OptPlatform: *Metaheuristic Optimisation Framework for Solving Complex Real-World Problems*

A thesis submitted for the degree of
Doctor of Philosophy

by

Ivars Dzalbs

Department of Electronic and Computer engineering

Brunel University London

January 2021

DECLARATION OF AUTHORSHIP

I, Ivars Dzalbs, declare that the work in the dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific references in the text, the work is the candidate's own work. The work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Signature: ____________________ Date: 31/05/2021

ABSTRACT

We optimise daily, whether that is planning a round trip that visits the most attractions within a given holiday budget or just taking a train instead of driving a car in a rush hour. Many problems, just like these, are solved by individuals as part of our daily schedule, and they are effortless and straightforward. If we now scale that to many individuals with many different schedules, like a school timetable, we get to a point where it is just not feasible or practical to solve by hand. In such instances, optimisation methods are used to obtain an optimal solution.

In this thesis, a practical approach to optimisation has been taken by developing an optimisation platform with all the necessary tools to be used by practitioners who are not necessarily familiar with the subject of optimisation.

First, a high-performance metaheuristic optimisation framework (MOF) called OptPlatform is implemented, and the versatility and performance are evaluated across multiple benchmarks and real-world optimisation problems. Results show that, compared to competing MOFs, the **OptPlatform outperforms in both the solution quality and computation time**.

Second, the most suitable hardware platform for OptPlatform is determined by an in-depth analysis of Ant Colony Optimisation scaling across CPU, GPU and enterprise Xeon Phi. Contrary to the common benchmark problems used in the literature, the supply chain problem solved could not scale on GPUs.

Third, a variety of metaheuristics are implemented into OptPlatform. Including, a new metaheuristic based on Imperialist Competitive Algorithm (ICA), called ICA with Independence and Constrained Assimilation (ICAwICA) is proposed. The ICAwICA was compared against two different types of benchmark problems, and results show the versatile application of the algorithm, **matching and in some cases outperforming the custom-tuned approaches.**

Finally, essential MOF features like automatic algorithm selection and tuning, lacking on existing frameworks, are implemented in OptPlatform. Two novel approaches are proposed and compared to existing methods. Results indicate the **superiority of the implemented tuning algorithms within constrained tuning budget environment**.

ACKNOWLEDGMENTS

Firstly, I would acknowledge my family and their unconditional support throughout my studies.

I would like to also thank my supervisory team, principal supervisor Dr. Tatiana Kalganova for providing me with the opportunity of pursuing PhD; and second supervisor Dr. Hongying Meng for the moral support.

Final thanks go to Tony Grichnik for providing endless interesting real-world problems to solve.

CONTENTS

Abstract .... **i**
Acknowledgments .... **ii**
Contents .... **iii**
List of Tables .... **vi**
List of Figures .... **ix**
List of Abbreviations .... **xiv**
1. Introduction **1**
1.1. Motivation 2
1.2. Thesis Contributions 2
1.3. List of publications 4
1.4. Thesis contents 4
2. Literature review **7**
2.1. Optimization methods 7
2.1.1. Heuristics 9
2.1.2. Metaheuristics 10
2.2. Metaheuristic Optimization Frameworks (MOFs) 25
2.2.1. MOF trends and limitations 33
2.3. Optimization Problems 35
2.3.1. Benchmark problems 35
2.3.2. Real-world problems 39
2.4. Summary 49
3. Optimization platform (OptPlatform) **50**
3.1. Target users and requirements 50
3.2. Technologies used 53
3.3. Fundamental concepts 54
3.4. Architecture 55
3.5. User workflow 56
3.6. Search cores module 62
3.6.1. Ant Colony Optimization (ACO) 64
3.6.2. Evolutionary Strategy (ES) 67
3.6.3. Imperialist Competitive Algorithm (ICA) 68
3.7. Visualisation tools 72

3.8. Solution transition optimisation ... 75
3.8.1. Numerical examples ... 77
3.9. MOF comparisons ... 78
3.10. Summary ... 81
4. The Imperialist Competitive Algorithm with Independence and Constrained Assimilation (ICAwICA) ... 83
4.1. Motivation and related work ... 83
4.2. Methods and implementation ... 85
4.2.1. Classic ICA ... 85
4.2.2. ICAwICA ... 86
4.2.3. Constrained assimilation ... 86
4.2.4. ICAwICA solution encoding for MKP and MDVRP ... 89
4.3. Experiments ... 90
4.3.1. Benchmark instances ... 90
4.3.2. Experimental setup ... 91
4.3.3. Comparison to classic ICA ... 92
4.3.4. Sensitivity analysis of independence rate ... 93
4.3.5. Comparison to the state-of-the-art metaheuristics for MKP ... 94
4.3.6. Comparison to the state-of-the-art metaheuristics for MDVRP ... 96
4.4. Summary ... 97
5. Accelerating supply chains with Ant Colony Optimization across a range of hardware solutions ... 99
5.1. Motivation and related work ... 99
5.1.1. Parallel Ant Colony Optimization ... 101
5.1.2. CPU ... 103
5.1.3. Xeon Phi ... 104
5.1.4. GPUs ... 104
5.1.5. Comparing hardware performances ... 105
5.2. Background ... 106
5.2.1. Parallel processing with OpenMP ... 106
5.2.2. CUDA programming model ... 106
5.2.3. Xeon Phi Knights Landing architecture ... 107
5.3. Methods and implementation ... 108
5.4. Experiments ... 112
5.4.1. Benchmarks ... 113

5.4.2. Speed performance 115
5.4.3. Hardware Comparison and speed of convergence 119
5.4.4. Comparisons using the Travelling Salesman Problem 121
5.5. Summary 122
6. Simple generate-evaluate strategy for tight-budget parameter tuning problems 125
6.1. Motivation 125
6.1.1. Parameter tuning problem 127
6.2. Related work 128
6.3. Proposed methods 130
6.3.1. Elitist Tuner - eTuner 131
6.3.2. Elitist tuning with pre algorithm selection – eTunerAlgo 132
6.4. Experiments 134
6.4.1. Experimental setup 134
6.4.2. Experimental results 136
6.5. Summary 141
7. Conclusions and Future Work 143
7.1. Conclusions 143
7.2. Future work 144
8. References 146
9. Appendix 165

LIST OF TABLES

Table 1. Summary of the most popular metaheuristic optimization frameworks sorted by creation year. 27

Table 2. MOF supported features. 35

Table 3. Solution transition plan for MKP gk01 with maximizing profit as objective fST. NSL represents the number of element-order pairs that differ from the final solution. 77

Table 4. Solution transition plan for Transcom scheduling and routing problem with minimizing total cost (in million $) as objective fST. NSL represents the number of element-order pairs that differ from the final solution. 78

Table 5. Parameters used for an experiment on a various algorithm on different MOFs 79

Table 6. Metaheuristic Optimization Framework comparisons. Best and average expressed as error per cent from an optimal solution, colour coded from the best error (in green) to the worse (in red). Google OR-tools is added for reference only and is not considered a MOF. 80

Table 7. Comparison of best and average scores between Classic ICA and ICAwICA across six test problem instances. Average and best out of 10 runs with standard deviation (std), BKS – Best Known Solution. 92

Table 8. Sensitivity analysis of Independence rate as an average error per cent gap for six test problem instances. With 0 representing ICA with no independence operator, $tavgs$ representing the average time in seconds to converge to the best solution, BKS – Best Known Solution 93

Table 9. Algorithm comparison across large Glover and Kochenberger (GK) knapsack instances. Results are expressed as average error percentage gap % against best-known profit. Colour coded from the best gap (green) to worst gap (red) for any given dataset. With dash (-) representing results that are not available. BKS – Best Known Solution, Std – Standard Deviation of the absolute value. 95

Table 10. Best solution obtained by ICAwICA compared to other algorithms in the literature across Cordeau's MDVRP benchmark instances and the best-known solution (BKS). The best scores represented in bold, N representing the number of customers, M – the number of depots. Average error percentage calculated using BKS as a reference, $tavg(m)$ – average time to converge to a solution, in minutes, Std – Standard Deviation 97

Table 11. ACO architecture and hardware configurations explored. LAC - Longest Common Subsequence Problem, MKP - Multidimensional Knapsack Problem, TSP - Travelling Salesman problem. IAC – Independent Ant Colonies, IntAC – Interactive Ant Colonies, PA – Parallel Ants. 102

Table 12. Meta-data required during solution creation based on problem type 102

Table 13. Comparison of Independent Ant Colonies (IAC), Parallel Ants (PA) and parallel Ants with Vectorisation (PAwV) architectures. 111

Table 14. Ant Colony System set of parameters for all configurations and architectures 112

Table 15. Parallel Ants fitness value baseline for different configurations of the number of parallel instances and the number of iterations. Each Parallel Instance data point is an average of 10 individual runs (table derived from 11*10 =110 runs). Expressed as a percentage of the proximity of the best-known solution (2,701,367.58). Colour-coded from worse – in red, to the best – in green. 114

Table 16. The number of iterations required to reach a specific solution quality. Each data point in the table is an average of 10 individual runs. Empty fields (-) represent instances where ACO did not obtain specified solution quality in 768k solutions explored. The solution quality is expressed as a percentage of the proximity of the best-know solution (2,701,367.58). 115

Table 17. Hardware A wall-clock time per iteration, in seconds. KMP config is environment variable set as part of KMP_PLACE_THREADS, for all instances KMP_AFFINITY=scatter, optimisation level /O3, favour speed /Ot. 117

Table 18. Hardware B wall-clock time per iteration, in seconds. KMP config is environment variable set as part of KM_PLACE_THREADS, for all instances KMP_AFFINITY=scatter, optimisation level /O3, favour speed /Ot. 118

Table 19. Hardware C wall-clock time per iteration, in seconds. The total number of parallel instances are adjusted for the thread-block dimensions. Compiled with CUDA 9.0. 1x, 2x and 4x correspond to the number of devices used to compute............ 119

Table 20. Estimated time (in seconds) required to converge to specific solution quality. Calculated by multiplying the number of iterations by the time taken for iteration for individual best performing hardware configuration. Solution quality is expressed as a percentage of the proximity of the best-know solution (2,701,367.58). 120

Table 21. The algorithms and hyperparameters used for tuning. Each of the parameters has a discrete set of values that can be used for the candidate. The total number of candidate configurations for Ant Colony Optimization is 12,000, for Evolutionary Strategy – 1000 and Imperialist Competitive Algorithm – 5760. Thus, the total number of candidate configurations is 18,760. 135

LIST OF FIGURES

Figure 1. Connections between the thesis chapters. 5

Figure 2. Taxonomy of optimization methods.. 9

Figure 3. Ant behaviour (inspired by [105]).. 21

Figure 4. Evolution cycle of Evolution Strategy 22

Figure 5. Convergence representation of ICA [123]. With stars representing empires and circles – their colonies. 24

Figure 6. Example of an MDVRP with ten customers (as circles) and two depots (A and B as rectangles) 37

Figure 7. Graphical representation of simplified Aerial Surveying Problem. Each rectangle (A, B, C) represent a base station and each of the circles (1-9) pose a task/location that needs to be visited. In this example, there are four different routes, route A-1-8-4-A in black, route A-7-B-3-A in blue, route A-C-5-9-6 in red and finally route A-2-C in yellow. 40

