is the subject of this thesis. We study local search operator called *rebuild*, which is described in detail, and compared against other local search operators. The combination of rebuild and another operator is also considered. Rebuild produces promising results in most test cases. However, the current implementation of rebuild has some weaknesses in large-scale TSP instances which can be fixed by using it jointly with k-opt operator. The combination of rebuild and a small amount of 3-opt operator provided the best results in a test with ten TSP instances. The final benchmark revealed that the combination of rebuild and 3-opt provided 1.5% shorter paths than another operator combination random mix, and 3.1% shorter paths than 3-opt alone.

Keywords: traveling salesman problem; local search; heuristics; randomized algorithms; open-loop TSP; rebuild.

Acknowledgments

This thesis was written at the University of Eastern Finland (UEF) during the later stages of 2021, and the winter and spring of 2022.

Firstly, I would like to thank the University of Eastern Finland as a whole. The teachers, professors, and other students have helped me greatly to obtain the computer science skills and knowledge that I have today. The personal growth I have had has been immense in the past almost six years that I have been studying at UEF. I would also like to thank all the other UEF staff members that supported me by helping with studying related issues.

I want to express my gratitude to my supervisor, Professor Pasi Fränti, for helping at the selection of the master's thesis topic and guiding me through the research and writing phases of the thesis. The knowledge and insights helped me greatly to build this thesis to the form that it is today. I am also incredibly grateful for the chance to work at UEF from spring to fall in 2021. That experience had a considerable impact on this thesis as I got to see the research side of computer science more deeply.

I am also very thankful to my almost supervisor and now reviewer, Dr. Radu Mariescu-Istodor. Being one of the best teachers at the time at UEF, his courses were both motivating and fantastic knowledge-rich experiences. He also supported me in the completion of my IT project and provided enormous amount of assistance in all education and career based matters. I highly appreciate all the insightful talks and the support that you have given.

Finally, I want to thank all my lovely family members and friends that have provided support throughout the whole studying period. I am thankful for everything.

List of abbreviations

TSP	Travelling Salesman Problem
VRP	Vehicle Routing Problem
Rb	Rebuild. The analyzed local search operator.
LKH	Lin-Kernighan Heuristic
R&R	Ruin and Recreate. A solving method for TSP and VRP.

Table of contents

1 Introduction

The traveling salesman problem (TSP) is a classical problem in the field of computer science. It has been under diligent researching for several decades because of its challenging nature. The problem's task is simple: travel through all the given points with the least amount of movement needed. The original problem was formed with the idea that if a salesman were to travel through many cities, what would be the optimal route to do so. Nowadays the problem has been generalized into numerous fields. A traveling salesman problem can be any problem that has a distance measure and a set of points.

TSP is an NP-hard problem. This means that it belongs to a class of problems that are difficult to produce good answers for and the time-complexity of exact algorithms is exponential. A naive approach of checking all possible solutions is unusable beyond small TSP instances. The number of different solutions increase factorially in relation to *N*, the size of the problem.

The exact number of different paths depend on the rules of the problem. For example, TSP can be differentiated to *symmetric* and *asymmetric* TSP. In the simpler variant, symmetric TSP, the direction of movement does not matter as the distance between all points is equal to both directions. This is not the case in asymmetric TSP where the distance between two points might be different in opposite directions. All tests in this thesis are done in symmetric TSP cases. However, most of the introduced solving methods are applicable in both variations.

An example of a simple traveling salesman problem is given in Figure 1. The exact number of different paths to this open-loop problem is given by the formula *N!/2* as it is a symmetric TSP. If it was an asymmetric TSP, the amount would be given by *N* factorial. Half of the paths can be ignored because the distance is the same in both directions. Therefore, the number of different paths for the problem given in Figure 1 is approximately 10^{33} given by the formula above. The

brute-force approach is not realistic even with this small 31 point TSP instance. Therefore, other strategies are needed.

TSP is an interesting problem to study because its applicable to numerous real-life situations. Various application areas include logistics, genome sequencing, drilling problems, aiming telescopes, data clustering and many more (Applegate & al., 2007). Logistics itself include countless applications areas for instance traveling of salesmen, school bus routing and postal deliveries. The drilling problems are classical instances of TSP use. Drilling problem instances are also included in the testing section of this thesis as they are part of TSPLIB (Reinelt, 1991).

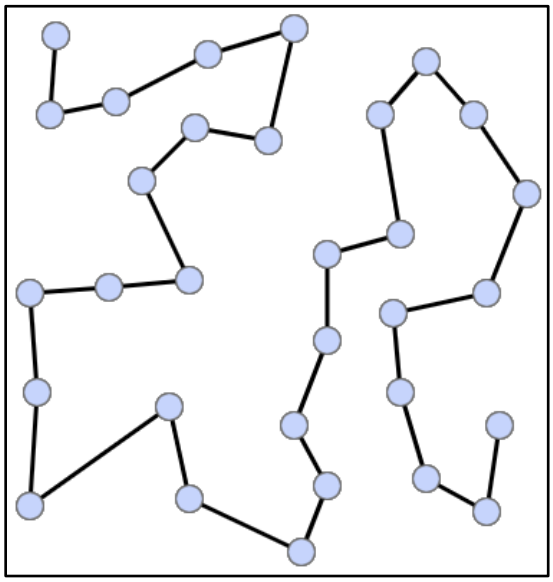

Figure 1. A simple TSP instance with a sample solution. *N* = 31.

2 Background of TSP

Chapter is organized as follows. Different TSP variations are described in Chapter 2.1. Several well-known TSP solving methods are introduced in Chapter 2.2, and the subject of this thesis, local search, is described in-depth in Chapter 2.3. Chapter 2.4 focuses on different meta heuristics and search strategies. Chapter 2.5 explains the motivation behind this research.

2.1 TSP variations

There are several variations of TSP. The variations are similar to each other and the solving methods for all variants are the same for the most part. *Closed-loop* TSP is the most common variation followed by *open-loop* TSP. In the closed-loop case, the traveler must return to the starting point, making the choice of the start point irrelevant. The start and end points do not matter in open-loop TSP. The only rule is that the traveler must visit all locations. One might have an idea that an optimal open-loop solution can be produced from an optimal closed-loop solution by removing the longest connection in the path. This strategy might work in trivial TSP instances but rarely works beyond the simple cases. The reason is that closed-loop paths will always form a loopy structure while open-loop usually does not. An example is given in Figure 2. Sengupta & al. (2019) also stated that the strategy usually does not work.

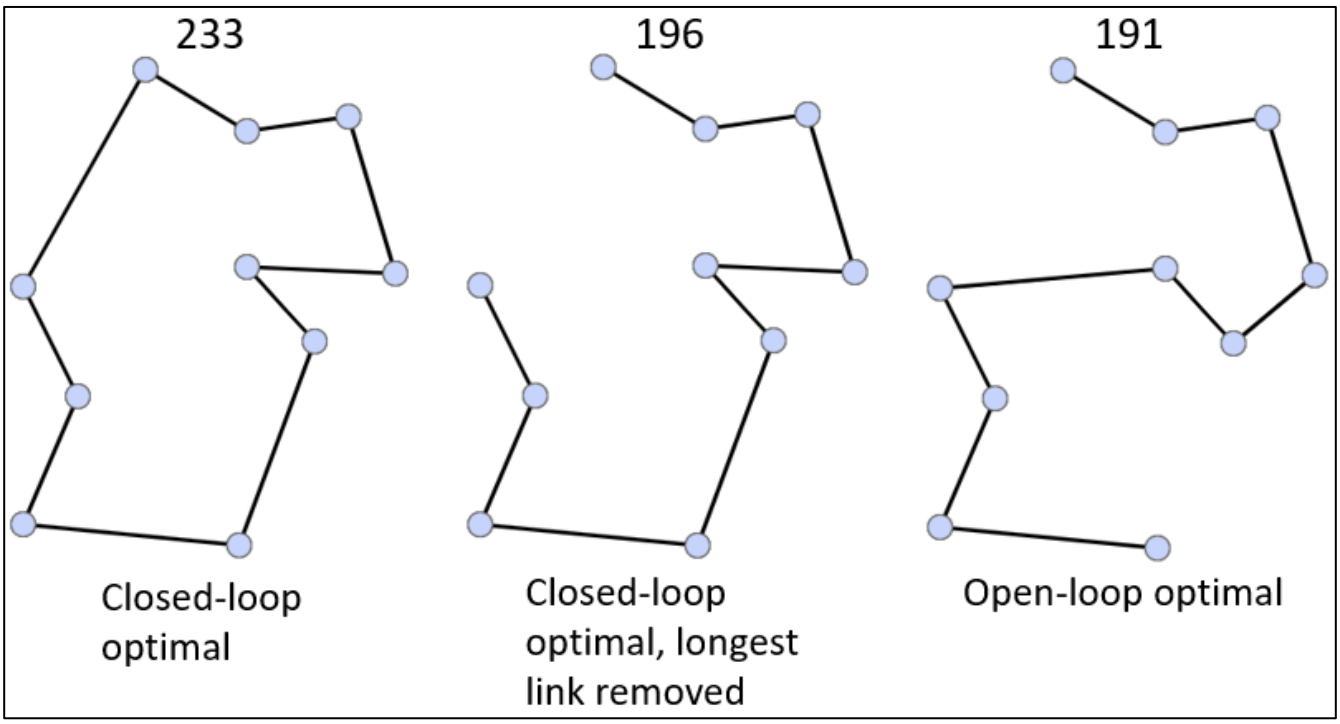

Figure 2. The difference between closed- and open-loop TSP. The optimal open-loop solution cannot be produced by cutting the longest link from the closed-loop optimal solution in most cases.

The rarer TSP variants include for example *fixed-start* and *multiple tours TSP* (Mariescu-Istodor & Fränti, 2021). The fixed-start TSP has a straightforward restriction: the path must always start from the given location. The idea of multiple tours TSP is to divide the problem to *k* travelers. The generalized variant of TSP, *vehicle routing problem* (VRP), is also worth mentioning. It is a more complex problem than TSP, as it takes the restrictions of real-world vehicle routing and service delivery into account. The added complexity is created by factors like time windows, multiple vehicles, and service depots. There are at least a dozen of VRP variations categorized by the type of restrictions that are present (Helsgaun, 2017).

2.2 Different heuristics and solving strategies

Different *heuristics* and solving strategies have been used for TSP as the brute-force approach is highly inefficient. A heuristic is an approach to problem solving where practical search rules are applied to the given problem in attempt to generate satisfactory results. Therefore, heuristics are

not guaranteed to find optimal or even good results but they manage to do so in most cases. TSP heuristics can be divided to three categories. There are (1) tour construction and (2) improvement methods (Nilsson, 2003), and (3) the combinations of these two.

Heuristic approaches are not the only viable option as optimal solvers also exist and they have been improving significantly over the years. The most famous one is called Concorde (Applegate & al., 2007) which holds the record for solving the largest TSP instance optimally. The instance contained 85,900 targets. However, it took Concorde approximately 136 CPU-years to solve as the time complexity is exponential. Heuristic solvers exist for this reason, and they are more popular than the optimal solvers (Mariescu-Istodor & Fränti, 2021).

The tour construction heuristics generate an initial solution to the given problem. These heuristics are usually simple in nature and they are not sufficient on their own as the constructed paths are nearly always inefficient. There are many straightforward construction heuristics for example nearest neighbor, shortest link (greedy), nearest insertion, cheapest insertion, random insertion, and farthest insertion (Weru, 2022). The nearest neighbor and shortest link approaches can easily be mixed up. The idea of the nearest neighbor approach is to move from city to city always visiting the closest unvisited city. The shortest link approach works by sorting all links between cities and picking the shortest link to be added to the path if it is legitimate to do so.

There also exists a more complicated approximation algorithm called Christofides which uses a minimum spanning tree in its process. This technique is more refined and it guarantees that the constructed path is at worst 50% longer than the optimal path (Nilsson, 2003). The time complexity of this algorithm is $O(N^3)$ whereas the simpler approaches usually have $O(N^2)$. The nearest neighbor approach is used in the thesis.

The minimum spanning tree conversion to a valid travelling salesman problem solution was also studied by Fränti & al. (2021). The algorithm works by branch elimination and provides clearly better results than the Christofides algorithm and other MST-based approaches.

Tour improvement techniques are the key for reliable results as the tour construction heuristics almost always fail to produce satisfactory path lengths. These techniques require an initial solution which is then improved with some type of strategy. The most common approach is to use local search operators or evolutionary algorithms such as genetic algorithms or ant colony optimization (Nilsson, 2003, Braekers & al., 2016).

The most advanced methods perform both the tour construction and improvement. *The Lin-Kernighan algorithm* (LK) is a well-known local search based heuristic which produces state-of-the-art level results for TSP. Discussion of the Lin-Kernighan algorithm is presented in Chapter 3.5.

2.3 Local search

Optimization of TSP with local search is the focus of this thesis. The idea of local search is to perform changes to an initial solution so that a better solution can be obtained. The candidate solutions are made iteratively by performing changes to random locations in the path with local search operators that perform changes in different ways. The search continues until a given time limit has been reached or if the path stops improving. As local search is a heuristic method, it does not guarantee an optimal result. However, near-optimal results are often found with good efficiency using this method.

Local search relies heavily on different operators and their usefulness. Each operator has a limited search space on which they generate new candidate solutions for the given problem. The effectiveness of this search depends on the properties of the chosen operators. In the context of TSP, this generally means using a variation of k-opt as it has been found to be the best method of solving TSP, at least in large-scale TSP instances (Mariescu-Istodor & Fränti, 2021). However, even if k-opt is the current dominant search operator, this does not mean that there could not exist a better operator or a combination of operators for TSP. These ideas started the implementation of a new operator for local searching which is analyzed in this thesis.

One of the weaknesses of local search is that most of the generated candidate solutions are meaningless (Mariescu-Istodor & Fränti, 2021). As only improving changes are usually accepted during local search, it is often inefficient to attempt changes that are not made within the local neighborhood. Especially, if the current solution is quite good already. These moves will rarely cause improvements in the path. The candidate solutions can be calculated very fast so this behavior is not relevant in small-scale TSP. However, large-scale TSP optimization suffers significantly as only a small number of changes are made in places where it matters. Therefore, localization of search space is needed in large-scale TSP instances.

There are two options to restrict the search space (Mariescu-Istodor & Fränti, 2021). The first one is a clustering strategy where the data points are placed in a cluster. The operators are restricted to perform changes to nodes and links within the same or neighboring clusters. The second option is to create a neighborhood graph by forming graphs such as Delaunay, Gabriel or XNN for the dataset, and use the same restriction logic. This is a powerful addition to local search in large-scale data and it was stated to be the most important design choice in local search (Mariescu-Istodor & Fränti, 2021).

2.4 Metaheuristics and search strategies

Metaheuristics are additional strategies added to the chosen heuristic. The metaheuristics decide which candidate solutions are accepted as new solutions. Different metaheuristic and heuristic combinations perform differently. Metaheuristics are deployed to either quicken the optimization process or give the operators a chance to escape local minima situations.

Greedy acceptance, *threshold accepting*, *tabu search* and *simulated annealing* are examples of different metaheuristics (Schrimpf & al., 2000) (Glover, 1989) (Rutenbar, 1989). Essentially, they all operate in a comparable manner. The metaheuristics decide whether changes with negative gain are allowed or not, and in what capacity. Greedy acceptance is the most straightforward one.

When using greedy acceptance, only beneficial moves are allowed. Threshold accepting applies a constant limit of how much negative gain is allowed in each move. Therefore, greedy acceptance is a special case of threshold accepting where the threshold has been set to zero. Greedy acceptance was used in this thesis.

Simulated annealing and tabu search are slightly more advanced. Simulated annealing employs the threshold idea from above but also allows it to change. The threshold starts at a high value and it decreases slowly as the search proceeds forward. This means that the number of negative gain moves can be initially high but only improving moves are allowed at the end of the search. Tabu search uses a different strategy. When positive moves are not found anymore, the tabu search allows moves with negative gain. This allows the search to move forward in local minima situations. However, the negative move must be remembered as if this were not the case, the counteracting positive gain move would be made in majority of cases. Therefore, the negative gain moves are placed in a tabu list to avoid this behavior, and this is where name tabu search is derived from. The tabu search can be applied in numerous different ways.

Search strategies include strategies such as random search, first improvement, best improvement, and prioritization (Sengupta & al. 2019) (Mariescu-Istodor & Fränti, 2021). Random search is a strategy where the operators are used in randomized places and all moves passing the threshold are accepted. First improvement approach is similar but instead of random order, the neighborhood is studied in an organized manner. Best improvement method is a time-consuming one. Therefore, it is not a practical one, especially in large-scale TSP instances. In this strategy, all available moves are inspected and the most beneficial move is selected. The prioritization method is an advanced technique used in LKH (Helsgaun, 2000). The operator's search is guided by a criterion called *alpha-nearness*. The alpha-nearness score is based on sensitivity analysis using minimum spanning trees. The links are scored based on their relatedness to the links in the formed spanning tree.

2.5 Motivation for new local search methods

As stated earlier, heuristics are needed for solving TSP efficiently because of its exponential nature. Exact solvers can be effective on small-scale TSP instances but they are unusable on large-scale TSP. Therefore, improvement of heuristic approaches is important. Local search has been a promising approach to TSP, and it has been used to optimize impressively large datasets. For example, a recent study arranged a Santa TSP challenge where the current methods for optimizing a dataset with approximately 1.4 million targets in one hour were analyzed (Mariescu-Istodor & Fränti, 2021). All of the working algorithms that were submitted to this competition were based on local search. If local search methods were to be improved further, increasingly better results could be seen in less time used.

