University of Lancaster

Finding the cheapest flight connection between airports

Yaroslav Pylyavskyy

32083008

Supervisor

Ahmed Kheiri

This dissertation is submitted to the Department of Management Science and the Board of Examiners of Lancaster University in partial fulfilment of the requirements for the degree of Master of Science in Business Analytics. I hereby attest that this dissertation is entirely my own work and that all sources of information have been fully referenced.

September 2019

Abstract

Kiwi.com proposed a real-world NP-hard optimisation problem with a focus on air travelling services, determining the cheapest connection between specific areas. Despite some similarities with the classical TSP problem, more complexity is involved that makes the problem unique. It is *Time-dependent* with *Time-Windows*, *Asymmetric* and *Generalised* (i.e. it involves areas that contain sets of cities from which exactly one is visited). In addition to this, infeasibility adds more complexity to the problem since there are no flights available between specific points in the network for certain days. While solving such computationally difficult problems, exact methods often fail, particularly when the problem instance size increases; Then alternative approaches, such as heuristics, are preferred in problem solving. In this study, Kiwi.com problem is solved in Python 3.7 by the implementation of 6 different Hyper-Heuristic methods, namely SR-IE, SR-GD, RD, RPD, RP-IE, and RL-OI. These methods are compared and RL-OI is found to have the best performance. In addition, 4 extra Hyper-Heuristic methods with different sets of 'Guided' low level heuristics are used to speed up significantly the feasibility of the initial solutions. The empirical results show the success of the Hyper-Heuristic methods, which use a set of four low level heuristics (Swap, Insert, Reverse, and Change Airport), on most of the released instances. The most significant achievement of this study was that when RL-OI run without time restrictions, it managed to find better solutions than the best known in 5 problem instances.

Acknowledgements

I would like to express my sincere gratitude for the successful completion of this dissertation to my supervisor Ahmed Kheiri. His consistent support and guidance have been invaluable throughout this dissertation and helped me to exceed my capabilities. He introduced me to the world of Hyper-Heuristics and stimulated my inspiration. I would also like to thank my friends, Vasilis Goumas and Periklis Gatsoulis, for supporting me and offering me a great environment to study.

Contents

List of Figures

List of Tables

Abbreviations

ACO	Ant Colony Optimisation
AM	All Moves
CF	Choice Function
DRD	Domincance-based and Random Descent hyper-heuristic
EMC	Exponential Monte Carlo
FTP	Flying Tourist Problem
GD	Great Deluge
GR	Greedy
IE	Improving and Equal
INMAX	Maximum index of the infeasible connections
INMIN	Minimum index of the infeasible connections
INR	A random number within the range of INMIN and INMAX
LS	Local Search
MC	Exponential Monte Carlo with Counter
MCF	Modified Choice Function
MWW	Mann-Whitney-Wilcoxon statistical test
NN	Nearest Neighbor
NP	Non-Deterministic Polynomial-Time
OI	Only Improving
PSO	Particle Swarm Optimisation
RD	Random Descent
RP	Random Permutation
RPD	Random Permutation Descent

RRT	Record-to-Record Travel
RL	Reinforcement Learning
SA	Simulated Annealing
SR	Simple Random
SSHH	Sequence-based Selection Hyper-heuristic
TS	Tabu Search
TSP	Travelling Salesman Problem

This dissertation is dedicated to my beloved parents, who have been always by my side in nice and difficult times.

Chapter 1

Introduction

1.1 Background

Over the last few decades, the airline industry has experienced a vast growth and air travel is considered by far the best option for long distance journeys. Every year more than 30 million flights are scheduled worldwide and this number keeps increasing. Even a simple journey between two cities in a given day might have thousands flight combinations. For instance, Figure 1.1 displays 10,000 paths for a single journey from San Francisco to Boston arriving at the same day. In fact, the same journey has around 30,000 flight combinations in total [1]. With such a complex air network, it is extremely hard for air travellers to find the cheapest connection possible between two airports, and this becomes even more challenging for a multi-city journey. As a result, several online travel agencies have developed search engines to provide cheap flights for travellers, such as Kiwi.com, Scyscanner.net, Expedia.co.uk, Sabre.com, Priceline.com, Orbitz.com, and many others. Some of these agencies in their attempt to improve their customer service have launched different challenges and projects, which have drawn the attention of many researchers from optimisation and computer science fields of studies. OpenFlights.org is an online searching tool associated with air travel, which launched *The Air-Traveling Salesman* project [2]. Similarly, Kiwi.com ran the *Travelling Salesman Challenge* in 2017, which contributed to the development of the current algorithm kiwi.com uses, called *NOMAD* [3]. In 2018, Kiwi.com proposed the *Travelling Salesman Challenge 2.0* which is the subject of this study. The *Travelling Salesman Challenge 2.0* is a real-world NP-hard optimisation problem determining the cheapest connection between specific areas. It is a modified version of the ordinary TSP and it is described as follows: Given a

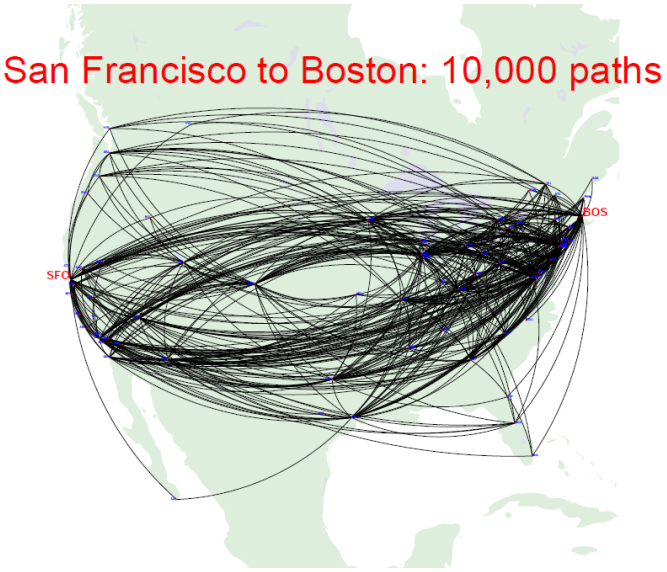

FIGURE 1.1: A map showing 10,000 paths from San Francisco to Boston arriving on the same day taken from [1]

set of N areas, the airports in each area, the starting point (the staring airport), the costs of travelling between those airports and the corresponding flight scheduled days, the goal is to find the cheapest possible route that visits each area exactly once by landing on one airport in each area, starting and ending at a given starting area. Even though the problem has some similarities with the classical Travelling Salesman Problem (TSP), there is more complexity involved that makes the problem unique. It is *Time-dependent* with *Time-Windows*, *Asymmetric* and *Generalised* (i.e. involves areas that contain sets of cities from which exactly one is visited). In addition to this, infeasibility adds more complexity to the problem since there are no flights available between specific points in the network for certain days. Hence, previously developed algorithms for TSP are not suitable for this problem and the need for new algorithms arises. Combinatorial problems, such as this, are of a great interest for scientists and researchers as they represent many real-life problems. Such problems are solved with exact methods, heuristic methods or by combining exact and heuristic methods together. Exact methods provide optimal solutions but for large size problems they are inefficient in terms of time. Therefore, heuristic methods are used to improve computational time efficiency and provide decent or near optimal solutions [4]. In this study, Kiwi.com problem is solved by means of Hyper-Heuristics.

1.2 Research Objectives

The objectives of this dissertation are:

1. Conduction of a thorough literature review on TSP Variants, Optimisation methods in Air Travel, Hyper-Heuristics in general, and on Hyper-Heuristic techniques used in this study.
2. Optimisation of Kiwi.com problem by means of Hyper-Heuristic methods, namely SR-IE, SR-GD, RD, RPD, RP-IE, and RL-OI.

1.3 Academic Publications

- Maab Alrasheed, Wafaa Mohammed, Yaroslav Pylyavskyy and Ahmed Kheiri (In Press) Local search heuristic for the optimisation of flight connections. International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)
- Yaroslav Pylyavskyy and Ahmed Kheiri (In Preperation) A Reinforcment Learning Hyper-Heuristic algorithm for the optimisation of flight connections.

1.4 Dissertation Structure

The thesis is structured as follows: In Section 2 the literature review regarding the variations of TSP, Optimisation methods in Air Travel, and Hyper-Heuristics is presented. Section 3 describes the Kiwi.com problem and the solution criteria in details. In Section 4 the construction of the algorithm, and the Hyper-Heuristic methods implemented are discussed. The implemented experiments and the obtained results are presented in Section 5. Finally, the conclusion to the results and suggested future study are presented in Section 6.

Chapter 2

Literature Review

2.1 Variations of TSP

TSP is a well known problem in the field of computer science and operations research. It was firstly defined by the Irish mathematician William Rowan Hamilton and by the British mathematician Thomas Kirkman in the 19th century. Despite being an old problem, it is still one of the most challenging problems in operations research field. The definition is as follows:

"Let $G = (V, A)$ be a graph where V is a set of n vertices. A is a set of arcs or edges, and let $C : (C_{ij})$ be a distance (or cost) matrix associated with A. The TSP consists of determining a minimum distance circuit passing through each vertex once and only once. Such a circuit is known as a tour or Hamiltonian circuit (or cycle). In several applications, C can also be interpreted as a cost or travel time matrix." [5]

TSP is an NP-hard problem and has been studied by many researchers due to its various applications in real-world problems. Some of these problems are; Computer Wiring [5] [6], Dashboard Design [5], Drilling of circuit boards [6], Hole Punching [5], Job Sequencing [5], Mask plotting in PCB production [6], Overhauling gas turbine engines [6], Vehicle Routing [6], VLSI circuits [6], Wallpapaer Cutting [5], Warehouse Automation System [6], and X-ray Crystallography [5] [6]. Over the last few decades, many variations of TSP have been created in order to fit and tackle more real-world problems. All the variants of the TSP are NP-hard as well. Some of them are [7]:

- General: The distance or cost is arbitrarily assigned between cities.
- Metric: The distance or cost d is metric and satisfies the triangle inequality; $\forall x, y, z \in X, d(x, z) \leq d(x, y) + d(y, z)$.
- Symmetric TSP (STSP): The cost of travelling from city i to city j is the same as travelling from city j to city i, $c_{ij} = c_{ji}$.
- Asymmetric TSP (ATSP): The cost of travelling from city i to city j is not the same as travelling from city j to city i, $c_{ij} \neq c_{ji}$.
- TSP with multiple visits (TSPM): It is allowed to visit cities more than once.
- Maximum benefit TSP (MBTSP): Salesman derives some benefit from visiting the cities.
- Prize collecting TSP (PCTSP): Salesman gets a prize for every visited city and pays a penalty for every city not visited.
- Multiple TSP: Instead of only one salesman, multiple salesmen are allowed.
- Open tour TSP: The salesman does not have to end the tour where it started.
- Time-Dependent TSP (TD-TSP): The travel cost depends on distance and the day of travel [8].
- Generalised TSP (GTSP): Cities are divided into clusters and salesman visits exactly one city of each cluster [9] . This variant is also known as Equality Generalised TSP (EGTSP) [10].
- Blind TSP (BTSP): Any city is allowed as a starting point for the tour [11].
- TSP with Neighbourhoods (TSPN): Salesman must visit a number of clients who are located in regions (or neighbourhoods) instead of cities. Each neighbourhood represents a space in which the client is willing to travel in order to meet the salesman [12].
- TSP with Time-Windows (TSPTW): Salesman must visit each city within a specified time. Failing to do so, results in an infeasible solution [13].

Numerous of algorithms have been developed for solving TSP. Such algorithms are classified into:

1. Exact algorithms, such as Branch-and-Bound, Dynamic programming, Lagrangian relaxation, and Integer Programming (IP) methods. Among these, the application of IP on the TSP has lead to essential improvements. While exact methods always provide the optimal solution, they often require an enormous amount of computational time. This becomes even worse when implemented in large size problems resulting in an unreasonably great computational time. [5] [14] [15]

2. Heuristic algorithms are used to provide a near to optimal solution for TSP within reasonable time. Examples include Local search algorithms [8], Genetic algorithms [6], Simulated annealing [14], Ant colony optimisation algorithm [16], Particle swarm optimisation algorithm [17], and Hyper-Heuristics [18].The advantages of these heuristics approaches are that they are simple and relatively easy to understand. They require less programming and storage requirements and produce multiple solutions within short time.[14]

3. Collaborative Combinations make use of both exact and heuristic algorithms separately. Particularly, these algorithms are combined in order to exchange information. Puchinger and Raidl [15] refer to such a combination used to solve TSP to optimality. Initially, a broad set of solutions is produced by an iterated local search algorithm. Then, by merging the edge-sets of the solutions, the optimal solution is found.

4. Integrative Combinations are exact algorithms embedded in heuristic algorithms or vice versa. Puchinger and Raidl [15] mention a local and variable neighbourhood search heuristic with an embedded exact algorithm used to solve the ATSP. The exact algorithm is used within the local search part to explore huge areas within the solution space. This integrative combination algorithm successfully escapes local optima and produce solutions of high class.

Kiwi.com problem shares some characteristics with several TSP variants. These characteristics are; Time-Dependence with Time-Windows, Asymmetry and Generalisation. While TSP variants are complex and difficult to solve on their own, KIWI.com problem combines four variants of the TSP. One could think of the Kiwi.com problem as a Generalised-Asymmetric-Time-Dependent TSP with Time-Windows.

2.2 Optimisation in Air Travel

Over the last few years, more and more researchers are interested in optimisation techniques applied in the air travel. One of the main reasons for this growing interest is that air travelling is very common nowadays, and it has become an affordable means of transportation for the majority of the global population. Air travellers usually refer to online search engines to find a cheap trip. While finding the cheapest flight connection between two airports might not be that difficult, finding the cheapest possible multi-city trip is extremely complicated. Therefore, several studies have been conducted in order to provide better quality solutions for multi-city trips by air travel.

Kiwi.com released a challenge in 2017, called *Travelling Salesman Challenge*, which was studied by Duque et al. [19]. This challenge is actually a previous version of the current Kiwi.com challenge, *Travelling Salesman Challenge 2.0*, that is studied in this thesis. The main difference between them is that while in the former version the traveller had to visit a number of cities, in the current version cities are divided into areas from which exactly one city has to be visited. Moreover, the running time allowed for the previous challenge was 30 seconds. Duque et al. [19] applied SA, ACO, and a hybrid algorithm combining SA and ACO to the problem, including large instances up to 100 cities. Additionally, they parallelized the ACO algorithm and meta-optimised the parameters of SA and ACO algorithms with the aid of a genetic algorithm. To improve further the quality of their solutions, they applied a K-Opt technique to good solutions obtained from the algorithms. They used Greedy search and Backtracking as benchmarks and compared all these algorithms. Their results showed that their hybrid algorithm outperformed the rest algorithms. As far as K-Opt technique and meta-optimisation of parameters are concerned, they only improved solutions of medium size instances.

The *Flying Tourist Problem* (FTP) [3] is a variation of the Kiwi.com problem with more constraints. The FTP study involves a tourist who wants to make a multi-city flight journey with the best possible schedule, route and set of flights. The tourist should also spend a number of days in each city. The objective of their problem is not only to minimise the total cost but also the flight duration and the cost-duration combination. The FTP is an NP-hard problem due to its relation to the TSP. In addition, the amount of possible solutions, given by $N!$, is significantly greater to the TSP possible solutions, which are $(N-1)!/2$. Marques et al. [3] tested three well known algorithms for their problem with a maximum running time of 1 second; the Simulated Annealing (SA), the

Ant Colony Optimisation (ACO) and the Particle Swarm Optimisation (PSO). PSO proved to be the most effective algorithm between them. Their proposed algorithm was compared with the current algorithm that Kiwi.com uses, called *NOMAD*, and their results showed it provided cheaper solutions for 95% of the times. However, their proposed algorithm was only tested for up to 20 cities in total, which is a very small size problem.

Saradatta and Pongchairerks [8] have discussed the Time-Dependent ATSP with Time Window and Precedence Constraints in Air Travel. However, in this work, the traveller visits countries instead of cities, where some countries might have to be visited straight-away after some other countries have been visited. The traveller has to spend one week in each country and then travel to the next one. The authors suggested two Local Search (LS) algorithms for the problem; LS with Swap Operator, and LS with Insert Operator. A modified version of the Nearest Neighbor was used as a benchmark. Both LS algorithms performed better than the Modified Nearest Neighbor when tested in instances of up to 20 countries. Their results suggest that LS with Insert is more appropriate for instances with easy constraints, and LS with Swap is better for hard constraints. This is mainly because the Insert Operator has a greater probability of generating an infeasible solution when applied on instances with hard constraints. Specifically, when Insert Operator is used, all travel dates in the solution have to change and thus the infeasibility chance increases.

Another interesting study is *The Air-Traveling Salesman* project launched by the OpenFlights.org [2]. In this project, they are given a list of airports to visit and the goal is to find the best possible route that minimises the travel distance. For some airports, a direct flight does not exist and thus an intermediate airport might need to be visited. The heuristic method used to tackle this problem was a Nearest Neighbor (NN) algorithm with a 3-opt Swap Operator. This operator simply changes the visiting order of 3 airports in the solution. Initially, they modified the NN algorithm and used it to generate a list of feasible initial solutions. Then, the 3-opt Swap Operator is applied in all the initial solutions exploring every possible combination. Such a process, generates multiple local minima from which the best is chosen as a final solution to the problem. Their algorithm was tested on several STSP and ATSP instances in order to evaluate its performance. The results showed that the algorithm performed quite well for both types of instances with small number of cities. However, the performance was poor for larger size problems, especially for ATSP instances.