Figure 8. Graphical representation of the outbound supply chain. Each warehouse i is connected to one or many origin ports p. The shipping lane between origin port p and destination port j is a combination of courier c, service level s, delivery time t and transportation mode m. 42

Figure 9. Pseudocode for calculating order transportation cost.............................. 43

Figure 10. Simplified Transcom supply chain example... 46

Figure 11. A high-level overview of modules in OptPlatform. Optimization platform uses two languages – C++ for low-level high-performance search and C# for user interfacing and other accessory tools. Split into user domain, where only problem details are specified and the abstracted backend - platform. 56

Figure 12. User workflow for implementing an optimization problem. Icons represent the modules used (in Figure 11) during the process, some of them can be optional. 56

Figure 13. Search space representation and solution element encoding. Constructed as a 2D matrix with sizes $Emax$ and $Omax$. Search algorithm selects one or multiple cells to be added to the final solution. 57

Figure 14. Simple bin packing problem encoding and decoding with five identical bins (represented as orders) and eight items (elements). 58

Figure 15. The interface between the search algorithms in the Search Core module and user-defined problem in Opt Problem. Flowchart on the left is a generic model that all search algorithms in the Search Cores follow. Methods isBetterPerformance and userSyncAfterIteration are optional and therefore greyed out. 60

Figure 16. Two example encodings for Travelling Salesman Problem (TSP). In Sequence encoding, only the selected element sequence in the solution is needed for encoding. Graph encoding represents nodes as a 2D graph, where the nodes themselves are represented as orders and the inter-connections as elements. Therefore, cells $o0e0$ and $o1e1$ would be invalid in TSP as it is a connection to itself. 62

Figure 17. Memory allocation and parallelism in Search Cores architecture. Areas of the process, where problem-specific methods are called, are in orange. Iterations are executed in sequence, however, in each iteration, multiple solutions are constructed and evaluated up to the maximum number of parallel instances - $PImax$. 63

Figure 18. High-level pseudo code for Ant Colony Optimization algorithm in OptPlatform. 66

Figure 19. High-level pseudo-code for Evolutionary Strategy algorithm in OptPlatform 67

Figure 20. Example of the mutation process of Evolutionary Strategy in OptPlatform. PossibleElement pairs with red are removed and replaced with PossibleElement pairs in blue. 68

Figure 21. High-level pseudo code for Imperialist Competitive Algorithm in OptPlatform 69

Figure 22. Example of Imperialist Competitive Algorithm assimilation process in OptPlatform. PossibleElements in red indicating the cells that are merged to create a new country. 70

Figure 23. The output of the global pheromone visualization tool. Each pixel represents a pheromone change for given element across multiple iterations. With red pixels indicating when evaporation happens, green – pheromone deposit, white – no pheromone left for the specific element and in black – no change between the iterations........ 72

Figure 24. OptPlatform's search visualization tool........ 73

Figure 25. Simulation summary graphical interface........ 74

Figure 26. Automatically generated solution animation of Transcom problem using Google Earth........ 75

Figure 27. A high-level overview of transition optimisation, where two solutions (sub-optimal and optimized) are used as inputs to generate a transition plan based on the provided goal. In this example, seven stages are generated starting from the sub-optimal solution at Stage 0 to optimized solution at Stage 7........ 76

Figure 28. Flowchart of classic ICA [124] (to the left) and the proposed ICAwICA (on the right), with red indicating the changes........ 87

Figure 29. The pseudocode for new assimilation and local search method for ICAwICA........ 88

Figure 30. Imperialist and colony constrained assimilation process with solution repair. With integer values corresponding to solution entries (item indices in MKP case or depo indices in MDVRP case)........ 89

Figure 31. Customer assignment to depots in MDVRP using ICAwICA assimilation. Where C1-C10 are customer indices and the encoded integers are depot indices that are assigned to a given customer, with bold representing assimilated changes........ 90

Figure 32. Comparison between Classic ICA [124] and ICAwICA for six test problem instances. Expressed as average error percentage to the best know solution. The graph demonstrates ICAwICA achieves average error of 0.62% while Classic ICA achieves 1.3%, relative improvement of over two times........ 93

Figure 33. The average error of the mean profit across all WEISH (1-30) instances. Average of 30 independent runs........ 94

Figure 34. Knights Landing tile with a larger processor die [282]........ 108

Figure 35. High-level pseudocode for Independent Ant Colonies (IAC) search algorithm 109

Figure 36. High-level pseudocode Parallel Ants (PA) search algorithm 110

Figure 37. High-level pseudocode for Parallel Ants with Vectorization (PAwV) search algorithm. Expanding on Figure 36' lines 3-7. 111

Figure 38. Parallel Ants best estimated computation time per solution quality for supply chain problem to converge to specific solution quality. Solution quality is expressed as a percentage of the proximity of the best-know solution (2,701,367.58). 121

Figure 39. Parallel Ants computation time per solution quality for lin318 TSP to converge to specific solution quality. Solution quality is expressed as a percentage of the proximity of the best-know solution (a distance of 42029). 122

Figure 40. Comparison between typical user workflow and automated workflow. Yellow boxes are indicating areas where user expertise is necessary for optimal results. In the automated workflow, algorithm selection and tuning are performed automatically. 126

Figure 41. A high-level overview of the process flow. The underlying problem is optimised by one or several metaheuristic algorithms. The metaheuristic algorithm(s)' parameters are optimised by the hyperparameter tuner. 128

Figure 42. Graphical representation of the allocation of configuration evaluations by variations of Elitism Rate and a brute force method for reference. All approaches are allowed to perform the same number of total experiments (100-hour tuning budget, with 60 second compute time for each configuration); thus, all three figures cover the same surface area 132

Figure 43. Pseudocode of the proposed Etilist Tuner - eTuner algorithm 132

Figure 44. Example of new candidate configuration generation for metaheuristic algorithm selection. Where given three parameters P, one "average" configuration is generated and six other candidate configurations. 133

Figure 45. The baseline for Aerial Surveying Problem (ASP) with a simple exhaustive search (brute force) approach, where each evaluation represents a single run for each of the 18,760 configurations. Error bars represent the minimum and maximum values

achieved during 10 simulations—average total cost, in a million dollars (minimisation problem, lower costs are better). 136

Figure 46. Comparison of eTuner and eTunerAlgo approaches for Aerial Surveying Problem. Error bars represent the minimum and maximum values achieved during 10 simulations—average total cost, in a million dollars (minimisation problem, lower costs are better). 138

Figure 47. Tuning algorithm comparison for Aerial Surveying Problem. Error bars represent the minimum and maximum values achieved during 10 simulations—average total cost, in a million dollars (minimisation problem, lower costs are better). eTuner and eTunerAlgo are the proposed methods, Iterative F-Race is the implementation of [301]. 139

Figure 48. Tuning algorithm comparison for Aerial Surveying Problem. Average total cost in a million dollars of 10 simulations (minimisation problem, lower costs are better). eTuner and eTunerAlgo are the proposed methods, Iterative F-Race is the implementation of [301]. 140

Figure 49. Tuner performance comparison for MKP-gk10 instance. Error bars represent the minimum and maximum values achieved during 10 simulations—average profit (maximisation problem, higher profits are better). eTunerAlgo is the proposed method, Iterative F-Race is the implementation of [301]. 141

Figure 50. Scenario A – Simplified Transcom supply chain example 166

Figure 51. Scenario B pallet flow between military bases and Commercial Partners (CPs). 167

Figure 52. Scenario B: Realistic Transcom supply chain example 168

Figure 53. Transcom Scenario B timeline 169

LIST OF ABBREVIATIONS

ABC	Artificial Bee Colony
ACO	Ant Colony Optimization
ACS	Ant Colony System
AS	Any System
BAT	Bat Algorithm
CoEA	Coevolutionary Algorithm
CSA	Cuckoo Search Algorithm
CPU	Central Processing Unit
DE	Differential Evolution
DVRP	Dynamic Vehicle Routing Problem
EA	Evolutionary Algorithm
EC	Evolutionary Computing
ES	Evolutionary Strategy
FA	Firefly Algorithm
GA	Genetic Algorithm
GP	Genetic Programming
GPU	Graphics Processing Unit
GRASP	Greedy Randomized Adaptive Search Procedure
GUI	Graphical User Interface
GWO	Grey Wolf Optimizer
HPC	High Performance Computing
ICA	Imperialist Competitive Algorithm
ILS	Iterated Local Search
JSS	Job Shop Scheduling
KP	Knapsack Problem
LS	Local Search
MFO	Moth-flame Optimization
MKP	Multidimensional Knapsack Problem
MOF	Metaheuristic Optimization Framework
mQAP	Multi-objective Quadratic Assignment Problem

MVO Multi-Verse Optimizer

OOP Object Oriented Programming

PRNG Pseudo Random Number Generator

PFSP Permutation Flow shop Problem

PSO Particle Swarm Optimization

QAP Quadratic Assignment Problem

SA Simulated Annealing

SAT Boolean Satisfiability Problem

SDK Software Development Kit

SI Swarm Intelligence

TS Tabu Search

TSP Travelling Salesman Problem

VNS Variable Neighbourhood Search

VRP Vehicle Routing Problem

WOA Whale Optimization Algorithm

1. INTRODUCTION

An optimisation is part of our daily schedule. A school timetable is an excellent example of a scheduling problem that could still be performed by hand, though it would be impractical. Once the timetable gets more involved with many students, classrooms and teachers, the naïve exhaustive search becomes infeasible in polynomial time. We refer to these kinds of problems as NP-complete or NP-hard. These NP-hard problems can be found in scheduling, timetabling, routing, logistics, supply chain management, finance and engineering.

Finding a global optimum in a complex optimisation problem is not trivial. In some cases, the search space's size is so big that even combined computation power of the whole world would struggle to solve the problem exhaustively in our lifetimes. In these instances, approximate solutions might be a reasonable trade-off if the solution found is near-optimal, and the computation time and resources are acceptable. Heuristic approaches offer this trade-off as a practical method for solving real-world problems, where the near-optimal solution may be enough. In a real-world optimization model, not all parameters are known or are recorded correctly and are usually approximated. Therefore, even if the exact optimization method is used, it is still likely to find a non-optimal solution while requiring more compute time and resources.

Metaheuristics, or *generic heuristics*, are optimisation algorithms that offer more generalisation than heuristic algorithms and are not problem limited. The generic nature allows the same algorithm to be applied to a wide variety of problems. However, the no-free-lunch theorem [1] suggests that no single algorithm would be the best for all possible problems; thus, multiple different metaheuristics exist. The ability to apply metaheuristics to various problems, or rather, solving the same problem with multiple metaheuristics, has been an inspiration of many Metaheuristic Optimisation Frameworks (MOFs) in the last two decades. MOFs are standardised frameworks that utilises metaheuristic methods for optimisation. This thesis implements such MOF for real-world optimisation problems.

1.1. Motivation

As the world gets more interconnected, companies and governments try to optimise their processes and lower the cost. The ever-growing data availability and increase in computing power is the perfect storm for global, intercontinental optimisation. Unexpected events such as the Covid-19 outbreak have made many companies re-plan their businesses, especially their supply chains. A more resilient or faster-adapting business is an edge against competition and competition drives more efficient use of limited resources.

Gaining an edge against competition involves robust and scalable optimisation frameworks. These platforms are required to not only be able to produce useful solutions in a reasonable time frame but also have all the essential supporting tools to implement the results to generate the most impact. Existing MOFs are mostly made for academia for research and new algorithms development. They are limited in applicability to real-world and expect some expert knowledge in metaheuristics.