The current methods are quite effective but they could always be developed further. Figure 3 is taken as an example. The most common local search operators could not optimize a seemingly simple TSP instance. The studied method, rebuild, is capable to escape the local minimum where the other operators are stuck in. The other operators are capable of finding the optimal solution in this case if the search is repeated several times. However, restarting the search in large-scale TSP is too time consuming. Therefore, the studied method could be used either as a main operator or a fine-tuning method alongside other operators.

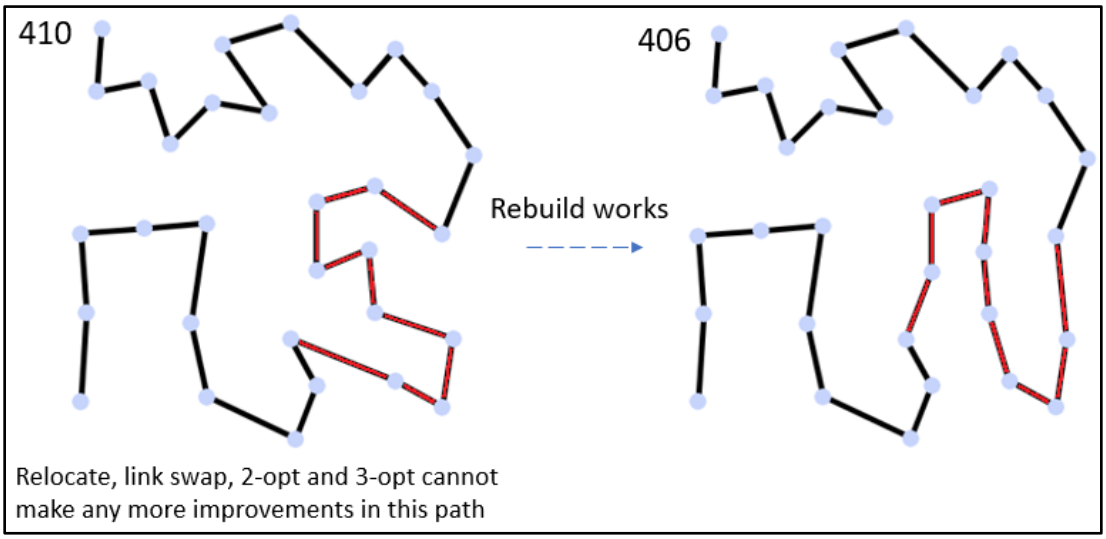

Figure 3. An example of the problems present in the most common operators. The problem was solved with the studied method.

3 Local search operators

Local search is based on using different local search operators. The operators decide the logic of how new candidate solutions are generated. They can be deployed either as a single operator or a group of operators. The most common operators are explained in this Chapter. Information will be given on their background, how they work and how big are their search spaces. The exact search spaces are given considering the open-loop TSP case, where there is one less link compared to closed-loop. The operators are relocate, link swap, 2-opt, 3-opt and the last two's generalized version k-opt. The uniqueness of operators is also discussed in the last Subchapter. The images are generated from real TSP instances (Mopsi dots dataset).

3.1 Relocate

Relocate is one of the recent proposed local search operators (Sengupta & al, 2019). It was developed to be used as a part of the random mix local search which also uses link swap and 2-opt operators. The idea came from a TSP path construction method called GENI, where new nodes are put into a solution one by one while changing other links in the process (Gendreau & al, 1992). This gave the creators the idea to make a similar move as a tour improvement operator. The operator is simple: it works by moving a random node from its current position to another random position in the tour (Figure 4). In this case, the move removed and added two links. If neither the moved node nor the position it is moved to is not the start or the end of the solution, 3 links are going to be changed. Tilo Strutz (2021) recently mentioned that relocate is a special case of k-opt. The size of the search space at any given time is $O(N^2)$ (Sengupta & al, 2019). There are N possible nodes to move and N-2 different destinations to choose from.

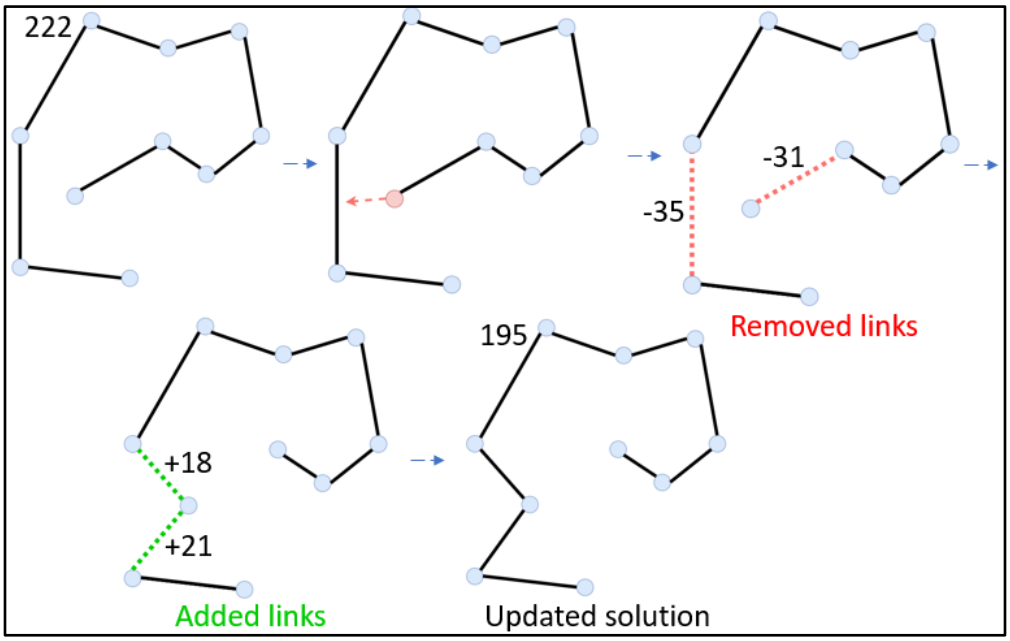

Figure 4: A demonstration of how the relocate operator works. The tour was made 27 units shorter in this example.

3.2 Link swap

Link swap is also one of the recently developed ideas for a local search operator (Sengupta & al, 2019). The operator is similar to relocate. Instead of a node relocation, a link is being relocated in this case. Because only one link is being moved, this operator is only applicable in open-loop TSP. An example of link swap is presented in Figure 5. Link swap considers one link, marked AB, and swaps it into a new position in the path. Only three new locations for the link keep the new solution valid (S = start, E = end): AE, SE, and SB. This move makes one or both nodes A and B new terminal nodes (Sengupta & al, 2019).

Link swap has a search space of size 3×(*N*-1) at any given time, which is O(*N*). The operator has *N*-1 choices for the link to be removed, and the path can be reconnected using three different options. Strutz (2021) mentioned that link swap is also a special case of k-opt. The creators also realized that link swap is a special case of 3-opt and relocate operators. The usefulness of the operator was justified with the fact that the size of the search space is linear, which is not the case for relocate or 3-opt.

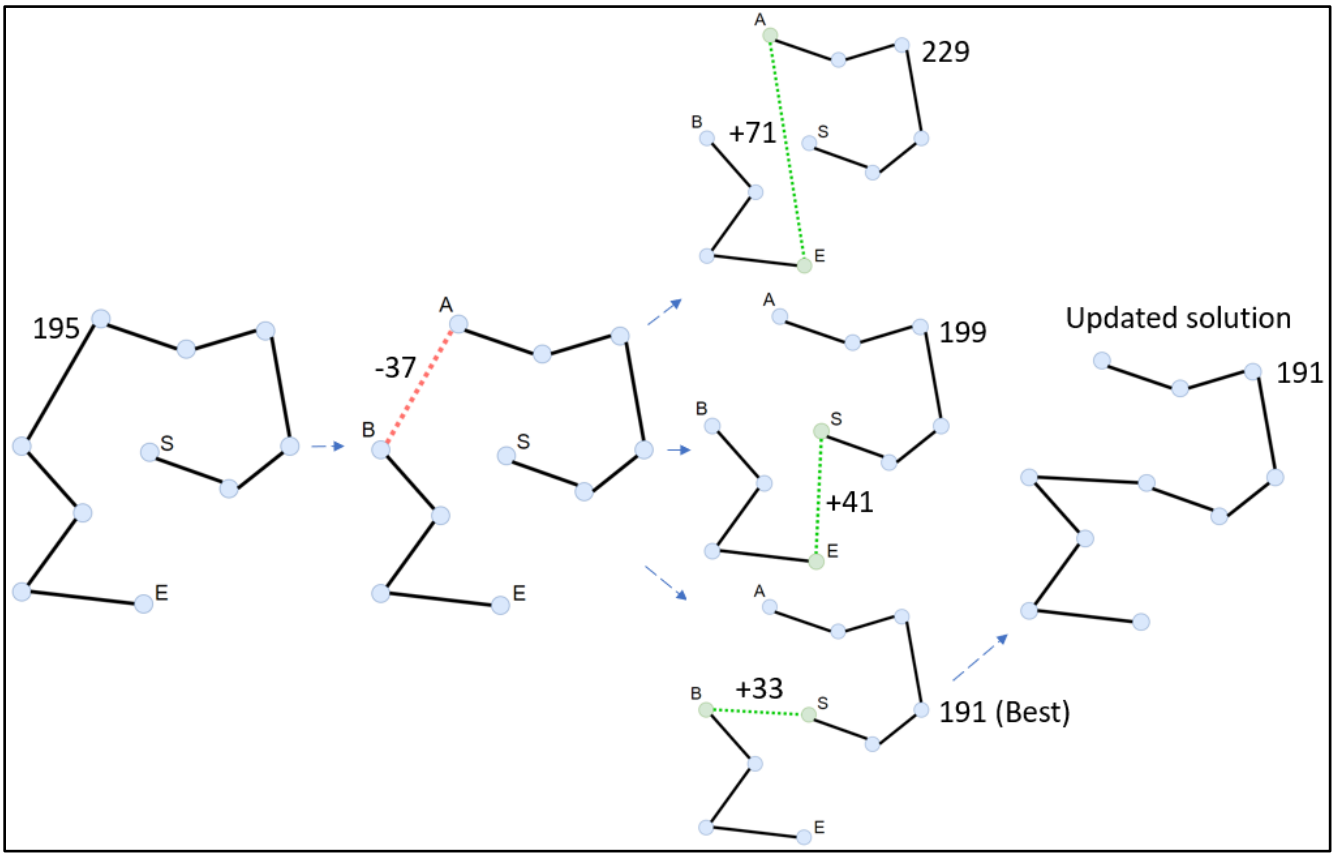

Figure 5: A demonstration of how the link swap operator works. A link is removed, and then the path is linked again with one of the three valid connection options. The tour was improved by 4 units in this case.

3.3 2-opt

The 2-optimization method, *2-opt*, is an old and common operator. It was first proposed by Croes (1958), and it has been through extensive research since then as it is a capable and a simple operator that is used for local searching. For example, Johnson and McGeoch (1997) made an extended study on the behavior and limitations of 2-opt. The 2-opt operator works by removing two links between two pairs of consecutive nodes, which are then replaced by two other links that complete the path (Figure 6). Another way to formulate the idea is that the path left between the two removed links is reversed in order. In case of 2-opt, there is only one valid way to reconnect the paths. The method was originally developed for closed-loop TSP, but it is also applicable to open-loop TSP (Sengupta & al, 2019).

The 2-opt operator is guaranteed to remove all crossovers in the path, but it is also capable of producing improvements that do not involve crossovers. For example, Sengupta et al. (2019) has examples of this behavior. The operator's search space at any given time is (N-1)×(N-2), which equals to O(N^2). This is because the first link can be selected in N-1 different ways and the second link can be chosen in N-2 different ways.

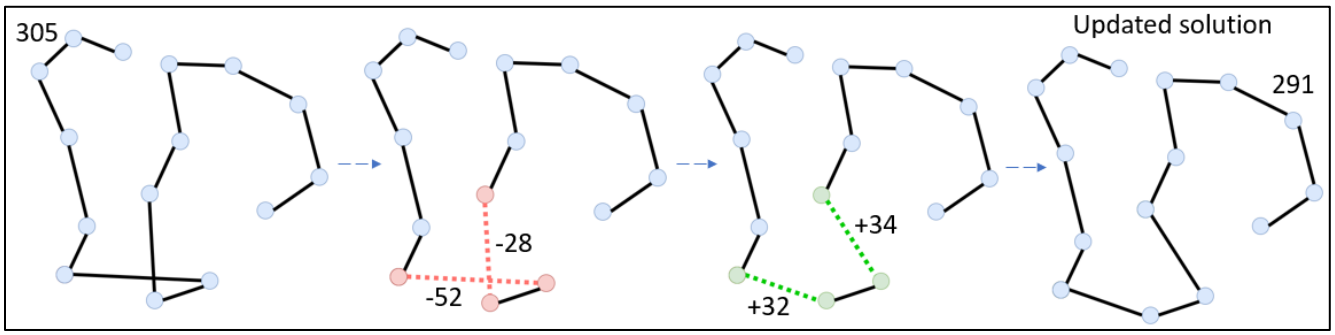

Figure 6: A demonstration of how the 2-opt operator works. Two links are being removed and the path is reconnected with the other way that involves these links. The path was improved by 14 units in this case.

3.4 3-opt

The *3-opt* operator works with the same logic as 2-opt. Intuitively, 3-opt manipulates 3 links instead of 2. The additional link greatly increases the number of different moves the operator can do. The operator was mentioned quite soon after 2-opt by Bock in 1958 (Johnson and McGeoch, 1997). One of the first papers done in the subject was by Lin (1965).

An example of the 3-opt operator is given in Figure 7. Three random links have been selected and those three links have been replaced with three other links that complete the path. When a path is broken like this, there are 7 new possible ways to reconnect the path. However, only four of these moves are unique to 3-opt. The remaining three can be obtained by 2-opt. In these cases, one of the broken links was reinserted in 3-opt. Because of the added complexity compared to 2-opt, the computational steps also increase. The three links can be chosen in (N-1)×(N-2)×(N-3)

ways and there are 7 options to choose from in all these cases. Therefore, the 3-opt operator's search space at any given time is in 7×((*N*-1)×(*N*-2)×(*N*-3)), which is equal to O(N^3).

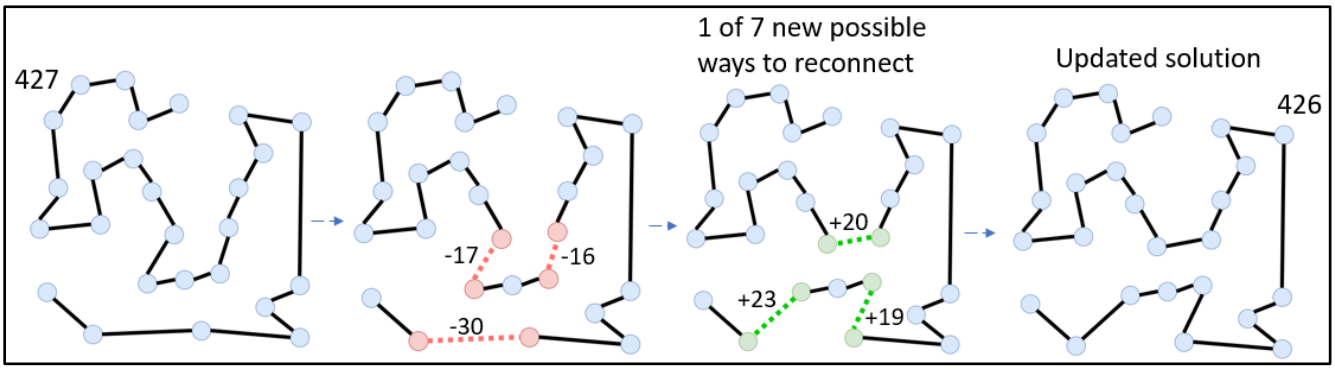

Figure 7: A demonstration of how the 3-opt operator works. Three links are being removed and the path is reconstructed considering all the 7 new cases that make a valid path. The path was improved by only 1 unit in this case. However, this example demonstrates that 3-opt can make quite complex moves.

3.5 K-opt

The *k-opt* operator is once again an extension of the 2-opt and 3-opt operators. It is a generalized version of these operators and follows the same logic. In this case, *K* number of links are removed, and the path is reconnected with *K* other links. The reconnection options grow exponentially as *K* increases. The effectiveness of different *K* values has been considered (Helsgaun, 2006). With a measure that gives equal weight to time taken and solution quality, *K* = 4 and *K* = 5 were the two best choices for *K* in a test performed by Helsgaun. However, if the best possible tour using this strategy is needed, *K* should be as large as possible. The parameter *K* still cannot be much larger as the processing time grows exponentially with *K*. The operator was first introduced by Lin and Kernighan (1973), which is why k-opt is also known by the name Lin-Kernighan heuristic.

The k-opt method is a famous and successful heuristic for the travelling salesman problem, and it is capable of record-breaking results in most TSP instances. LKH is an effective implementation of this heuristic implemented by Helsgaun (2000). However, the LKH package is not only an imple-

mentation of k-opt. In addition of using k-opt, other strategies for guiding the search are applied. LKH also received an update by Helsgaun later (2006) called LKH-2. This is a highly effective heuristic solver for TSP. For example, this package has held the record for the WorldTSP, a TSP problem containing 1 904 711 locations, almost without a break since 2003 (TSP, 2022) (Mariescu-Istodor & Fränti, 2021).

3.6 The uniqueness of operators

It is important to consider the uniqueness of each operator's search spaces. With this type of analysis, the trouble spots of different operators can be found. Additionally, a single operator might not be enough to provide good solutions for TSP. A combination of operators with complementary search spaces could be considered in this case. Another theory regarding a combination of operators is that one single operator could be stuck in a locally optimal solution. Then, another operator could find a better path while giving the original operator a chance to find improvements to a slightly modified situation.