A transportation problem, including air travel, was studied by Li et al. [20] in 2016. The *Travel Itinerary Problem* involves a traveller who wants to make a multi-city trip without having any preference about the means of transportation. The aim is to find the cheapest possible combination of itineraries for the whole trip within the time frame specified. Li et al. [20] have solved the problem via IP and an implicit enumeration algorithm. In addition, the authors developed a Smart Travel System, where users input their travel preferences and the optimal tour is returned by using real online data. Their method is very effective and runs within reasonable amount of time for small size problems. However, since an exact method is used its main drawback is the exponentially increasing computational time with the increase of cities.

Last but not least, Touyz [21] tackled the *Travelling Tourist Problem* in 2013. In this work, a tourist wants to make a multi-city journey within specified time frames. The travel cost between cities depends on time and the objective is to minimise it. The approach of the author is to divide the main problem into two interconnected sub problems; 1) For a given set of days, the objective is to find the minimum cost of journey and 2) Find a set of days that minimises the cost of journey. Touyz [21] then combines SA, parallel chains and evolutionary Markov Chain Monte Carlo algorithms that exchange information amongst themselves to solve the problem. Firstly, a population of possible solutions is initialised and SA is used for improvement. In the next phase, solutions are sorted in an ascending order and a corresponding temperature is assigned in a descending order to each solution. In the final step, mutation and crossover operators are applied to produce new generations and the fitness criteria is re-evaluated until the termination criteria is met. The author generated data for 10 cities trips with different travel days in order to test the proposed method. Results showed that the algorithm produced good solutions within reasonable amount of time. Yet, the performance of the proposed algorithm is unknown for larger size problems.

2.3 Local Search

One of the most successful heuristic methods for hard combinatorial optimisation problems is Local Search (or Neighborhood Search). The main benefits of LS algorithms are that they are easy to understand, easy to implement, and they are very effective in terms of execution time and solution quality. A common combinatorial optimisation problem has a finite number of feasible solutions, where each solution has an associated

cost and the aim is to minimise (or maximise) the cost. The functionality of LS algorithms is based on a neighborhood structure, where a neighborhood is defined for each solution. A neighborhood represents a set of solutions, which are slightly modified than the original solution [22] [23]. In TSP, for example, a neighborhood solution would be a tour where the visiting order of two cities has been changed. A Swap Operator is a well-known and widely used method to generate neighborhoods. This operator, simply changes a small part of the solution and returns a new one. A 2-change move, for instance, changes the position of two elements in a solution. In the same vein, 3-changes swaps three elements and λ-changes swaps λ elements in order to create a neighborhood solution [22]. There are also other operators, such as Reverse Operator which reverses the order of the elements within a specified range of the solution. Insert Operator inserts a selected element from the solution into a specified position of the solution. Yet, as far as large size problems are concerned, the former mentioned operators are not as efficient as Swap Operator. This is because, they a perform a large change in the solution and therefore, have a greater chance of generating infeasible solutions. After an operator has been applied, a criterion is used on whether to accept the new solution or not. A typical criterion is to accept the new solution only if it is better than the old one [24]. However, the classical LS algorithm has a significant drawback. It can be easily trapped in a local minima when the neighborhood search space is poorly structured. For instance, figure

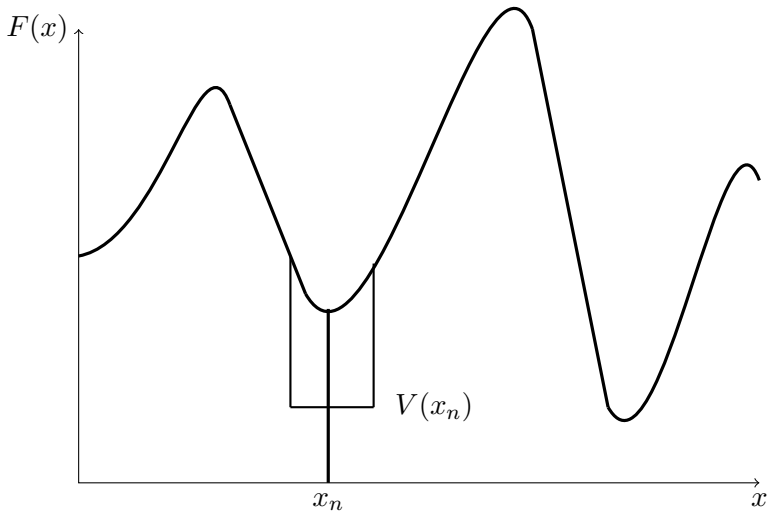

FIGURE 2.1: An example of LS being trapped in a local minima

2.1 shows a neighborhood search space $V(x_x)$, where $F(x)$ is a cost function and x is a set of solutions. The global minima in figure 2.1 is out of the neighborhood search

range and thus, the best solution would be a local minima x_n. One way to overcome this problem is to use a random restart approach. In this approach, multiple random initial solutions are generated and LS is applied in each one of them. This, allows the search space to increase significantly and consequently increases the probability of reaching a lower local minima [25]. Another way to avoid being trapped in a local minima is to use more sophisticated algorithms, such as Iterated LS, Simulated Annealing, Genetic Algorithms, Particle Swarm Optimisation or Ant Colony Optimisation [23] [24].

2.4 Hyper-Heuristics

Many hard combinatorial computational problems have been effectively solved by means of simple heuristics and meta-heuristics algorithms. Yet, these algorithms, such as genetic algorithms, simulated annealing, tabu search, etc. have certain drawbacks. That is, they require advanced knowledge in order to be applied in different domains, and thus lack in terms of re-usability [26] [27]. Their performance is highly based on the embedded neighborhood operators and sufficiently tuned parameters which are developed by the user. Furthermore, whenever a problem changes then the algorithm has to be appropriately adjusted. This is usually time consuming and in some cases the whole algorithm needs to be redesigned [28]. Due to these weaknesses, many researchers focused on building a method which would automatically design heuristics and would be applicable in a wide range of problems without requiring any expertise knowledge of the domain of a problem. This idea was firstly stated in 1960s but was not implemented until 2000 [29]. Nowadays, this automated process is known as Hyper-heuristic and is widely applied to many combinatorial problems. Hyper-heuristics are divided into two main categories; 1) *generation* hyper-heuristics which generate new heuristics and 2) *selection* hyper-heuristics which select a heuristic to apply among a set of low level heuristics [18]. In addition to this, hyper heuristics are further divided into two subcategories; 1) *constructive* methods which are constructing a solution from the beginning by implementing a set of heuristics at different phases of the construction process and 2) *local search* methods which use a complete initial solution and apply a set of heuristics in a perturbative way to improve the solution [28]. Selection local search hyper-heuristics, which are the focus of this thesis and from now on will be referred simply as hyper-heuristics, are methods that choose a heuristic from a set of low level heuristics at different stages and apply it to the solution for improvement. This particular method is very forceful since certain heuristics might perform better than others at certain stages and vice

versa. Hence, the sequence in which low level heuristics are applied is essential because altered sequences would generate different solutions [30]. Hyper-heuristics are capable of learning which sequences or heuristics are more effective during the search process by using feedback. Based on the origin of feedback, the learning process is classified as *online* or *offline*. In the former process, the algorithm is directly learning by solving the main problem, while in the latter process the algorithm is learning by solving a set of training instances which prepare the algorithm to effectively encounter completely new instances [29]. Hyper-heuristics operate without the need of any information regarding the functionality of low level heuristics. They require only the type of optimisation (min or max) and the number or content of low level heuristics [30]. In return, they provide useful feedback such as the amount of time required to apply a heuristic, the utilisation rate of each heuristic, the change in the objective function, etc. which are vital for the learning process [31] [32]. Moreover, their purpose is to find an effective method of solving a problem rather than finding a good solution [30]. Another significant strength of hyper-heuristics is that instead of exploring the space of solutions directly, they explore the space of heuristics [33]. Compared to other algorithms, they are fast and relatively easy to implement [34]. Their framework usually has two sequential steps; 1) *heuristic selection* and 2) *move acceptance* [26]. The first step is accountable for selecting a low level heuristic among a set of heuristics and apply it to the solution, whereas the second step is a decision regarding the acceptance or rejection of the new solution [18].

2.4.1 Heuristic selection

The process of heuristic selection is divided into *deterministic* and *non-deterministic*. Deterministic processes select the next heuristic to apply in a predefined specific sequence at each iteration. On the other hand, non-deterministic processes choose the next heuristic to apply according to some probability distribution or by using their learning mechanism. Some of the heuristic selection strategies developed follows [26]:

- Simple Random (SR): selects a low level heuristic in a random manner according to a uniform probability distribution.
- Random Descent (RD): selects a low level heuristic randomly and applies it repeatedly for as long it yields better solutions.
- Random Permutation (RP): selects low level heuristics in a predefined randomly generated sequence.

- Random Permutation Descent (RPD): is basically a mix of RD and RP strategies. In other words, it selects a low level heuristics based on RP strategy and applies it repeatedly until no further improvement.
- Choice Function (CF): assigns a score to each low level heuristic according to a synthesis of three different measures and selects the one with the best score to apply.
- Roulette Choice (RC): uses the CF strategy and assigns a probability of selection to each low level heuristic which is proportional to $F(LLH_i)/\sum_i F(LLHi)$ [31].
- Reinforcement Learning (RL): assigns a minimum score to every low level heuristic in the initialisation of the algorithm. These scores are increased or decreased based on the improvement or deterioration of the solution respectively. RL chooses and applies the heuristic with the best score at each stage [30].
- Tabu Search (TS): uses some principles of RL to rank the low level heuristics. Whenever an applied heuristic improves the solution, its rank is increased. Otherwise, its rank decreases and it enters the tabu list while the current solution is unchanged. TS chooses and applies the heuristic with the best rank that is not in the tabu list [33].
- Greedy (GR): applies all low level heuristics at each iteration and selects the one with the biggest improvement in the objective function.
- Sequence-based Selection Hyper-heuristic (SSHH): is a learning strategy in which the goal of the algorithm is to identify the sequences of heuristics with the best performances [27].
- Dominance-based and Random Descent hyper-heuristic (DRD): is a hybrid multi-stage hyper-heuristic. A set of effective heuristics is identified with a GR strategy at the dominance-based stage and those heuristics are then applied with an RD strategy. The set might be updated by the dominance-based stage if no progress is made during the search process [27].

Random strategies are usually used as a benchmark due to their main drawback of generating "good" quality solutions by pure randomness [30]. CF consists of three different measures; f_1, f_2 and f_3. f_1 measures the previous performance of each low level heuristic by using the following formula:

$$f_1(LLH_i) = \sum_n \alpha^{n-1} \frac{I_n(LLH_i)}{T_n(LLH_i)}$$

where $I_n(LLH_i)$ is the change in the objective function, $T_n(LLH_i)$ is the time needed to apply LLH_i in n previous call, and α is a value between 0 and 1. f_2 measures the performance of LLH_k and LLH_i when called in this sequence by using the following formula:

$$f_2(LLH_k, LLH_i) = \sum_n \beta^{n-1} \frac{I_n(LLH_k, LLH_i)}{T_n(LLH_k, LLH_i)}$$

where $I_n(LLH_k, LLH_i)$ is the change in the objective function, $T_n(LLH_k, LLH_i)$ is the time needed to apply both LLH_k and LLH_i in n previous call, and β is a value between 0 and 1. f_3 records the time elapsed $\tau(LLH_i)$ since the previous time that LLH_i was called.

$$f_3(LLH_i) = \tau(LLH_i)$$

These three measures are used to calculate the score of each low level heuristic with the following formula:

$$F(LLH_i) = \alpha f_1(LLH_i) + \beta f_2(LLH_k, LLH_i) + \delta f_3(LLH_i)$$

where α and β weight the f_1 and f_2 respectively to intensify the search process, and δ weights f_3 to diversify the process [18]. Drake et al. [33] in their work focused on tackling two major drawbacks of CF strategy. The first one is that some heuristics are over rewarded for large improvements at early stages, while at later stages, where only small essential improvements are possible, heuristics are under rewarded. The second one is that diversification, f_3, is out of control and is not contributing at the time when its needed the most. Therefore, Drake et al. [33] proposed the Modified Choice Function (MCF) in order to overcome those limitations. In MCF, α and β weights are replaced by a single parameter ϕ to intensify the search process as shown in 2.1 and δ is defined by the equation 2.2.

$$F(LLH_i) = \phi_t f_1(LLH_i) + \phi_t f_2(LLH_k, LLH_i) + \delta_t f_3(LLH_i) \tag{2.1}$$

$$\delta_t(LLH_i) = 1 - \phi_t(LLH_i) \tag{2.2}$$

where t is the number of times LLH_i was called since the last improvement made by the same heuristic. For every improvement achieved ϕ is rewarded and gets a value close to 1, whereas δ gets a value close to 0. For every non-improving move ϕ and δ are linearly decreased and increased respectively. The authors proved that these modifications achieve an improved utilisation of intensification and diversification. Chakhlevitch and Cowling [30] highlight a main drawback of the RL strategy. While at early stages some heuristics might be more effective than others, on later stages this condition might change leading the RL to repeatedly select ineffective heuristics until their score decrease sufficiently. The authors recommend to place in the tabu list any non-improving heuristics until a better solution is found so as to overcome this issue. Bai et al. [28] proposed another way to overcome this drawback by utilising shorter term memories. Hence, the algorithm will gradually forget any repetitive improvements performed by certain heuristics at early stages. Moreover, Bai et al. [28] emphasise the importance of distinguishing between the instances in which heuristics fail to yield feasible solution and the instances where heuristics yield a feasible but worse solution. Heuristics of the latter type should get more chances of being selected, especially when a local optima is reached. Some interesting variations of TS have been developed by Kendall and Hussin [35]. In [35], TS places the heuristic applied in the tabu list regardless of its result on the solution. A short and fixed tabu duration prevents any heuristic that belongs in the tabu list to be applied. In each iteration, only the best non-tabu heuristic is allowed to be applied. The advantage of this strategy is that it is steadily yielding good quality solutions (not the best though) given that there is enough CPU time. Another version of TS, in [36], repeatedly applies a certain heuristic for as long as it leads to improvements and places it in the tabu list when no further improvement is made. Tabu durations are random and within a fixed range. GR, on the one hand, is a learning strategy with the least memory consumption because at each iteration previous information is discarded [29]. On the other hand, it is slower than random strategies because it evaluates each heuristic in each iteration [30].

2.4.2 Move acceptance

According to the type of the move acceptance, hyper-heuristics are categorised into two different types; *mutational heuristics* and *hill climbers*. The principle of hill climbers is to accept only improved solutions (or sometimes equal to previous solution) in each stage

of the search process, while mutational heuristics might accept worse solutions at certain stages during the search process. Mutational heuristics are particularly important for escaping local optima. Sometimes, a better performance might be achieved by applying a hill climber after a mutational heuristic [26] [32]. Several move acceptance strategies have been developed from which some are the following [26]:

- All Moves (AM): accepts all moves.
- Only Improving (OI): accepts only improving moves.
- Improving and Equal (IE): accepts moves that improve the objective value or moves that are equal to the previous objective value.
- Exponential Monte Carlo (EMC): accepts all improving moves and some non-improving moves according to a dynamic probability function as shown in 2.3.

$$p_t = e^{-\frac{\frac{\Delta f}{N}}{1-\frac{t}{D}}} \tag{2.3}$$

  where Δf is the change in the objective value at iteration t, D is the maximum iterations number, and N is an expected range for the maximum change in the objective value.

- Exponential Monte Carlo with Counter (MC): is a modified version of EMC where a counter increases the probability of accepting a worse move when no improvement has been made for a fixed number of iterations.
- Simulated Annealing (SA): is similar to EMC but uses a different dynamic probability function as shown in 2.4 [18].

$$p_t = e^{-\frac{\Delta f}{\Delta F(1-\frac{t}{T})}} \tag{2.4}$$

  where Δf is the change in the objective value at t iteration, T is the maximum number of iterations, and ΔF is the range for the maximum change in the objective value after a heuristic is applied.

- Great Deluge (GD): accepts all moves within a dynamic level of the objective value. The initial level is equal to the initial objective value and is linearly updated towards the expected objective value by using the 2.5 formula.

$$\tau_t = f_0 + \Delta F \times (1 - \frac{t}{T}) \tag{2.5}$$

where τ_t is the threshold level at iteration t, T is the maximum number of iterations, ΔF is the expected range for the maximum change in the objective value, and f_0 is the final expected objective value.

- Record-to-Record Travel (RRT): is a variation of GD which accepts worse moves only if the move is not significantly worse than the previous solution. In addition for minimisation problems, a small value, called *DEVIATION* parameter, is added to the current solution to increase the acceptance probability [37].

- Late Acceptance: stores previous objective values in a set of size L. Then, the new solution is accepted if it is better than the Lth solution [38].

Chapter 3

Problem Description

Kiwi.com problem requires the minimisation of the travelling cost by finding the best possible flight route for a given number of areas, where an area is a set of cities (airports). Costs differ according to the direction and day of travel. It is assumed that cost is the same for all hours of any day (i.e. a morning flight has the same price as an evening flight). The trip begins from a given city and everyday exactly one city of each area is visited. Throughout the trip, the arrival city is also the departure city for each area. This means that it is not allowed to arrive to a city and continue the trip by departing from another city of the same area. The trip ends in the area (not necessarily the city) where it began.