1.2. Thesis Contributions

The work presented in chapters 3, 4, 5 and 6 discuss the proposed methods of efficient metaheuristics optimization platform aimed for complex real-world optimization problems. These methods have been accepted and published in two journal papers and two conference papers. The original contributions of this thesis can be summarized as follows:

1. **OptPlatform**: high-performance metaheuristic optimization platform aimed at solving a class of complex real-world optimisation problems. The developed software system incorporates necessary toolset for efficient optimization problem implementation and analysis. It comprises of three metaheuristic algorithms – Ant Colony Optimization (ACO), Evolutionary Strategy (ES) and Imperialist Competitive Algorithm (ICA). The developed OptPlatform can derive optimal solutions quicker than comparable existing metaheuristic optimization frameworks and introduces tools that are not available on other platforms, such as automatic algorithm selection and tuning. The superior efficiency of the OptPlatform has been considered for several optimisation problems, both benchmark and real-life models. The OptPlatform architecture is inspired based on previously published work in [2].

2. **Imperialist Competitive Algorithm with Independence and Constrained Assimilation (ICAwICA):** an improved metaheuristic algorithm based on classical Imperialist Competitive Algorithm (ICA). The proposed algorithm introduces the concept of colony independence – a free will to choose between classic ICA assimilation to the empire's imperialist or any other imperialist in the population. Furthermore, a constrained assimilation process is introduced that replaces classical ICA assimilation and revolution operators. ICAwICA shows definite improvement over classical ICA and outperforms most of the competition across a variety of optimization problems. The proposed algorithm was published in [3].
3. **A study of parallel Ant Colony Optimization:** an in-depth analysis of parallel Ant Colony Optimization architecture scaling across numerous hardware solutions – high-end workstation CPU, Intel Xeon Phi architecture and General Processing Units (GPUs). Although previous research indicates that GPUs are the most suitable for benchmark routing problems, this study empirically demonstrates how the scaling dynamics do not translate to a real-world optimisation problem due to memory access patterns necessary. The contradictory findings with the supporting dataset were published in [4].
4. **eTuner and eTunerAlgo hyperparameter tuning algorithms**: two simple generate-evaluate algorithms are developed for automated metaheuristic and their hyperparameter selection. A benchmark dataset, containing three metaheuristics and their performance for set of hyperparameters (18,760 configurations), is generated and published in [5]. Later, this benchmark dataset is used to evaluate and compare the different methods with the current state-of-the-art. Results show that the presented approach is more suited for low tuning budgets than the competition. Methods used and the results are published in [6].

1.3. List of publications

The following lists the work that has been publicised as part of this research:

- I. Dzalbs and T. Kalganova, “Simple generate-evaluate strategy for tight-budget parameter tuning problems,” in IEEE Symposium Series on Computational Intelligence (SSCI), 2020, doi 10.1109/SSCI47803.2020.9308348.
- I. Dzalbs, T. Kalganova and I. Dear, “Imperialist Competitive Algorithm with Independence and Constrained Assimilation,” in 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), 2020, pp. 1–11, doi: 10.1109/HORA49412.2020.9152916.
- I. Dzalbs and T. Kalganova, “Accelerating supply chains with Ant Colony Optimization across a range of hardware solutions,” Comput. Ind. Eng., vol. 147, p. 106610, Sep. 2020, doi: 10.1016/j.cie.2020.106610.
- I. Dzalbs and T. Kalganova, “Forecasting Price Movements in Betting Exchanges Using Cartesian Genetic Programming and ANN,” Big Data Res., vol. 14, pp. 112–120, 2018, doi: 10.1016/j.bdr.2018.10.001.

1.4. Thesis contents

This thesis consists of seven chapters. The first chapter familiarises the reader with a brief background, motivation, and significance of this research.

Chapter 2 covers an in-depth literature review of optimization methods, with a focus on metaheuristics. Furthermore, Chapter 2 also presents and analyzes various existing metaheuristic optimization frameworks in the literature. Finally, Chapter 2 introduces multiple optimization problems that will be solved throughout the further chapters. **Figure 1** summarises the connections between different optimization problems across chapters.

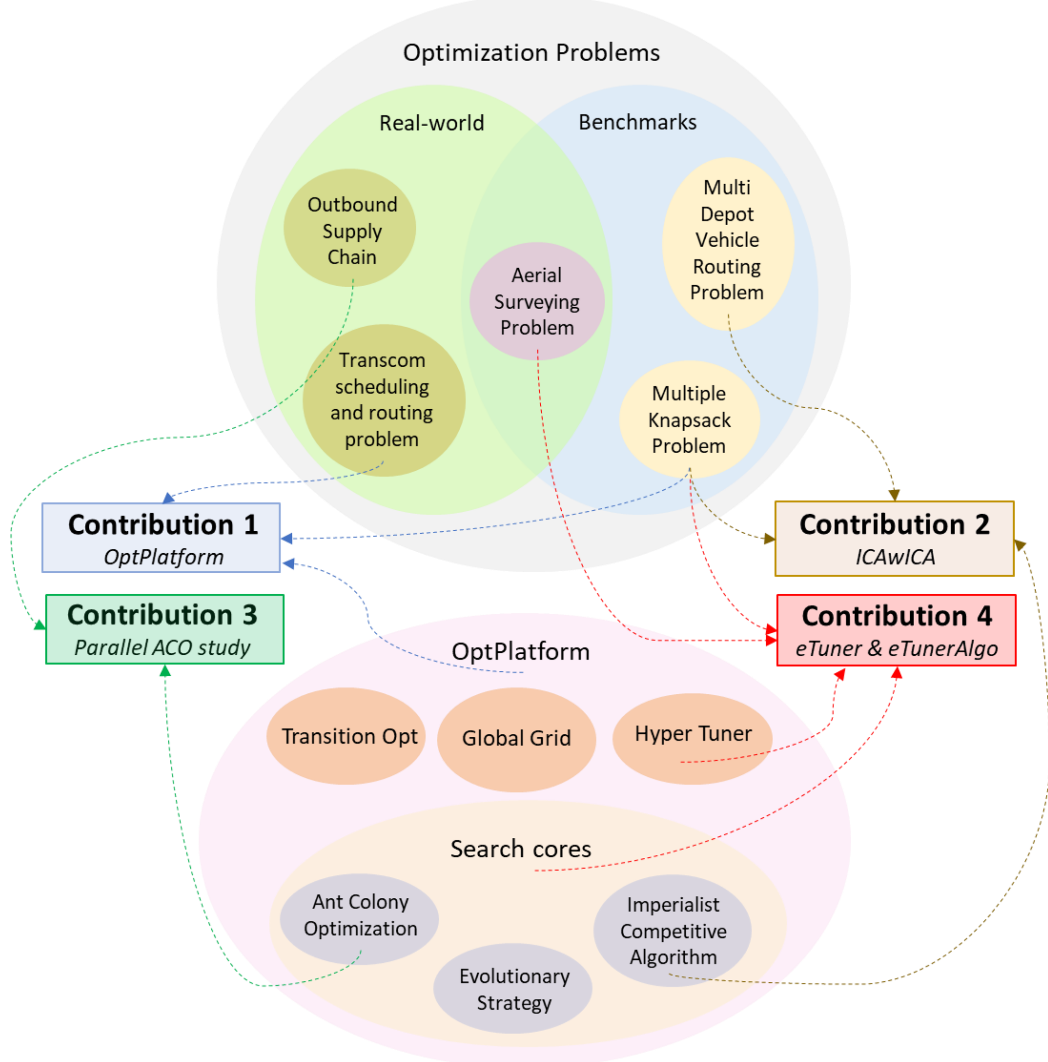

Figure 1. *Connections between the thesis chapters.*

Chapter 3 introduces the optimization platform – OptPlatform. It starts by defining the target users and software requirements as well as justifies the technologies used. Furthermore, Chapter 3 describes the software architecture and gives user workflow examples. The three implemented metaheuristic algorithms are described in detail, including supporting modules such as Search Visualizer, Transition Opt and Global Grid. Search visualizer module creates a graphical report of the statistical metrics about the search, while Transition Opt module creates a step by step transition plan of implementing the resulting optimized solution. Global Grid animates the geographical paths in the map for more straightforward analysis and solution understanding. A real-world Transcom scheduling and routing problem is solved as a case study, to demonstrate the platform and its modules. Finally, the developed

OptPlatform is compared to existing metaheuristic optimization frameworks using the Multiple Knapsack Problem.

In Chapter 4, a new algorithm based on the Imperialist Competitive Algorithm is developed – Imperialist Competitive Algorithm with Independence and Constrained Assimilation (ICAwICA). ICAwICA improves on existing implementations by replacing traditional assimilation and revolution operators with constrained assimilation. Furthermore, independence operator is used for local search. The algorithm's performance is evaluated on two benchmark problems – Multi Depot Vehicle Routing Problem (MDVRP) and Multiple Knapsack Problem (MKP). The experimental results demonstrate the superiority over classic ICA and universality of the local search.

Chapter 5 presents a detailed exploration of parallel Ant Colony Optimization (ACO) algorithm and its scaling dynamics on various hardware types. Academic literature indicates that Graphical Processing Units (GPUs) can speed-up the search process for benchmark problems by 172 times. Chapter 5 investigates if the same ACO architectures can be applied for a real-world supply chain optimization problem. Results indicate that although suitable for simple benchmark problems, GPU ACO architectures cannot scale for more complex supply chain problems.

In Chapter 6, two simple generate-evaluate hyperparameter tuners are introduced for automated metaheuristic algorithm selection and evaluation. A benchmark dataset is generated based on all three metaheuristics – ACO, ICA, and Evolutionary Strategy (ES) and used to evaluate the performance. Two optimization problems were used for the underlying optimization – Aerial Surveying problem (a real-world problem adapted as a benchmark) and MKP. Results demonstrate the superiority over existing parameter tuning algorithms.

Finally, Chapter 7 concludes the thesis and lists potential future research directions.

2. LITERATURE REVIEW

Although the main research area is in Metaheuristic Optimization Frameworks (MOFs), the first section of this paragraph introduces the background of optimisation methods and introduces the reader to various metaheuristics found in the literature. In particular, section 2.1 highlights an overview of the current state of the art optimization methods with a focus on metaheuristics (section 2.1.2). Section 2.2 reviews and analyzes the current state of the art MOFs, where research gaps are established (section 2.2.1). The chapter continues by introducing various optimization problems that are solved throughout the consecutive paragraphs in section 2.3. Finally, the chapter is summarized in section 2.4.

2.1. Optimization methods

Optimization is the process of finding the best solution among a pool of possible solutions. Optimization is applied to a wide range of engineering, economic and even social systems to minimize cost or maximize profits. There is no single optimization technique that can be efficiently applied across all optimization problems. Hence several optimization methods have been developed for different kinds of optimization problems [7]. The optimum pursuing behaviour is also referred to as *mathematical programming* in o*perations research*. Operations research is a branch of mathematics focusing on applying scientific methods and techniques for the decision-making process. The research area's roots can be traced down to World War II, where the British military faced the problem of allocating constrained resources, such as aeroplanes, radars, and submarines to different destinations. At the time, there were no systematic methods for resource allocation, and hence a group of mathematicians was called for assistance. The mathematical methods developed were instrumental in the winning of the Air Battle by Britain. Techniques, such as linear programming, were created as part of military research operations and therefore came to be known as operations research [7].