Two example situations are considered with relocate, link swap, 2-opt and 3-opt. In the first one, a strong operator, 3-opt, cannot find an improving move while slightly weaker operators, relocate and link swap, are able to improve the path (Figure 8). In this case, both 2-opt and 3-opt cannot find an improvement to the path. When relocate is used, an improvement of 8 units is guaranteed to be found, which is most likely the optimal solution for this instance. Link swap has a 50% chance of finding either a 6 units better solution or the 8 units better solution. Interesting thing is that after the 6-unit change by link swap, 3-opt can then find the optimal 384 units long route.

There is an important aspect to consider when implementing 3-opt. It seems that with a special implementation of 3-opt this case could be handled with this operator also. The limitation here is that the operator cannot operate normally because it is operating at the end of the solution. This case could be handled with an implementation that considers a pseudo node at both ends of the solution. This way the operator can make the needed move properly.

In the second example, a similarity between 3-opt and relocate is shown (Figure 9). These operators work in separate ways, but they have colliding search space in one situation. This example shows quite clearly that relocate is a special case of k-opt as stated earlier. When a node is relocated in the middle of the path, the move removes and replaces three links (3-opt). If the move involves either end of the solution, then only two links are being removed and replaced. This case is similar to 2-opt. It is important to note that while in this example 3-opt and relocate function the same that is not the case in a usual case. Search space for 3-opt is much larger as shown in the experiments section.

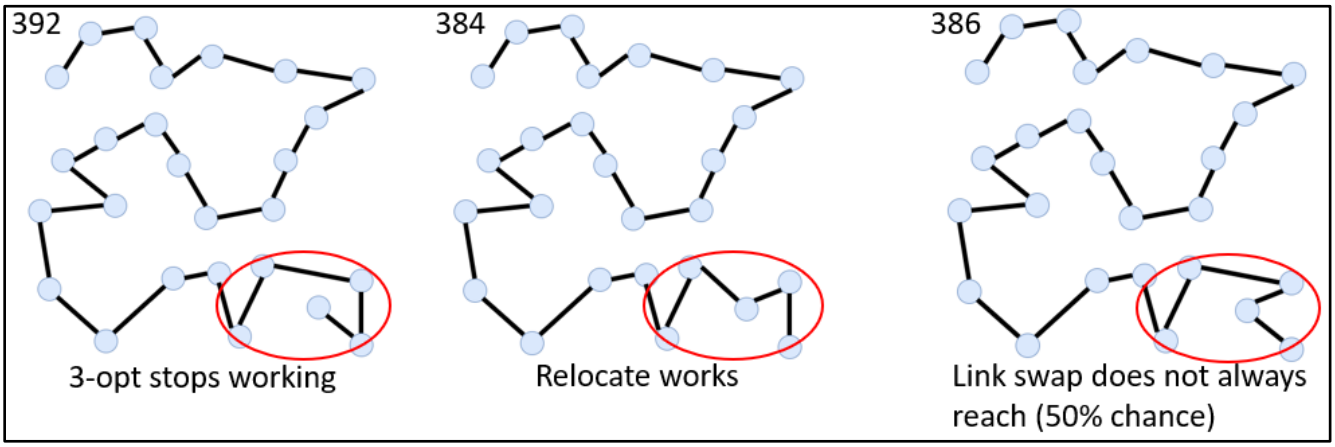

Figure 8: An example of different search spaces. The 3-opt operator cannot find an improvement in the path on the left. Both relocate and link swap can find the optimal path. However, link swap has a 50% chance to do so as it depends on which link is chosen to be operated on.

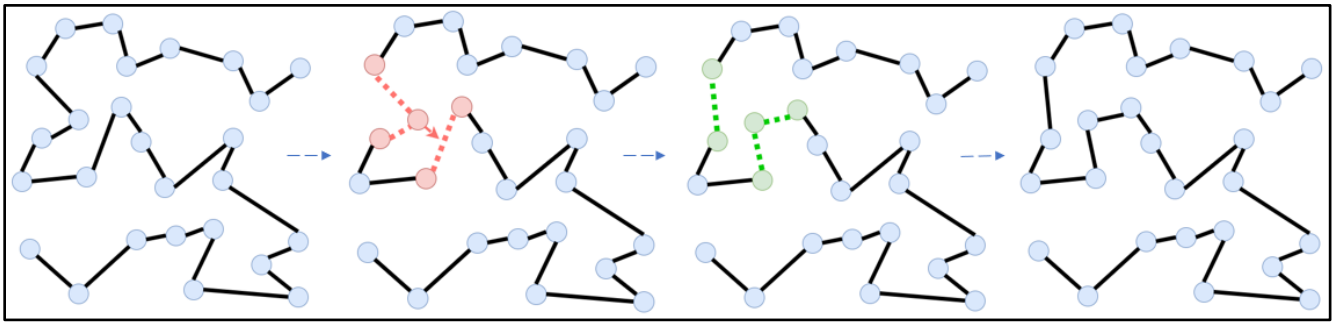

Figure 9: The move that is the same between 3-opt and relocate. The dotted lines indicate what links must be chosen by 3-opt for this change to happen. The red arrow indicates how the move works from relocate operator's point of view.

4 Rebuild operator

The new operator was created with an idea that destroying a part of a TSP solution and constructing it again could make the solution more efficient. Additionally, this operator could help with escaping local minima as it produces significantly larger variations to the candidate solutions than other operators, especially if the deconstructed path is large enough. The operator was given the name *rebuild*.

The basic idea of rebuild is to take an initial solution and remove some number of consecutive nodes from the solution, which are then placed back to the solution one by one. Like all local operators, rebuild is an iterative process. Each iteration consists of the following steps. First, the number of nodes to remove is decided. That is indicated with a *K* value. After that, a target node A is randomly picked and the end point B is calculated, so that *K* nodes are left between A and B. The nodes are removed by connecting A and B. The disconnected nodes are now connected back to the solution one by one in a random order. An example of one iteration of rebuild is pictured in Figure 10.

The reconnection is done to the nearest node in the path at every step. However, there is one decision that must be made when putting the nodes back. When the closest node in the path is calculated, the node that is joining the solution can be put back before or after that node (Figure 11). Both situations are always considered and the more favorable one at that moment is chosen. Alternatively, the destination of the relocated node could be decided by selecting the lowest cost position for the node. That might be an effective strategy.

After all nodes have been returned to the solution, the distance change of all these link additions and removals are known. If the cost change is advantageous, the current state of the solution is kept. However, if the cost change is unfavorable, returning to the old solution is needed. The easiest way to achieve this is by taking a copy of the solution before making any changes. This ap-

proach might be unfeasible when working with large-scale data. Another more complex approach involves making a list of all the changes made to the solution. This list could then be used to bring back the old solution in case of an unsuccessful iteration. This method was not implemented. If implemented, it could provide additional speed-up in large-scale TSP problems.

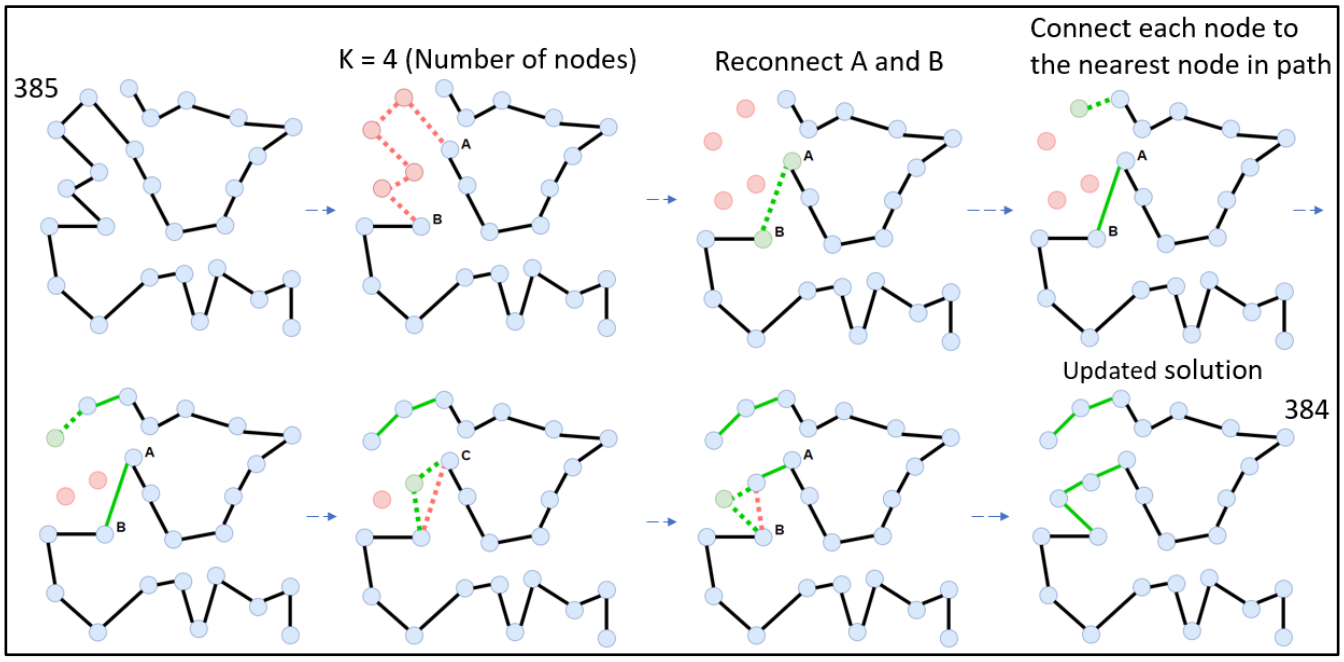

Figure 10: An example iteration of rebuild. First, the value of *K* is decided, which declares how many nodes are removed from the solution. Then, end-points A and B are randomly chosen and linked. The removed nodes are placed back into the path one by one. Red and green lines show the removing and adding of links, respectively.

The following Subchapters provide additional details about the design choices made for this version of rebuild. These include a discussion of how the cost calculation is done, notes on deciding *K* values and how rebuild can be implemented in open-loop TSP. Additionally, the order of adding nodes back in a path and time complexity of rebuild are examined.

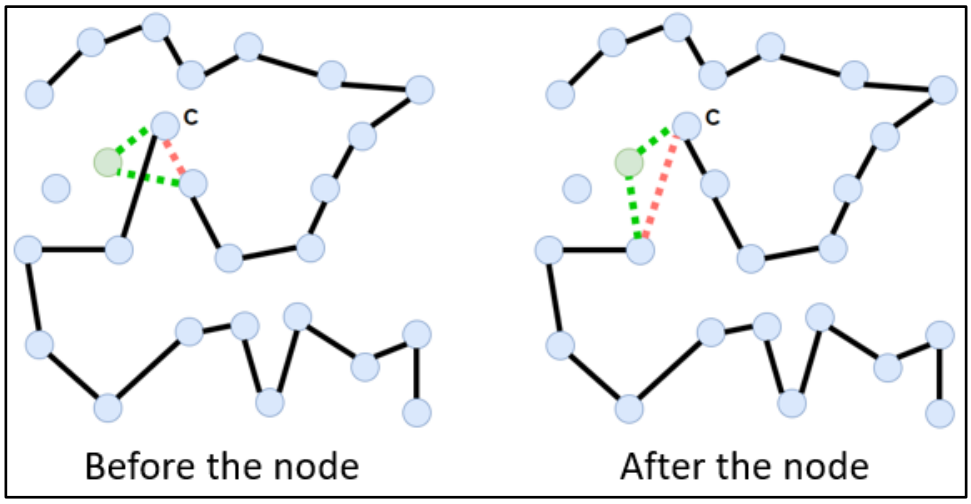

Figure 11: When putting nodes back in the path, their placement can be made before or after the node that is nearest (C). In this case, the node marked with green was being placed before and after the node C.

4.1 Cost calculation

The cost calculation for rebuild is relatively simple. However, due to how this operator functions, the cost calculation differs from other local search operators. In this context, cost calculation means calculating the change of distance when links are added or removed. It is essential to calculate the cost in this way because this allows measuring the effects of operations before they have been made to the data structure itself. Furthermore, calculating the length of a TSP solution by simply adding the length of each link together after every iteration is too slow during local search. Cost calculation for rebuild had to be done while adding the nodes back to the solution.

Cost calculation is generally done before any changes are made when using local search operators. This was the starting point for the implementation of rebuild. However, during the initial stages of implementation, it seemed that this approach would be either overly complex or even impossible to make work with rebuild. The problem is that when putting the nodes back to the solution, one insertion cannot be made without considering where all the other removed nodes are going to be placed. The path has already been altered when the *K* number of nodes were removed and when some of the removed nodes were put back to the solution. In other words, the

rebuild operation has multiple steps whereas other operators produce one simultaneous move. There could be a way to solve this with a process where all the moves are recorded in a data structure and then taken account at each step. This would vastly speed up the processing of rebuild. Additional complexity is added by the fact that a relocated node can be placed before or after the chosen node.

A variant with only $K = 1$ was also implemented in the initial stages of designing rebuild. This was a very straightforward implementation, but as expected, it lacked the power to do anything meaningful. Logically, it makes sense as rebuild with $K = 1$ is just a limited variant of relocate. It can only relocate a node to its closest neighbor. Therefore, without generalizing this to any K, this would have been an ineffective way to implement this operator. Some reconsideration was needed and that lead to the ideas that were presented earlier in this Chapter.

4.2 Deciding K value

The most important design choice when using rebuild is the K value. The variable K indicates how many nodes are being removed and reconnected each iteration between the nodes A and B. Also, the number of links removed in the initial step of rebuild when the nodes A and B are connected is K. Between A, all the removed nodes and B, there are $K + 1$ links to remove but one link is inserted back between A and B, which gives the total amount of K removed links. This means every time a node is reconnected to the path, one link is also made.

In this thesis, two main types of K values are considered. One of which is constant and the other one is a varying value of K. With the varying value, K is simply randomly selected between a range of values each iteration.

The nature of local searching by default is random. The optimization is based on making a large amount of solution candidates which allows the optimization to move forward. Rebuild has one fixed property that other operators do not have. The reconnection of the nodes is only consid-

ered to the nearest node in the path. This property makes rebuild one of the least random operators. While being not as random is not a bad property to have, it reduces the number of candidates the operator can produce in this case. Therefore, a varying *K* value was considered early in the development of rebuild.

The varying *K* value turned out to be a superior design choice as seen in the experiments Chapter. With a constant *K* value, rebuild would only have *N* different spots to consider for improving the tour. This makes it likely to find a situation where no more improvements can be found. The varying *K* value alleviates this problem considerably. Fundamentally, different *K* values can solve different kind of problems in the path.

4.3 Strengths and weaknesses of rebuild

Rebuild has few clear strengths and weaknesses which are discussed in this section. Analyzing these traits is important in few ways. By finding the trouble spots with rebuild, the rebuild operator can be tuned to fix these problems. This can be done either as an extension of rebuild or as a completely new operator. Another, maybe a more valid reason for such analysis is that the problem cases of rebuild could be covered with another operator. Combining rebuild with other operators is assessed later in the thesis.

One of the clear weaknesses include crossovers in the path. The worst-case scenario for rebuild in small data is presented in Figure 12. In this case, the path has a large loop structure and a crossover that is unsolvable for rebuild. The *K* value would need to be large enough to cover the entire loop or at least most of it. The loop in this case consists of about 25 nodes.

The lowest *K* value that was able to continue optimizing the worst-case configuration was 22. However, it took considerable number of iterations to do so as the operator had to find the exact starting position and ordering to find this move. Higher values were much more feasible for solv-

ing this case. For example, K = 25 was able to continue optimization after just a couple hundred iterations on average. A shorter path is shown in Figure 13.

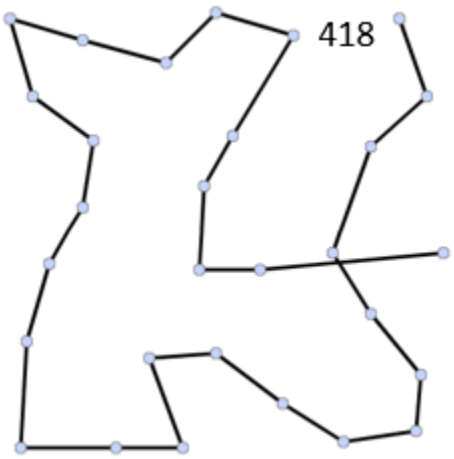

Figure 12: The worst-case scenario for rebuild. A sizable loop structure followed by an unsolvable crossover. Dataset: Mopsi dots-4339.

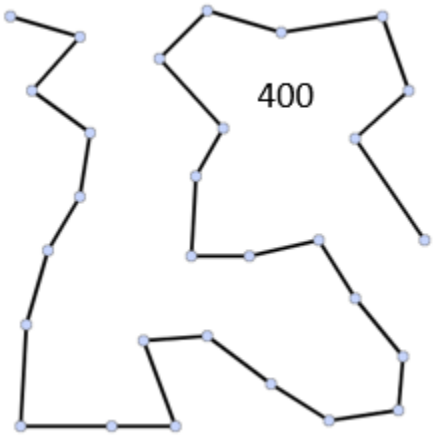

Figure 13: The worst-case scenario for rebuild solved by a large enough K value. Dataset: Mopsi dots-4339.

The problems with crossovers intensify with larger datasets. With small instances, it is possible to increase K enough to cover these cases. This is not possible realistically with larger data. Figure 14 shows an example of the problems with crossovers. The loops are simply too large to be fixed by bigger K values. The only remedy to this situation is combining rebuild with some other operator. The k-opt operator would be a likely candidate as it is known for breaking crossovers easily.