Mathematically, let $Area = \{area_1, area_2, \ldots, area_n\}$ be a set of n areas, where each area $r \in Area$ is composed of a set of airports $\{airport_1, airport_2, \ldots\}$; and let c_{ij}^d be the flight cost between the departure airport i and the arrival airport j on day d, which has two properties: c_{ij}^d is not necessarily equal to c_{ji}^d (i.e. the problem is asymmetric); and for $d1 \neq d2$, c_{ij}^{d1} is not necessarily equal c_{ij}^{d2}, (i.e. the problem is time dependent). For some cities, where $d = 0$, flights are available everyday. Moreover, in some instances there are multiple flights between the same cities for the same day with different costs (i.e. different airline companies for the same connection). The objective of the problem is to find the best possible flight route that connects all the given areas and minimises the cost within the time given, subject to the following constraints:

- The trip starts from the starting airport (city) given.
- Exactly one city is visited in each area (but we can choose which one).
- Every day a different area is visited.
- The trip continues from the arrival airport.
- The entire trip ends in any airport of the area where it began.

A simple problem instance consisting of 4 areas and 8 airports is presented in Figure 3.1. In this example, the 4 areas are; Greece, Italy, Spain, and the UK, where each area contains two cities; Athens and Thessaloniki, Rome and Milan, Madrid and Barcelona, and London and Liverpool, respectively. The trip begins from Athens to Madrid, then continues from Madrid to London, then from London to Rome and ends in Thessaloniki. Notice that in this trip the starting and finishing airports are different, but the trip is acceptable because it ends in the area where it began.

FIGURE 3.1: Simple problem instance with a possible solution

Kiwi.com provided small, medium, and large datasets, which are 14 in total, covering a range from 10 to 300 areas and a range from 1 to 6 airports (All 14 datasets can be found at https://code.kiwi.com/travelling-salesman-challenge-2-0-wrap-up-cb4d81e36d5b). Each dataset is structured as follows (An example of Instance-1 is provided in Appendix B):

- The first input in the instance is the number of areas the user wants to visit.
- The second input is the starting airport in which the user is currently located.
- Then the areas and the corresponding airports are listed.
- Then follows a list of the travelling costs between each pair of airports for the corresponding day.

The time limits suggested by Kiwi.com for the instances are the following:

- For small test cases, i.e. test cases where the number of areas ≤ 20 and the total number of airports < 50, the time limit is 3 seconds.
- For medium test cases, i.e. test cases where the number of areas ≤ 100 and the total number of airports < 200, the time limit is 5 seconds.
- For large test cases, i.e. test cases where the number of areas > 100, the time limit is 15 seconds.

Chapter 4

Methodology

In this thesis, Kiwi.com problem is solved by means of Hyper-heuristics in Python 3.7 programming language. Hyper-heuristics are fast, relatively easy to implement, they exploit the search space of low level heuristics instead of the search space of the solution, and they are capable of learning. Hence, they are an ideal candidate to provide good solutions for the Kiwi.com problem and overcome the time limit challenges. In total, 6 Hyper-heuristic algorithms with different heuristic selection and move acceptance methods are implemented in all instances. Briefly, the algorithm generates an initial solution, then low level heuristics are applied until the solution becomes feasible, and finally Hyper-heuristics improve the cost until the given time limit. The solution is represented in a two dimensional vector (see equation 4.1), the first dimension indicates the index of the area (see equation 4.2) and the second dimension indicates the airport of the corresponding area (see equation 4.3).

$$Solution = (Sol_{area}, Sol_{airport})_N \tag{4.1}$$

Where $N = (1, 2, \ldots, n)$ is the total number of given areas.

$$Sol_{area} = \{area_1, area_2, \ldots, area_n\} \tag{4.2}$$

$$Sol_{airport} = \{airport_1, airport2, \ldots, airport_n\} \tag{4.3}$$

Where each pair $(area, airport)_i$ represents the area and the corresponding airport visited on day i. Due to the size and complexity of the problem, the algorithm is divided into smaller meaningful parts which are discussed in detail in the following sections.

4.1 Initial Solution

All datasets provided by Kiwi.com are in $'dataset'.in$ format and entirely consist of string objects. Thus, beside reading the dataset, all numbers are converted into integer objects. This task is done while reading the data. Otherwise, it would slow down the algorithm for large datasets and exceed the time limit. Moreover, all datasets are separated into general information and flights information vectors. The former represents the total number of areas, the starting airport, the areas and the airports, while the latter represents all the available flights among airports with the corresponding day and cost. The first input in the instances, which is the number of areas to be visited, plays a key role in the accomplishment of this task. This is because, since the total number of areas is known then the length of lines containing the general information could be computed. The remaining length of lines are the flights information. Similarly, by using a counter variable to keep track of the line number, the conversion process is done concurrently with the reading dataset process.

The next step is to create a dictionary object that uses as a key a vector containing the departure airport, the arrival airport and the day. The value of the key is the corresponding cost. However, some instances have more than one available flights to choose from with different costs. Therefore, if multiple flights exist on the same day and for the same route then the cheapest among them is stored.

After the completion of these tasks, a random initial solution is produced. According to the datasets, the initial solution could be feasible or infeasible. In the first phase, the solution areas are generated by enumerating all areas in a vector. Then the area that contains the starting airport is found and swapped with the first area of the vector. In addition, the starting area is also added at the end of the vector. In the second phase, the solutions airports are generated by using the solution areas vector as indexes in the general information vector to get the first airport of each area. An initial solution example for Instance-1 follows:

$$Sol_{area} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$$

$$Sol_{airport} = \{AB0, AB1, AB2, AB3, AB4, AB5, AB6, AB7, AB8, AB9, AB0\}$$

4.2 Low Level Heuristics

- Swap Operator: randomly selects two areas in the solution and their corresponding airports and swaps them. An example from Instance-1 follows:

  Before Swap:

  $Sol_{area} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$

  After Swap:

  $Sol_{area} = \{0, 8, 2, 3, 4, 5, 6, 7, 1, 9, 0\}$

- Insert Operator: randomly selects an area in the solution and an index within the solution. Then the area is inserted to the determined index by moving all the areas between the previous area index and the determined index to the left or right as required to maintain the solution size. An example from Instance-1 follows:

  Before Insert:

  $Sol_{area} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$

  After Insert:

  $Sol_{area} = \{0, 2, 3, 4, 5, 6, 7, 8, 1, 9, 0\}$

- Reverse Operator: randomly selects two areas in the solution and reverses all the solution instances between these two areas. An example from Instance-1 follows:

  Before Reverse:

  $Sol_{area} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$

  After Reverse:

  $Sol_{area} = \{0, 1, 8, 7, 6, 5, 4, 3, 2, 9, 0\}$

- Change Airport Operator: randomly selects two airports in the solution and replaces them with another airport of the corresponding area. An example from Instance-2 follows:

  Before Change Airport:

  $Sol_{airport} = \{EBJ, NBP, OMG, NCA, NUJ, OHT, GSM, EFZ, QKK, SSC, TKT\}$

  After Change Airport:

  $Sol_{airport} = \{EBJ, NBP, OMG, XJE, NUJ, OHT, GSM, EFZ, QKK, SSC, EBJ\}$

4.3 Delta Evaluation

In this section, the methods for calculating feasibility, cost, delta feasibility and delta cost evaluation are explained. Delta feasibility and cost are extremely useful functions so as to avoid unnecessary calculations and speed up the algorithm. In other words, they evaluate only the changed parts of the solution [39]. These functions are easy to develop when the solution exhibits small changes. For instance, Swap operator changes only two parts of the solution. However, when Insert and Reverse operators are applied they perform large changes in the solution and developing these functions becomes quite complex. Additionally, Change Airport operator introduces more complexity when the last airport is selected.

The feasibility calculation function checks if all airport connections in the solution exist in the keys of the dictionary for the corresponding day or for day equal to 0. For every non existing connection a counter variable is increased by 1. In addition, a vector is used to save the indexes of the non existing connections and is later on used to enhance the process of improving the feasibility. The cost calculation function operates in the same manner as feasibility calculation. It extracts the cost from the dictionary by using the keys. In some cases, a flight connection is available every day and is also available for a certain day. Whenever this is true, the cheapest cost is extracted.

The delta feasibility function evaluates only the changed parts of the solution. When a Swap operator is applied then the feasibility is evaluated between the swapped and the previous airport in the solution, and between the swapped and the next airport in the solution. As far as Reverse and Insert operators are concerned, a binary variable is introduced to instruct the algorithm for a different feasibility evaluation procedure whenever one of these operators improves the cost. This variable drives the function to evaluate the feasibility all over the changed part of the solution. When the algorithm enters the cost improvement process the solution is always feasible. Thus, whenever delta feasibility function finds an infeasible connection, the function is aborted immediately without evaluating any remaining parts of solution. Regarding the Change Airport operator, the evaluation process follows the same procedure as Swap operator but with an exception if the last airport of the solution is changed. Another binary variable is introduced that drives the evaluation function towards a different procedure. In this procedure, the single difference is that the function evaluates the feasibility between the last airport and the previous airport. The delta cost function operates in the same manner as delta feasibility function. It calculates the partial cost for a selected part

of the solution. Just as cost calculation function, it chooses the cheapest cost in cases when a flight connection is available every day and is also available for a certain day.

4.4 Feasibility Improvement

The process of feasibility improvement for the given time limits was challenging, particularly for several instances. While for some instances a certain method produced feasible solutions in time, it was ineffective for other instances. Therefore, a Hyper-heuristic method with three different sets of low level heuristics is implemented based on the instance during the feasibility improvement process. In addition, 'guided' operators are introduced to enhance the process. These operators use the vector with the infeasible connections provided by the feasibility calculation function. This function returns the minimum (INMIN) and maximum (INMAX) indexes of the infeasible connections. Then, a random number is generated within that range and is used by the 'guided' operators (INR). Whenever INMIN and INMAX are equal means that there is only one infeasible connection left and at this stage the particular index is passed. A general framework of the Hyper-heuristic method used for the feasibility improvement is shown in Algorithm 1.

Algorithm 1: Feasibility Improvement

```
1  Let O represent the set of operators
2  Let S represent the current solution
3  Let S_new represent the new solution
4  Let F represent the feasibility of the current solution
5  Let F_new represent the feasibility of the new solution
6  S ← Initialise();
7  repeat
8  |   F ← CalculateFeasibility(S);
9  |   O_i ← Select(O);
10 |   S_new ← ApplyOperator(O_i, S);
11 |   S_new ← ApplyChangeAirportGuidedOperator(S_new);
12 |   F_new ← CalculateFeasibility(S_new);
13 |   if F_new ≤ F then
14 |   |   S ← S_new;
15 |   |   F ← F_new;
16 |   end
17 until F = 0;
```

The Hyper-heuristic method uses a set of five low level heuristics for all small and medium size instances (from Instance-1 up to Instance-6), and Instances 8 and 10. The set of low level heuristics includes; Swap operator, Insert operator, Swap Guided operator, Insert Guided operator and Change Airport Guided operator. The method performs iterations until the solution becomes feasible. In every iteration a low level heuristic is selected randomly. Guided operators select a random area from the solution and replace it with a potentially infeasible area (INR). After the application of a low level heuristic, the Change Airport Guided operator follows. This operator changes the airport of a randomly selected area, the airport of INMAX, and the airport of the next area of INMAX. Then, the new solution is accepted if the feasibility is equal or less to the previous. An example is demonstrated in Figure 4.1, where Swap and Insert heuristics

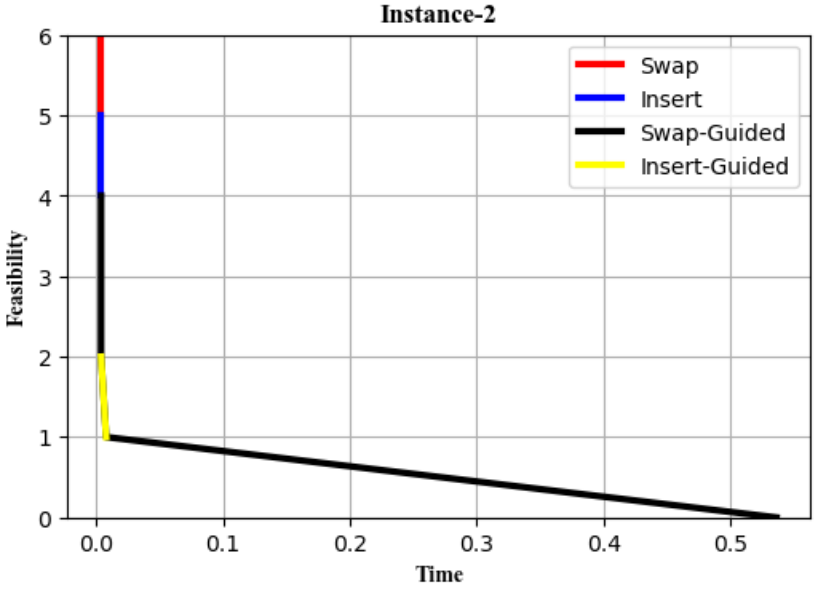

FIGURE 4.1: Plot of Feasibility improvement versus time for Instance-2

decrease the infeasibility by two units and then Swap-Guided along with Insert-Guided heuristics lead to a feasible solution. The same Hyper-heuristic method is implemented for Instance-7 but with a different set of low level heuristics. In this set, four low level heuristics are used; Swap operator, Reverse operator, Reverse Guided operator, and Change Airport Guided operator. The Reverse Guided operator reverses the order of all areas in the solution between INMIN and INMAX. For the remaining Instances except for Instance-14 the following set of low level heuristics is used; two Swap Guided operators, Reverse Guided operator, and Change Airport Guided operator. The first Swap Guided operator selects a random area in the solution and swaps it with the INMIN area, whereas the second Swap Guided operator swaps it with the INMAX area. The Reverse Guided

operator reverses the order of areas in the solution between INMIN and a potentially infeasible area (INR). The Change Airport Guided operator changes the airport of a randomly selected area, the airport of INMIN, the airport of the next area of INMAX, and the airport of a potentially infeasible area (INR). In Table 4.1, the average times of a feasible solution generation over 10 runs are shown when the process of feasibility improvement was used. When this process was not used, the average times were greater than one minute. Instances not shown in the table are initially feasible and do not require any improvement. In Instance-14, the low level heuristics used are; two Swap Guided

TABLE 4.1: Average times of feasible solution generation over 10 runs

Name	Average Time
Instance-2	1.24(s)
Instance-6	0.23(s)
Instance-7	4.9(s)
Instance-8	3.53(s)
Instance-9	4.27(s)
Instance-10	4.21(s)
Instance-11	5.02(s)
Instance-12	10.34(s)
Instance-13	12.58(s)
Instance-14	48.55(s)

operators, Swap operator and, Insert operator. The Swap Guided operators have the same functionality as mentioned above. The utilisation rates for each low level heuristics are illustrated in Figure 4.2. During the feasibility improvement process, Swap-Guided operator has a utilisation rate of 68%, Swap follows with a rate of 28% and, Insert with a rate of 4%. Although, this method generates a feasible solution within reasonable time, it does exceed the time limit of 15 seconds. When the method was executed for 10 runs, it generated a feasible solution within 48.55 seconds in average. The fastest and slowest feasible solutions were given within 24.32 seconds and 97.45 seconds respectively.

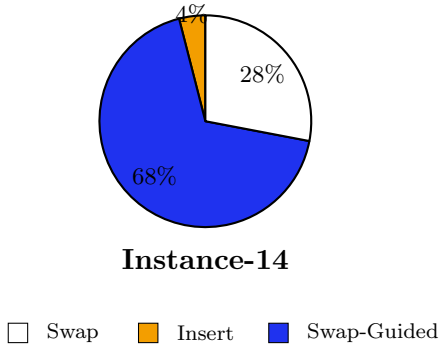

FIGURE 4.2: Utilisation rate of each operator for improving Feasibility in Instance-14

4.5 Hyper-Heuristics

In this section, the Hyper-heuristics methods used in the Kiwi.com problem are discussed. In total, 6 Hyper-heuristics are implemented; SR-IE, SR-GD, RD, RP-IE, RPD, and RL-OI. All of them have the same set of low level heuristics; Swap operator, Insert operator, Reverse operator, and Change Airport operator.

SR-IE method selects randomly a low level heuristic and applies it to the solution. If the low level heuristic selected is not Change Airport operator, then Change Airport operator has a probability of 50% to be applied after the selected low level heuristic. Then, if the new solution is feasible, its cost is compared with the cost of the previous solution. Whenever, the new cost is equal or less than the previous, the new solution is saved. Otherwise, either the new solution is infeasible or worse, it is scraped and the previous solution is restored, as shown in Algorithm 2.

Algorithm 2: Simple Random - Improving or Equal

```
1  Let O represent the set of operators
2  Let S represent the current solution
3  Let S_new represent the new solution
4  Let F_new represent the delta feasibility of the new solution
5  S ← Initialise();
6  repeat
7  |  O_i ← Select(O);
8  |  S_new ← ApplyOperator(O_i, S);
9  |  if O_i ≠ ChangeAirportOperator then
10 |  |  if Probability > 50% then
11 |  |  |  S_new ← ChangeAirportOperator(S_new);
12 |  |  end
13 |  end
14 |  F_new ← CalculateDeltaFeasibility(S_new);
15 |  if F_new = 0 then
16 |  |  if S_new ≤ S then
17 |  |  |  S ← S_new;
18 |  |  end
19 |  end
20 until TimeLimit;
```

In SR-GD method, a low level heuristic is randomly selected and applied to the solution. The same Change Airport and feasibility rules are applied as described in SR-IE method. Then, the new solution is accepted if it is not worse than τ, which is given by equation 2.5 in section 2.4.2. GD move acceptance strategy might also accept a worse solution.