The optimization problem needs to be modelled first before it can be solved. To develop the mathematical model of an optimization problem, the following components should be fully characterized [8]:

1) The set of optimization variables $x_1, x_2, \dots, x_n$.
2) The objective function $f(x)$ that applies to the optimization variables and returns a real value. The objective function can be either minimized or maximized.
3) A set of constraints that should hold on the optimization variables.
4) The domain sets $D_1, D_2, \dots, D_n$ as the domains of the optimization variables $x_1, x_2, \dots, x_n$.

However, some optimization problems can be described without constraints; similarly, optimization variables' domain set can be the entire space [7].

The optimal or near-optimal solution can then be found by either exact (or *deterministic, classical*) or approximate (or *random, modern*) methods, hierarchy shown in Figure 2. Exact methods offer a mathematically provable optimal solution; however, because large proportion of real-world optimization problems are NP-hard, deterministic methods are not always suitable due to computation expense. NP-hard refers to problems that are impossible to predict whether an optimal solution can be computed in less than exponential time [9]. Furthermore, it is not always possible to define an exact technique for every optimization problem. In contrast, approximate methods offer short-time solving of NP-hard problems while finding optimal or near-optimal solutions [10]. Due to the shortcomings of the exact techniques and ever-increasing complexity of problems being solved, approximate methods have gained traction in the last few decades. These methods cannot guarantee the optimality of the final solution; however, offer a near-optimal solution with reasonable computation time.

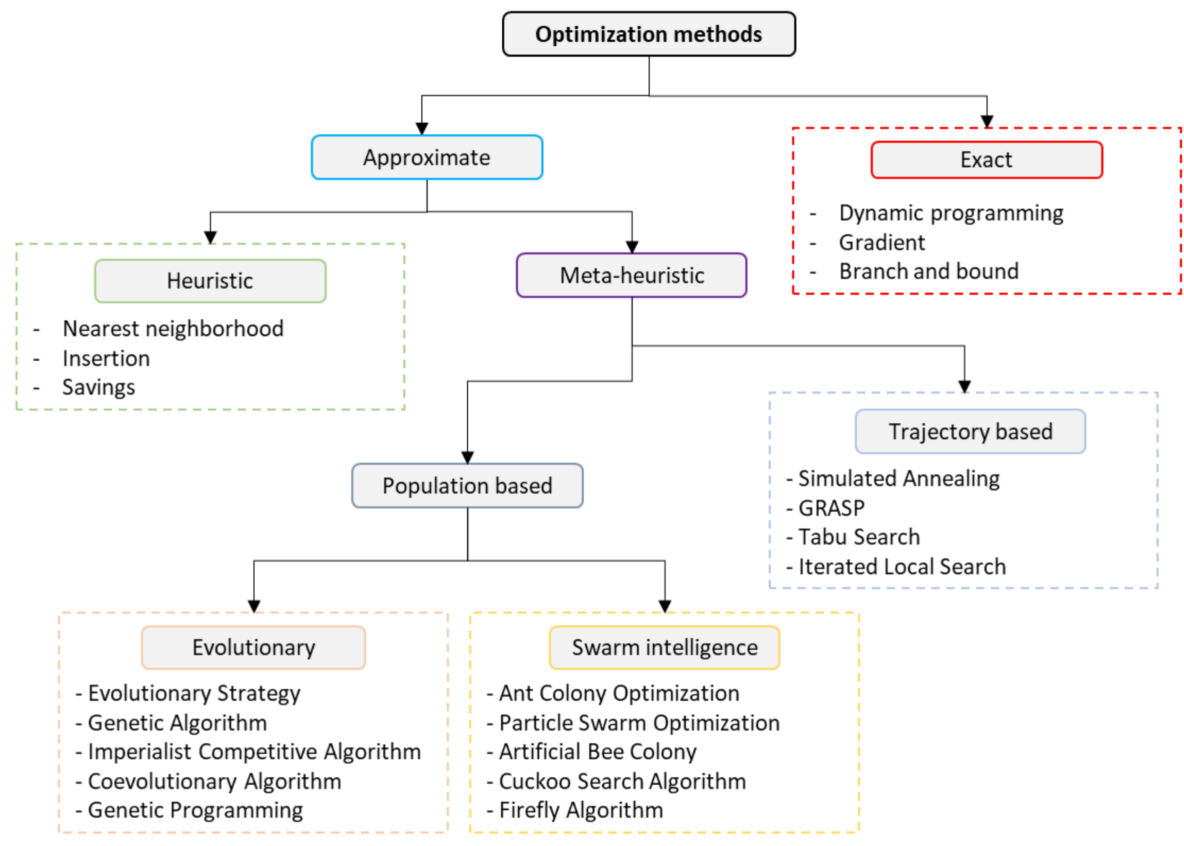

Figure 2. Taxonomy of optimization methods

2.1.1. Heuristics

Even though a theoretically provable optimal solution is desirable, it is not always possible or practical. Due to runtime complexity, most exact methods are not applicable to high dimensionality real-world problems. In these instances, approximate solutions might be a reasonable trade-off if the solution found is near-optimal, and the computation time and resources are acceptable. Heuristic approaches offer this trade-off as a practical method for solving real-world problems, where the near-optimal solution may be enough. In a real-world optimization model, not all parameters are known or are recorded correctly and are usually approximated. Therefore, even if the exact optimization method is used, it is still likely to find a non-optimal solution while requiring more compute time and resources. Furthermore, in real-time systems, a good enough solution is necessary in a matter of seconds.

Heuristic approaches offer practical implementation of hard to solve optimization problems based on knowledge gained from experience. Examples of this method include using a rule of thumb, an educated guess, an intuitive judgement, or common sense [10]. Constructive heuristics are usually the fastest to implement, and they

construct a solution from scratch by iteratively adding solution sub-components till a complete solution is obtained. Determining constructive heuristic is easy in most combinatorial problems; however, the resulting solution quality strongly depends on the level of the expertise used to design the implementation.

In the last three decades, the interest of more generic heuristics has been growing exponentially [11]. These general heuristics, called *metaheuristics*, combine the basic heuristics with a higher-level search framework for efficient exploration of the search space and are entirely independent of the application domain.

2.1.2. Metaheuristics

In the 70ies, a new paradigm was introduced that promised to combine basic heuristic methods at a higher level to explore search space more effectively – called *modern heuristics* [12]. These days more commonly referred to as *metaheuristics*. Metaheuristics include simulated annealing, evolutionary strategy, tabu search, ant colony optimization and many more. The specific implementation of metaheuristics varies from one another; however, they all implement two operators – *diversification* and *intensification* [13]. Diversification refers to the exploration of the search space, while intensification – exploitation of the knowledge about the search space. All metaheuristics need to balance the exploration and exploitation – too high intensification and search may get stuck into local optima (sub-optimal solution); too high diversification and global optima may never be found or convergence takes too long [13]. Fundamental characteristics of metaheuristics are summarized based on [13] and [14]:

- Metaheuristics are usually approximate and non-deterministic.
- Metaheuristics are not problem-specific.
- Metaheuristics explore the search space to find "good enough" solutions.
- Metaheuristics may incorporate domain-specific knowledge in the form of heuristics.
- Metaheuristics can be described by an abstraction level.
- Metaheuristics may use various strategies to avoid premature convergence.
- Metaheuristics usually allow highly parallel implementation.
- Metaheuristics may use search experience as a form of memory to guide the search.

There are many ways that metaheuristics can be classified, and some of the groupings are subjective and depend on the author's viewpoint. Some of the algorithms may fit into one or two taxonomies and sometimes overlap [10].

For example, some metaheuristic algorithms can be classified as nature-inspired, like ant colony optimization and genetic algorithm, while others as non-nature inspired – tabu search and iterated local search. Work in [15] performed extensive taxonomy on nature and bio-inspired metaheuristics by comparing 300 papers over different years. Furthermore, authors in [15] also categorized nature-inspired algorithms as follows: Breeding-based Evolution, Swarm Intelligence, Physics and Chemistry based, Social Human Behaviour algorithms, plant-based and other miscellaneous.

Furthermore, metaheuristic algorithms can also be divided into trajectory-based (sometimes referred to as single-point) and population-based search [16]. This division specifies how many solutions are created at any given iteration. Trajectory-based methods include most local search approaches such as iterated local search, tabu search and simulated annealing, where the current best solution is replaced by a new one. On the other hand, population-based algorithms maintain many solutions (population) in the search space (in evolutionary methods), or they perform search processes that alter the distribution probability over the search space (ant colony optimization for example) [12]. Usually, population-based algorithms start with a random population that is enhanced throughout the search. Trajectory-based algorithms tend to favour exploitation, while population-based are more exploration oriented. Often additional methods are implemented for local search when using population-based algorithms and a global search for trajectory-based.

Authors in [14] used taxonomy of metaphor-based and non-metaphor based metaheuristics. The former are algorithms that simulate natural phenomena, human behaviour or mathematics, the latter, metaheuristics that do not use any form of simulation for determining their search strategy.

Metaheuristics can also be with or without memory. Majority of population-based algorithms are with-memory as they use previous search history to guide and assist the search processes. In contrary, memory-less algorithms only use current state to determine the next action, i.e. follow the Markov process.

Although numerous metaheuristic algorithms exist in the literature [17],[14],[11], the following section aims to provide an overview of the most common metaheuristic algorithms, with the focus on Ant Colony Optimization, Evolutionary Strategy and

Imperialist Competitive Algorithm, all three used as optimization algorithms in section 3.6.

2.1.2.1. Trajectory-based

The main characteristics of trajectory-based metaheuristics are that they start from a single solution and iteratively move away from it, describing a search space trajectory. These techniques aim to improve local search in a more intelligent way. Trajectory methods consist of the Simulated Annealing (SA) method, the Greedy Randomized Adaptive Search Procedure (GRASP) method, the tabu search, and many local search variations.

- **Simulated Annealing (SA)**

The algorithm is inspired by the annealing process of metal or glass, where the material's temperature is slowly reduced till low energy state is reached. First proposed by [18] and then adopted by [19] for optimization problems. The fundamental idea is to use temperature as an explicit strategy to guide the search. The algorithm starts by generating a random or heuristic-based initial solution. In the beginning, when the temperature is still high, the algorithm prefers exploration and accepts good and bad solutions. But as the temperature is reduced, the requirements for improving existing solution becomes stricter and stricter. The strengths of SA is the ability to avoid getting stuck in local minima which is directly linked to the cooling schedule [20]. The cooling schedule determines the functional form of the change in the temperature needed in SA. SA has been applied to multiple discrete and continuous optimization problems [21], though rarely on combinatorial problems as a standalone algorithm [17]. Most commonly, SA is used as a form of local search in hybridization with other algorithms. Variations of SA such as Microcanonic Annealing [22], Threshold accepting method [23] and Noising method [24] aim to improve the generic form of SA.