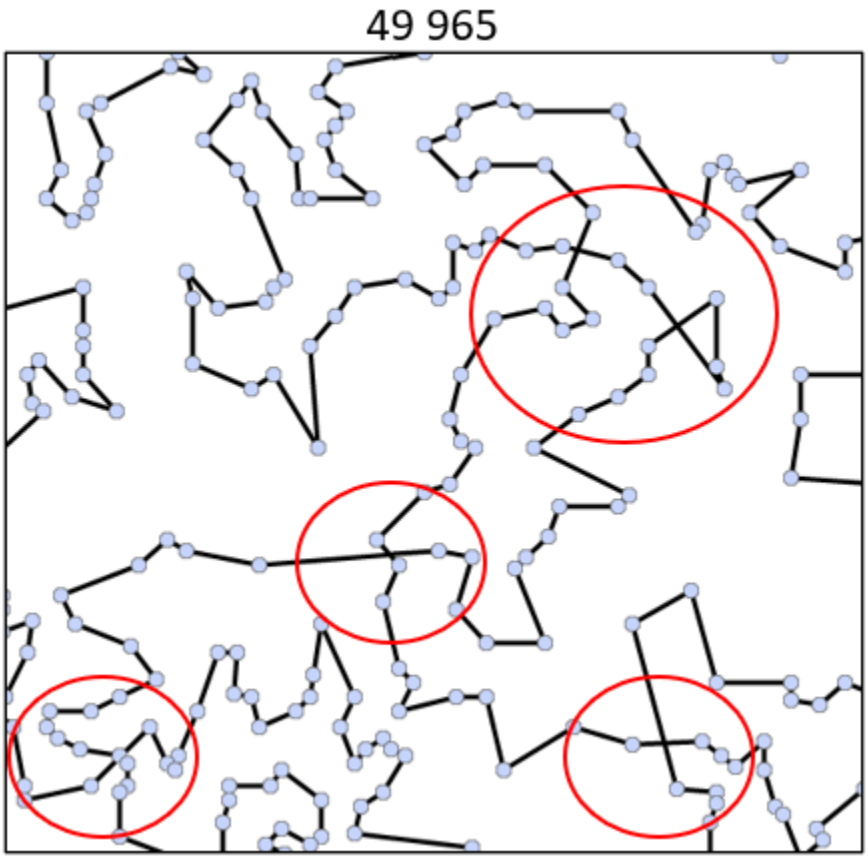

Figure 14: Crossovers are a problem for the rebuild operator. The situation was obtained after rebuild operator (with K = 5–15) was used until no longer improvements were found. The example was generated using a TSPLIB instance d657 (Reinelt, 1991).

Another weakness of the rebuild operator could be that removing consecutive nodes is too restrictive search sometimes. An alternative strategy would, for example, involve destroying the paths in some defined area, which should make the search space larger.

Strengths of rebuild include the removal of inefficient detours and its capability to make larger scale changes compared to other operators. Additionally, no other operator can optimize a consecutive line of nodes like rebuild can. The capabilities of rebuild are going to be shown in pictures. In larger instances, rebuild must be combined with at least 2-opt or 3-opt to eliminate crossovers. Therefore, 3-opt was first iterated for this visualization until no further improvements were found. After that, rebuild was applied to improve the path further. Results are shown in Figure 15 using the TSPLIB instance d657 again. The images display a portion of the complete

path. The path improved significantly overall, and the number of inefficient detours was reduced. Especially, the clearly unwanted path leading out of the picture to the top right was removed. The total path length was reduced from 50979 to 48500 with these kind of changes to the path.

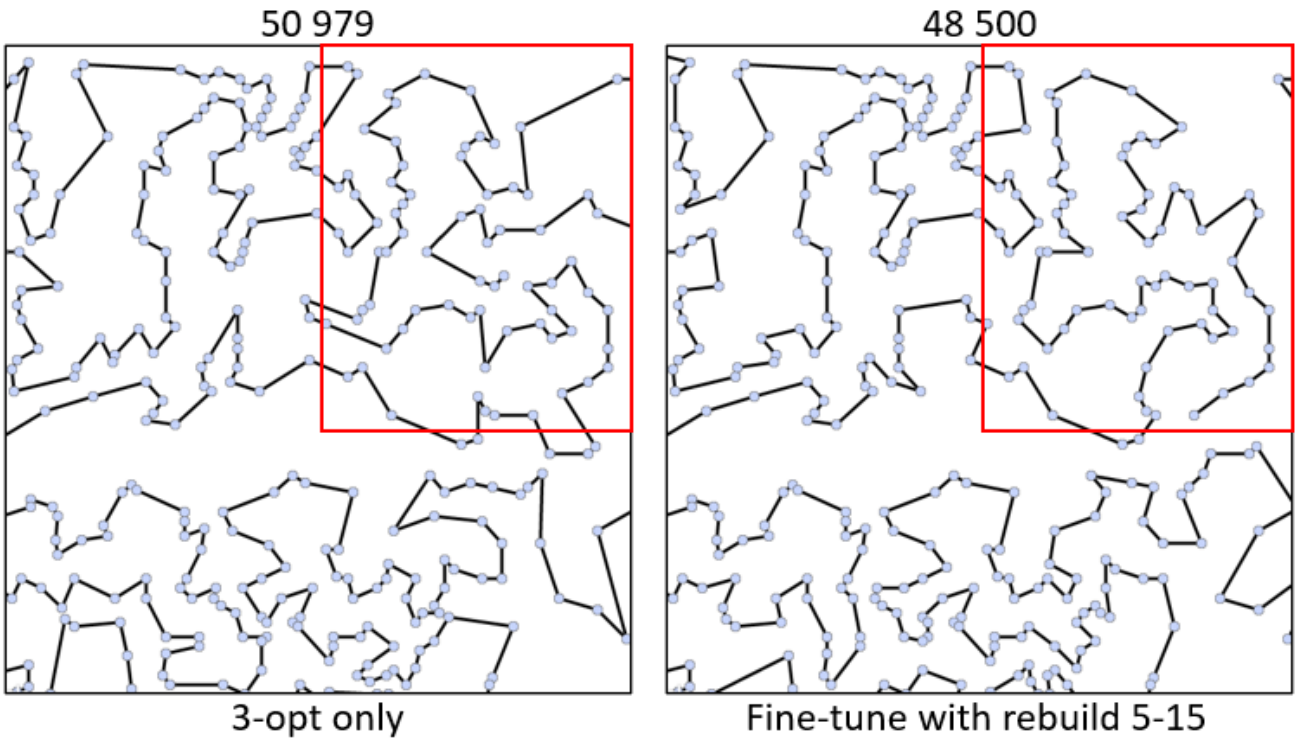

Figure 15: Improvement capabilities of rebuild visualized. The total path length reduced from 50979 to 48500. The example was generated using a TSPLIB instance d657 (Reinelt, 1991).

A similar test was performed on TSPLIB instance u1060 (Reinelt, 1991). The results are shown in Figure 16. The overall length was reduced from 243602 to 230861. The most notable changes are marked with red circles. In this case, the fine-tuning was also applied with larger *K* values afterwards. These results are shown in Figure 17. Additional improvements were found in this case and the overall path length improved by 618 units.

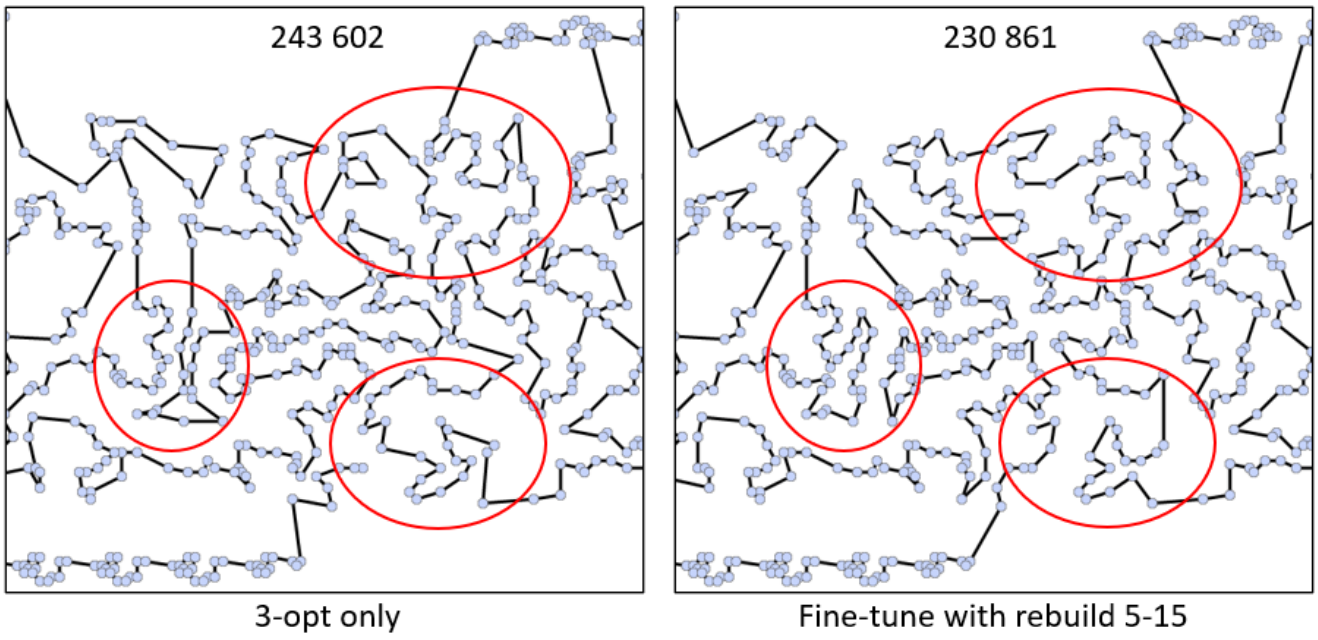

Figure 16: Improvement capabilities of rebuild visualized. The most notable changed are marked with circles. The example was generated using a TSPLIB instance u1060 (Reinelt, 1991).

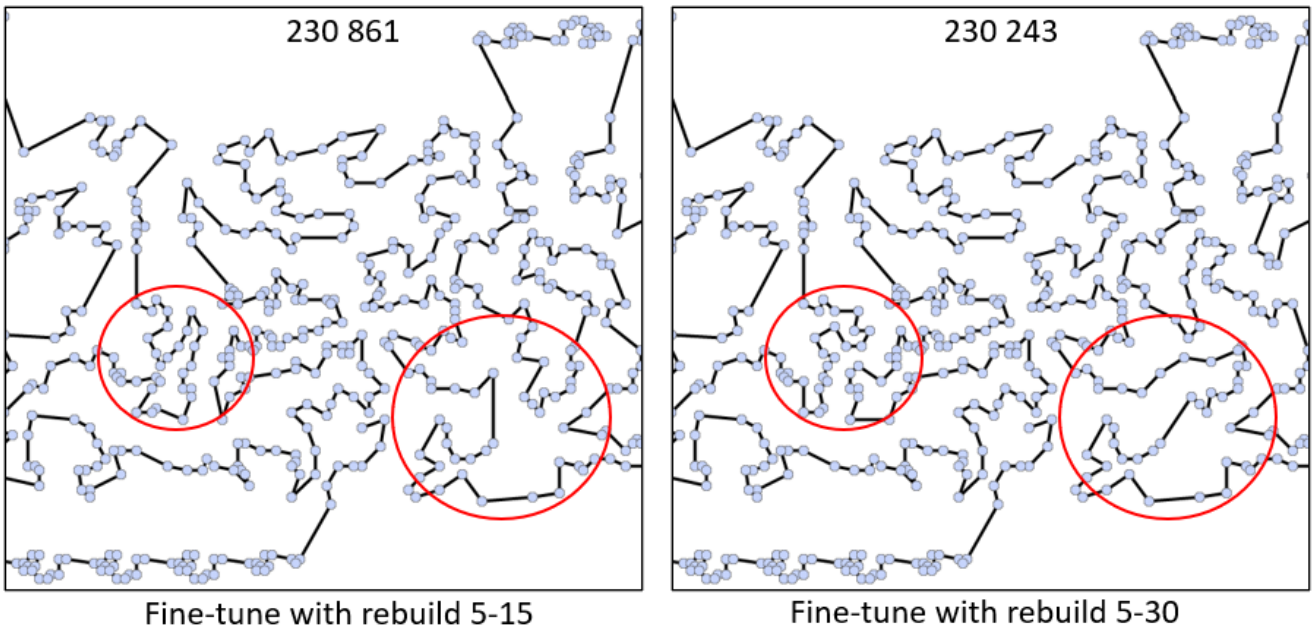

Figure 17: Improvement capabilities of rebuild visualized. The most notable changed are marked with circles. The example was generated using a TSPLIB instance u1060 (Reinelt, 1991).

4.4 Rebuild in open-loop case

Using rebuild in closed-loop case of TSP is trivial. The solution path is continuous, therefore the cutoff point between the start and end point are meaningless. However, this is not the case in open-loop TSP. In open-loop TSP, the start and end points can be in separate locations. There is a decision to make here. The rebuild operator can be either allowed to loop in open-loop TSP, allowing changes to both the start and the end of the solution at the same time or this looping feature is not allowed. In the latter case, the target needs to be selected between the indexes 0 and $N - K - 1$ to avoid overflowing. All the results in the thesis are done without looping. However, the looping strategy is likely advantageous and rebuild should be implemented in this way. The improvement of managing the small-scale instances is most likely noticeable. In these cases, the start and end points can be quite close to each other.

4.5 The order of adding nodes back into the path

We consider two ways of inserting the removed nodes back into the path when using rebuild, random and default order. Default order adds them back in the same order in which they were removed. Both strategies were evaluated on one instance of data. The results are shown in Table 1. Randomized order clearly provides better solutions. With K = 5–10, K = 5–15 and K = 5–20, the default order solutions were 3.5%, 4.6% and 2.4% worse on average, respectively. The improvement is caused by the larger search space that the random order provides. Arranging the solution back in a random order will provide much more solution candidates compared to default order.

The better solutions are not obtained without a cost of performance. However, the additional time taken is negligible in all cases. Shuffling K number of nodes in an array has O(K) time complexity with Fisher–Yates shuffle. This is repeated every iteration alongside the default O($K \times N$) iterations done by rebuild. Some additional overhead is caused by this operation. The effect was evaluated with small and large data (Table 2). The results show that randomized order on small

data causes about 50% slower iterations on average. However, this slowdown is redundant as optimizing a short path like this takes only few milliseconds as shown in the testing section. On a larger TSP instance, the performance is the same between randomized and default order as the shuffling operation takes no time compared to the whole operation. In this case, the randomized order was a 5–10% faster. This behavior was repeatable, and no definitive cause was noticed. Other steps of rebuild must benefit from randomizing the order of nodes in some way.

Table 1: The results of testing rebuild with random and default order node insertion. Results are shown in path lengths. Different ranges of *K* were tested. Data was collected from 50 runs of each variation on Mopsi dots-4400 instance. Same starting solution each time (length = 498).

Order	Random			Default		
K	5–10	5–15	5–20	5–10	5–15	5–20
Best	406	406	406	412	406	412
Average	424	409	410	439	428	420
Worst	444	427	420	468	468	460

Table 2: The results of testing random and default order in terms of time taken. The numbers are formed as a 10-run average. Before collecting the results, an additional 10 runs were allowed to run to standardize processor output.

N	31						5000					
Iterations	1 000 000						10 000					
Order	Random			Default			Random			Default		
K	5–10	5–15	5–20	5–10	5–15	5–20	5–10	5–15	5–20	5–10	5–15	5–20
Time (ms)	1179	1532	1963	742	1027	1326	983	1359	1659	1089	1432	1770

4.6 Confirming linear time complexity in relation to K

Rebuild is expected to have a linear time complexity in relation to *K*. It is different from k-opt which scales exponentially in relation to k, so bigger *K* values with rebuild are possible to be used. A simple test was performed to guarantee that this was the case. Figures 18 and 19 show these results for both small and large data. The time complexity of rebuild is O(*K* × *N*). In more detail, the array manipulation operations such as removing K nodes and copying also take O(*N*) time. The steps taken by the algorithm equals to *K* x 3*N*, which is equal to O(*K* × *N*). In small TSP instances, the overall time complexity can be close to O(N^2) if K is almost equal to *N*. However, in a usual case it can be assumed that *N* >> *K*.

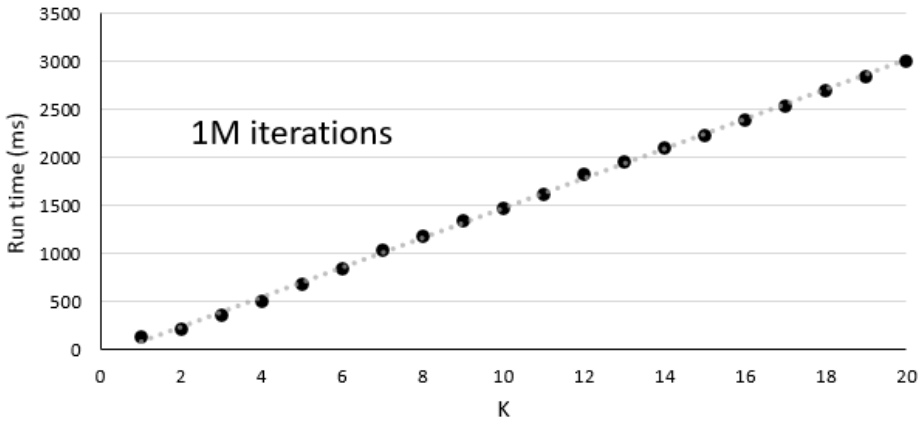

Figure 18: Rebuild run time with various *K* values (10 run average) with linear regression line. *N* = 31.