Therefore, all accepted solutions are saved in a vector and the one with the lowest cost is returned when the time limit is about to be reached.

RD method selects randomly a low level heuristic and repeatedly applies it for as long as it yields better solutions. The same Change Airport and feasibility rules are applied as described in SR-IE method.

In RP-IE method, a vector is generated containing the application sequence of the low level heuristics. In each iteration a low level heuristic is selected based on the sequence vector. The move acceptance strategy is IE and rules for Change Airport operator and feasibility are as in SR-IE.

In RPD method, a vector is generated containing the application sequence of the low level heuristics. In each iteration a low level heuristic is selected based on the sequence vector and is repeatedly applied until no further improvement, as shown in Algorithm 3. Rules for Change Airport and feasibility are as in SR-IE.

Algorithm 3: Random Permutation Descent

```
1  Let O represent the set of operators
2  Let j represent the current iteration
3  Let P represent the random operators sequence
4  Let S represent the current solution
5  Let S_new represent the new solution
6  Let F_new represent the delta feasibility of the new solution
7  S ← Initialise();
8  repeat
9      O_i ← Select(P_j);
10     S_new ← ApplyOperator(O_i, S);
11     F_new ← CalculateDeltaFeasibility(S_new);
12     while S_new < S and F_new = 0 do
13         S ← S_new;
14         S_new ← ApplyOperator(O_i, S);
15         F_new ← CalculateDeltaFeasibility(S_new);
16     end
17 until TimeLimit;
```

In RL-OI method, low level heuristics compete with each other for selection and the best one is selected. Each LLH is assigned a score and based on their performance are either rewarded or penalised. In addition, any LLH that produces an infeasible solution is penalised slightly higher as suggested by Bai et al. [28]. All rewards and penalties are weighted based on the iteration number. This idea is introduced to avoid over-rewarding LLHs in the early phase of the search process, as mentioned by Chakhlevitch

and Cowling [30], and maintain a fair reward and penalty system. However, penalties are less weighted than rewards and work as if the LLH is given a second chance. In this method, only improving and sometimes worse solutions are accepted. The procedure of accepting a worse solution is controlled by a counter variable β, which keeps track of the consecutive number without improvements. Variable β increases even more when an infeasible solution is generated. Another variable, called α, represents the tolerance number for consecutive non-improvements and needs to be tuned according to the size of the Instance. Whenever, the solution is not much worse than the previous and $\beta > \alpha$, a worse solution is accepted and all scores of LLHs are set on default. This idea is implemented so as to escape local minima, as discussed in Section 2.3. Furthermore, since worse solutions are accepted, the best solution is always recorded. Note that in this algorithm the Change Airport probability rule is not applied. An example is provided in Algorithm 4.

Algorithm 4: Reinforcement Learning - Only Improving

```
1  Let O represent the set of operators
2  Let OS represent the scores of operators
3  Let j represent the current number of iteration
4  Let β represent the number of iterations without improvement
5  Let α represent a tolerance number for non-improvements
6  Let S represent the current solution
7  Let S_new represent the new solution
8  Let F_new represent the delta feasibility of the new solution
9  S ← Initialise();
10 repeat
11     O_i ← Select(OS_best);
12     S_new ← ApplyOperator(O_i, S);
13     if F_new = 0 then
14         if S_new < S or β > α then
15             S ← S_new;
16             OS_i ← OS_i + j × 2 × 0.015;
17             β ← 0;
18         else
19             OS_i ← OS_i − (j/2) × 0.01;
20             β ← β + 1;
21         end
22     else
23         OS_i ← OS_i − (j/2) × 0.0125;
24         β ← β + 10;
25     end
26 until TimeLimit;
```

Chapter 5

Results

In this section, the results of each Hyper-heuristic method for all instances are presented, a statistical test is conducted for the best method against the rest methods to identify any significant difference in performance, and a further analysis of the best method is presented.

5.1 An Overview of the Results

The experiments were performed on an i5-8300H CPU Intel Processor at 2.30GHz with 8.00GB RAM. The summary of experimental results are presented in Table 5.1 and Table 5.2. Each method was executed for 30 runs with different random seed values and within the time limits that Kiwi.com specified. In both Tables, the name of the instance is shown in the first column, then the Hyper-heuristic methods are shown on the top followed by the best cost, the average cost, and the standard deviation obtained over 30 runs. For most of the instances, the results suggest that RL-OI had the best performance, random methods had a moderate performance, and SR-GD had the worst performance. All methods performed well for small size instances (Instance-1 to Instance-3) and found the best known cost. RL-OI managed to obtain the best known solution in all 30 runs for Instance-1, and had the best average solution and standard deviation for Instance-3. Instance-2 is a special dataset because it has only one feasible solution. Thus, the challenging part of that instance was to find the feasible solution and not to optimise it. In medium size instances (Instance-4 to Instance-6), none of the methods was able to find the best known solution within time limits. In Instance-4, RL-OI obtained better results than the rest methods. In Instance-5, both RL-OI and SR-IE found the best

TABLE 5.1: The performance of SR-IE, SR-GD, and RD algorithms over 30 runs

	SR-IE			SR-GD			RD		
Name	best	average	std.	best	average	std.	best	average	std.
Instance-1	1396	1452	50	1396	1420	31	1396	1468	72
Instance-2	1498	1498	0	1498	1498	0	1498	1498	0
Instance-3	7672	7913	297	7672	7953	334	7672	7929	332
Instance-4	14611	15403	440	14828	17129	1020	14485	15420	609
Instance-5	735	952	102	953	1374	178	749	975	120
Instance-6	3614	4574	435	5890	10035	1446	3829	4744	509
Instance-7	34482	36217	1095	86598	91695	2622	34212	36333	1136
Instance-8	10516	14790	1622	18192	24854	2302	10172	14646	2013
Instance-9	205121	242804	18090	288043	311054	9897	209537	235814	13776
Instance-10	116527	134137	7812	357156	374482	9989	115304	138660	9323
Instance-11	73441	80178	3689	95741	101254	3433	74931	79885	3002
Instance-12	122651	134118	4377	141492	147971	4505	123234	131992	4932
Instance-13	179496	189218	4619	180819	190891	5032	179137	189775	5567
Instance-14	-	-	-	-	-	-	-	-	-

TABLE 5.2: The performance of RP-IE, RPD, and RL-OI algorithms over 30 runs

	RP-IE			RPD			RL-OI		
Name	best	average	std.	best	average	std.	best	average	std.
Instance-1	1396	1444	45	1396	1448	46	**1396**	1396	0
Instance-2	1498	1498	0	1498	1498	0	**1498**	1498	0
Instance-3	7672	7961	358	7672	8004	473	**7672**	7776	231
Instance-4	14535	15539	575	14535	15647	661	**14251**	14755	395
Instance-5	751	993	124	750	962	112	**735**	925	86
Instance-6	3874	4664	408	**3197**	4693	578	3322	4434	386
Instance-7	34408	36263	1213	34306	36147	982	**32785**	34202	793
Instance-8	10352	13801	1900	10495	14948	2024	**10162**	13728	2338
Instance-9	201151	230808	12903	199515	227873	13521	**190844**	225528	18178
Instance-10	123417	137739	7786	127507	142462	9433	**102710**	116809	9262
Instance-11	73518	79387	3908	74354	80884	3651	**71939**	79554	4116
Instance-12	124776	134897	4932	123766	132151	4788	**110122**	124597	4682
Instance-13	182559	190052	4090	180775	190861	4496	**174572**	185178	4972
Instance-14	-	-	-	-	-	-	-	-	-

solution compared to the rest methods. However, RL-OI had better average cost and less standard deviation than SR-IE. RPD found the best solution for Instance-6, whereas RL-OI had the best average and standard deviation. In large size instances (Instance-7 to Instance-14), RL-OI had the best results for most of the instances. In general, the results are quite poor for large size instances. However, if more time was allowed, results could be improved. In Instance-14, the algorithm failed to yield a feasible solution within time limit.

As shown above, RL-OI generated the best results for most of the instances. In order to confirm this, the Mann-Whitney-Wilcoxon (MWW) test is conducted to identify any

TABLE 5.3: MannWhitney-Wilcoxon test of RL-OI at 5 % significance level

Name	SR-IE	SR-GD	RD	RP-IE	RPD
Instance-1	$<$	$<$	$<$	$<$	$<$
Instance-2	$<$	$<$	$<$	$<$	$<$
Instance-3	$*$	$<$	$<$	$<$	$<$
Instance-4	$<$	$<$	$<$	$<$	$<$
Instance-5	$*$	$<$	$*$	$<$	$*$
Instance-6	$*$	$<$	$<$	$<$	$<$
Instance-7	$<$	$<$	$<$	$<$	$<$
Instance-8	$<$	$<$	$<$	$*$	$<$
Instance-9	$<$	$<$	$<$	$*$	$*$
Instance-10	$<$	$<$	$<$	$<$	$<$
Instance-11	$*$	$<$	$*$	$*$	$*$
Instance-12	$<$	$<$	$<$	$<$	$<$
Instance-13	$<$	$<$	$<$	$<$	$<$

significantly different behaviour between RL-OI and the rest methods. The MWW test is a non-parametric statistical test which identifies whether two independent samples have different distributions. In Table 5.3, the results of the MWW test at 5% significance level are presented, where $<$ symbol indicates that RL-OI generates statistically better results than the compared method, and $*$ symbol indicates that no significant difference exists. As seen in Table 5.3, RL-OI generates better results for most of the instances. Furthermore, the results suggest that SR-IE has a similar performance to RL-OI in Instances-3/5/6/11. In Instances-5/11, RD and RPD performances are not significantly different to RL-OI. RP-IE performs similar to RL-OI in Instances-8/9/11, and RPD performs similar to RL-OI in Instance-9.

5.2 An Analysis of the RL-OI method

The performance of RL-OI method over 30 runs is shown in Table 5.4, where the first four columns are the characteristics of each instance, then the time limit follows along with summary statistics and, the last two columns present the best known cost and the gap percentage between the best cost found by RL-OI and the best known cost. The results

TABLE 5.4: The performance of RL-OI algorithm over 30 runs

Name	areas	airports	airports in areas	time(s)	best	average	std.	best known	gap (%)
Instance-1	10	10	1 (min) – 1 (max)	3	1396	1396	0	1396	0%
Instance-2	10	15	1 (min) – 2 (max)	3	1498	1498	0	1498	0%
Instance-3	13	38	1 (min) – 6 (max)	3	7672	7776	231	7672	0%
Instance-4	40	99	1 (min) – 5 (max)	5	14251	14755	395	14024	1.59%
Instance-5	46	138	3 (min) – 3 (max)	5	735	925	86	698	5.03%
Instance-6	96	192	2 (min) – 2 (max)	5	3322	4434	386	2159	35%
Instance-7	150	300	1 (min) – 6 (max)	15	32785	34202	793	31681	3.37%
Instance-8	200	300	1 (min) – 4 (max)	15	10162	13728	2338	4052	60.23%
Instance-9	250	250	1 (min) – 1 (max)	15	190844	225528	18178	76372	59.98%
Instance-10	300	300	1 (min) – 1 (max)	15	102710	116809	9262	21167	79.39%
Instance-11	150	200	1 (min) – 4 (max)	15	71939	79554	4116	44153	38.62%
Instance-12	200	250	1 (min) – 4 (max)	15	110122	124597	4682	65447	40.57%
Instance-13	250	275	1 (min) – 3 (max)	15	174572	185178	4972	97859	43.94%
Instance-14	300	300	1 (min) – 1 (max)	15	-	-	-	118811	-

show that RL-OI performed well for small and medium size instances except for Instance-6. This method performed well mainly due to the Reinforcement Learning system and the parameters used to escape local minima. However, RL-OI performed poorly for large size instances except for Instance-7. As shown in Table 5.4, the performance decreases dramatically for more than 200 areas. The main factors of this poor performance are the sizes of the instances and the short time available. In addition, an improved initial solution would probably increase the performance.

TABLE 5.5: The performance of RL-OI algorithm without time limits

Name	time	best found	best known	gap (%)
Instance-1	1.88(s)	1396	1396	0%
Instance-2	1.86(s)	1498	1498	0%
Instance-3	2.3(s)	7672	7672	0%
Instance-4	3.38(m)	**13952**	14024	**-0.52%**
Instance-5	1.51(m)	**694**	698	**-0.58%**
Instance-6	10.54(m)	**1733**	2159	**-19.73%**
Instance-7	8.51(m)	**31218**	31681	**-1.46%**
Instance-8	28.13(m)	**4033**	4052	**-0.47%**
Instance-9	> 30(m)	77892	76372	1.95%
Instance-10	> 30(m)	43542	21167	51.39%
Instance-11	> 30(m)	51358	44153	14.03%
Instance-12	> 30(m)	76298	65447	14.22%
Instance-13	> 30(m)	145233	97859	32.62%
Instance-14	> 30(m)	198573	118811	40.17%

In further analysis, the performance of RL-OI is tested without time restrictions and the results are presented in Table 5.5. As shown in Table 5.5 the solutions are remarkably improved. Specifically, in Instance-4/5/6/7/8 the RL-OI method managed to achieve even a lower cost than the best known so far. In general, the RL-OI method provided

best known and better than known solutions for all small and medium size instances, as well as for Instance-7 and Instance-8 (see Apendix C for solutions in detail). For the remaining large instances, the algorithm lacks time efficiency and did not yield near best known solutions. This is probably because of the huge sizes of the instances.

5.2.1 Analysis of small size instances

In Figure 5.1, the cost versus time is shown when low level heuristics are applied jointly and separately in small size instances. In addition, Fiugre 5.2 shows the utilisation rate of each operator when combined together. The plots indicate that:

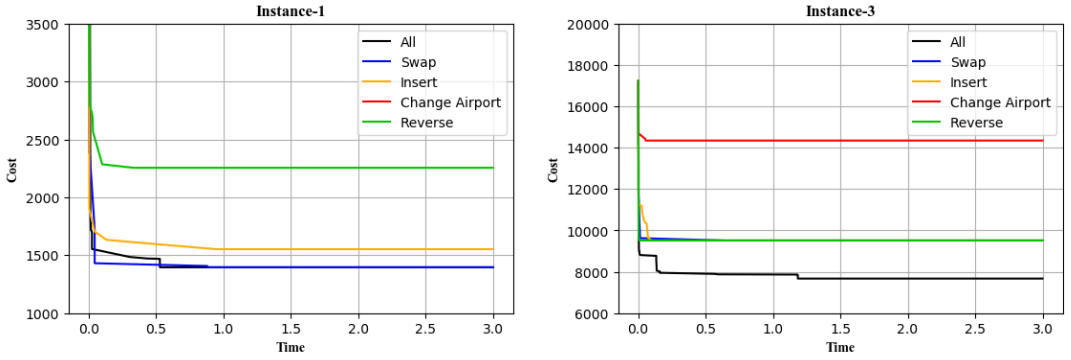

FIGURE 5.1: Plots of cost versus time with combined operators and without for small size instances

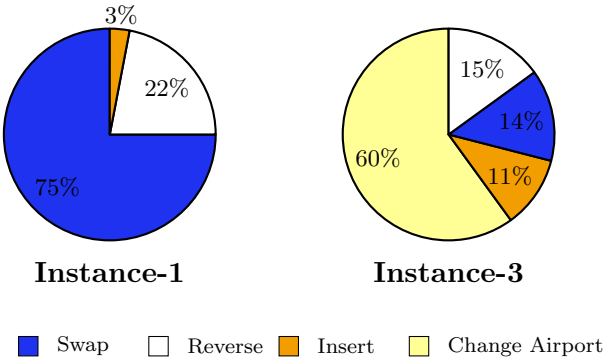

FIGURE 5.2: Utilisation rate of each operator for small size instances

- In Instance-1, the best known solution is reached by combining operators and by Swap operator. Insert operator managed to reach close enough to the best known solution, while Reverse operator had the worst performance. Change Airport operator is not used because Instance-1 has only one airport in each area. The

utilisation rate plot shows that Swap operator contributed the most. Although, Reverse operator had worse performance than Insert operator when applied separately, it contributed more than Insert operator when all operators were combined.

- In Instance-3, the best known solution is reached only when all operators are combined. Swap, Insert, and Reverse operators achieved the same performance and yielded a cost not far from the best known solution. Change Airport operator did not perform well when applied separately. The utilisation rate plot indicates that Change Airport operator had the greatest contribution among all operators in reaching the best known solution. Reverse and Swap operators had a similar contribution, whereas Insert operator had the least contribution.

5.2.2 Analysis of medium size instances

In Figure 5.3, low level heuristics are applied jointly and separately in medium size instances. Furthermore, the utilisation rate of each operators are shown in Figure 5.4. Change Airport operator did not improve significantly the solution when applied separately. Overall, the plots show that:

- In Instance-4, the best solution is given by the combination of all operators. Insert and Reverse operators have a similar performance, while Swap operator is slightly worse. Change Airport operator has the greatest utilisation rate as shown in Figure 5.4, followed by Reverse and Swap operators and, Insert operator follows last.

- In Instance-5, the combination of all operators lead to the best cost. Reverse operator had the best performance when applied separately followed by Insert and Swap operators. According to the utilisation rate figure, Reverse operator contributes the most, Insert and Change Airport operators are equally utilised and, Swap operator is the last one.

- In Instance-6, the best cost is achieved through the combination of all operators. When applied separately, Insert and Swap operators stood out from the rest operators. Likewise, as can been seen from Figure 5.4, Insert operator has the greatest utilisation rate followed by Swap operator. Reverse operator has a utilisation rate of 16% and Change Airport is following last.