- **Greedy Randomized Adaptive Search Procedure (GRASP)**

The GRASP algorithm [25],[26] is a multi-start or iterative metaheuristic with two phases in each iteration: solution construction and local search. The construction phase uses a Greedy Randomized Adaptive algorithm to build a solution. If the solution is not valid, a repair procedure is applied. The solution is then improved by local search. The improved solution is the final result of the search. In the greedy part

of the heuristic, a solution is built iteratively by adding partial solutions. It means that list of partial solution entries needs to be created beforehand. In each iteration the list is sorted based on a greedy function. Each of the partial solutions are selected randomly from the set of restricted candidates (RCL) [17]. A comprehensive summary of common approaches of GRASP and problem domains are provided by the survey in [27] and a more recent survey of [28]. Most commonly GRASP is combined with other local search techniques, such as simulated annealing, variable neighbourhood search and iterated local search [29], [30].

- **Tabu Search (TS)**

The Tabu search (TS) first introduced in 1986 by [31]. Its main characteristic is based on the use of mechanisms inspired by human memory [17]. Improvement on the Local Search which can avoid local minima by use of memory methods in three schemes: 1) use of flexible memory structures to search and evaluate information based on previous moves; 2) control the actions to be applied on the time of search process; 3) use of memory functions of long term and short term memory to diversify and intensify the search [32]. The main idea of the TS is the restrictions of already visited areas of the search – tabu list. The length of the tabu list controls the memory dynamics of the search process. Small list leads to concentration on small areas – intensification, while long list encourages exploration of larger regions – diversification. A detailed description of the TS with its variations can be found in [33]. Although attempts to apply tabu search for continuous problems exist [34], most of the TS research focuses on discrete combinatorial optimization [33].

- **Iterated Local Search (ILS)**

The Iterative local search (ILS) [35] iteratively applies the local search algorithm to the candidate solutions. Each move is only performed if the new solution is better than the current one based on the *acceptance criterion*. The algorithm selects the starting point in the search space either randomly or based on domain-knowledge. The acceptance criterion alongside perturbation mechanism allows altering the dynamics of search intensification and diversification [17]. Because the algorithm lacks the ability to detect or escape local minima, it is generally not used in its standard form. Variations of iterative local search are described in [36], which typically implements more sophisticated termination criteria.

2.1.2.2. Swarm Intelligence

The Swarm Intelligence (SI) is a population-based paradigm for solving optimization problems and has been inspired by the collective behaviour of a group of insect colonies or other animal societies. Many such organisations can be observed in nature, such as ant foraging behaviour, fish schooling, animal herding, bird flocking, and many more. Although there is usually no central entity dictating how the swarm individuals should behave, the interaction between the agent and the environment often leads to the emergence of global and self-organizing behaviour [17]. For example, individual ants do not exhibit sophisticated behaviour; however, many ants in an ant colony working together can achieve more complex tasks [37]. In recent times, swarm intelligence has seen growth in popularity for solving NP-hard problems where finding global optima becomes increasingly hard in real-time scenarios [38]. Most common problems include the travelling salesman problem (TSP), feature selection, robot swarm learning, clustering and scheduling [39].

Although other emerging SI algorithms based on the behaviour of glow-worms, lions, wolfs, bats and monkeys exist [38], [40], this section focuses on the most common SI algorithms. Ant Colony Optimization (ACO) is discussed separately in section 2.1.2.4.

- **Particle Swarm Optimization (PSO)**

Particle Swarm Optimization (PSO) was first introduced in 1995 by [41] as global optimization algorithm; however, the concepts of autonomous agents (particles) can be traced back to 1983 [38], where the idea of many individuals working together for creating a fuzzy graphical object was used in early animation [42]. PSO is inspired by the flocking behaviour of birds or fish schooling. The technique is often compared to that of an evolutionary optimization, where the population is randomly sampled and evaluated to determine the best solution, and the process is repeated over many iterations. However, unlike evolutionary algorithms, each particle also has a velocity and memory assigned to it [14]. Individuals in the population – particles, move around in a search space. During movement, each particle adjusts their position according to that of their own experience or the whole population's experience. Therefore, PSO combines the local search (through self-experience) with global search methods

(through neighbouring experience), balancing the intensification and diversification of the search [43].

A detailed description of PSO types and a survey of its hybrids are available in [44]. Because PSO is population-based, meaning, each of the agents can build solution independently at any given iteration, many parallel implementations have been explored. For example, [45] summarizes all parallel PSO implementation, including the usage of graphical processing units (GPUs). The PSO is popular across a range of research areas; thus, many surveys have been carried out. For instance, authors in [46] summarize recent advances of PSO in the solar energy domain. Furthermore, [47] carried out a PSO survey in filter-based classification and [48] focused on PSO for feature selection.

- **Artificial Bee Colony (ABC)**

There are multiple bee-inspired optimization algorithms like Bee Colony Optimization (BCO) [49], Virtual Bee Algorithm (VBA) [50], beehive algorithm [51], Discrete Bee Dance Algorithm (DBCA) [52] and other variations. This section focuses on the most popular honeybee inspired algorithm – Artificial Bee Colony (ABC) [53]. Just like honeybee colonies in nature, the ABC algorithm divides all the bees into three categories based on their purpose in the colony. A colony is composed of Employed Bees (EB), Onlooker Bees (OB) and Scout Bees (SB). The employed bees are responsible for searching for new food sources and providing feedback to the bees in the hive (onlooker bees). Based on the provided information by waggle dance, onlooker bees start exploiting these food sources. As the nectar amount of a food source increase, the probability of visiting the source by onlooker bees rises. Once the food source is exhausted (due to intensification), scout bee is responsible for finding a new food source. The bees iteratively look for new food sources while improving solution till termination criteria is reached [54].

A comprehensive summary of the latest advances in ABC algorithms is provided in [54]. ABC algorithm has been applied to many NP-hard problems, both in benchmarks such as travelling salesman problem (TSP) and real-life applications like image segmentation [55], well placement [56], solving sudoku, job shop scheduling and many others [57]. Furthermore, a survey in [58] summarizes ABC's many application areas in a wide range of engineering domains.

- **Cuckoo Search Algorithm (CSA)**

Cuckoo search algorithm (CSA), created by [59] in 2009, is a novel population-based algorithm that mimics the obligate brood parasitism behaviour of a bird called the cuckoo. Some cuckoos have involved a way that allows their parasitic females to imitate the eggs of few chosen host species. This phenomenon reduces the probability of the host birds detecting the parasitic egg. If the host birds discover alien egg, they either dispose of the intruder egg or abandon their nest altogether [60]. The CSA combines this obligate brood parasitic behaviour with *Lévy flight*, a type of random walk with step-lengths calculated according to heavy-tailed probability distribution [61]. The CS algorithm is based on three simplified and idealized rules: a) each cuckoo lays one egg at a time and dumps it in a randomly chosen nest; b) the best nests with high quality of eggs (solutions) will carry over to the next generations; c) the number of available host nests is fixed, and a host can discover an alien egg with probability following a normal distribution. In the last case, the host bird can either throw away the egg or abandon it to build a new nest in different locations [59].

Even though CSA is one of the newer SI family algorithms, numerous implementations and applications are found in the literature [62]. For instance, [63] explores standard CSA modifications, commonly used parameter settings and different hybrids in detail. Furthermore, [61] investigates the broad area of real-world CSA applications, such as medical applications, clustering and data mining, image processing, energy and economic load dispatch problems, to name a few.

- **Firefly Algorithm (FA)**

Firefly Algorithm (FA) proposed by [64] in 2008 is another recent metaheuristic optimization algorithm. It is based on how fireflies attract mating partners or warn potential predators by their flashing light, produced by the biochemical process – bioluminescence. In the FA implementation, all fireflies are assumed to be unisexual so that any individual firefly is attracted to all other fireflies. The attractiveness is proportional to the brightness of the flash, and they both decrease as distance increases. Thus, for any two flashing fireflies, the less bright individual will move towards the brighter one. If the brightness of both fireflies is the same, the fireflies will move randomly. The landscape of the objective function determines the brightness of a firefly.

Compared to other metaheuristics, FA can solve both continuous [65] and discrete [66] optimization problems. Multiple variations of the algorithms where explored and compared in the study conducted in [67]. Application areas include various optimization, classification and wide range of engineering applications, all summarized in [68] and [69]. Recently, FA has been explored as viable option in combination with neural networks [70].

2.1.2.3. Evolutionary algorithms

Evolutionary Computation (EC) is a category of population-based metaheuristics inspired by the Darwinian principles of evolution of living beings. The beginnings of applying Darwinian principles to solve computing problems can be traced back to sixties, where three different implementations of the idea developed separately for many years [71]. In the USA Fogel introduced the evolutionary programming [72], while Holland referred to his as genetic algorithm [72]. Furthermore, in Europe, Rechenberg [73] and Schwefel [74] called theirs – Evolution strategies. Only in the early nineties, these different representatives of one technology were labelled under one name – evolutionary computing. After a while, concepts of genetic programming [75] were also introduced [71].

These days there are numerous variations and adaptions of the classical evolutionary algorithms (surveys in [76], [77] and [76] describe them in detail); however, they all follow principles of natural selection (survival of the fittest individual) in a population. Evolutionary algorithms can therefore be structured based on [78] as follows: a) one or more individuals are competing for constrained resources; b) population changes dynamically due to the cycle of death and birth of individuals; c) a notion of fitness, which reflects the ability of individual to survive and reproduce; d) offspring closely resembles their parents, but are not identical.

The following section briefly describes the most common evolutionary algorithms. Evolution Strategy algorithm (ES) and Imperialist Competitive Algorithm (ICA) are discussed separately in section 2.1.2.5 and section 2.1.2.6, respectively.

- **Genetic Algorithm (GA)**

The Genetic Algorithm (GA) is one of the most well-known and most applied algorithms out of evolutionary computation family. Developed by John Holland in the early 1970s [72], it has gained interest in various research communities. The genetic

algorithm starts with a set of solutions called *population*. Each solution is represented by a *chromosome*, that encodes a set of genes. The simplistic implementation of GA is very generic and is usually adapted based on the problem solved: representation of the chromosome, selection strategy, crossover and mutation operators. Chromosomes are evaluated based on their *fitness* – ability to survive and reproduce, over iterative process called *generations*. In each generation, individuals are selected for reproduction by exchanging some of their parts – crossover. After crossover, individuals are subjected to a mutation operator, based on mutation rate. Mutation operator introduces some randomness in the search. This process continues till a termination criterion is met.

There are multiple GA selection schemes, for instance, roulette-wheel selection, tournament selection, ranking selection and others. A comprehensive comparison of the selection methods used is described in [79]. Furthermore, the crossover is another important GA operator with many different implementations, single point and n-point crossover being the most common. However, more sophisticated implementations like *uniform, three-parent, arithmetic, partially mapped* crossovers have also been proposed. Both [80] and [81] provides detailed descriptions of different crossover methods. There are also multiple mutation strategies; the GA survey in [82] describes them in detail.

The popularity of GA has resulted in numerous variants of the algorithm and its application to a wide range of optimization problems. Overviews of recent advances tend to be surveyed in specific research fields, such as genetic algorithms applied in operation management [83], supply chain management [84], lens design [85], composite structure design [86], scheduling [87] amongst others. Moreover, parallel implementations of GA were explored in [88].