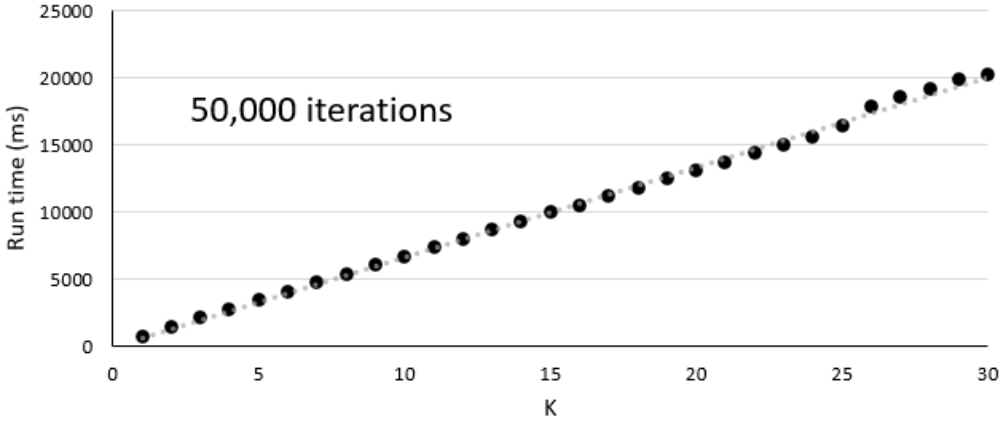

Figure 19: Rebuild run time with various *K* values (10 run average). *N* = 5000.

5 Similar techniques in literature

After the implementation of rebuild, it was found out that similar methods have already been researched. One of the first similarities can be found few decades ago. Dees and Karger (1982) developed a circuit board optimization method called rip-up and reroute. This technique was used to optimize the wiring paths. It worked by removing existing wiring on the board, which was then reinserted back to the path in a different orientation. The exact algorithm was not described in this paper.

Almost the same method as rebuild was proposed more recently. The creators call it *"the ruin and recreate"* method (R&R) (Schrimpf & al, 2000). The differences of rebuild and R&R are that rebuild considers only a sequential rebuild operation when R&R combines sequential, radial, and random removal operations, or ruins as they are described. Radial ruin works by removing *K* closest neighbors from a selected node. The *K* value was calculated in a way that it is proportional to the size of the problem. Proportions of 1, 2, 5, 10, 20 and 50 percent were considered. This forms the upper bound for *K* and the lower bound is always 1. The paper showed promising results for solving TSP compared to 2-opt.

The R&R heuristic considers also different metaheuristic compared to rebuild. Rebuild uses only greedy acceptance. Different metaheuristics for R&R were discussed in Schrimpf & al (2000) and the best results were obtained with linear simulated annealing. The path recreation phase is done using best insertion for the nodes whereas rebuild uses nearest insertion. The R&R paper mentions that it would be possible to study more sophisticated recreation methods for better results. Lastly, the initial solution of R&R was formed using R&R for the whole problem as the first iteration. The usefulness of this method remained unclear as it was compared to a randomized initialization which always creates rather poor results.

The ideas of rip-up and reroute, and R&R were developed further by Pisinger and Røpke (2007) with their *adaptive large neighborhood search* (ALNS) heuristic. This heuristic was designed specifically for the vehicle routing problem but applying it to TSP would be trivial. The new major features of ALNS include additional destruction and recreation methods, and an adaptive layer which controls where the destruction and recreation steps are applied. The paper considers seven different destruction methods and two recreation methods. It suggests that combining several methods together provides better quality results. The adaptive layer chooses where the operation is performed. Successful iterations in a particular area improves the chances of choosing that area again.

The ALNS algorithm also uses simulated annealing as metaheuristic. It is mentioned that other metaheuristics can also be applied but simulated annealing was chosen. Additionally, it is suggested that noise should also be added into the objective function. It is claimed that this helps to diversify the search.

The K value for ALNS changes depending on the size of the problem. Their past research indicated that large moves are rarely accepted because the insertion methods are too weak. Small moves are also ineffective as they seldom provide any major improvements to the solution. It was decided that the lower bound for K is calculated to be the minimum of values $0.1 \times N$ and 30. The upper bound is calculated to be the minimum of values $0.4 \times N$ and 60. This means that the lower bound of 30 is reached in problems bigger than 300 locations, and the upper bound of 60 is reached in problems bigger than 150 locations.

The ALNS method's success was also noticed by Microsoft Bing. They recently developed a cloud service for *Multi-Itinerary Optimization* (MIO) using a slightly modified version of ALNS as their optimization method (Cristian & al, 2019, 2021). In this case ALNS was applied to VRP as well. MIO is the same as ALNS but with different destructions and repair methods. MIO's destruction methods are called *skips*, *high cost*, *replace item*, *neighborhood*, and *transfers*. The repair methods are called *early*, *late*, *nearest neighbor*, and *greedy*.

The results of MIO were compared against another state-of-the-art heuristic, LKH. The third version of LKH is an extension to LKH1 and LKH2, and it was specifically designed for different VRP variants (Helsgaun, 2017). MIO performed better in the tests done by Cristian & al (2021). MIO managed to find better solutions in all test cases. When the optimization speed was considered, MIO was only slower in modest problem sizes. MIO had noticeably better running times otherwise.

6 Experiments

The logical step after implementing a new operator is to evaluate it against other operators. The tests also include comparison of different *K* values, and the combination of rebuild and another operator. The experiments are constructed in a way that a very straightforward instance is considered first. After that, small-scale solving ability is examined followed up by large-scale testing where operators attempt to solve an instance with 5000 points. It is important to note that a problem with 5000 points is not a very large problem by today's standards (Mariescu-Istodor & Fränti, 2021). This test was considered large-scale in this case because the effectiveness of plain local search starts to deteriorate with large neighborhoods. If local search is deployed on orders of magnitude bigger data, additional strategies are needed, like divide and conquer or localization of operators. Finally, few chosen operators are tested with 10 different TSP instances of varying sizes (76 to 5000). All testing was done on machine with a Ryzen R7 5800X processor and 16 GB of 3800 MHz RAM.

The structure of this Chapter is the following. First, the test instances are described in the first Subchapter. The simple test with one instance is considered next with all operators. The results of combing rebuild with another operator are presented in the third Subchapter. The Chapters 6.4 and 6.5 test rebuild with different *K* values in terms of accuracy and efficiency with small and large TSP instances. The combination of rebuild and 3-opt is studied more closely in Chapter 6.6. Finally, few chosen operators are evaluated in a final benchmark. The TSP instances of the final benchmark are presented in the last Subchapter.

6.1 Datasets

The datasets have been chosen so that each operator is tested in simple small-scale instances and more complex large instance. It is important to consider both cases. The small-scale testing will provide concrete examples of how the operator performs changes. It will also give an indica-

tion of how capable the operator is in the larger scale tests. The large-scale testing is the more important test case. These tests will show the operator's strength more clearly as the original solution will need major tuning to reach a satisfactory solution.

The small-scale testing is performed using Mopsi Dots datasets (Sengupta & al., 2019). These are small computer-generated open-loop TSP instances with *N* varying between 5 and 31. Six datasets have been chosen from the whole dataset which contains 6449 instances. This quantity seems suitable enough to draw conclusions from given that each test case includes dozens of repeats. The instances have been picked by random selection while trying to pick the ones with close to maximum *N*. This has been done to avoid instances that are too simple. Dots datasets were selected because they seemed presentative of simple small TSP instances that are easy to analyze and could be attempted to be solved in real-world situations. The datasets were initialized with random start nearest neighbor approach which gives them their initial lengths. The instances are visualized in Figure 20.

The large-scale testing section was done using only one TSP instance called *Santa5000*. This was a small version of the dataset used in Mariescu-Istodor & Fränti (2021) which has approximately 1.4 million targets. These targets represent the households of Finland, and they were gathered using OpenStreetMap (OSM) and the Overpass API. The 1.4 million targets were reduced to 5000 for the purpose of this testing. It is a random subset of the original data. This seemed like a suitable dataset for large-scale testing. The instance includes a good variety of sparse and dense areas as can be seen from Figure 21. This should assess the operators general capability adequately.

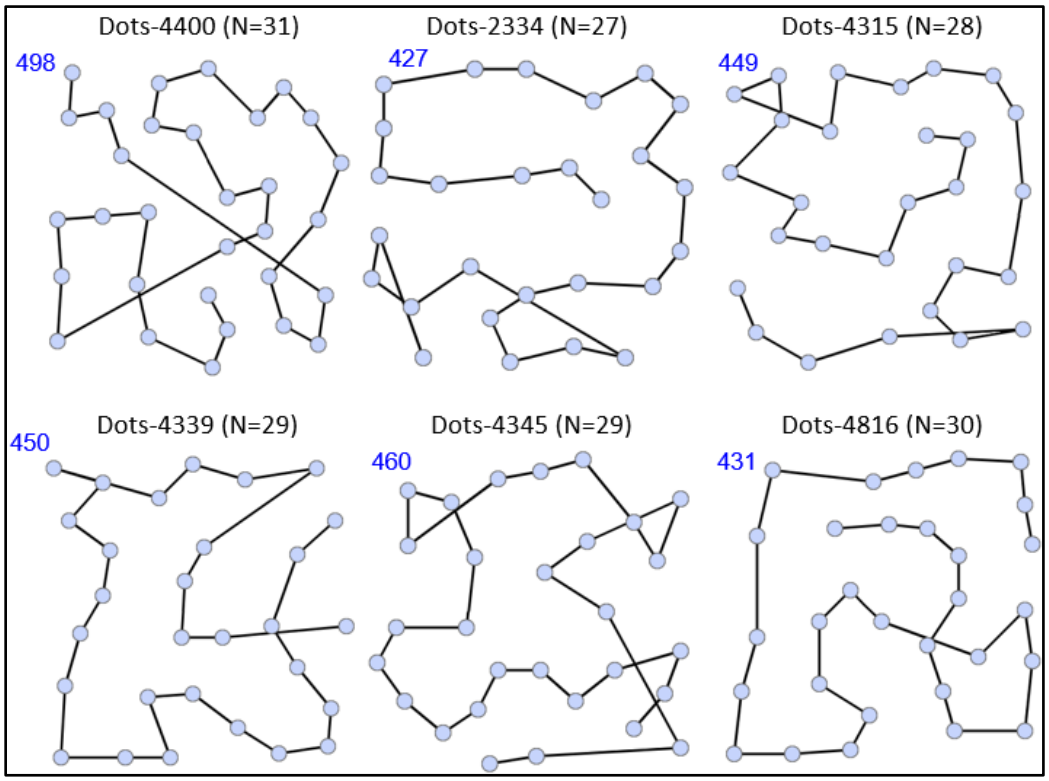

Figure 20. Visualization of the Dots datasets that are used for small-scale testing. The lengths are marked with blue text.

Figure 21. The santa5000 instance.

6.2 All operators on one small TSP instance

The first test is a simple one. Relocate, link swap, 2-opt, 3-opt, and rebuild are evaluated on dots-4400. Random mix (Sengupta & al., 2019) is also included in the results which is a combination of relocate, link swap and 2-opt. On each iteration one of the three operators is chosen at random with equal probability. The purpose of this test is to illustrate the relative power of each operator and visually analyze the results.

The numerical results are shown in Table 3, and the visual results are shown in Figures 22 and 23. The results are formed from 50 runs of each operator until no further improvements was found. There are 3 reported values in these results, the best, average and the worst run. Initial solution length was 498 and the optimal length for this problem is 406.

Table 3. Comparison of different operators capabilities in dots-4400. The lengths of best, average, and worst run are provided.

Operator	Best	Average	Worst
Relocate	441	464	479
Link swap	435	454	476
2-opt	422	436	469
3-opt	420	424	439
Random mix	406	418	434
Rebuild (*K* 5–15)	406	409	427

The results are ordered from worst operator to best in this test. The numerical results show that the operators differ heavily from each other. Relocate and link swap are the weakest operators when deployed on their own. That result is not surprising as they are the simplest operators on paper. Additionally, relocate and link swap are intended to be used as part of random mix. The 2-opt operator gives more solid results overall while also being a straightforward operator in the

way it works. Those results are improved even further with 3-opt. Especially, the worst case between 2-opt and 3-opt improves drastically. The random mix provides surprisingly reliable results as it is ranked 2nd in this experiment. It also managed to find the optimal route in 6 out of the 50 runs. The studied operator, rebuild, managed to produce the best results in this test. Rebuild found the optimal path in 34 attempts out of 50. The *K* values ranged from 5 to 15. The different *K* value ranges are discussed in-depth in later Chapters.

The visual results provide insights on the abilities of each operator. It is important to note that the average result does not always align between the numerical and visual results. This happens because the average result is calculated from 50 runs. The value does not represent any individual run. In these cases, the closest actual run is shown in the visual results. When relocate and link swap are separated from the random mix search, they become weak. They both have a limited amount of moves they can do and especially struggle with crossovers. Link swap does relatively well when considering that all of its moves consider only the ends of the solution. Relocate would provide better results than link swap if the problem were larger as can be seen in the large-scale test.

The 2-opt operator can make a wider variety of improving moves which lead to better results compared to relocate and link swap. Along with other moves, 2-opt is guaranteed to break any crossovers which explains most of the difference seen between 2-opt and link swap. However, large amount of potential improvement is also left to be gained. The worst-case visualization of 2-opt illustrates that operating with only 2 links is too restrictive as the middle part of the solution is highly inefficient. The 3-opt operator considerably improves the situation as the worst-case for 3-opt is 30 units better, and the average case is better by 12 units. Interestingly, the best solution improves only by 2 units. This indicates that there is more variance in the 2-opt results, and this happens because 3-opt can escape from bad situations more often.

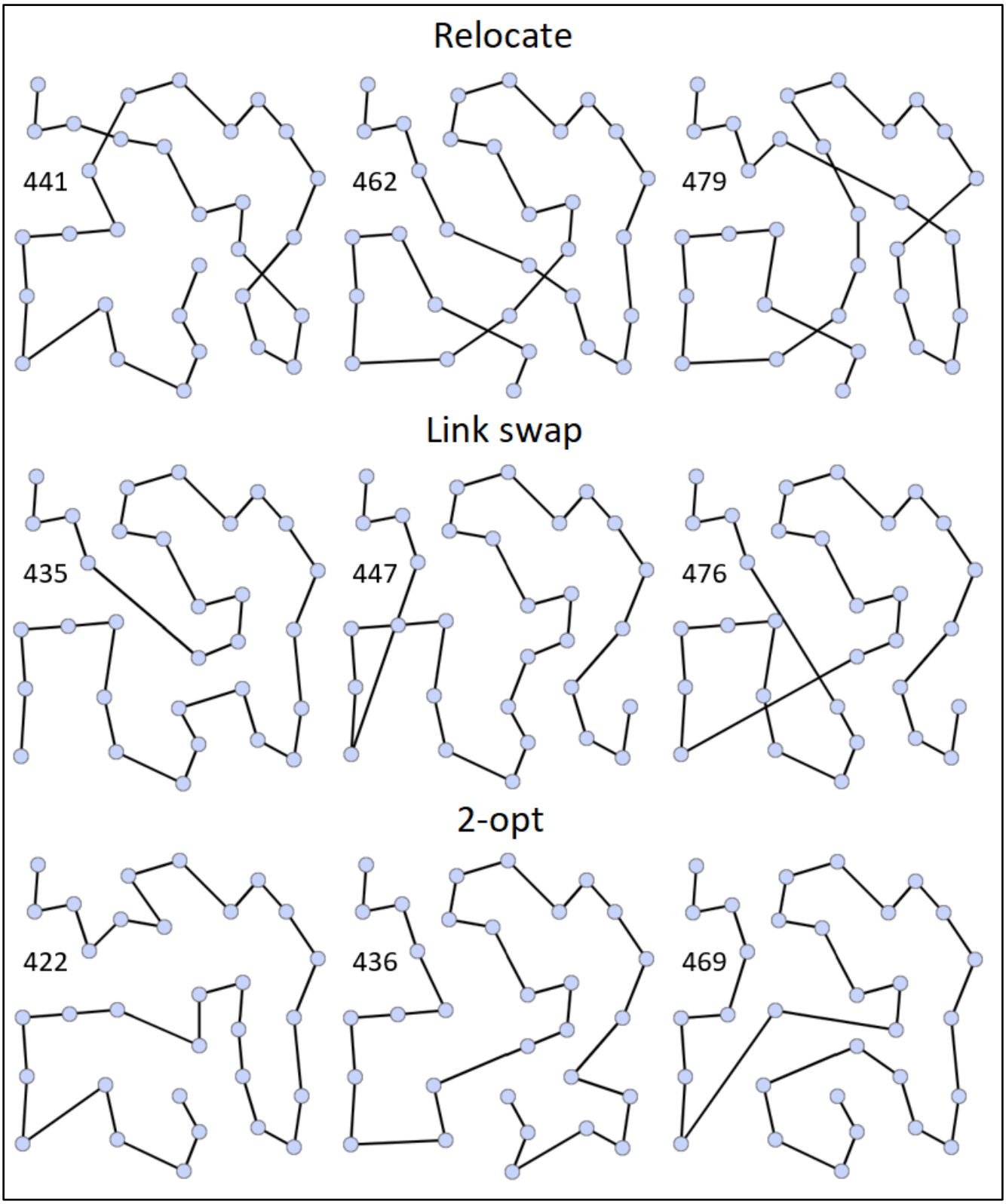

Figure 22. A visual comparison of relocate, link swap and 2-opt in dots-4400. The images of best, average, and worst run are provided.