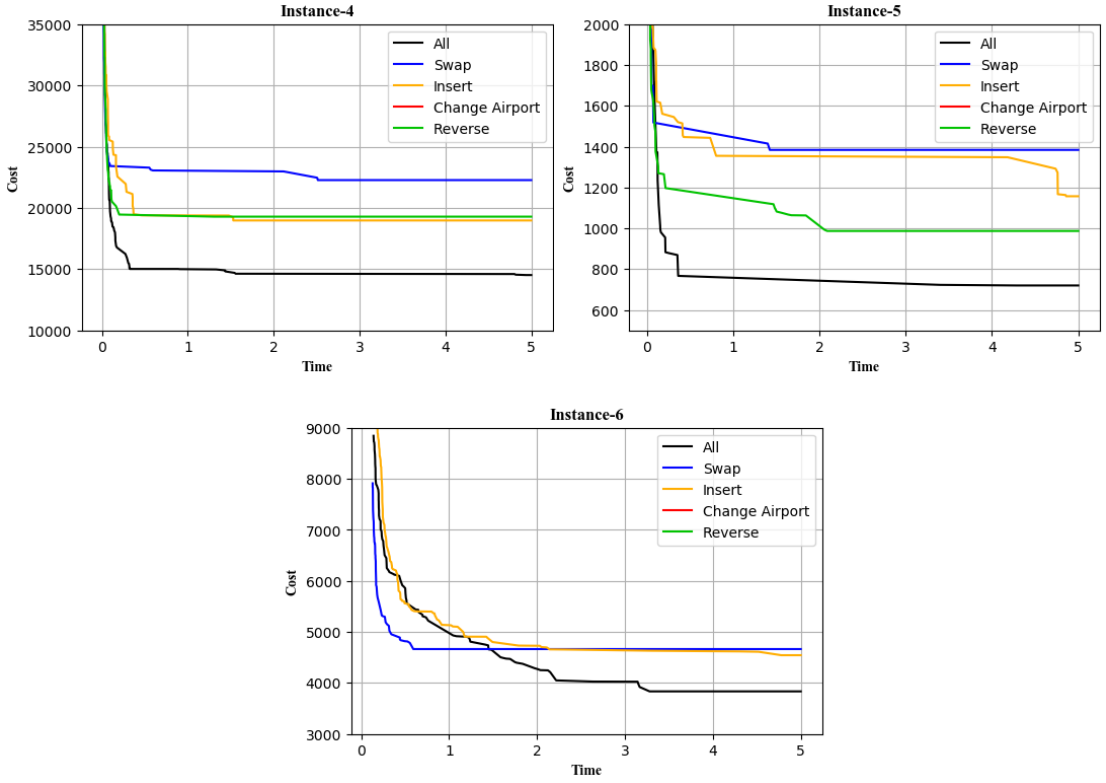

FIGURE 5.3: Plots of cost versus time with combined operators and without for medium size instances

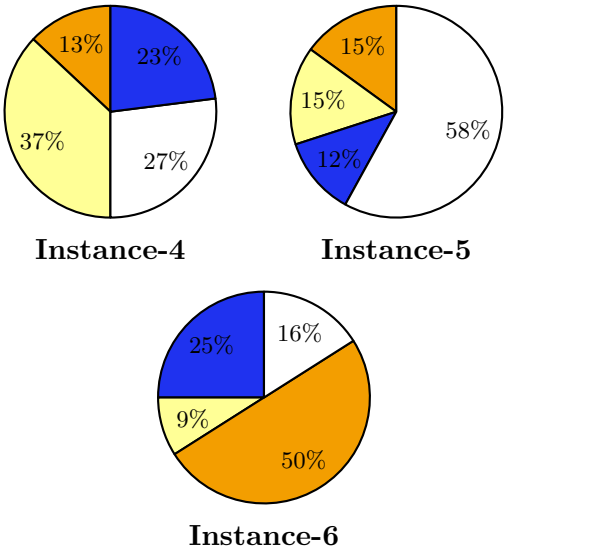

FIGURE 5.4: Utilisation rate of each operator for medium size instances

5.2.3 Analysis of large size instances

In Figure 5.5 and Figure 5.7, the cost versus time is shown when low level heuristics are applied jointly and separately. Likewise in medium size instances, Change Airport operator is not improving greatly the cost when applied separately. Moreover, Figure 5.6 and Figure 5.8 illustrate the utilisation rates of operators for large size instances. The summary of plots suggest that:

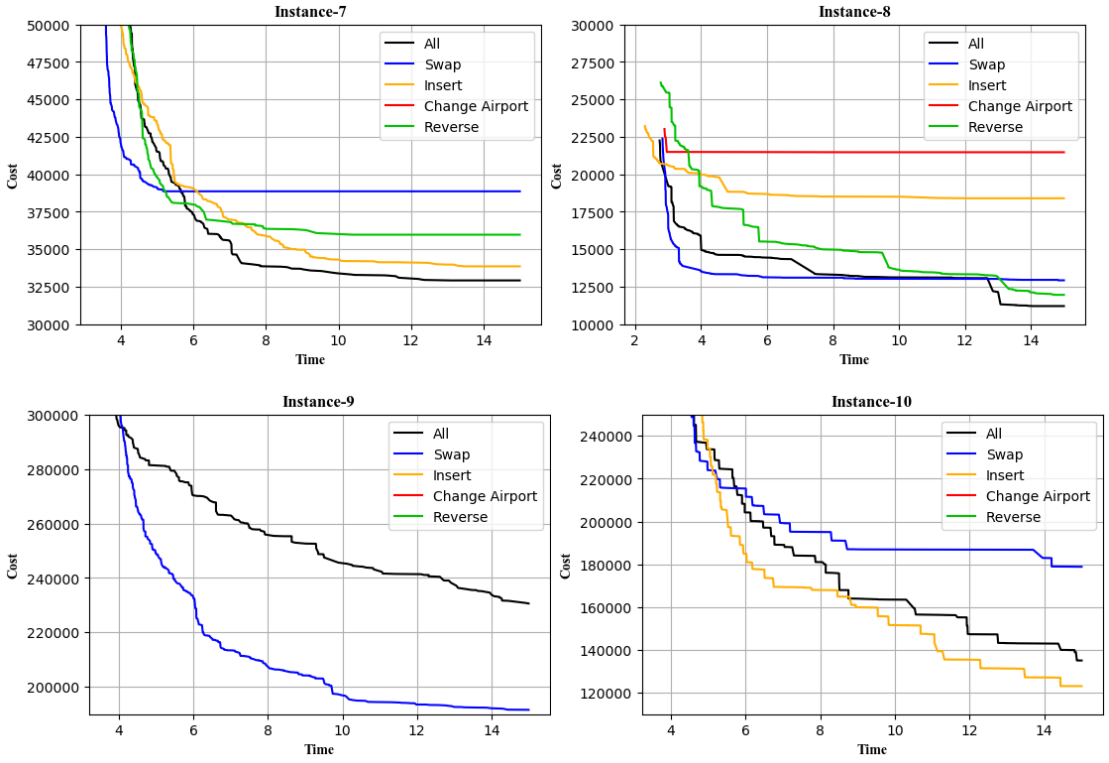

FIGURE 5.5: Plots of cost versus time with combined operators and without for large size instances-7/8/9/10

- In Instance-7, Figure 5.5 shows that the best solution is achieved via combination of operators. Insert operator yields the second best solution and then Reverse and Swap operators follow respectively. As shown in Figure 5.6, Reverse operator contributes the most in cost reduction and is followed by Insert and Swap operators respectively.

- In Instance-8, according to Figure 5.5, the best cost is yielded through the combination of all operators. Reverse and Swap operators generate the next two best solutions, whereas Insert and Change Airport operators generate worse solutions.

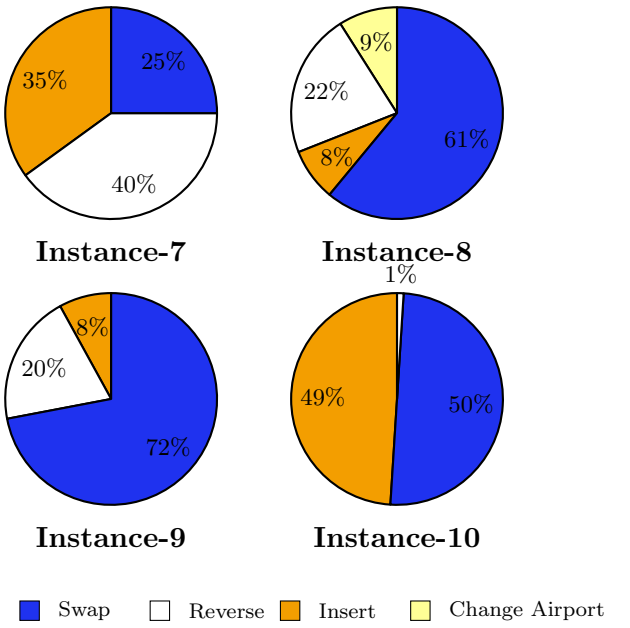

FIGURE 5.6: Utilisation rate of each operator for Instances-7/8/9/10

As can be seen in Figure 5.6, the utilisation rate of Swap operator is the greatest, Reverse operator comes second with a rate of 22%, Change Airport follows with 9% and, Insert is last with 8%.

- In Instance-9, Figure 5.5 suggests that Swap operator achieved a better solution than the combined operators. The rest operators did not decrease significantly the cost when applied separately. Figure 5.6 shows that, indeed, Swap operator is the greatest contributor when operators are combined. Reverse and Insert operators obtained a utilisation rate of 20% and 8% respectively. Instance-9 has only one airport in each area and hence, Change Airport operator is not utilised.

- In Instance-10, Insert operator obtained a better solution than combined operators as shown in Figure 5.5. The third best solution was achieved by Swap operator. A similar behaviour is noticed in Figure 5.6, where Swap and Insert operators are the main contributors when all operators are combined. Reverse operator contributes by only 1%. Instance-10 has only one airport in each area and therefore, Change Airport operator is not utilised.

- In Instance-11, Figure 5.7 shows that Swap operator obtained a better solution than combined operators. When operators are combined, Swap operator has the greatest utilisation rate, as shown in Figure 5.8. The next greatest contributor is Change Airport operator and is followed by Reverse and Insert operators.

- In Instance-12, Swap operator yielded a better cost than combined operators, as shown in Figure 5.7. According to Figure 5.8, the utilisation rate of Swap operator is the greatest with a value of 80%. Then, Change Airport follows with a rate of 14%, Reverse operator with 5% and, Insert operator with only 1%.

- In Instance-13, Figure 5.7 illustrates the best cost that was achieved by Swap operator. When operators are combined, the greatest contribution is made by Swap operator, as shown in Figure 5.8. The next biggest contributor is Change Airport operator, Reverse operator follows last and, Insert operator is not utilised.

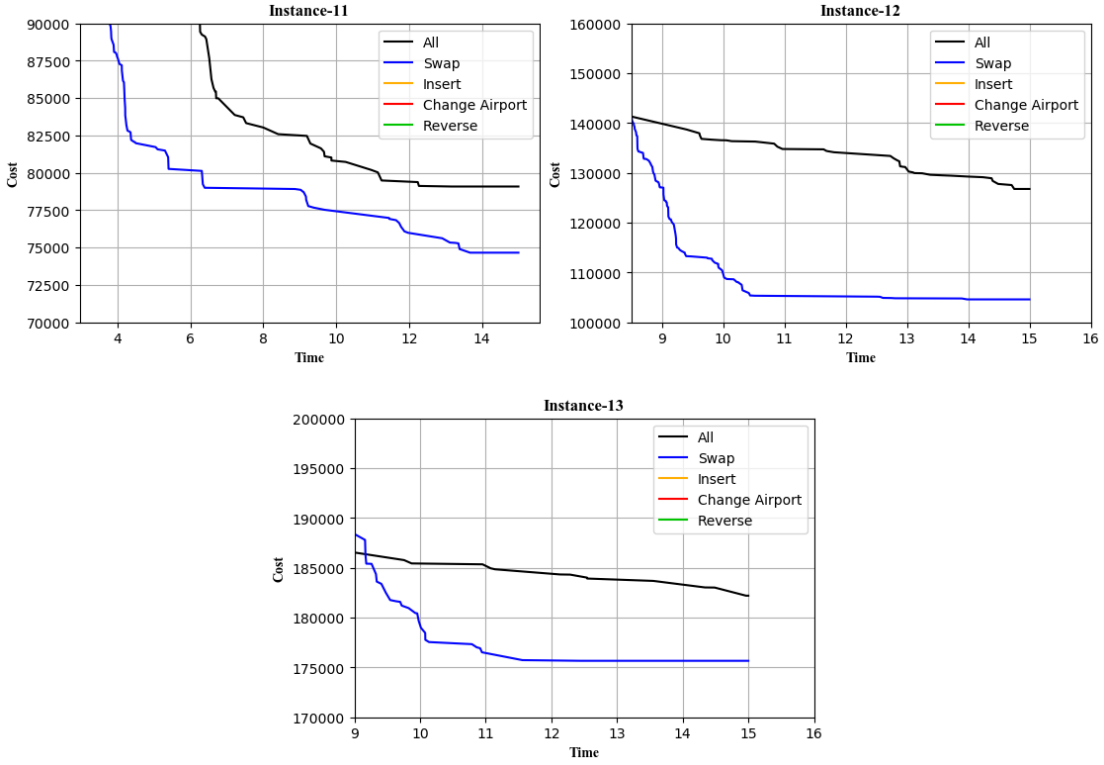

FIGURE 5.7: Plots of cost versus time with combined operators and without for large size instances-11/12/13

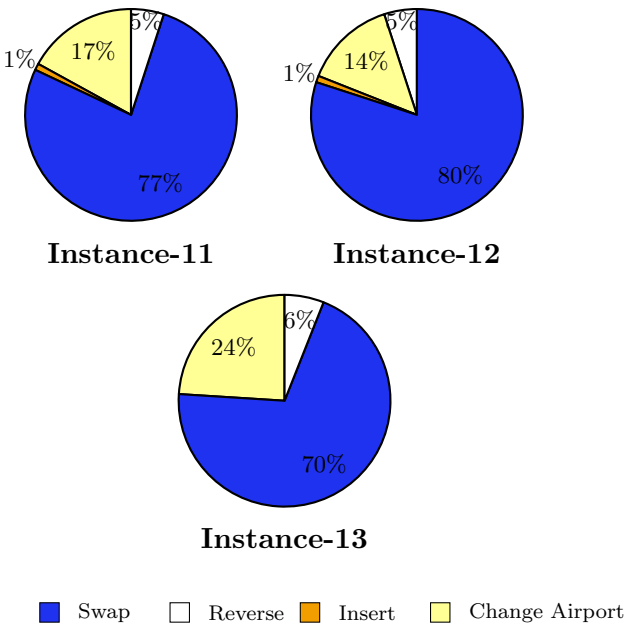

FIGURE 5.8: Utilisation rate of each operator for Instances-11/12/13

Chapter 6

Conclusion

6.1 Summary of Work

In this dissertation, Hyper-heuristic methods are used to solve the Kiwi.com problem. In total, 6 Hyper-heuristic methods with the same set of LLHs (Swap, Insert, Reverse, Change Airport) were implemented; SR-IE, SR-GD, RD, RP-IE, RPD, and RL-OI. Among the methods, RL-OI was proved to have the best performance for most of the instances. Overall, the combination of all LLHs achieves a high performance for small, medium, and some large size instances. In small size instances, Swap and Change Airport are the main contributors, whereas for medium size instances the main contributors are Insert, Reverse, and Change Airport. In large size instances, the combination of LLHs is recommended only for Instance-7 and Instance-8, where the main contributors are Swap and Reverse. For Instance-9 and onward, Swap LLH achieves a better performance than the combined LLHs with an exception in Instance-10, where Insert LLH has the best performance. As far as the feasibility improvement is concerned, Hyper-heuristics with different sets of Guided LLHs improved significantly the speed of generating a feasible solution. For Instance-14, however, they did not succeed to yield a feasible solution within time limit. In addition, one of the most significant achievements of this study was that the RL-OI method yielded best known and better than known solutions up to Instance-8, when time restrictions were removed. Yet, the algorithm failed to achieve a high performance in terms of solutions and time efficiency for large size instances.

6.2 Future Work

In this section, some ideas are presented for future study based on the findings and the weaknesses of this dissertation. These ideas are the following:

- **Development of different Initialisation methods:** The initialisation method used in this study was the random approach. However, the implementation of a greedy approach would likely improve the results as proved in [40].
- **Experimentation with more sophisticated Hyper-heuristic methods:** In this work, the Hyper-heuristic methods implemented were mainly random based except for RL-OI method. It would be interesting to investigate the performance with more sophisticated methods such as, Record-to-Record Travel, Dominance-based and Random Descent hyper-heuristic, Sequence-based Selection Hyper-heuristic, Tabu Search, and Choice Function. Particularly, the SSHH method is very interesting so as to discover any hidden higher performances, if any exist, when LLHs are applied in a controlled sequence.
- **Formulation of an Exact method:** The formulation of an exact method for Kiwi.com problem could be implemented by means of Integer Programming. On the one hand, it would be extremely complex and inefficient in terms of running time for a such complicated problem. On the other hand, it would be helpful to find the optimal solutions for the released instances.
- **Implementation of the algorithm in $C++$ program language:** Although, Python 3.7 language is used in this work, which is fast and relatively easy to understand, better results would be achieved by using $C++$ language. The greatest benefit of $C++$ language is that it is one of the fastest programming languages available.