- **Coevolutionary Algorithms (CoEAs)**

Coevolution is the mechanism by which two or more species evolve in tandem by interacting with each other. Examples of coevolutionary processes include hosts and parasites, predators and prey, insects pollinating the flower and other cooperative or symbiotic relationships. Biological coevolution occurs in many natural processes and has been the inspiration for Coevolutionary Algorithms (CoEAs). Compared to single population evolutionary algorithms, CoEA consists of two or more populations of species that continuously interact and co-evolve simultaneously [17]. Although there

are many variants of CoEAs, the most common categories are *competitive coevolution* and *cooperative coevolution.* As names suggest, in competitive coevolution populations compete during optimization and individuals are rewarded at the expense of those they interact with. Conversely, in cooperative coevolution, individuals are rewarded when working with other individuals and punished when they perform poorly together [89].

In the cooperative coevolution, the different species live together for a mutual benefit – symbiosis. The first cooperative coevolution algorithm was proposed in 1994 by Potter and De Jong [90]. The idea was to divide a complex problem into sub-problems, where each of the sub-problems is assigned to a population. These populations evolve independently and only interact to obtain fitness. Cooperative CoEAs are often integrated into other metaheuristics like cooperative PSO ([91],[92]) and cooperate ABC [93].

Competitive coevolution, however, simulates competing forces in nature like predators and prey, where prey evolve to defend themselves better, while predators develop better attack strategies. It was first introduced by Hillis [94] for sorting networks, where one population was assigned a set of sorting networks (the hosts), and another population was assigned the test cases (the parasites). Fitness was given to each of the sorting network based on the ability to solve the test case. Furthermore, each test case was assigned fitness based on the number of times the networks incorrectly sorted it. This process allowed both populations to evolve simultaneously while interacting only through fitness function evaluations [89]. Moreover, competitive coevolutionary models are well suited for models where it is difficult or impossible to formulate an objective fitness function explicitly [16].

Summaries of recent advances of coevolutionary algorithms are discussed in [95]. Furthermore, CoEAs applications to multi-objective optimization problems reviewed in [89].

- **Genetic Programming (GP)**

Genetic Programming (GP) was first introduced in by Koza [75]. Although based on a similar strategy as a GA, the GP offers a more high-level automated approach for creating a computer program based on the goal of the problem [17]. GP still uses the same genetic operators as selection, crossover and mutation; however, the solutions are based on the decision rules (variables or *terminals*) and arithmetic operations –

functions and not fixed-length string like in a GA. GP solutions are usually expressed as a tree, where the tree leaves are the terminals, and the arithmetic operators are the internal nodes. Like in a GA, the initial population of computer programs (individuals) is usually generated randomly and evolved over many generations to improve the fitness value. In every generation, each program is evaluated based on the fitness function, which determined the program's ability to survive and reproduce [96].

There are many types of GP algorithms, and authors in [97] classified them in eight major types: Tree-based GP, Stack-based GP, Linear GP, Extended Compact GP and Grammatical Evolution GP. Applications of GPs range from biological and genome structure optimization [98] to image processing [99], scheduling [96], forecasting [100] and many more. Summary of all types of GPs and their application areas is structured in a recent survey in [97].

2.1.2.4. Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) takes inspiration from ant cooperative behaviour of finding a food source. The origins of the phenomena can be traced to the double bridge experiment by Deneubourg et al. in [101]. A controlled experimented of ants' movement was conducted by constructing two variable-sized bridges between the food source and the ant nest. At the start, ants moved randomly in all experiments but eventually converged to the single shortest path. In experiments where both bridges were equal size, ants converged towards using one of the two bridges. When the experiment was repeated multiple times, each of the two bridges was used an equal number of times. It was concluded, that because ants lay down pheromone, their behaviour is influenced by the pheromone's concentration on the chosen path, i.e. higher concentration – more likely ant will choose the given path. Furthermore, naturally, pheromone scent evaporates on the longer, no more used paths, giving a higher probability of re-deposit of pheromone on shorter, more appealing paths [102]. This behaviour is modelled in **Figure 3**, where initially ants move randomly, before converging to the single shortest path at the final stage of the search.

This ant behaviour was a base for the development of Ant System (AS) algorithm by Dorigo [103]. The initial work in [104] looked at three different AS – *ant-density*, *ant-quantity* and *ant-cycle,* and solved small TSP instances. A pheromone is deposited on the graph's edges in the ant-density approach if ant moves between two connected nodes. Furthermore, in ant-quantity, the pheromone is deposited over the distance of

the edge between two connected nodes. Finally, ant-cycle only deposited pheromone when ant completes the full tour. It was concluded that ant-cycle approach offers the best results [104].

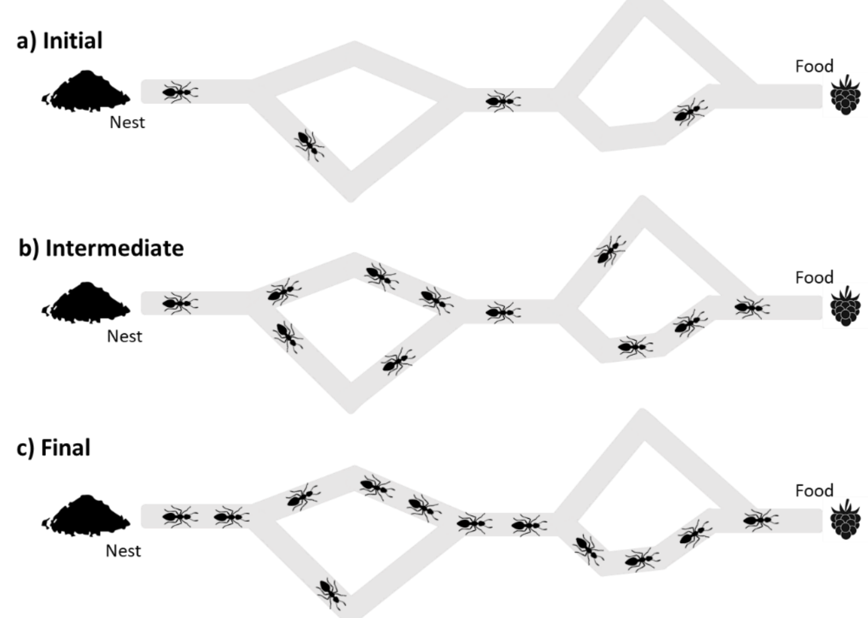

Figure 3. *Ant behaviour (inspired by [105])*

Although AS could solve small TSP instances, it was not competitive to other state-of-the-art algorithms at the time for solving TSP. One of the first improvements was the introduction of elitist ant in the Elitist Ant System [104], where the best performing ant (elitist) in the iteration has more pheromone weight compared to other ants. Since then, many other improvements for the original AS have been explored [102], with Ant Colony System (ACS) being one of the most adopted by researchers.

The ACS was first introduced by Dorigo et al. [106] for effectively solving TSP instances as an improvement to the original AS. The improvements were as follows: first, the ant state transition rule included a direct way of controlling the balance exploitation and exploration, by the introduction of the *pseudo-random-proportion rule.* Second, the *global pheromone updating rule* is only applied on the edges of the best performing tour in the iteration. Finally, a *local pheromone update rule* is applied while ants are construction solutions. Therefore, as in AS, ants build their solution based on the greedy state-transition rule and local pheromone. Once all ants have completed their tours, the pheromone is updated again based on the global pheromone update rule. Furthermore, just like in AS, ants are guided by heuristic information (preference for shorter routes) and pheromone information (routes with more pheromone are more

desirable). The pheromone update rules are designed such that pheromone is deposited on the trails that should be visited by future ants [106].

Ant Colony Optimization (ACO) has a long history for solving TSP instances [105]; however, it can also be applied to other NP-hard optimization problems, like vehicle routing [107], various types of assignment problems [108], scheduling [109], subset selection [110] and even machine learning [111] and bioinformatics [112]. This metaheuristic algorithm's versatility and its recent application areas are detailed in both [113] and [105]. Moreover, the parallel implementations of ant colony optimization surveyed in [114].

2.1.2.5. Evolution Strategy (ES)

Just like genetic algorithm, Evolution Strategies (ES) was inspired by the principles of natural evolution. Initially created as a technique for automated experimental design optimization by Rechengerb [73] and later adopted by Schwefel [74]. The first implementation of ES was a simple algorithm that used only mutation and selection, called *two membered ES*. In the two membered ES, each parent produces a single child by means of mutation. Furthermore, the selection process determines the fittest individual to become the parent of the next generation. This scheme is also referred to as *(1+1)-ES* as it contains a population of one parent individual and one descendant [115]. The basic flow diagram is shown in **Figure 4**.

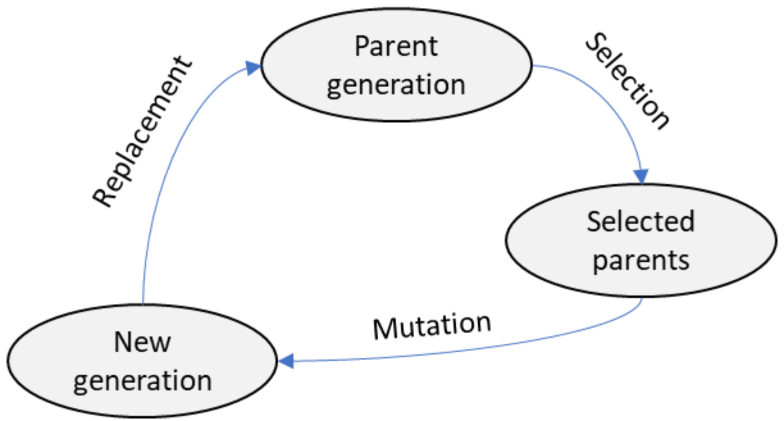

Figure 4. *Evolution cycle of Evolution Strategy*

The (1+1)-ES got extended to a population-based ES, referred to as *multimember* ES. In multimember ES, multiple parents ($\mu > 1$) can participate in the creation of the offspring individual, therefore denoted as (μ+1)-ES. As there are multiple parents, additional recombination operators are possible – two of the μ parents are selected

randomly to create the child. The selection undergoes “survival of the fittest” evolution, by eliminating the worst offspring while keeping population size constant. The rise of parallel computers was the motivation of the extensions of (μ+1)-ES, such as (μ+λ)-ES and (μ,λ)-ES [115]. In the (μ+λ)-ES, multiple parents μ are able to produce multiple children λ by means of mutation and recombination, worst performing individuals are discarded to maintain the population size constant. Moreover, in the (μ,λ)-ES the parents μ, that are creating the next generation λ, are discarded unconditionally of their fitness values [116].

In contrast to GA, where most of the research focuses on the recombination and crossover operators, ES mutation is the dominating operator for the search [117]. The mutation is usually implemented as a normal (Gaussian) distribution with a step size σ. The most straightforward implementation keeps mutation step constant, while more sophisticated implementations adopt σ dynamically. Dynamic mutation step approaches include 1/5 success rule [73], the σ-self-adaption [73], the meta-ES [118] and others [119].

Variations of ES are commonly applied to machine learning [120], constrained optimization [121], finance [122], among others. Detailed theoretical investigation of evolution strategy algorithm search performance is available in [123] and [119].

2.1.2.6. Imperialist Competitive Algorithm (ICA)

Imperialism is a policy to extend an empires or nation's rule or jurisdiction over other nations or establish and retain colonies and dependencies. In modern colonialism, more developed countries are attempting to dominate less developed countries to extend their power by political-military alterations. The drive for influence motivates imperialist competition, which consequently creates a race of political, military and economic development amongst the imperialist countries. Once the country has been colonized, its empire attempts to spread its cultural values, by building schools, libraries, railroads and other public infrastructure. An excellent example of such influence on culture is British colonization of India, where English was taught extensively in schools and gradually became India's second language [124].