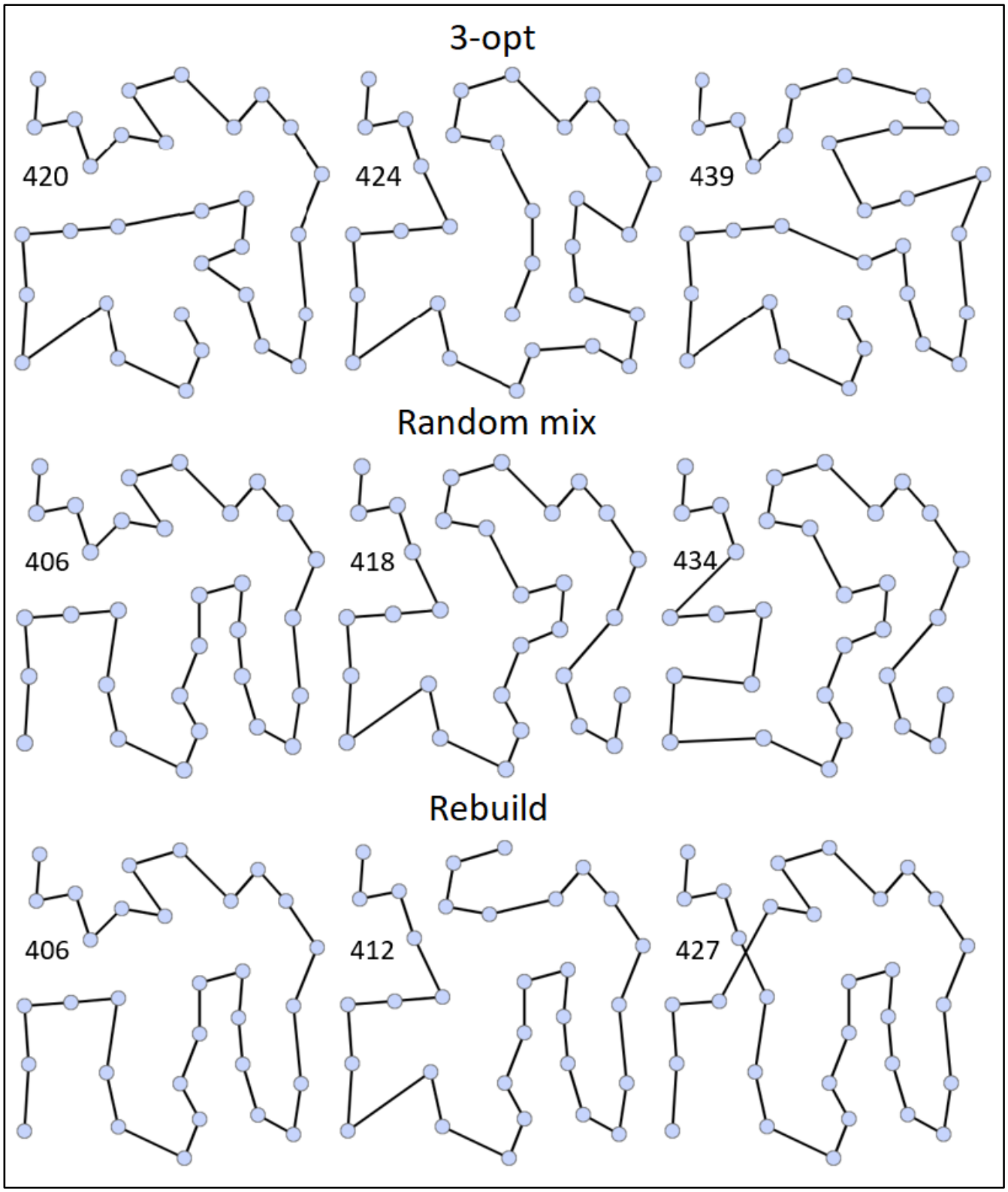

Figure 23. A visual comparison of 3-opt, random mix and rebuild in dots-4400. The images of best, average, and worst run are provided.

The visual results of random mix and rebuild reveal only few insights. The worst-case of rebuild shows again that rebuild has a clear problem with crossovers that can be fixed with a k-opt operator. The random mix and rebuild results show the most clearly that the successfulness of local searching in the small-scale depends on which moves the operators make in each step. If the operators make such moves that the overall path starts from top left corner and proceeds right, then a near optimal route is obtained most of the time. If the route proceeds downwards, then the optimal path is never found as the local operators are not capable of such large direction change. Therefore, repeating the whole local search multiple times is beneficial as a more efficient path is often found (Sengupta & al., 2019).

6.3 Rebuild combined with one operator on small data

Next, we consider the combination of rebuild and another operator. The idea was to evaluate which operator has the most synergy with rebuild. The combination was done by randomly selecting which operator is being used. To compensate the fact that iterations of rebuild are more time consuming than iterations of other operators, the selection was done so that there was 10% change for rebuild and 90% for the other operator. The results are compared to plain rebuild, 3-opt and random mix. This testing is done with all 6 Dots instances, so that more comprehensive results can be obtained. The results are presented in Figure 24.

The results include the outcomes of every instance individually, but the average section is the most important. The comparison operators rebuild, 3-opt and random mix rank themselves so that 3-opt is the worst, random mix places second and plain rebuild (*K* 5–15) achieves the best solutions with small margins. It follows the same pattern as the previous test.

Out of the operators that are combined with rebuild, relocate is the worst candidate. Although, the average result is still better than plain rebuild. The result is explainable with the fact that these operators are the most similar in the way of how they work. The second worst is 2-opt. This is an unanticipated result as 2-opt is a better operator than link swap on its own. The reason for

this can be that although 2-opt can break crossovers and assist the search in other ways, link swap can change the end points of the path effectively which rebuild benefits from. The ranking of these two operators would be the opposite if a larger dataset were considered. The 2-opt operator would have more opportunities for optimizing and link swap would not be as strong as it only considers moves done to the end points. The combination of rebuild and 3-opt intuitively achieves the best results. However, the margins are small when compared to 2-opt and link swap.

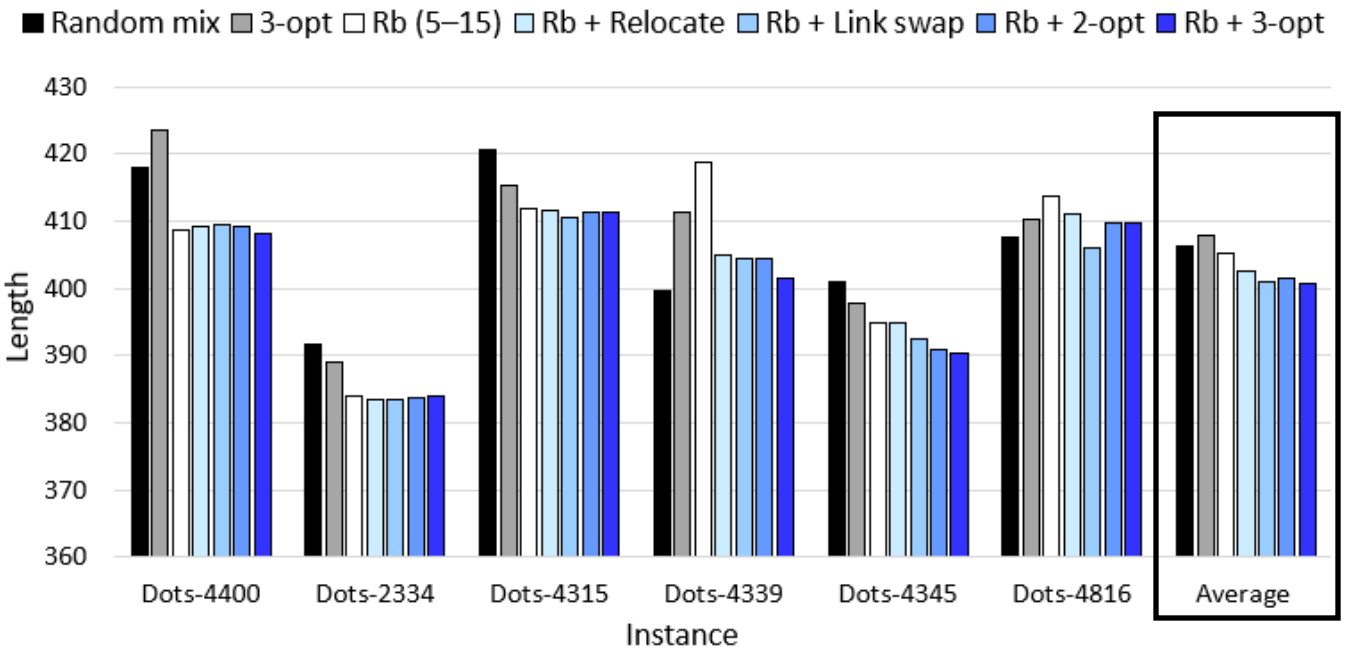

Figure 24. Results of testing different rebuild and operator combinations. Paths are compared to plain rebuild, 3-opt and random mix. 50 repeats each test case.

6.4 Different K values on small data

The testing of different *K* values is considered next. A small test on Dots-4400 with singular *K* values is analyzed first. Different *K* values ranging from 1 to 20 were used and the final length was collected after the solution stopped improving. The results of the test can be seen in Figure 25.

As can be seen from the figure, low *K* values are not powerful enough to provide good solutions in this case. It can be assessed that small *K* values are useless in all cases. The moves found with

small *K* values can also be found by larger *K* values, so it is not wise to spend iterations on trying to find improvements with small *K* values. Medium *K* values are quite useful, however, as can be seen in later tests. They can provide faster changes than large *K* values. Larger *K* values are needed for making more substantial changes to the path. With this TSP instance, rebuild needs a *K* value of at least around ten as this is where the lengths stabilize. Further *K* value increases only provide minor improvements to the solutions.

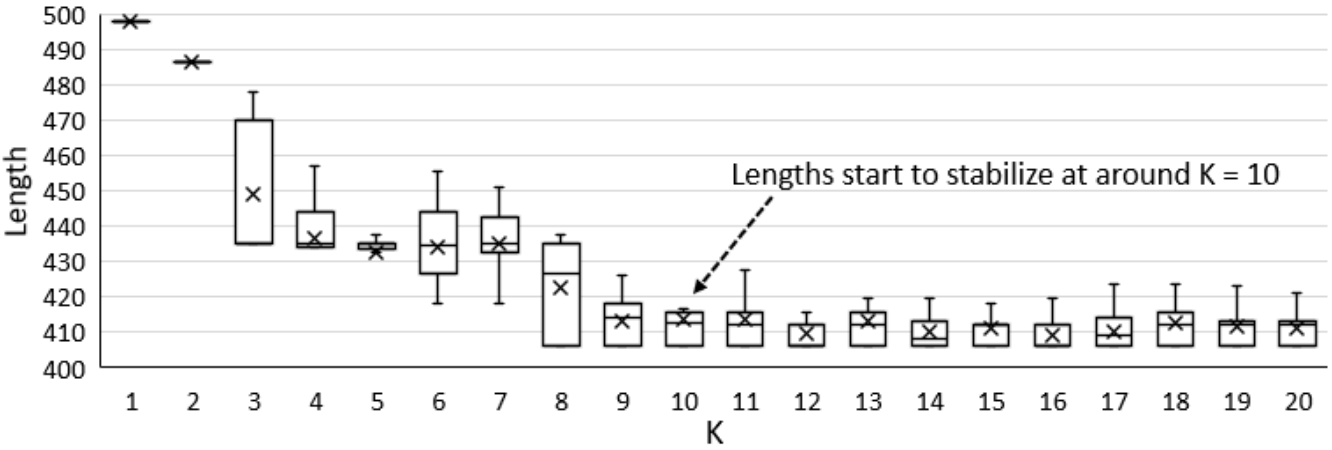

Figure 25. The optimization capabilities of each *K* value on Dots-4400. Each value is formed from a twenty run average.

The next test involves testing of different *K* value ranges with all six Dots instances. The results are compared to 3-opt and random mix again, and they are shown in Figure 26. The average results are the most important as the results in singular instances can vary in a major way. The datasets characteristics change the outcome.

The results clearly show that rebuild 5–10 lacks the power to produce good solutions as it produces worse solutions than 3-opt and random mix. It can be interpreted that using only the larger *K* values is beneficial in small TSP instances. Rebuild 15–20 and 10–20 provide the best results in this case. The reason may be that small *K* values might make small-scale changes to a solution that are irreversible even with large *K* values. It seems that large *K* values can make changes that are meaningful towards building an optimal solution. Numerically, rebuild 5–15 which was used in earlier test has a gap of 0.77% to rebuild 15–20. Rebuild 15–20 was the best performer overall. That leaves 3-opt and random mix with a gap of 1.52% and 1.35% respectively to rebuild 15–20.

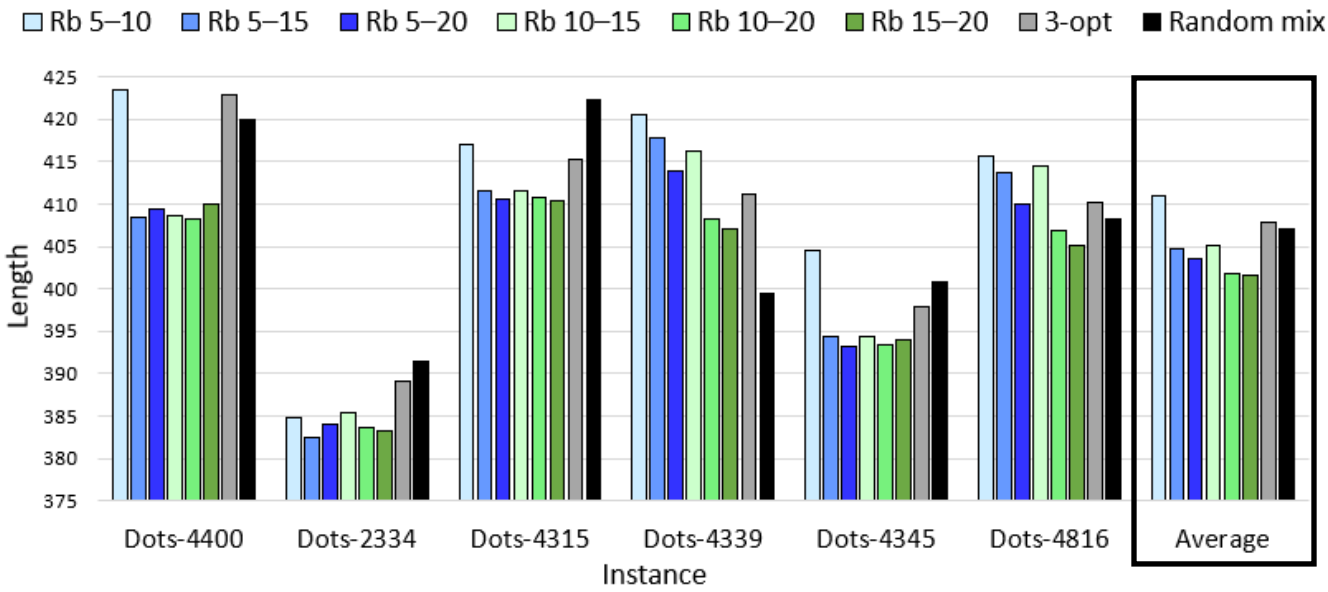

Figure 26. The testing of different *K* value ranges on six Dots instances. Twenty repeats each test case.

6.5 Operator efficiency on small data

The efficiency of different operators is considered next on Dots-4400. The test is done to different singular *K* values first, and two aspects of the search are considered, length and time taken. The results are shown in Figure 27. One aspect to note here is that iterations were capped to ten thousand in this particular test. This was done to prevent inconsistencies in the data. Large K values would occasionally find a very minor improvement to a solution late in the search, and that caused flawed results.

The result shows that as the *K* value increases, the length of the final path decreases as shown before. This has an inverse effect on the time taken. When the *K* value increases, the time taken also increases. Conclusions can be drawn on the efficiency of different *K* values. Small values of *K* (1–5) are fast but lack the power to do major changes. Medium values of *K* (6–15) provide a good balance of efficiency and good solutions. Large *K* values (15+) lead to inefficiency in this case as lower *K* values can provide the same results with less time. The large *K* values are needed for satisfactory results with different datasets. It should be noted that the results are presented on the millisecond level. They have more of a theoretical meaning instead of a practical one.