Appendix A

Code Listings

Main_Driver.py

```
import Read_File
import Calculations
import time
import Improve_Feas_A as IFeas
#Use Improve_Feas_A for Instances: {1,2,3,4,5,6,8,10}
#Use Improve_Feas_B for Instances: {7}
#Use Improve_Feas_C for Instances: {9,11,12,13}
#Use Improve_Feas_D for Instances: {14}
import SR_IE as Method
#Methods available: {SR_IE, RD, RP_IE, RPD, SR_GD, RL_OI}

start_time = time.time()
#Type below the maximum time allowed (i.e. 2.99, 4.99, 14.99)
run_time = 2.99
#Type below the problem instance (i.e. "1.in", "2.in", etc.)
File = Read_File.ReadFile("1.in")

areas_n = int(File[0][0][0])
start_air = File[0][0][1]
areas_and_airports = File[0][1:]
mydict = Read_File.CreateDistances(File[1], areas_n)

sol = list(Calculations.Initial_Sol(start_air, areas_and_airports, mydict,
    areas_n))
IFeas.ImproveFeasibility(sol, areas_and_airports, mydict, areas_n)
```

```
Method.HH(sol, areas_and_airports, mydict, areas_n, start_time, run_time)
print('Run Time: ', time.time() - start_time)
print('Feasibility :',Calculations.CalculateFeasibiity(sol,mydict, areas_n)[0])
print('Total Cost :',Calculations.CalculateCost(sol,mydict, areas_n))
print("Solution areas: ", sol[0])
print("Solution airports: ", sol[1])
```

Read_File.py

```
def ReadFile(f_name):
    dist = []
    line_nu = -1
    with open(f_name) as infile:
        for line in infile:
            line_nu += 1
            if line_nu == 0:
                index = int(line.split()[0]) * 2 + 1
            if line_nu >= index:
                temp = line.split()
                temp[2] = int(temp[2])
                temp[3] = int(temp[3])
                dist.append(temp)
            else:
                dist.append(line.split())
        info = dist[0:int(dist[0][0])*2+1]
        flights = dist[int(dist[0][0])*2+1:]
    return info, flights

def CreateDistances(File, areas_n):
    mydict = {}
    for i in File:
        if ((i[0],i[1],i[2])) in mydict.keys():
            if mydict[(i[0],i[1],i[2])] > i[3]:
                mydict[(i[0],i[1],i[2])] = i[3]
        else:
            mydict[(i[0],i[1],i[2])] = i[3]
    return mydict
```

Calculations.py

```
import Operators

def Initial_Sol(start_air, areas_and_airports, mydict, areas_n):
    start_area = 0
    for i in range(areas_n):
        if start_air in areas_and_airports[i*2 + 1]:
            start_area = i
            break
    solArea = list(range(areas_n + 1))
    Operators.Swap(solArea, 0, start_area)
    solArea[areas_n] = start_area
    solAirport = [start_air]
    for i in range(1,areas_n + 1):
        solAirport.append(areas_and_airports[solArea[i]*2 + 1][0])
    return solArea, solAirport

def CalculateFeasibiity(sol,mydict, areas_n):
    feas = 0
    indexes = []
    for i in range(areas_n):
        if ((sol[1][i],sol[1][i+1],i+1)) in mydict.keys() or
    ((sol[1][i],sol[1][i+1],0)) in mydict.keys():
            next
        else:
            feas = feas + 1
            if i > 0 and i < areas_n:
                indexes.append(i)
    if len(indexes) == 0:
        return feas, 1, areas_n - 2
    else:
        ix = min(indexes)
        ixx = max(indexes)
    return feas, ix, ixx

def PFeas(sol,mydict, areas_n,x,y,Rev,CAL):
    feas = 0
    if CAL == True:
        if ((sol[1][x-1],sol[1][x],x)) not in mydict.keys() and
    ((sol[1][x-1],sol[1][x],0)) not in mydict.keys():
```

```
            feas += 1
            return feas
        if ((sol[1][x],sol[1][x+1],x+1)) not in mydict.keys() and
    ((sol[1][x],sol[1][x+1],0)) not in mydict.keys():
            feas += 1
            return feas
        if ((sol[1][y-1],sol[1][y],y)) not in mydict.keys() and
    ((sol[1][y-1],sol[1][y],0)) not in mydict.keys():
            feas += 1
            return feas
    else:
        if Rev == False:
            if ((sol[1][x-1],sol[1][x],x)) not in mydict.keys() and
    ((sol[1][x-1],sol[1][x],0)) not in mydict.keys():
                feas += 1
                return feas
            if ((sol[1][x],sol[1][x+1],x+1)) not in mydict.keys() and
    ((sol[1][x],sol[1][x+1],0)) not in mydict.keys():
                feas += 1
                return feas
            if ((sol[1][y-1],sol[1][y],y)) not in mydict.keys() and
    ((sol[1][y-1],sol[1][y],0)) not in mydict.keys():
                feas += 1
                return feas
            if ((sol[1][y],sol[1][y+1],y+1)) not in mydict.keys() and
    ((sol[1][y],sol[1][y+1],0)) not in mydict.keys():
                feas += 1
                return feas
        else:
            if x > y:
                t = x
                x = y
                y = t
            for i in range(x-2,y+1):
                if ((sol[1][i],sol[1][i+1],i+1)) not in mydict.keys() and
    ((sol[1][i],sol[1][i+1],0)) not in mydict.keys():
                    feas += 1
                    return feas
    return feas
```

```
def CalculateCost(sol,mydict, areas_n):
    cost = 0
    for i in range(areas_n):
        if ((sol[1][i],sol[1][i+1],i+1)) in mydict.keys():
            temp1 = mydict[(sol[1][i],sol[1][i+1],i+1)]
        else:
            temp1 = 500000
        if ((sol[1][i],sol[1][i+1],0)) in mydict.keys():
            temp2 = mydict[(sol[1][i],sol[1][i+1],0)]
        else:
            temp2 = 500000
        if temp1 < temp2:
            cost += temp1
        else:
            cost += temp2
    return cost

def PCost(sol,mydict, areas_n,x,y,Rev,CAL):
    cost = 0
    pen = 500000
    if CAL == True:
        if ((sol[1][x-1],sol[1][x],x)) in mydict.keys():
            temp1 = mydict[(sol[1][x-1],sol[1][x],x)]
        else:
            temp1 = pen
        if ((sol[1][x-1],sol[1][x],0)) in mydict.keys():
            temp2 = mydict[(sol[1][x-1],sol[1][x],0)]
        else:
            temp2 = pen
        if temp1 < temp2:
            cost += temp1
        else:
            cost += temp2
        if ((sol[1][x],sol[1][x+1],x+1)) in mydict.keys():
            temp1 = mydict[(sol[1][x],sol[1][x+1],x+1)]
        else:
            temp1 = pen
        if ((sol[1][x],sol[1][x+1],0)) in mydict.keys():
            temp2 = mydict[(sol[1][x],sol[1][x+1],0)]
        else:
            temp2 = pen
```

```
            if temp1 < temp2:
                cost += temp1
            else:
                cost += temp2
            if ((sol[1][y-1],sol[1][y],y)) in mydict.keys():
                temp1 = mydict[(sol[1][y-1],sol[1][y],y)]
            else:
                temp1 = pen
            if ((sol[1][y-1],sol[1][y],0)) in mydict.keys():
                temp2 = mydict[(sol[1][y-1],sol[1][y],0)]
            else:
                temp2 = pen
            if temp1 < temp2:
                cost += temp1
            else:
                cost += temp2
        else:
            if Rev == False:
                if ((sol[1][x-1],sol[1][x],x)) in mydict.keys():
                    temp1 = mydict[(sol[1][x-1],sol[1][x],x)]
                else:
                    temp1 = pen
                if ((sol[1][x-1],sol[1][x],0)) in mydict.keys():
                    temp2 = mydict[(sol[1][x-1],sol[1][x],0)]
                else:
                    temp2 = pen
                if temp1 < temp2:
                    cost += temp1
                else:
                    cost += temp2
                if ((sol[1][x],sol[1][x+1],x+1)) in mydict.keys():
                    temp1 = mydict[(sol[1][x],sol[1][x+1],x+1)]
                else:
                    temp1 = pen
                if ((sol[1][x],sol[1][x+1],0)) in mydict.keys():
                    temp2 = mydict[(sol[1][x],sol[1][x+1],0)]
                else:
                    temp2 = pen
                if temp1 < temp2:
                    cost += temp1
                else:
```

```
            cost += temp2
        if ((sol[1][y-1],sol[1][y],y)) in mydict.keys():
            temp1 = mydict[(sol[1][y-1],sol[1][y],y)]
        else:
            temp1 = pen
        if ((sol[1][y-1],sol[1][y],0)) in mydict.keys():
            temp2 = mydict[(sol[1][y-1],sol[1][y],0)]
        else:
            temp2 = pen
        if temp1 < temp2:
            cost += temp1
        else:
            cost += temp2
        if ((sol[1][y],sol[1][y+1],y+1)) in mydict.keys():
            temp1 = mydict[(sol[1][y],sol[1][y+1],y+1)]
        else:
            temp1 = pen
        if ((sol[1][y],sol[1][y+1],0)) in mydict.keys():
            temp2 = mydict[(sol[1][y],sol[1][y+1],0)]
        else:
            temp2 = pen
        if temp1 < temp2:
            cost += temp1
        else:
            cost += temp2
    else:
        if x > y:
            t = x
            x = y
            y = t
        for i in range(x-2,y+1):
            if ((sol[1][i],sol[1][i+1],i+1)) in mydict.keys():
                temp1 = mydict[(sol[1][i],sol[1][i+1],i+1)]
            else:
                temp1 = pen
            if ((sol[1][i],sol[1][i+1],0)) in mydict.keys():
                temp2 = mydict[(sol[1][i],sol[1][i+1],0)]
            else:
                temp2 = pen
            if temp1 < temp2:
                cost += temp1
```

```
                else:
                    cost += temp2
    return cost
```

Operators.py

```
import numpy as np

def Swap(arr,a,b):
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp

def Swap2(arr,arr2,a,b):
    temp = arr[a]
    temp2 = arr2[a]
    arr[a] = arr[b]
    arr2[a] = arr2[b]
    arr[b] = temp
    arr2[b] = temp2

def Reverse(arr,arr2,a,b):
    if a > b:
        t = a
        a = b
        b = t
    temp = arr
    temp2 = arr2
    c = a
    for i in temp[b:a:-1]:
        arr[a+1] = i
        a += 1
    for i in temp2[b:c:-1]:
        arr2[c+1] = i
        c += 1

def Insert(arr,arr2,a,b):
    temp = arr[a]
    temp2 = arr2[a]
    arr.remove(temp)
    arr2.remove(temp2)
```

```
    arr.insert(b,temp)
    arr2.insert(b,temp2)

def Change_Airport(arr,arr2,a,b,areas_and_airports):
    np.random.shuffle(areas_and_airports[arr2[a]*2 + 1])
    np.random.shuffle(areas_and_airports[arr2[b]*2 + 1])
    arr[a] = areas_and_airports[arr2[a]*2 + 1][0]
    arr[b] = areas_and_airports[arr2[b]*2 + 1][0]
```

Improve_Feas_A.py

```
import numpy as np
import Operators
import Calculations

def ImproveFeasibility(sol, areas_and_airports, mydict, areas_n):
    LLH0 = lambda: Operators.Swap2(sol[0],sol[1],x,y)
    LLH1 = lambda: Operators.Insert(sol[0],sol[1],x,y)
    LLH2 = lambda: Operators.Swap2(sol[0],sol[1],x,yy)
    LLH3 = lambda: Operators.Insert(sol[0],sol[1],x,yy)
    LLHS = [LLH0,LLH1,LLH2,LLH3]
    feas = list(Calculations.CalculateFeasibiity(sol,mydict,areas_n))
    while feas[0] != 0:
        x = np.random.randint(1,areas_n)
        yy = np.random.randint(feas[1],feas[2]+1)
        y = np.random.randint(1,areas_n)
        l = np.random.randint(0,len(LLHS))
        copy_0 = sol[0].copy()
        copy_1 = sol[1].copy()
        LLHS[l]()
        Operators.Change_Airport(sol[1],sol[0],x,feas[2],areas_and_airports)
        Operators.Change_Airport(sol[1],sol[0],x,feas[2]+1,areas_and_airports)
        feas_new = list(Calculations.CalculateFeasibiity(sol,mydict,areas_n))
        if feas_new[0] <= feas[0]:
            feas = feas_new
        else:
            sol[0] = copy_0
            sol[1] = copy_1
```

Improve_Feas_B.py

```
import numpy as np
import Operators
import Calculations

def ImproveFeasibility(sol, areas_and_airports, mydict, areas_n):
    LLH0 = lambda: Operators.Swap2(sol[0],sol[1],x,y)
    LLH1 = lambda: Operators.Reverse(sol[0],sol[1],feas[1],feas[2])
    LLH2 = lambda: Operators.Reverse(sol[0],sol[1],r,rr)
    LLHS = [LLH0,LLH1,LLH2]
    feas = list(Calculations.CalculateFeasibiity(sol,mydict,areas_n))
    while feas[0] != 0:
        x = np.random.randint(1,areas_n)
        r = np.random.randint(1,areas_n-2)
        rr = np.random.randint(r+2,areas_n)
        y = np.random.randint(1,areas_n)
        l = np.random.randint(0,len(LLHS))
        copy_0 = sol[0].copy()
        copy_1 = sol[1].copy()
        LLHS[l]()
        Operators.Change_Airport(sol[1],sol[0],x,feas[2],areas_and_airports)
        Operators.Change_Airport(sol[1],sol[0],x,feas[2]+1,areas_and_airports)
        feas_new = list(Calculations.CalculateFeasibiity(sol,mydict,areas_n))
        if feas_new[0] <= feas[0]:
            feas = feas_new
        else:
            sol[0] = copy_0
            sol[1] = copy_1
```

Improve_Feas_C.py

```
import numpy as np
import Operators
import Calculations

def ImproveFeasibility(sol, areas_and_airports, mydict, areas_n):
    LLH0 = lambda: Operators.Swap2(sol[0],sol[1],x,feas[1])
    LLH1 = lambda: Operators.Swap2(sol[0],sol[1],x,feas[2])
    LLH2 = lambda: Operators.Reverse(sol[0],sol[1],feas[1],yy)
    LLHS = [LLH0,LLH1,LLH2]
```

```
    feas = list(Calculations.CalculateFeasibiity(sol,mydict,areas_n))
    while feas[0] != 0:
        x = np.random.randint(1,areas_n)
        if feas[2] == feas[1]:
            yy = feas[1]
        else:
            yy = np.random.randint(feas[1],feas[2])
        l = np.random.randint(0,len(LLHS))
        copy_0 = sol[0].copy()
        copy_1 = sol[1].copy()
        LLHS[l]()
        Operators.Change_Airport(sol[1],sol[0],feas[1],yy,areas_and_airports)
        Operators.Change_Airport(sol[1],sol[0],x,feas[2]+1,areas_and_airports)
        feas_new = list(Calculations.CalculateFeasibiity(sol,mydict,areas_n))
        if feas_new[0] <= feas[0]:
            feas = feas_new
        else:
            sol[0] = copy_0
            sol[1] = copy_1
```

Improve_Feas_D.py

```
import numpy as np
import Operators
import Calculations

def ImproveFeasibility(sol, areas_and_airports, mydict, areas_n):
    LLH0 = lambda: Operators.Swap2(sol[0],sol[1],x,feas[1])
    LLH1 = lambda: Operators.Swap2(sol[0],sol[1],x,feas[2])
    LLH2 = lambda: Operators.Insert(sol[0],sol[1],x,y)
    LLH3 = lambda: Operators.Swap2(sol[0],sol[1],x,y)
    LLHS = [LLH0,LLH1,LLH2,LLH3]
    feas = list(Calculations.CalculateFeasibiity(sol,mydict,areas_n))
    while feas[0] != 0:
        x = np.random.randint(1,areas_n)
        y = np.random.randint(1,areas_n)
        l = np.random.randint(0,len(LLHS))
        copy_0 = sol[0].copy()
        copy_1 = sol[1].copy()
        LLHS[l]()
        feas_new = list(Calculations.CalculateFeasibiity(sol,mydict,areas_n))
```

```
        if feas_new[0] <= feas[0]:
            feas = feas_new
        else:
            sol[0] = copy_0
            sol[1] = copy_1
```

SR_IE.py

```
import Calculations
import Operators
import time
import numpy as np

def HH(sol, areas_and_airports, mydict, areas_n, start_time, run_time):
    cost = Calculations.CalculateCost(sol,mydict,areas_n)
    LLH0 = lambda: Operators.Swap2(sol[0],sol[1],x,y)
    LLH1 = lambda: Operators.Insert(sol[0],sol[1],x,y)
    LLH2 = lambda: Operators.Reverse(sol[0],sol[1],r,rr)
    LLH3 = lambda:
    Operators.Change_Airport(sol[1],sol[0],x,yy,areas_and_airports)
    LLHS = [LLH0,LLH1,LLH2,LLH3]
    while time.time() - start_time < run_time:
        Rev = False
        CAL = False
        copy_0 = sol[0].copy()
        copy_1 = sol[1].copy()
        x = np.random.randint(1,areas_n)
        y = np.random.randint(1,areas_n)
        yy = np.random.randint(1,areas_n+1)
        r = np.random.randint(1,areas_n-2)
        rr = np.random.randint(r+2,areas_n)
        l = np.random.randint(0,len(LLHS))
        c = float(np.random.random(1))
        if l == 3 and yy == areas_n:
            CAL = True
            y = areas_n
        if l == 1 or l == 2:
            Rev = True
            x = r
            y = rr
        else:
```

```
            Rev = False
        LLHS[l]()
        if l != 3 and c >= 0.5:
            LLH3()
        f_after = Calculations.PFeas(sol,mydict,areas_n,x,y,Rev,CAL)
        if f_after == 0:
            cost_new = Calculations.CalculateCost(sol,mydict,areas_n)
            if cost_new <= cost:
                cost = cost_new
            else:
                sol[0] = copy_0
                sol[1] = copy_1
        else:
            sol[0] = copy_0
            sol[1] = copy_1
```