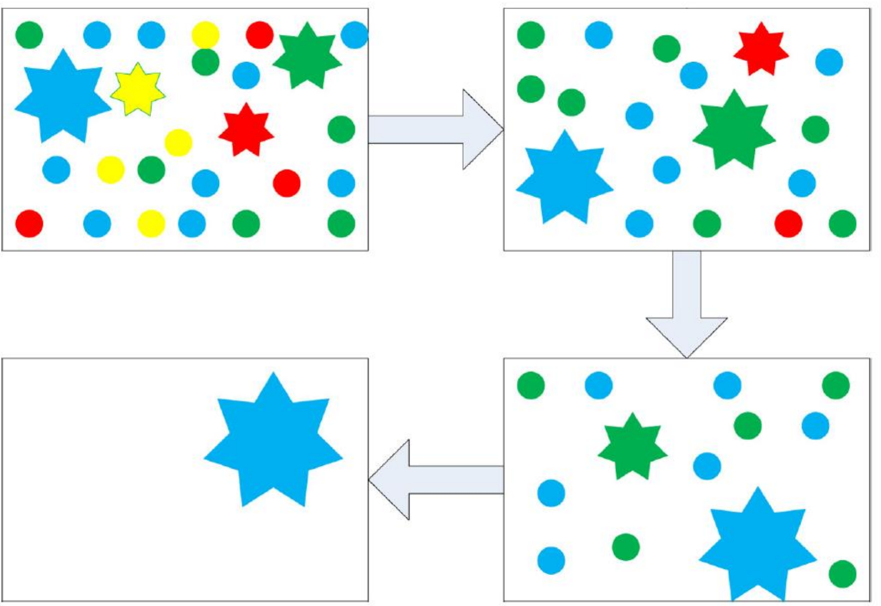

***Figure 5**. Convergence representation of ICA [124]. With stars representing empires and circles – their colonies.*

Imperialist Competitive Algorithm (ICA) is a subset of metaheuristic algorithms modelled based on geopolitical behaviour. It can also be classified as a social Darwinism that follows evolutionary computing principles. Atashpaz-Gargari and Lucas first proposed the ICA in [125] for solving continuous cost functions and since has generated interest amongst many researchers. Like many other population algorithms, ICA starts its search by generating a random initial population where each individual of the population represents a *country*. Countries within ICA can be thought of as chromosomes in a genetic algorithm. The initial population is separated into multiple groups (so-called *empires*). Strongest countries become *imperialist* within the empire, while weakest - their colonies. Each colony within empire moves closer to their imperialist in the form of *assimilation* operator. In order to provide diversity amongst countries, a *revolution* operator (mutation in GA) is implemented. If a colony becomes stronger than its imperialist at any point, then the two countries are swapped, such that imperialist is the strongest country in the empire. The search follows an iterative process. After each iteration, the weakest colony within the most powerless empire is assigned to one of the stronger empires – following the *imperialist competition* process. An empire is eliminated once it contains no more colonies. The search usually continues until the termination criteria are met. Ideally, the search is terminated once all empires are eliminated and only one, the best, empire remaining. This convergence process is shown in **Figure 5**, where a star represents an empire

and circles – their colonies. As the search process progresses, the number of empires shrink, while the remaining empires gain power. At the final stage, only a single empire remains.

Although initial ICA was created for continuous problems, researchers have extended the algorithm to solve various NP-hard discrete problems. According to the survey performed in [124], ICA is most commonly applied for industrial engineering problems, scheduling in particular. Though most recently applications such as prediction [126][127], clustering [128] and encryption [129] have emerged, demonstrating the versatility and wide application areas of the algorithm.

2.2. Metaheuristic Optimization Frameworks (MOFs)

One of the characteristics of metaheuristics described in the previous section – problem non-specificity, has inspired many metaheuristic optimization frameworks (MOFs) in the last two decades. These frameworks aim to provide simplified and standardised techniques for solving a wide range of optimization problems using pre-implemented metaheuristics. Furthermore, the user can take advantage of already implemented and debugged high-performance algorithms with little additional effort. All the supporting tools, like monitoring, reporting, parallel and distributed computing, are already integrated, and therefore the user can focus the efforts only on the problem on hand.

However, generic optimizers are argued as impossible [130]. Similarly, The No Free Lunch (NFL) theorem [1] states that no single strategy or algorithm always performs better than another for all possible problems. And the ideology behind MOFs follows this logic – there is no single metaheuristic algorithm that will solve all problems to an optimal solution; therefore a wide range of metaheuristics are available to be matched to the specific problem. The saying "Jack of all trades, master of none, oftentimes better than a master of one" is applicable when referred to MOFs. By definition, metaheuristics cannot guarantee optimal solutions but offer close to optimal solutions for wide range of problems. Furthermore, MOFs facilitate re-use and comparisons of metaheuristics, therefore allowing user to focus only on the algorithms that perform the best for the problem.

Numerous frameworks for problem-solving using metaheuristics are found in the literature with similar features and usage scenarios. Authors in [131] identified three

main MOF usage scenarios: industrial application, research and teaching. In the industrial application scenario, the MOFs reduce the implementation burden. For practitioners, optimal search performance and the ease of use are the most valuable features. Furthermore, when the frameworks are used for research on metaheuristics and optimization problems, the monitoring and analysis tools are preferred. Finally, graphical representations of the solutions, reports and the ease of use are likely valued for MOFs used in teaching. Moreover, a recent study in [132] focused on comparing and analysing different MOFs, especially multi-agent structures and the hybridization of metaheuristics.

Both [131] and [132] list numerous MOFs found in the literature, the sixteen of the most relevant MOFs are summarized in **Table 1**, sorted by the year of inception. Furthermore, technology such as programming language and platform used is also listed. The table also shows the available metaheuristic algorithms and the benchmark discrete combinatorial problems solved. If the framework has had a software update or any research contributions in the past year, it is labelled as active.

The next section briefly introduces each of the MOF, while overall comparisons and limitations are discussed in section 2.2.1.

- **MALLBA**

The MALLBA project was started in 2000 by [133] and is based on the concept of skeletons in C++. The aim is to create a library of skeletons for combinatorial optimization (including exact, heuristic and hybrid methods) for easy and efficient parallelism. It is targeted to sequential computers and LAN or WAN clusters. The skeletons refer to classes that are required to be implemented for any given algorithm. Although the framework offers multiple population-based algorithms like GA, ES, ACO and PSO, it lacks documentation and examples. The latest version of this framework[1] was presented in [134] and since has been abandoned.

[1] MALLBA project website at http://neo.lcc.uma.es/mallba/easy-mallba/

Table 1. *Summary of the most popular metaheuristic optimization frameworks sorted by creation year.*

Framework	Year	Active	Technology	Algorithms	Discrete combinatorial examples
MALLBA [133]	2002	No	C++	GA, SA, ES, ACO, PSO	-
ParadisEO [135]	2004	No	C++	PSO, GA, EA	TSP, DVRP
HeuristicLab [136]	2004	Yes	C#	ES, GA, PSO, TS, VNS, GP	JSS, KP, QAP, TSP, VRP
BEAGLE [137]	2006	No	C++	GA, ES, GP	KP, TSP
JCLEC [138]	2008	No	Java	GA, GP	KP, TSP
JCOP [139]	2009	No	Java	GA, SA	JSS, KP, SAT, TSP
OptFrame [140]	2010	Yes	C++	EA, SA, TS, LS	VRP, TSP, KP
EvA2 [141]	2010	No	Java	ES, GA, DE, PSO, SA	KP
jMetal [142]	2011	Yes	Java	GA, PSO, ES, DE	mQAP
Opt4j [143]	2011	No	Java	EA, DE, PSO, SA	KP, TSP
ECJ [144]	2012	Yes	Java	ES, SA, AS, ACO, PSO, DE, GP	KP, SAT, TSP
HyperSpark [145]	2015	No	Scala, Apache Spark	GA, SA, TS, ACO	PFSP
JAMES [146]	2016	No	Java	TS, LS	KP, TSP
EvoloPy [147]	2016	Yes	Python	MFO, MVO, BAT, FA, CSA, GWO, WOA, PSO	-
jMetalSP [148]	2018	Yes	Java, Apache Spark	GA, PSO, ES, DE	mQAP
jMetalPy [149]	2019	Yes	Python, Apache Spark	GA, PSO, ES, DE	mQAP
OptPlatform (this work)	**2020**	**Yes**	**C#, C++**	**ES, ICA, ACO**	**MKP, MDVRP, TSP, ASP**

- **ParadisEO**

ParadisEO (Parallel and Distributed Evolving Objects) is a white-box C++ framework that offers parallel and distributed metaheuristics. Created in 2004 by [135] and has evolved to support multiple modules: EO provides a set of classes for the development of population-based metaheuristics (ES, GA, PSO); MO provides tools for trajectory-based metaheuristics; MOEO provides tools for implementation of evolutionary techniques for multi-objective optimization; PEO provides classes for parallel and distributed applications and finally MO-GPU for GPU implementation [150]. The GPU implementation is one of this framework's unique features; however, the authors concluded that the application areas for speedup might be limited. The

platform[2] is well documented, however, appears to be no longer supported, with the last update in 2012.

- **HeuristicLab**

HeuristicLab is another MOF that has been in constant development since 2002. Heuristic and Evolutionary Algorithm Laboratory (HEAL) first presented the framework in [136]. The long development has allowed HeuristicLab to be one of the leading and most feature-rich frameworks available today. It integrates population-based metaheuristics like GA, ES and PSO and multiple trajectory-based algorithms like LS, TS, VNS. Although initially developed for heuristic optimization, the software has evolved and integrated aspects of machine learning using genetic programming and classification techniques. Amongst many other features, HeuristicLab offers well established GUI, SDK and extension called HeuristicLab Grid for support of grid computing. Both [151] and [152] offer comprehensive reviews of the framework's features and limitations. One of the main limitations is the use of C#, therefore supported natively only on Windows. Furthermore, authors in [152] state the lack of documentation as another drawback on such a feature-rich framework, although many problem examples and tutorials exist. Latest version 3.3.16[3] was released in 2019 and therefore is still in active development.

- **Beagle**

Beagle is an open-source Evolutionary Computing (EC) framework proposed in 2006 [137]. The framework explicitly focuses on traditional EC, i.e. Genetic Algorithm (GA), Evolutionary Strategy (ES) and Genetic Programming (GP). It also introduces six basic configurable principles for creating new EC algorithms: representation of chromosome, fitness, operators, evolutionary model, parameter management and configurable output. Furthermore, the framework uses C++ implementation with XML structures for data management. The project directory[4] suggests that no advances or updates have been introduced since 2017. Examples include various GP benchmarks and Knapsack (KP) and Travelling Salesman Problems (TSP). The framework appears to be strictly limited to EC development and does not support non-EC metaheuristics.

---

[2] ParadisEO project website at http://paradiseo.gforge.inria.fr/index.php
[3] HeuristicLab project website at https://dev.heuristiclab.com/
[4] Beagle project directory available at https://github.com/chgagne/beagle

- **JCLEC**

In [137], the authors presented a Java-based framework for JCLEC (Java Class Library for Evolutionary Computing). The software is split into three modules: JCLEC core that specifies the data types and functionality; JCLEC experiments runner that is responsible for the algorithm execution and finally the GenLab – a graphical interface for rapid prototyping. Just like many other frameworks, it explicitly focuses on evolutionary algorithms. A case study of the Knapsack problem was performed in [138]; however, no comparisons or results were presented. Currently, the framework's development has been abandoned, with the last version of JCLEC 4.0.0[5] released in 2014.