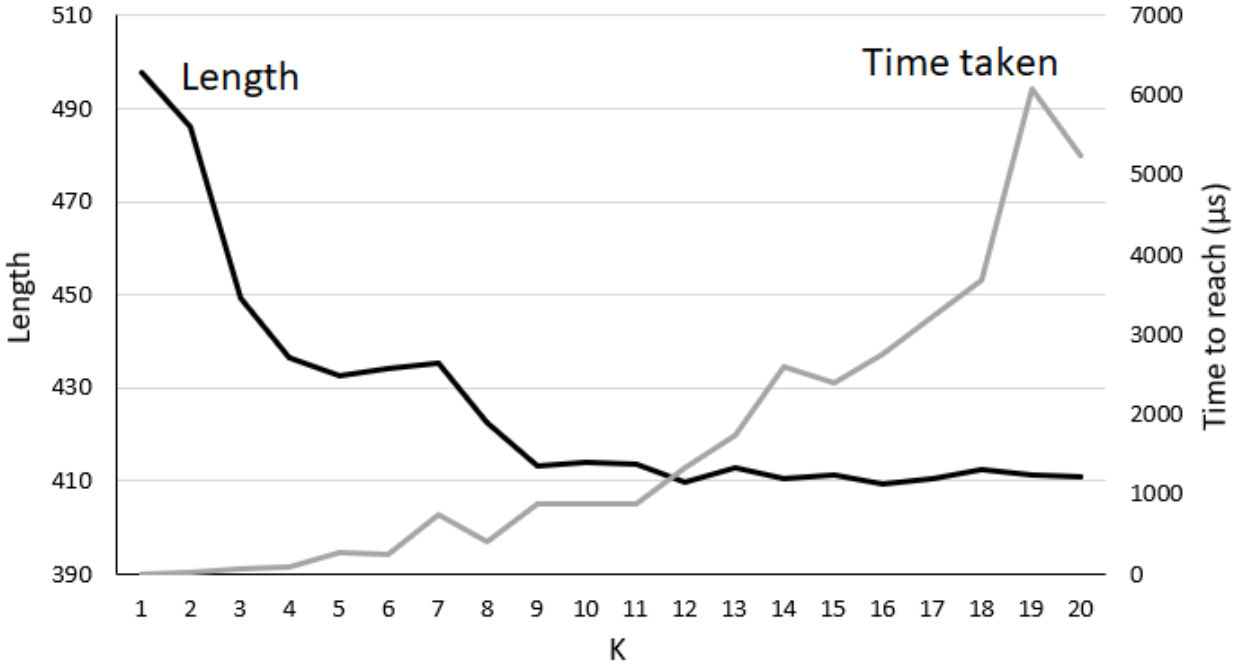

Figure 27. The comparison of different *K* values in terms of speed and solution quality. Twenty run average.

Next, the efficiency of different *K* value ranges is tested. The testing methodology is the same as before. The results are presented in Figure 28. This test produces a similar pattern to the last one. Small *K* values produce mediocre results with fast speed. Larger *K* values produce better results while compromising speed. With this dataset, rebuild 5–15 is sufficient enough to produce decent results. Larger *K* values should be used in a general instance because they will generate better solutions as seen in earlier testing.

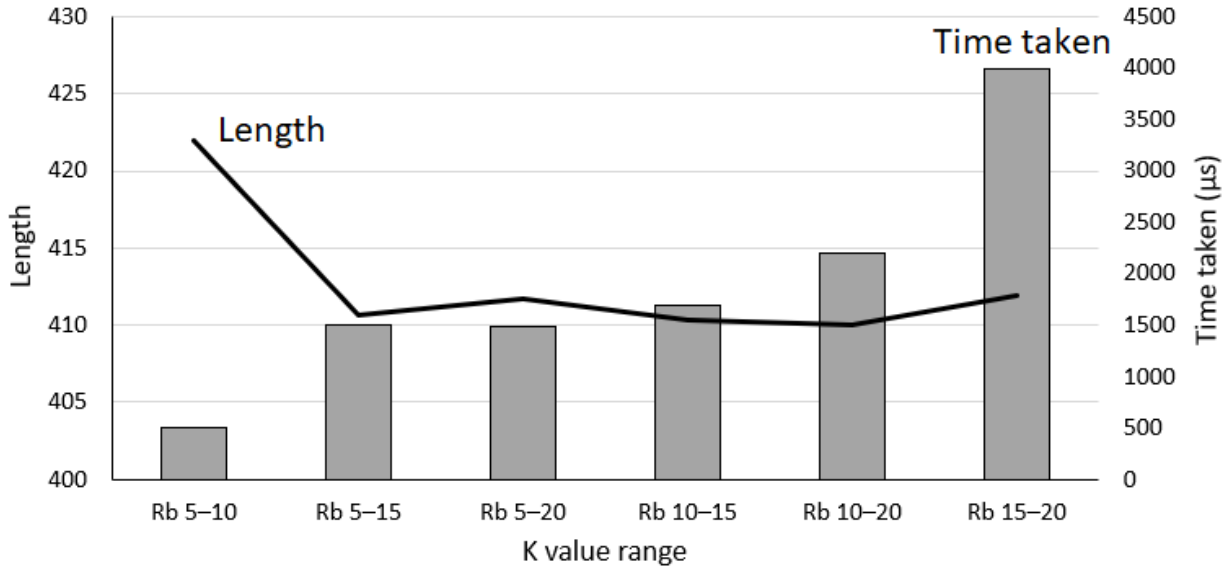

Figure 28. The comparison of different *K* value ranges in terms of speed and solution quality. Twenty run average.

The same testing was done to other operators and random mix to compare the results between all operator variants (Figure 29). Relocate, Link swap and 2-opt provide fast processing times with insufficient results on their own. When they are combined together as random mix, they start to produce relatively reliable results with fast processing times. This makes for an efficient combination of operators that slightly beats 3-opt in solution quality and is multiple times faster in small-scale TSP.

Finally, the best efficiency candidates are compared in Figure 30. The 3-opt operator alone is not the most suitable operator for small-scale TSP as it is in a disadvantage in this test. Random mix provides considerably faster results and rebuild 5–15 provides shorter paths with same speed. The best rebuild variant in terms of path length in earlier testing, rb 15–20, is also included in this chart as it provides best results with compromised speed. The best results do not show here as rebuild 5–15 was better in this Dots-4400 instance. A repeated random mix strategy was studied in Sengupta & al (2019), and it seems to be an effective way of solving small-scale TSP instances as this data suggests. Random mix can be repeated numerous times because of its processing time which makes it highly likely to find optimal or near-optimal solutions.

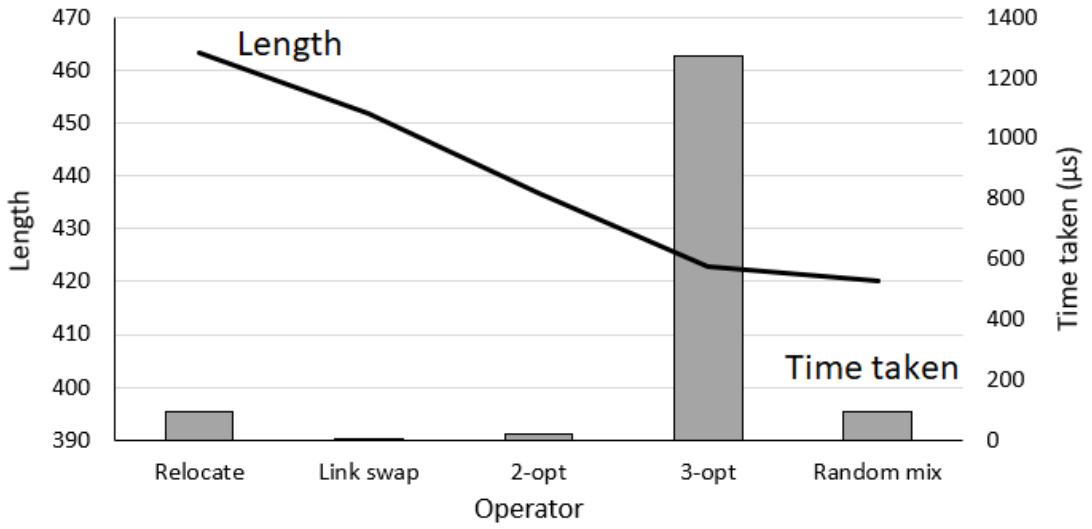

Figure 29. The comparison of different operators in terms of speed and solution quality. Twenty run average.

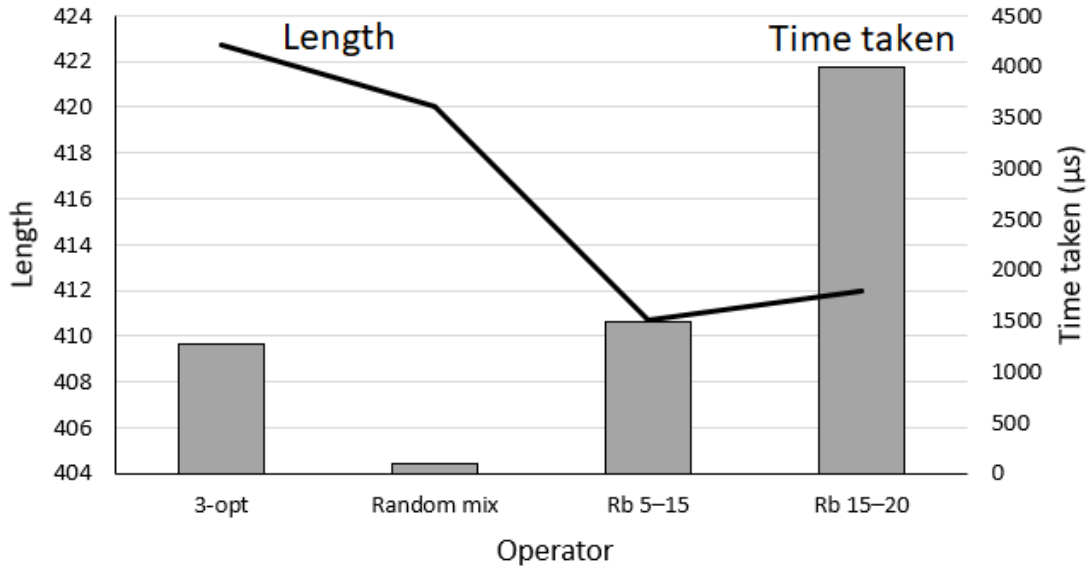

Figure 30. The comparison of the most relevant operators in terms of speed and solution quality. Twenty run average.

6.6 Different K values on large data

After testing operators on a small-scale TSP instance, a larger instance was considered. This test will measure each operator's scaling ability. The tests are performed on the Santa5000 instance with a starting length of 19 906 482. With large-scale TSP instances it is harder to dictate when the search has completely halted. This is why the tests are performed by running each test case for 30 seconds and then checking the result. This is enough time for the operator to have stopped improving the solution in most cases.

The first test compares different rebuild variations with singular *K* values similar to the small-scale testing. It is important to analyze if larger *K* values are needed in large-scale TSP instances. The results are shown in Figure 31. It can be seen that small *K* values fail to produce reliable results again. The outcomes start to stabilize at around $K = 10$ up to $K = 20$ where the trend starts to go upwards. A major drop in lengths can be seen at around $K = 25$. This is likely caused by an artifact in the dataset with this initialization. A problem spot in the tour is solved by the larger *K* values and this results in the large drop in lengths. When rebuild is combined with 3-opt later in the paper, we can confirm that this is the case. The trend from $K = 25$ upwards is towards worse results if the effect of the artifact is removed.

In theory, larger *K* values should always find better routes if enough time is given. The reason large *K* values create inefficiency is due to the reconstructing phase. The algorithm is not guaranteed to reconstruct the path in an intelligent way so if a large number of nodes are removed at once, there is an increasing chance that the reconstructed is not shorter than the original.

The same testing was done to different *K* value ranges. The results are presented in Figure 32. Here we can see again that *K* value ranges are better than singular values as rebuild 5–30, 10–30 and 15–30 beat the best singular *K* value (28) in this test. Other *K* value ranges would also beat the singular *K* value in another instance but that does not occur in this case. The artifact in the dataset causes this. The artifact makes the comparison of different *K* value ranges hard in this

instance. These results show that rebuild benefits from having large scale of *K* values included in the search as rebuild 15–30 is the third best after rebuild 5–30 and 10–30.

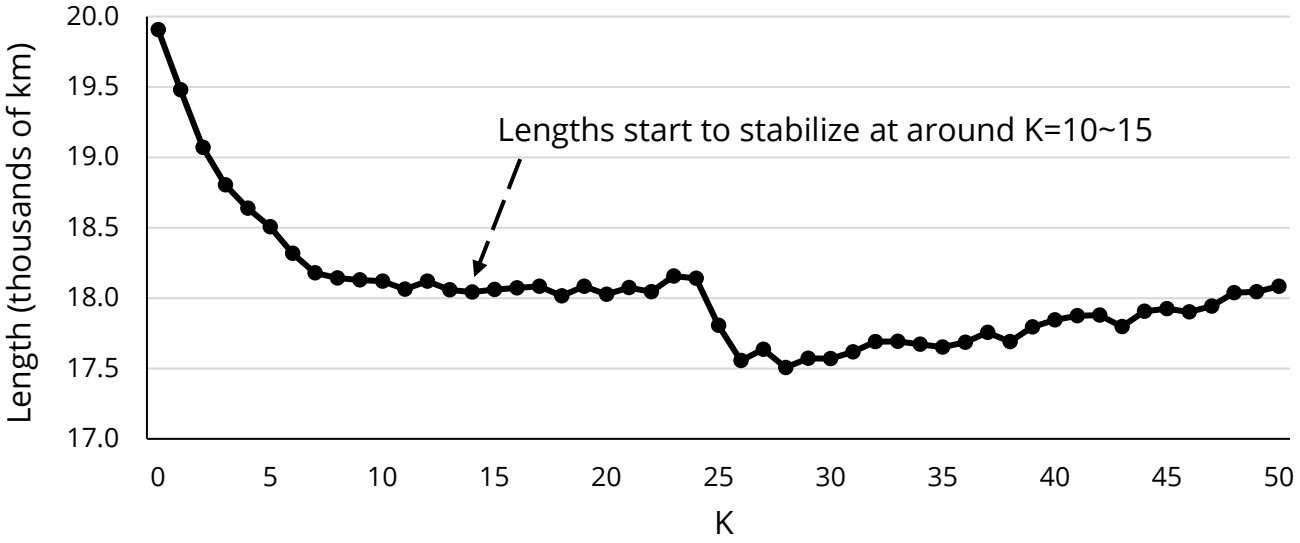

Figure 31. The optimization capabilities of each *K* value on Santa5000. Each value is formed from a ten run average.

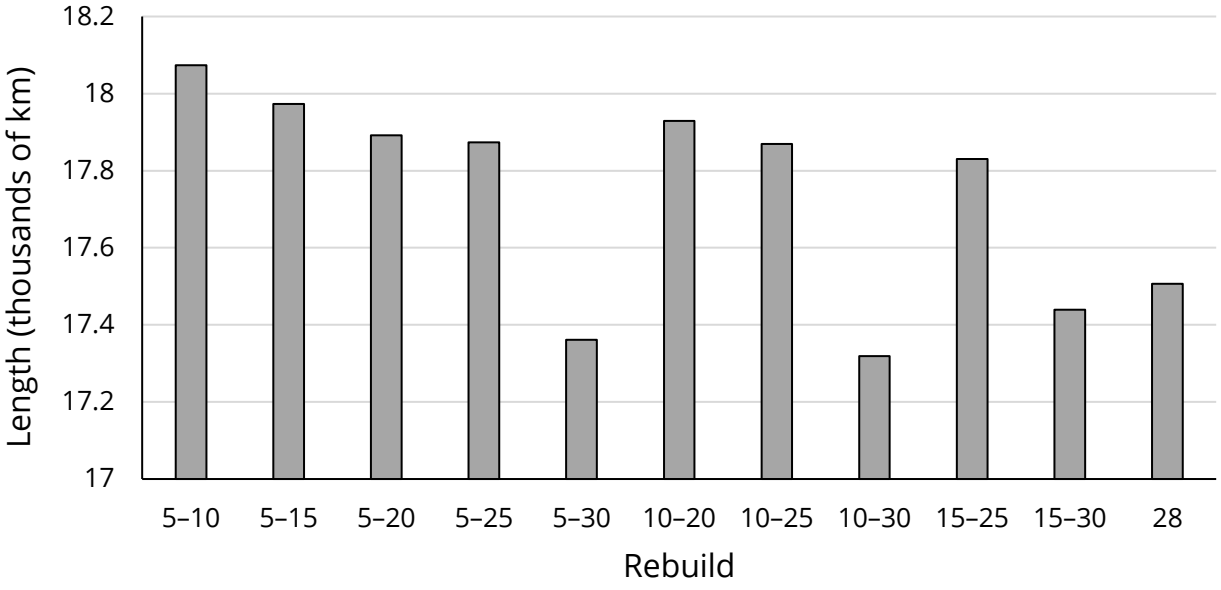

Figure 32. The testing of different *K* value ranges on Santa5000 instance. Ten repeats each test case. Singular *K* value 28 is used as an example.

Lastly, the testing was done to the other operators presented in this thesis. The results are presented in Figure 33. As also seen before, link swap and relocate are effective in random mix

search but do not provide satisfactory results alone. The 2-opt and 3-opt operators provide comparable results compared to each other but 3-opt is capable of some additional fine-tuning. Random mix is again proficient at providing decent results as the path length is approximately 500 000 units shorter compared to 3-opt. The weaknesses of rebuild start to appear on this large-scale test. The best rebuild variation has the same result as 3-opt, and thus loses to random mix.