SR_GD.py

```
import Calculations
import Operators
import time
import numpy as np

def HH(sol, areas_and_airports, mydict, areas_n, start_time, run_time):
    cost = Calculations.CalculateCost(sol,mydict,areas_n)
    cost_best = cost
    solBest_0 = sol[0].copy()
    solBest_1 = sol[1].copy()
    LLH0 = lambda: Operators.Swap2(sol[0],sol[1],x,y)
    LLH1 = lambda: Operators.Insert(sol[0],sol[1],x,y)
    LLH2 = lambda: Operators.Reverse(sol[0],sol[1],r,rr)
    LLH3 = lambda:
    Operators.Change_Airport(sol[1],sol[0],x,yy,areas_and_airports)
    LLHS = [LLH0,LLH1,LLH2,LLH3]
    f = 1396   #Type the optimal value
    delta = cost - f
    while time.time() - start_time < run_time:
        time_now = time.time() - start_time
        Rev = False
        CAL = False
        copy_0 = sol[0].copy()
```

```
        copy_1 = sol[1].copy()
        x = np.random.randint(1,areas_n)
        y = np.random.randint(1,areas_n)
        yy = np.random.randint(1,areas_n+1)
        r = np.random.randint(1,areas_n-2)
        rr = np.random.randint(r+2,areas_n)
        l = np.random.randint(0,len(LLHS))
        c = float(np.random.random(1))
        if l == 3 and yy == areas_n:
            CAL = True
            y = areas_n
        if l == 1 or l == 2:
            Rev = True
            x = r
            y = rr
        else:
            Rev = False
        LLHS[l]()
        if l != 3 and c >= 0.5:
            LLH3()
        f_after = Calculations.PFeas(sol,mydict,areas_n,x,y,Rev,CAL)
        if f_after == 0:
            cost_new = Calculations.CalculateCost(sol,mydict,areas_n)
            t = f + delta * (1 - (time_now / run_time))
            if cost_new <= t:
                if cost_new <= cost_best:
                    cost_best = cost_new
                    solBest_0 = sol[0].copy()
                    solBest_1 = sol[1].copy()
            else:
                sol[0] = copy_0
                sol[1] = copy_1
        else:
            sol[0] = copy_0
            sol[1] = copy_1
    sol[0] = solBest_0
    sol[1] = solBest_1
```

RD.py

```
import Calculations
import Operators
import time
import numpy as np

def HH(sol, areas_and_airports, mydict, areas_n, start_time, run_time):
    cost = Calculations.CalculateCost(sol,mydict,areas_n)
    LLH0 = lambda: Operators.Swap2(sol[0],sol[1],x,y)
    LLH1 = lambda: Operators.Insert(sol[0],sol[1],x,y)
    LLH2 = lambda: Operators.Reverse(sol[0],sol[1],r,rr)
    LLH3 = lambda:
    Operators.Change_Airport(sol[1],sol[0],x,yy,areas_and_airports)
    LLHS = [LLH0,LLH1,LLH2,LLH3]
    while time.time() - start_time < run_time:
        Rev = False
        CAL = False
        copy_0 = sol[0].copy()
        copy_1 = sol[1].copy()
        x = np.random.randint(1,areas_n)
        y = np.random.randint(1,areas_n)
        yy = np.random.randint(1,areas_n+1)
        r = np.random.randint(1,areas_n-2)
        rr = np.random.randint(r+2,areas_n)
        l = np.random.randint(0,len(LLHS))
        c = float(np.random.random(1))
        if l == 3 and yy == areas_n:
            CAL = True
            y = areas_n
        if l == 1 or l == 2:
            Rev = True
            x = r
            y = rr
        else:
            Rev = False
        LLHS[l]()
        if l != 3 and c >= 0.5:
            LLH3()
        f_after = Calculations.PFeas(sol,mydict,areas_n,x,y,Rev,CAL)
        if f_after == 0:
```

```
            cost_new = Calculations.CalculateCost(sol,mydict,areas_n)
            while (cost_new < cost and f_after == 0) and time.time() -
    start_time + 0.5 < run_time:
                cost = cost_new
                copy_0 = sol[0].copy()
                copy_1 = sol[1].copy()
                x = np.random.randint(1,areas_n)
                y = np.random.randint(1,areas_n)
                yy = np.random.randint(1,areas_n+1)
                r = np.random.randint(1,areas_n-2)
                rr = np.random.randint(r+2,areas_n)
                c = float(np.random.random(1))
                if l == 3 and yy == areas_n:
                    CAL = True
                    y = areas_n
                if l != 2:
                    Rev = False
                else:
                    Rev = True
                    x = r
                    y = rr
                LLHS[l]()
                if l != 3 and c >= 0.5:
                    LLH3()
                f_after = Calculations.PFeas(sol,mydict,areas_n,x,y,Rev,CAL)
                cost_new = Calculations.CalculateCost(sol,mydict,areas_n)
            else:
                sol[0] = copy_0
                sol[1] = copy_1
        else:
            sol[0] = copy_0
            sol[1] = copy_1
```

RP_IE.py

```
import Calculations
import Operators
import time
import numpy as np

def HH(sol, areas_and_airports, mydict, areas_n, start_time, run_time):
```

```
cost = Calculations.CalculateCost(sol,mydict,areas_n)
LLH0 = lambda: Operators.Swap2(sol[0],sol[1],x,y)
LLH1 = lambda: Operators.Insert(sol[0],sol[1],x,y)
LLH2 = lambda: Operators.Reverse(sol[0],sol[1],r,rr)
LLH3 = lambda:
Operators.Change_Airport(sol[1],sol[0],x,yy,areas_and_airports)
LLHS = [LLH0,LLH1,LLH2,LLH3]
perm = np.random.randint(0,4,size=1000000)
iteration = -1
while time.time() - start_time < run_time:
    Rev = False
    CAL = False
    iteration += 1
    copy_0 = sol[0].copy()
    copy_1 = sol[1].copy()
    x = np.random.randint(1,areas_n)
    y = np.random.randint(1,areas_n)
    yy = np.random.randint(1,areas_n+1)
    r = np.random.randint(1,areas_n-2)
    rr = np.random.randint(r+2,areas_n)
    c = float(np.random.random(1))
    if perm[iteration] == 3 and yy == areas_n:
        CAL = True
        y = areas_n
    if perm[iteration] == 1 or perm[iteration] == 2:
        Rev = True
        x = r
        y = rr
    else:
        Rev = False
    LLHS[perm[iteration]]()
    if perm[iteration] != 3 and c >= 0.5:
        LLH3()
    f_after = Calculations.PFeas(sol,mydict,areas_n,x,y,Rev,CAL)
    if f_after == 0:
        cost_new = Calculations.CalculateCost(sol,mydict,areas_n)
        if cost_new <= cost:
            cost = cost_new
        else:
            sol[0] = copy_0
            sol[1] = copy_1
```

```
        else:
            sol[0] = copy_0
            sol[1] = copy_1
```

RPD.py

```
import Calculations
import Operators
import time
import numpy as np

def HH(sol, areas_and_airports, mydict, areas_n, start_time, run_time):
    cost = Calculations.CalculateCost(sol,mydict,areas_n)
    LLH0 = lambda: Operators.Swap2(sol[0],sol[1],x,y)
    LLH1 = lambda: Operators.Insert(sol[0],sol[1],x,y)
    LLH2 = lambda: Operators.Reverse(sol[0],sol[1],r,rr)
    LLH3 = lambda:
    Operators.Change_Airport(sol[1],sol[0],x,yy,areas_and_airports)
    LLHS = [LLH0,LLH1,LLH2,LLH3]
    perm = np.random.randint(0,4,size=1000000)
    iteration = -1
    while time.time() - start_time < run_time:
        Rev = False
        CAL = False
        iteration += 1
        copy_0 = sol[0].copy()
        copy_1 = sol[1].copy()
        x = np.random.randint(1,areas_n)
        y = np.random.randint(1,areas_n)
        yy = np.random.randint(1,areas_n+1)
        r = np.random.randint(1,areas_n-2)
        rr = np.random.randint(r+2,areas_n)
        c = float(np.random.random(1))
        if perm[iteration] == 3 and yy == areas_n:
            CAL = True
            y = areas_n
        if perm[iteration] == 1 or perm[iteration] == 2:
            Rev = True
            x = r
            y = rr
        else:
```

```
            Rev = False
        LLHS[perm[iteration]]()
        if LLHS[perm[iteration]] != LLH3 and c >= 0.5:
            LLH3()
        f_after = Calculations.PFeas(sol,mydict,areas_n,x,y,Rev,CAL)
        if f_after == 0:
            cost_new = Calculations.CalculateCost(sol,mydict,areas_n)
            while (cost_new < cost and f_after == 0) and time.time() -
    start_time + 0.5 < run_time:
                cost = cost_new
                copy_0 = sol[0].copy()
                copy_1 = sol[1].copy()
                x = np.random.randint(1,areas_n)
                y = np.random.randint(1,areas_n)
                yy = np.random.randint(1,areas_n+1)
                r = np.random.randint(1,areas_n-2)
                rr = np.random.randint(r+2,areas_n)
                c = float(np.random.random(1))
                if perm[iteration] == 3 and yy == areas_n:
                    CAL = True
                    y = areas_n
                if perm[iteration] != 2:
                    Rev = False
                else:
                    Rev = True
                LLHS[perm[iteration]]()
                if LLHS[perm[iteration]] != LLH3 and c >= 0.5:
                    LLH3()
                f_after = Calculations.PFeas(sol,mydict,areas_n,x,y,Rev,CAL)
                cost_new = Calculations.CalculateCost(sol,mydict,areas_n)
            else:
                sol[0] = copy_0
                sol[1] = copy_1
        else:
            sol[0] = copy_0
            sol[1] = copy_1
```

RL_OI.py

```
import Calculations
import Operators
import time
import numpy as np

def HH(sol, areas_and_airports, mydict, areas_n, start_time, run_time):
    cost = Calculations.CalculateCost(sol,mydict,areas_n)
    LLH0 = lambda: Operators.Swap2(sol[0],sol[1],x,y)
    LLH1 = lambda: Operators.Insert(sol[0],sol[1],x,y)
    LLH2 = lambda:
    Operators.Change_Airport(sol[1],sol[0],x,yy,areas_and_airports)
    LLH3 = lambda: Operators.Reverse(sol[0],sol[1],x,y)
    LLHS = [LLH0,LLH1,LLH2,LLH3]
    scores = [0.5,0.5,0.5,0.5]
    cost_best = cost
    solBest_0 = sol[0].copy()
    solBest_1 = sol[1].copy()
    iteration = -1
    b = 0
    c = 2
    #c value depends on the instance
    #A value within the range of 1.05 and 4 is recommended
    a = 1000
    #For small instances a = 1000 is recommended
    #For medium instances a = 5000 is recommended
    #For large instances a = 30000 is recommended
    #Without time restrictions a = 1000 is recommended for Instance-9 and
    onward
    while time.time() - start_time < run_time:
        iteration += 1
        d = False
        CAL = False
        Rev = False
        copy_0 = sol[0].copy()
        copy_1 = sol[1].copy()
        x = np.random.randint(1,areas_n)
        y = np.random.randint(1,areas_n)
        yy = np.random.randint(1,areas_n+1)
        if d == True:
```

```
            scores = [0.5,0.5,0.5,0.5]
        best = max(scores)
        index = scores.index(best)
        if index == 2 and yy == areas_n:
            CAL = True
        if index == 2:
            y = yy
        if index == 1 or index == 3:
            Rev = True
        cost_before = Calculations.PCost(sol,mydict,areas_n,x,y,Rev,CAL)
        LLHS[index]()
        f_after = Calculations.PFeas(sol,mydict,areas_n,x,y,Rev,CAL)
        if b > a:
            d = True
        if f_after == 0:
            cost_after = Calculations.PCost(sol,mydict,areas_n,x,y,Rev,CAL)
            if cost_after < cost_before or (b > a and cost_after <
    cost_before*c):
                cost_new = Calculations.CalculateCost(sol,mydict,areas_n)
                scores[index] += 0.1 * iteration*2*0.015
                b = 0
                if cost_new < cost_best:
                    cost_best = cost_new
                    solBest_0 = sol[0].copy()
                    solBest_1 = sol[1].copy()
            else:
                sol[0] = copy_0
                sol[1] = copy_1
                scores[index] -= 0.1 * iteration/2*0.01
                b += 1
        else:
            sol[0] = copy_0
            sol[1] = copy_1
            scores[index] -= 0.1 * iteration/2*0.0125
            b += 10
    sol[0] = solBest_0
    sol[1] = solBest_1
```

Appendix B

Test Instances

All Instances can be found at:

https://code.kiwi.com/travelling-salesman-challenge-2-0-wrap-up-cb4d81e36d5b

The first few lines of Instance-1 follows:

10 AB0

Zona_0

AB0

Zona_1

AB1

Zona_2

AB2

Zona_3

AB3

Zona_4

AB4

Zona_5

AB5

Zona_6

AB6

Zona_7

AB7

Zona_8

AB8

Zona_9

AB9

AB0 AB1 1 1

AB0 AB2 1 3619

AB0 AB3 1 4618

AB0 AB4 1 2127

AB1 AB0 1 5639

AB1 AB2 1 4217

AB1 AB3 1 1305

AB1 AB4 1 2484

AB2 AB0 1 4079

AB2 AB1 1 3171

AB2 AB3 1 4461

AB2 AB4 1 2820

AB3 AB0 1 2689

AB3 AB1 1 4667

AB3 AB2 1 5890

AB3 AB4 1 4244

Appendix C

Best Solutions for Instances-4/5/6/7/8

Best Solution for Instance-4:

Starting Area: 27

Starting Airport: 'GDN'

Total Cost: 13952

Sol_{Area} = {27, 19, 18, 10, 30, 11, 35, 26, 9, 13, 20, 25, 3, 39, 16, 28, 34, 12, 36, 17, 33, 1, 8, 32, 15, 31, 4, 6, 24, 21, 0, 22, 14, 7, 37, 5, 23, 29, 38, 2, 27}

$Sol_{Airport}$ = {'GDN', 'VNO', 'RIX', 'TLL', 'LED', 'HEL', 'BLE', 'OSL', 'CPH', 'HAM', 'LUX', 'EIN', 'ANR', 'LTN', 'ORK', 'OPO', 'SDR', 'LYS', 'GVA', 'VCE', 'LJU', 'VIE', 'BRQ', 'BTS', 'BUD', 'BEG', 'SJJ', 'DBV', 'TGD', 'SKP', 'TIA', 'MLA', 'HER', 'AKT', 'IST', 'VAR', 'KIV', 'IAS', 'LWO', 'BQT', 'WAW'}

Best Solution for Instance-5:

Starting Area: 0

Starting Airport: 'RCF'

Total Cost: 694

Sol_{Area} = {0, 4, 5, 9, 10, 13, 7, 8, 3, 2, 14, 16, 21, 26, 25, 24, 20, 19, 15, 29, 17, 18, 23, 22, 28, 27, 45, 42, 41, 36, 31, 30, 44, 32, 37, 38, 33, 34, 35, 39, 40, 43, 12, 11, 6, 1, 0}

$Sol_{Airport}$ = {'RCF', 'AON', 'BPA', 'JRJ', 'EHO', 'UCF', 'LQP', 'RZA', 'FWF', 'LZS', 'MWE', 'LGI', 'MLL', 'AVC', 'MUD', 'SMM', 'GSW', 'JSF', 'EGM', 'YMJ', 'HGY', 'FNO', 'FED', 'QUZ', 'SQZ', 'PIG', 'SZR', 'PSS', 'WYX', 'QNY', 'KLW', 'GMU', 'BBZ', 'IXY', 'QNL', 'CLQ', 'UUZ', 'LDS', 'GMM', 'MWT', 'FIE', 'JLM', 'JAQ', 'EIF', 'MGY', 'EQP', 'RCF'}

Best Solution for Instance-6:

Starting Area: 0

Starting Airport: 'VHK'

Total Cost: 1733

Sol_{Area} = {0, 68, 67, 63, 95, 58, 47, 62, 76, 65, 66, 71, 42, 43, 38, 39, 40, 41, 64, 69, 70, 75, 74, 73, 72, 33, 44, 45, 46, 31, 32, 37, 36, 35, 34, 93, 92, 77, 88, 89, 84, 85, 80, 81, 82, 87, 86, 91, 90, 79, 78, 83, 59, 49, 48, 53, 52, 57, 56, 60, 61, 51, 50, 55, 54, 94, 17, 5, 4, 15, 14, 9, 10, 11, 12, 13, 2, 3, 8, 7, 6, 1, 29, 28, 18, 19, 23, 24, 25, 21, 20, 30, 16, 26, 27, 22, 0}