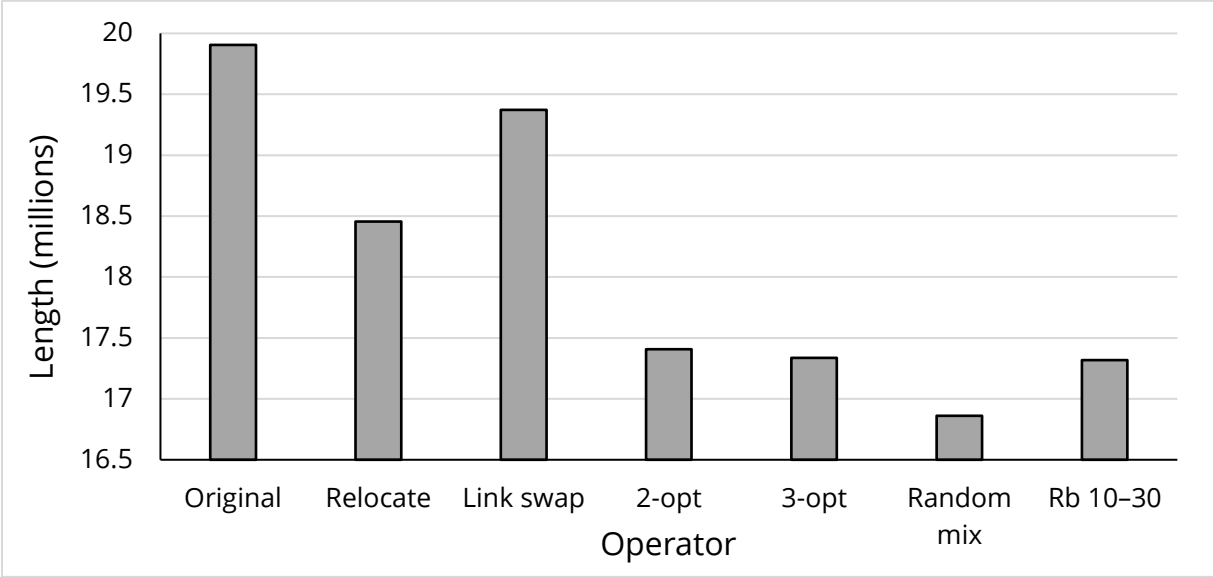

Figure 33. The testing of different operators on Santa5000. Ten repeats each test case.

6.7 Rebuild combined with 3-opt

As seen in the previous test, rebuild's optimizing capabilities suffer in large-scale TSP instances. Rebuild especially struggles with crossovers so that could be easily fixed by combining rebuild with any k-opt class operator. The 2-opt operator would be another suitable candidate but 3-opt was chosen for the purposes of this thesis. The testing was done so that there was a certain chance for an iteration to be using rebuild. An important thing to note here is that iterations of rebuild are much slower compared to 3-opt. With this five thousand point dataset, rebuild 5–30 iterations were approximately one thousand times slower. This means that if a combination has a 1% chance for rebuild and a 99% chance for 3-opt, the actual time used by each operator would

be roughly 90% for rebuild and 10% for 3-opt. The actual percentages depend on the rebuild variation (*K* values).

The results of running different rebuild variants with varying chance for a rebuild iteration is presented in Table 4. The general result is that only a small portion of 3-opt iterations are needed timewise to fix the problems rebuild has. The optimal chance for a rebuild iteration is between 0.1% and 1%, which roughly equates to using rebuild for 50% to 90% of the processing time. Another observation can be made that larger *K* value ranges provide better results as rebuild 5–20, 5–30 and 10–25 got all of the top five results. Rebuild was beaten by random mix in the last test, but the best rebuild and 3-opt combinations beat the random mix result by approximately 300 000 units on average.

Table 4. The results of testing different rebuild and 3-opt combinations with varying *K* value ranges and different proportions of rebuild and 3-opt. Results given in gap values compared to the best.

	5-10	**5-15**	**5-20**	**5-25**	**5-30**	**10-20**	**10-25**	**10-30**	**15-25**	**15-30**
0.05% rb	1.21 %	0.52 %	0.42 %	0.38 %	0.36 %	0.54 %	0.63 %	0.45 %	0.70 %	0.73 %
0.1% rb	1.09 %	0.43 %	0.28 %	0.21 %	0.19 %	0.36 %	0.21 %	0.24 %	0.58 %	0.50 %
0.5% rb	0.75 %	0.64 %	0.05 %	0.31 %	0.00 %	0.45 %	0.17 %	0.27 %	0.48 %	0.35 %
1% rb	0.83 %	0.19 %	0.08 %	0.48 %	0.24 %	0.38 %	0.15 %	0.28 %	0.61 %	0.75 %
5% rb	0.86 %	0.95 %	1.01 %	0.64 %	0.66 %	0.95 %	0.92 %	1.21 %	1.62 %	1.57 %
10% rb	1.89 %	1.22 %	1.25 %	1.16 %	1.42 %	1.50 %	1.52 %	1.62 %	2.32 %	2.37 %
20% rb	2.43 %	1.76 %	1.86 %	2.33 %	2.08 %	2.35 %	2.71 %	2.12 %	2.74 %	3.17 %

6.8 Operator efficiency on large data

Apart from testing only the result quality after 30 seconds of optimizing, the optimizing speed is also important. The speed testing will reveal when the operators hit their local minimum spots and which operators still have potential for further improvements. The results of testing five most relevant operators and operator combinations is presented in Figure 34.

The final result between operators at 30 seconds is still the same but the speed of improvement is the observable feature here. Plain rebuild 5–30 and 3-opt have the worst results in both speed and solution quality. Rebuild 5–30 beat the 3-opt in the 30 seconds test before but because of additional overhead created by the nature of this test, rebuild would need a bit more time to reach the 3-opt level of result. The 3-opt operator is faster at optimizing than rebuild 5–30 but rebuild achieves better results if enough time is given. Random mix has the fastest optimization speed initially in this test but falls behind the combination of rebuild and 3-opt after few seconds of optimizing as the operators fail to find new solutions frequently. Rebuild 5–20 with 3-opt and rebuild 5–30 with 3-opt have comparable results. The differences are that rebuild 5–30 with 3-opt has better results in the beginning and at the end. Rebuild 5–30 is capable of producing shorter paths if enough time is given but rebuild 5–20 finds beneficial chances faster in the middle of the search.

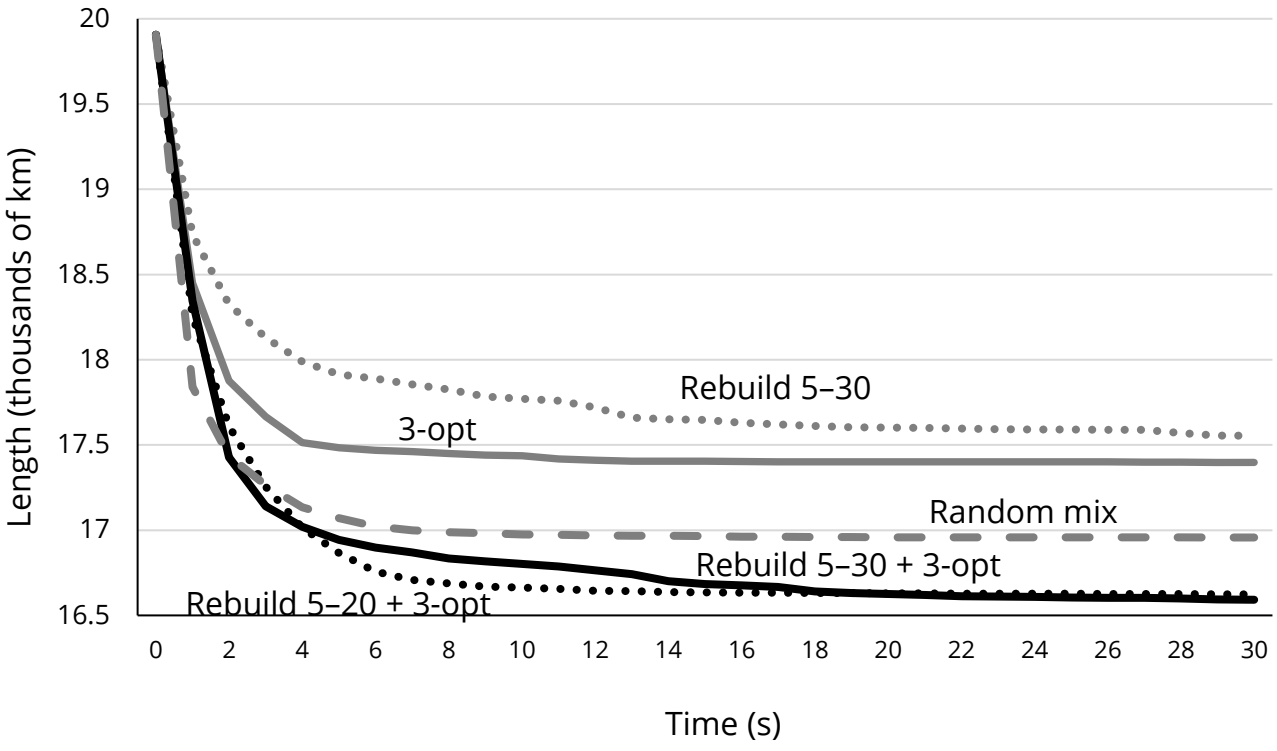

Figure 34. The visualization of each operator's optimization speed on Santa5000. Each datapoint is formed as a ten run average.

An observation can be made that combining operators with complementary search spaces makes optimization progress faster. The idea is not the most intuitive one as one might think the opposite would be true. This happens because the operator combination has more potential for optimization which makes iterations successful with higher probability.

6.9 The final benchmark

The final tests were done with a collection of ten different TSP instances, and it was conducted only to the most relevant operators to see how they perform compared to each other in a larger variety of instances. These tests were performed so that more definitive results could be extracted. The idea was to have a variety of TSP instances with different sizes and properties. The collection includes seven different TSPLIB instances (Reinelt, 1991), two modified Santa competition instances (Mariescu-Istodor & Fränti, 2021) and one clustering dataset (Fränti & Sieranoja, 2018). The TSPLIB instances are classical instances for TSP benchmarking. The Santa dataset was selected because it contains a good variety of sparse and dense areas. Lastly, the clustering dataset was selected because it seemed like a fitting addition to the test collection. The s4 variant of the clustering datasets introduces major overlap between the clusters. All datasets were initialized with random start nearest neighbor approach. The distance is calculated as Euclidean distance in all cases. The gr666 dataset originally uses the great-circle distance but Euclidean distance was used in this case to keep the distance function same across all tests. The details of each instance are given in Table 5.

The collection of all results is presented in Table 6. The relative power of each operator can be calculated when the operators are compared to each other. Rebuild 5–30 and 3-opt provide the worst results. When these two are compared, rebuild 5–30 is 0.26% worse than 3-opt on average. They provide comparable results on average but some differences can be seen in individual instances. Random mix provides better results than rebuild 5–30 and 3-opt. The paths provided by random mix are 1.62% shorter on average when compared to 3-opt.

Table 5. An overview of the datasets used in the final benchmark.

Dataset	*N*	Initial length	Origin
pr76	76	135,343	TSPLIB
u159	159	50,523	TSPLIB
pcb442	442	60,025	TSPLIB
d657	657	60,424	TSPLIB
gr666	666	4,374	TSPLIB
santa1000	1000	9,999,791	Santa
u1060	1060	278,126	TSPLIB
pr2392	2392	475,201	TSPLIB
santa5000	5000	19,906,482	Santa
s4	5000	37,484,958	Clustering

Table 6. The results of the final benchmark. Results given in gap values.

	Rb 5–30	3-opt	Rb 5–20 + 3-opt	Rb 5–30 + 3-opt	Random mix
pr76	2.03 %	5.55 %	0.19 %	0.00 %	2.51 %
u159	0.54 %	2.73 %	2.78 %	1.49 %	0.00 %
pcb442	0.99 %	1.86 %	0.19 %	0.00 %	1.54 %
d657	2.23 %	2.63 %	0.00 %	0.32 %	2.74 %
gr666	5.52 %	2.39 %	0.00 %	0.45 %	1.19 %
santa1000	7.37 %	2.78 %	0.60 %	0.00 %	2.45 %
u1060	4.42 %	3.68 %	0.21 %	0.00 %	2.41 %
pr2392	4.51 %	5.55 %	0.41 %	0.00 %	2.57 %
santa5000	5.46 %	4.43 %	0.00 %	0.34 %	1.44 %
s4	4.80 %	3.72 %	0.00 %	0.54 %	1.53 %
Average gap	**3.79 %**	**3.53 %**	**0.44 %**	**0.31 %**	**1.84 %**

The best results are obtained by the combinations of rebuild and 3-opt. When the two combinations are compared rebuild 5–30 and 3-opt has a slight advantage over rebuild 5–20 and 3-opt. Rebuild 5–20 and 3-opt is 0.12% worse than rebuild 5–30 and 3-opt. If the processing time were more than 10 seconds, a slightly larger gap would be seen. Rebuild 5–30 and 3-opt combination produces 3.09% shorter paths than plain 3-opt and 1.49% shorter paths than random mix.

7 Conclusions

In this thesis, traveling salesman problem and common approaches for solving it were studied. The operators of the local search approach were analyzed in detail. Rebuild operator was constructed and explained comprehensively. The experiments included the comparison of common existing operators and rebuild.

Rebuild produced promising results especially in small-scale TSP instances in both efficiency and solution quality. The main competitors are random mix and 3-opt which provided worse results than the rebuild operator. In large-scale TSP, rebuild did not provide sufficient results on its own. However, a combination with a small amount of k-opt operator produced promising results also in large-scale TSP. The final benchmark between few chosen operators revealed that a combination of rebuild 5–30 and 3-opt provided 1.5% shorter paths than random mix and 3.1% shorter paths than 3-opt.

Some limitations can be seen in the experiment part of the study. A portion of the results were provided as qualitative analysis instead of quantitative. However, quantitative analysis was provided in the sections where it was necessary. Quantitative analysis could have been more extensive with more diverse datasets. Additionally, the datasets were initialized only with one initialization only. This approach tests the operator's flexibility but also randomizing the initial solution each run has its merits.

There are numerous further research options. First, rebuild should be developed further using some of the features found in other similar methods discussed in Chapter 5. Sequential removal should not be the only removal method. Especially, a removal option that removes paths in a given area instead of a given path should be considered as rebuild struggles with breaking crossovers. For example, it could work by removing K closest neighbors from a selected node. This was the method used in Chapter 5.

Rebuild should also be tested in a very large-scale TSP instance and compared against existing methods. The Santa competition dataset would be a good choice. Rebuild's capabilities should be analyzed against 4-opt and 5-opt as well.

References

Applegate, D., Bixby, R., Chvátal, V., Cook, W. (2007) *The Traveling Salesman Problem*. Princeton University Press.

Braekers, K., Ramaekers, K., Nieuwenhuyse, I. (2016) The Vehicle Routing Problem: State of the Art Classification and Review. *Comput. Ind. Eng.* **99**, 300–313.

Nilsson, C. (2003). Heuristics for the Traveling Salesman Problem.

Croes, G. (1958) A Method for Solving Traveling-Salesman Problems. *Operations Res.* **6**(6)*,* 791–812.

Cristian, A., Marshall, L., Negrea, M., Stoichescu, F., Cao, P., Menache, I. (2019) Multi-itinerary optimization as cloud service (Industrial Paper). *In Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 279–288.

Cristian, A., Marshall, L., Negrea, M., Stoichescu, F., Cao, P., Menache, I. (2021) Multi-itinerary optimization as cloud service. *Communications of the ACM,* **64**(11). 121–129.

Dees, W., Karger, P. (1982) Automated Rip-Up and Reroute Techniques. *19th Design Automation Conference*, 432–439.

Fränti, P., Sieranoja, S. (2018) K-means properties on six clustering benchmark datasets. *Applied Intelligence*, **48**(12), 4743–4759.

Fränti, P., Nenonen, T., Yuan, M. (2021) Converting MST to TSP path by branch elimination. *Applied Sciences*, **11**(177), 1–17.

Gendreau, M., Hertz, A., Laporte, G. (1992) New insertion and postoptimization procedures for the traveling salesman problem. *Oper. Res.*, **40**, 1086–1094.

Glover, F. (1989) Tabu Search-Part I. *ORSA Journal on Computing* **1**(3), 190–206.

Helsgaun, K. (2006) An Effective Implementation of K-opt Moves for the Lin-Kernighan TSP Heuristic. Roskilde Universitet. Computer Science. Computer Science Research Report.

Helsgaun, K. (2017) An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems: Technical report. Roskilde Universitet. http://www.akira.ruc.dk/~keld/research/LKH-3/LKH-3_REPORT.pdf

Johnson, D., McGeoch, L. (1997) The traveling salesman problem: A case study in local optimization. *Local Search Comb. Optim.*, 215–310.

Lin, S. (1965) Computer solutions of the traveling salesman problem, *Bell Syst. Tech. J.* **44**(10), 2245–2269.

Lin, S., Kernighan, B. (1973) An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Res.* **21**(2), 498–516.

Mariescu-Istodor, R., Fränti, P. (2021) Solving the Large-Scale TSP Problem in 1 h: Santa Claus Challenge 2020. *Frontiers in Robotics and AI* 8:689908.

Pisinger, D., Røpke, S. (2007) A general heuristic for vehicle routing problems. *Computers & Operations Research*, **34**(8), 2403–2435.

Rutenbar, R. (1989) Simulated annealing algorithms: an overview. *IEEE Circuits and Devices Magazine* **5**(1), 19–26.

Reinelt, G. (1991) TSPLIB – A Traveling Salesman Problem Library. *ORSA Journal on Computing* **3**(4), 376–384.

Schrimpf, G., Schneider, J., Stamm-Wilbrandt, H. and Dueck, G., 2000. Record breaking optimization results using the ruin and recreate principle. Journal of Computational Physics, 159(2), 139–171.

Sengupta, L., Mariescu-Istodor, R., Fränti, P. (2019) Which Local Search Operator Works Best for the Open-Loop TSP? *Appl. Sci.* **9**(19)*, 3985*.

Strutz, T. (2021) Travelling Santa Problem: Optimization of a Million-Households Tour within one Hour, *Frontiers in Robotics and AI* 8:652417.

TSP (2022) World TSP. WWW page. https://www.math.uwaterloo.ca/tsp/world/ (21.2.2022).

Weru, L. (2022) 11 Animated Algorithms for the Traveling Salesman Problem. Stem Lounge. https://stemlounge.com/animated-algorithms-for-the-traveling-salesman-problem/ (24.3.2022)