$Sol_{Airport}$ = {'VHK', 'YQK', 'IXQ', 'JSP', 'FVJ', 'CWC', 'VTQ', 'PYW', 'NKE', 'UAX', 'AJK', 'BJW', 'BCT', 'LDS', 'CTQ', 'PJI', 'BML', 'YUT', 'QIU', 'VDG', 'EXJ', 'DBD', 'YMG', 'IFS', 'ZVP', 'ZML', 'EFU', 'LWM', 'CZA', 'OTN', 'XSR', 'QXW', 'CJF', 'WYU', 'KKE', 'LFE', 'AFF', 'YHH', 'SAN', 'TPR', 'NTJ', 'UHR', 'RXB', 'UXA', 'UYZ', 'IEI', 'MUF', 'LGF', 'RSE', 'EUV', 'VKQ', 'UXC', 'CPQ', 'GYZ', 'QEZ', 'CUV', 'KIS', 'AWR', 'HPO', 'DQJ', 'WJI', 'MXV', 'TXU', 'PFG', 'WTV', 'ZZY', 'DUK', 'LTN', 'LNP', 'RLV', 'UDQ', 'GCX', 'WBF', 'YCE', 'RNS', 'IWR', 'ASN', 'BYT', 'CCR', 'NOZ', 'UNM', 'ZVF', 'UDY', 'IKC', 'GCC', 'IXJ', 'BSK', 'BHK', 'IXR', 'XEH', 'FPD', 'QLA', 'UWL', 'URB', 'UGD', 'RNK', 'EJS'}

Best Solution for Instance-7:

Starting Area: 104

Starting Airport: 'AHG'

Total Cost: 31218

Sol_{Area} = {104, 52, 78, 31, 58, 111, 27, 79, 134, 45, 125, 97, 43, 143, 42, 75, 91, 87, 65, 74, 99, 28, 64, 92, 55, 62, 9, 80, 132, 141, 142, 100, 25, 46, 128, 113, 29, 34, 82, 49, 130, 110, 83, 61, 121, 118, 138, 93, 7, 131, 109, 70, 36, 123, 137, 101, 33, 13, 145, 84, 76, 69, 77, 59, 56, 14, 30, 1, 88, 8, 149, 21, 85, 60, 106, 72, 71, 135, 40, 16, 117, 107, 19, 146, 103, 67, 57, 66, 114, 48, 73, 136, 38, 26, 4, 140, 144, 139, 119, 3, 18, 120, 127, 98, 37, 32, 112, 129, 147, 17, 6, 23, 2, 124, 116, 81, 35, 68, 126, 122, 94, 5, 115, 0, 44, 53, 24, 41, 10, 105, 89, 12, 90, 96, 39, 95, 148, 20, 102, 63, 133, 54, 50, 11, 51, 22, 108, 15, 86, 47, 104}

$Sol_{Airport}$ = {'AHG', 'ARO', 'FWO', 'DKV', 'BTY', 'AKZ', 'AEN', 'DIZ', 'FJA', 'BKI', 'EZG', 'GWJ', 'AMR', 'ERX', 'CTU', 'COH', 'FRJ', 'FCD', 'EFY', 'FHS', 'EPQ', 'DDV', 'FAY', 'FFF', 'FDV', 'DNK', 'DUH', 'BXV', 'ETU', 'DHV', 'DVS', 'DER', 'DWI', 'CRI', 'GRU', 'BUF', 'FJO', 'EIH', 'GMM', 'FMZ', 'BPW', 'HCP', 'BAQ', 'HAF', 'DPO', 'FKV', 'BNK', 'FOE', 'ALM', 'DSM', 'DIB', 'BRF', 'GAR', 'DAU', 'ATB', 'GSL', 'FXP', 'BSP', 'BZH', 'GUT', 'ALX', 'BWF', 'HAP', 'ECE', 'CWC', 'AZS', 'BAJ', 'FIO', 'BHB', 'FOS', 'HFU', 'GWN', 'FRW', 'BZT', 'BBW', 'LSE', 'JLI', 'MRT', 'JUZ', 'NAD', 'INT', 'LIR', 'MGM', 'HON', 'LOU', 'NCU', 'KXI', 'JOQ', 'KXN', 'MAZ', 'MQY', 'IBM', 'MCH', 'KPR', 'HID', 'IWU', 'LHG', 'LAL', 'KYK', 'IIH', 'MED', 'KLO', 'KXM', 'JMP', 'HMD', 'HWB', 'NIZ', 'MOR', 'HXU', 'HVV', 'JHC', 'KLS', 'LAA', 'JGW', 'ICN', 'MIJ', 'JUG', 'IRN', 'LUI', 'LPA', 'IVN', 'JWN', 'MLJ', 'KMH', 'IDG', 'NFL', 'LYI', 'LBK', 'HTJ', 'KKA', 'ITE', 'HRV', 'MAS', 'MDX', 'MON', 'KXF', 'LID', 'LJA', 'KON', 'LZD', 'NFB', 'IRE', 'IOM', 'JOO', 'MYY', 'JBB', 'HUV', 'JQD', 'HGD', 'HFX', 'AHG'}

Best Solution for Instance-8:

Starting Area: 156

Starting Airport: 'AEW'

Total Cost: 4033

Sol_{Area} = {156, 57, 49, 187, 163, 77, 181, 37, 60, 46, 111, 138, 11, 27, 41, 53, 145, 159, 14, 184, 178, 197, 56, 47, 124, 121, 174, 165, 132, 76, 92, 33, 149, 16, 152, 151, 96, 58, 157, 65, 141, 119, 148, 10, 109, 43, 30, 103, 191, 17, 86, 108, 36, 146, 158, 183, 34, 55, 54, 154, 21, 190, 19, 42, 199, 8, 84, 161, 160, 2, 64, 196, 153, 12, 74, 62, 169, 135, 104, 167, 114, 85, 31, 129, 26, 150, 73, 95, 59, 71, 131, 87, 15, 155, 112, 0, 80, 127, 61, 5, 137, 93, 69, 168, 83, 120, 140, 182, 100, 23, 72, 18, 192, 173, 170, 40, 198, 81, 128, 110, 13, 50, 164, 143, 98, 195, 29, 22, 39, 193, 179, 32, 189, 123, 89, 176, 126, 79, 188, 1, 4, 25, 144, 20, 63, 177, 130, 94, 45, 90, 82, 147, 101, 88, 99, 24, 180, 51, 118, 9, 67, 194, 66, 105, 35, 3, 136, 44, 102, 91, 186, 70, 52, 116, 28, 122, 97, 133, 68, 139, 171, 38, 162, 106, 78, 113, 166, 134, 172, 185, 125, 117, 6, 107, 115, 48, 142, 7, 75, 175, 156}

$Sol_{Airport}$ = {'AEW', 'AUO', 'ZMT', 'IDB', 'LVN', 'FCJ', 'OAE', 'FMC', 'OBE', 'AEH', 'PDI', 'TRJ', 'OVD', 'ZZP', 'WRZ', 'AZF', 'OLQ', 'XMD', 'WCD', 'IHD', 'FWA', 'RLT', 'FCP', 'NPF', 'NFP', 'NBR', 'EDA', 'BYB', 'BKB', 'PDY', 'BIN', 'WUE', 'DWQ', 'NRS', 'VZK', 'BCU', 'QUI', 'CHF', 'MAE', 'MRE', 'AFH', 'AUL', 'QXU', 'EOB', 'MFT', 'KIC', 'LRL', 'RYJ', 'QZU', 'PFU', 'RWO', 'BGU', 'LTI', 'HNP', 'OCB', 'JHO', 'AME', 'WLL', 'HGU', 'DNZ', 'MSW', 'OIO', 'VGV', 'JFU', 'PEU', 'EOW', 'BQL', 'XLY', 'NPT', 'BPY', 'RGK', 'KIL', 'XGM', 'IYZ', 'DVQ', 'EBC', 'CKW', 'OOM', 'BNL', 'JKB', 'OTQ', 'JBS', 'SXJ', 'ILI', 'JQL', 'TGY', 'PCD', 'CJM', 'CHK', 'ECS', 'IID', 'EBY', 'ALA', 'CZJ', 'MYR', 'FKP', 'DRO', 'ANZ', 'QRL', 'SVZ', 'TLA', 'BFZ', 'EXV', 'JCY', 'LNC', 'KBN', 'GRH', 'IPE', 'MMN', 'AUJ', 'RVY', 'RJM', 'CGR', 'PUG', 'HMM', 'DCY', 'LRU', 'IVF', 'IUN', 'OPC', 'JRT', 'MHW', 'LTF', 'RAA', 'UPZ', 'VFT', 'GKF', 'JEL', 'AKF', 'MVV', 'FYA', 'LII', 'KIW', 'WRL', 'EBK', 'JSR', 'MDB', 'VNJ', 'GHI', 'HKD', 'JZU', 'LKE', 'RHQ', 'XSY', 'ASF', 'HPZ', 'EOG', 'QBR', 'PUW', 'PAV', 'PFI', 'WUL', 'PNH', 'WCS', 'XSF', 'XLT', 'FGF', 'RRT', 'PRO', 'WPH', 'LTP', 'RAR', 'TBS', 'FIG', 'DDZ', 'EGV', 'NZN', 'NVV', 'OXH', 'NOR', 'ROM', 'UZI', 'JAH', 'DSR', 'YHH', 'EQV', 'HTD', 'BZR', 'BMD', 'CUU', 'EHZ', 'PEP', 'JBJ', 'PJS', 'RLE', 'BLJ', 'QMS', 'FAO', 'JIM', 'MYZ', 'CAA', 'HIX', 'CSW', 'DSN', 'BAB', 'XNG', 'GCF', 'HIN', 'OVC', 'DCB', 'AEW'}

Bibliography

[1] Marcken C. D. Computational complexity of air travel planning, 2003. Public Notes on Computational Complexity. Retrieved June, 14, 2019 from `http://www.demarcken.org/carl/papers/`.

[2] OpenFlights. The air-traveling salesman, 2015. Retrieved June,13,2019 from `https://sites.google.com/site/travellingcudasalesman/`.

[3] Rafael Marques, Lus Russo, and Nuno Roma. Flying tourist problem: Flight time and cost minimization in complex routes. *Expert Systems with Applications*, 130: 172–187, 2019.

[4] Irina Dumitrescu and Thomas Stützle. Combinations of local search and exact algorithms. In Stefano Cagnoni, Colin G. Johnson, Juan J. Romero Cardalda, Elena Marchiori, David W. Corne, Jean-Arcady Meyer, Jens Gottlieb, Martin Middendorf, Agnès Guillot, Günther R. Raidl, and Emma Hart, editors, *Applications of Evolutionary Computing*, pages 211–223, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[5] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231 – 247, 1992.

[6] Akshay Vyas, Dashmeet Chawla, Anuj Mehta, Abhishek Chelawat, and Urjita Thakar. Genetic algorithm for solving the traveling salesman problem using neighbor-based constructive crossover operator. *INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES AND RESEARCH TECHNOLOGY (IJESRT)*, 7:101–110, 04 2018.

[7] Rasmus Rasmussen. TSP in spreadsheets - a guided tour. *International Review of Economics Education*, 10(1):94–116, 2011.

[8] Thanaboon Saradatta and Pisut Pongchairerks. A time-dependent atsp with time window and precedence constraints in air travel. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 9(2-3):149–153, 2017.

[9] Petrica C. Pop and Serban Iordache. A hybrid heuristic approach for solving the generalized traveling salesman problem. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 481–488, New York, NY, USA, 2011. ACM.

[10] Valentina Cacchiani, Albert Einstein Fernandes Muritiba, Marcos Negreiros, and Paolo Toth. A multistart heuristic for the equality generalized traveling salesman problem. *Networks*, 57(3):231–239, 2011.

[11] H. S. Lope and L. S. Coelho. Particle swarn optimization with fast local search for the blind traveling salesman problem. In *Fifth International Conference on Hybrid Intelligent Systems (HIS'05)*, pages 245–250, Nov 2005.

[12] Adrian Dumitrescu and Joseph S.B. Mitchell. Approximation algorithms for tsp with neighborhoods in the plane. *Journal of Algorithms*, 48(1):135 – 159, 2003. Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms.

[13] Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M. Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):367–371, 1995.

[14] Malik Muneeb Abid and Muhammad Iqbal. Heuristic approaches to solve traveling salesman problem. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, 15:390–396, 09 2015.

[15] Jakob Puchinger and Günther R. Raidl. Combining Metaheuristics and Exact Algorithms in Combinatorial Optimization: A Survey and Classification. In José R. Álvarez José Mira, editor, *First International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2005*, volume 3562, pages 41–53, Las Palmas, Spain, June 2005. Springer-Verlag.

[16] Marco Dorigo, Mauro Birattari, and Thomas Sttzle. Ant colony optimization artificial ants as a computational intelligence technique. *IEEE Comput. Intell. Mag*, 1:28–39, 2006.

[17] Jalandhar Regional Campus. A review of parameters for improving the performance of particle swarm optimization. *International Journal of Hybrid Information Technology*, 8(4):7–14, 2015.

[18] Leena N. Ahmed, Ender zcan, and Ahmed Kheiri. Solving high school timetabling problems worldwide using selection hyper-heuristics. *Expert Systems with Applications*, 42(13):5463 – 5471, 2015.

[19] Diogo Duque, José Aleixo Cruz, Henrique Lopes Cardoso, and Eugénio Oliveira. Optimizing meta-heuristics for the time-dependent tsp applied to air travels. In Hujun Yin, David Camacho, Paulo Novais, and Antonio J. Tallón-Ballesteros, editors, *Intelligent Data Engineering and Automated Learning – IDEAL 2018*, pages 730–739, Cham, 2018. Springer International Publishing.

[20] Xiang Li, Jiandong Zhou, and Xiande Zhao. Travel itinerary problem. *Transportation Research Part B: Methodological*, 91:332 – 343, 2016.

[21] Joshua Touyz. The travelling tourist problem: A mixed heuristic approach, 2013. Retrieved June,23,2019 from `http://issuu.com/jgetouyz/docs`.

[22] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37(1):79 – 100, 1988.

[23] Pierre Hansen and Nenad Mladenovi. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449 – 467, 2001.

[24] Marc Pirlot. General local search methods. *European Journal of Operational Research*, 92(3):493 – 511, 1996.

[25] Helena R. Loureno, Olivier C. Martin, and Thomas Sttzle. Iterated local search. In *Handbook of Metaheuristics, volume 57 of International Series in Operations Research and Management Science*, pages 321–353. Kluwer Academic Publishers, 2002.

[26] Ender Özcan, Burak Bilgin, and Emin Erkan Korkmaz. A comprehensive analysis of hyper-heuristics. *Intell. Data Anal.*, 12(1):3–23, January 2008.

[27] Alhanof Almutairi, Ender Özcan, Ahmed Kheiri, and Warren G. Jackson. Performance of selection hyper-heuristics on the extended hyflex domains. In Tadeusz Czachórski, Erol Gelenbe, Krzysztof Grochla, and Ricardo Lent, editors, *Computer and Information Sciences*, pages 154–162, Cham, 2016. Springer International Publishing.

[28] Ruibin Bai, Jacek Blazewicz, Edmund K. Burke, Graham Kendall, and Barry McCollum. A simulated annealing hyper-heuristic methodology for flexible decision support. *4OR*, 10:43–66, 2012.

[29] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, Dec 2013.

[30] Konstantin Chakhlevitch and Peter Cowling. *Hyperheuristics: Recent Developments*, pages 3–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[31] Peter Cowling, Graham Kendall, and Eric Soubeiga. A hyperheuristic approach to scheduling a sales summit. In Edmund Burke and Wilhelm Erben, editors, *Practice and Theory of Automated Timetabling III*, pages 176–190, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[32] Ender Özcan, Burak Bilgin, and Emin Erkan Korkmaz. Hill climbers and mutational heuristics in hyperheuristics. In Thomas Philip Runarsson, Hans-Georg Beyer, Edmund Burke, Juan J. Merelo-Guervós, L. Darrell Whitley, and Xin Yao, editors, *Parallel Problem Solving from Nature - PPSN IX*, pages 202–211, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[33] John H. Drake, Ender Özcan, and Edmund K. Burke. An improved choice function heuristic selection for cross domain heuristic search. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *Parallel Problem Solving from Nature - PPSN XII*, pages 307–316, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[34] G. Kendall and M. Mohamad. Channel assignment optimisation using a hyper-heuristic. In *IEEE Conference on Cybernetics and Intelligent Systems, 2004.*, volume 2, pages 791–796, Dec 2004.

[35] Graham Kendall and Naimah Mohd Hussin. An investigation of a tabu-search-based hyper-heuristic for examination timetabling. In Graham Kendall, Edmund K. Burke, Sanja Petrovic, and Michel Gendreau, editors, *Multidisciplinary Scheduling: Theory and Applications*, pages 309–328, Boston, MA, 2005. Springer US.

[36] Graham Kendall and Naimah Mohd Hussin. A tabu search hyper-heuristic approach to the examination timetabling problem at the mara university of technology. In Edmund Burke and Michael Trick, editors, *Practice and Theory of Automated Timetabling V*, pages 270–293, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[37] Gunter Dueck. New optimization heuristics: The great deluge algorithm and the record-to-record travel. *Journal of Computational Physics*, 104(1):86 – 92, 1993.

[38] Edmund K. Burke and Yuri Bykov. A late acceptance strategy in hill-climbing for examination timetabling problems. In *In Proceedings of the conference on the Practice and Theory of Automated Timetabling(PATAT*, 2008.

[39] John H. Drake, Ahmed Kheiri, Ender zcan, and Edmund K. Burke. Recent advances in selection hyper-heuristics. *European Journal of Operational Research*, 2019.

[40] Yaroslav Pylyavskyy Maab Alrasheed, Wafaa Mohammed and Ahmed Kheiri. Local search heuristic for the optimisation of flight connections. In *International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)*, (In Press).