- **JCOP**

JCOP (Java Combinatorial Optimization Platform) was developed as part of Ondřej Skalička's master thesis in 2009 [139]. One of the project's main aims was to develop a platform in which any of the implemented problems can be solved by any of the available algorithms without the need for customization per algorithm. Author of JCOP implies that the platform was not designed to be fast but rather a tool to choose the best amongst multiple algorithms. The framework implements basic GA and SA. Furthermore, the platform is well documented and includes numerous combinatorial problem examples. Project GitHub page[6] suggests that the project has not been updated since 2014.

- **OptFrame**

OptFrame [140] aims to provide a simple C++ interface for standard components of trajectory and population-based metaheuristics. Authors claim to deliver a smarter version of traditional methods to consider problem-specific characteristics. The software is structured based on two container classes – Solution and Evolution. Evaluator class allows the implementation of both single and multi-objective functions. Furthermore, the framework also supports parallelism with shared and distributed memory, and basic GA and TS algorithms. The project is well documented with multiple examples, and the latest OptFrame v4.0[7] integrates C++20 features.

---

[5] JCLEC project website at http://jclec.sourceforge.net/index.php
[6] JCOP project website at http://jcop.sourceforge.net/en/index.html
[7] OptFrame project website at https://github.com/optframe/optframe

- **EvA2**

EvA2 (an Evolutionary Algorithms framework, revised version 2) is a heuristic optimization framework with an emphasis on EA implemented in Java. It is an improved version of previous JavaEvA optimization toolbox. EvA2 is being used as a teaching aid in lecture tutorials and is aimed to two groups of users: non-expert user that wants to apply EA for solving application problem and scientist that want to use the platform for algorithm development or performance comparisons. EvA2 implements various population-based algorithms, like ES, GA, DE, PSO and trajectory-based techniques like SA. Furthermore, the framework also offers a simple GUI and integration with MATLAB. Based on the project page[8], the latest version of 2.2 was published in 2015 and is no longer in active development.

- **jMetal**

jMetal is another framework with a long history of development, dating back to the introduction in 2010 by [142]. jMetal stands for Metaheuristic Algorithms in Java and follows object-oriented principles. Although some implementations of single-objective optimization problems exist, the framework mostly focuses on multi-objective problems. Based on evolutionary algorithms, it follows the structure of an *Algorithm* that solves a *Problem* using one or more *SolutionSet* and a list of *Operator* objects. Both *SolutionSet* and *Solution* classes allow the representation population and individuals in population-based metaheuristics. The framework incorporates multiple multi-objective tools, such as Pareto convergence quality indicators, statistical tests, and GUI due to the multi-objective problem focus. Just like the majority of other frameworks, jMetal implements GA, ES and PSO. The platform offers detailed instructions and documentation[9]. More recently jMetal migrated to Maven and version 6 is in active development.

- **Opt4j**

Another evolutionary computing framework based on Java was presented in [143], called Opt4j. It uses modular in design and uses the genotype-phenotype principles for the solution encoding. Compared to other similar frameworks, Opt4j explicitly implements functions that translate genotype into phenotype and vice versa. It also

[8] EvA2 project website at http://www.ra.cs.uni-tuebingen.de/software/EvA2/
[9] jMetal project website at https://jmetal.github.io/jMetal/

implements a graphical interface where users can view and analyse the performed tests and perform optimization without code. Unfortunately, the framework[10] has not been updated since 2015 and is limited to EA class algorithms.

- **ECJ**

ECJ is a research Evolutionary Computing system written in Java [144]. The ECJ framework is one of the most established metaheuristic frameworks with extensive documentation. It is developed by dozens of research contributors and has covered many features such as GUI with charting and support for parallelism and distributed computing. The GUI allows loading and executing algorithms based on checkpoint and parameter files, editing parameters and charting statistics. Although metaheuristic algorithms like AS and PSO are supported, the framework mainly focused on EC and GP for continuous optimization problems. Work in [153] summarizes the recent advances of the ECJ and concludes that support for combinatorial optimization is lacking. Furthermore, out of the 23 available benchmark examples, only three – KP, SAT, TSP – are for discrete combinatorial optimization. The framework is in active development, and the latest version (27th iteration) is available on their website[11].

- **HyperSpark**

HyperSpark is a cloud computing oriented metaheuristic framework first introduced in 2015 as part of Master thesis [145]. The framework focuses on the area of Big Data processing with the use of Scala and Apache Spark. It supports population-based algorithms like GA and ACO, as well as a trajectory-based search like SA and TS. More recently, the authors refined HyperSpark to solve the Permutation Flow-Shop Problem (PFSP) [154]. The work in [154] shows that HyperSpark is struggling to scale across the cluster due to the overheads.

Furthermore, when compared to other MOFs, HyperSpark lacks in the ability to produce good quality solutions. Moreover, the framework is implemented in a less-used programming language (Scala), leading to slower adoption. Similarly, the software lacks documentation and working examples. The framework was last updated in 2016 and is available on the project GitHub page[12].

---

[10] Opt4j project website at http://opt4j.sourceforge.net/index.html
[11] ECJ project website at https://cs.gmu.edu/~eclab/projects/ecj/
[12] HyperSpark project website at https://github.com/deib-polimi/hyperspark

- **JAMES**

JAMES (Java Metaheuristic Search)[13] was developed to solve discrete optimization problems using local search algorithms [155]. The software emphasises the separation of a problem specification and the search algorithm. Although limited to forms of iterative local search, the platform is well documented and has multiple examples. Unfortunately, since the introduction in 2016, the development on the framework has stopped.

- **EvoloPy**

EvoloPy is a relatively new nature-inspired optimization framework in Python [147]. It focusses on the implementation of the most recent metaheuristic algorithms such as Grey Wolf Optimizer (GWO), Multi-Verse Optimizer (MVO), Moth-flame Optimization (MFO), Whale Optimization Algorithm (WOA), Bat algorithm (BAT), Cuckoo Search Algorithm (CSA) and Firefly algorithm (FA). As it is a new framework, documentation is limited, and currently, only 23 benchmarks based on math equation optimization are made available. However, activity on the project GitHub[14] suggest that the project is in active development.

- **jMetalSP**

jMetalSP [148] is another extension of jMetal that offers parallel computing features based on apache Spark. Mainly aimed for dynamic multi-objective Big Data optimization problems. The case study of 100 city TSP showed the advantages and limitations of the system. More recently, authors in [156] adopted the framework to integrate various streaming services for dynamic multi-objective optimization. The framework[15] is still in active development.

- **jMetalPy**

Just recently, a Python implementation of jMetal was proposed in [149]. The framework implements most features from jMetal while leveraging Python's visualization and statistical tools for easier analysis. The framework[16] is very recent and still in active development.

[13] JAMES project website at http://www.jamesframework.org/
[14] EvoloPy project website at https://github.com/7ossam81/EvoloPy
[15] jMetalSP project website at https://github.com/jMetal/jMetalSP
[16] jMetalPy project website at https://github.com/jMetal/jMetalPy

2.2.1. MOF trends and limitations

Analysing the MOFs discussed above leads to conclude that generally these frameworks are created for academic research – in which algorithm dynamics and comparisons are performed on benchmark datasets, but rarely adopted by practitioners for solving real-world problems. This leads most (9 out of 16 MOFs discussed here) platforms to be abandoned after the research project is complete. On the other hand, the platforms that have gained traction in research community like jMetal have multiple adaptions to other languages and architectures – jMetalSP and jMetalPy. The active development in jMetal's github indicate multiple researchers participating in the project. Moreover, some MOFs like HeuristicLab has a long-established history with good documentation and support forums that leads to ever-increasing adoption in the research community. However, only a few references of real-world usage by practitioners can be found in the literature.

The lack of supporting tools for deploying the final solution in real-life, limits the practical applications. For example, automatic parameter tuning is beneficial for users that are not specialists in metaheuristic algorithms; however, none of the above frameworks supports automatic algorithm selection or tuning. Furthermore, only a couple of MOFs, like HeuristicLab and jMetal, support the results' statistical analysis. Another consideration for adaption is the ease of use, and only a handful of MOFs (ECJ, EvA2, HeuristicLab, JCLEC and Opt4j) support graphical interface. Similarly, solution visualisation is limited and supported only on a few platforms. None of them implements tools to guide the user on implementing the proposed solution most effectively.

Moreover, as multi-core CPU architectures become mainstream, these computing resources must be utilised efficiently. Optimisation algorithms are compute-intense, and thus, any MOF should implement parallelism to speed-up the search process. Unfortunately, not all existing frameworks even support basic parallelism, and even fewer do it effectively. The efficiency of parallelism implementation varies and may sometimes can be lost at the cost of higher-level generalisation. However, at least in three MOFs, namely, HyperSpark, jMetalSP and jMetalPy, distributed computing is in the framework's core. Other frameworks, such as HeuristicLab and ParadiseEO, implement parallel computing as separate modules that are generally loosely coupled with the base framework and may not be as efficient. Furthermore, BEAGLE and

JAMES frameworks only support parallelism partially. It is worth noting that recently MOFs based on Apache Spark has become popular, with three out of five MOFs reviewed using the technology in the past five years.

Examining the supported metaheuristic algorithms by each of the frameworks, indicate the overwhelming majority is only supporting solution representation that is well suited for Evolutionary Computing algorithms and their variations – GA, ES, DE and GP. This encoding is also suitable for algorithms like PSO and trajectory-based searches like AS, SA, LS and TS. Only a few platforms, namely MALLABA, ECJ and EvoloPy, are generic enough to implement metaheuristics that do not follow Evolutionary Computing (EC-style) encoding, like the ACO. In particular, EvoloPy is the only MOF that covers a comprehensive and diverse set of metaheuristic algorithms. Even when platform supports multiple metaheuristic algorithms, not all of them can be applied successfully. For example, some frameworks, namely HeuristicLab, limit the usage of metaheuristic algorithm depending on the solution encoding. This is the main limitation of most existing metaheuristic frameworks, as they are not generic enough to accommodate a wide range of metaheuristic algorithms for any problem.

Table 2 summarizes the available features for all sixteen platforms. From this table, only a few features are covered by all evaluated frameworks. None of them covers all of them – which presents a research opportunity in this area – for example, none of the analysed MOFs supports automatic algorithm selection and parameter tuning. Automatic algorithm selection and tuning accelerates the development process and reduces the expert knowledge required to use metaheuristics effectively.

Table 2. *MOF supported features.*

	Characteristics				
Framework	**Statistical analysis**	**Graphical interface**	**Automatic algorithm and parameter selection**	**Parallelism**	**Support for non-EC solution encoding**
MALLBA	No	No	No	Yes	Yes
ParadisEO	No	No	No	Yes	No
HeuristicLab	Yes	Yes	No	Yes	No
BEAGLE	No	No	No	Yes, limited	No
JCLEC	No	No	No	No	No
JCOP	No	No	No	No	No
OptFrame	No	Yes	No	Yes	No
EvA2	No	No	No	Yes	No
jMetal	Yes	Yes	No	Yes	No
Opt4j	No	No	No	Yes	No
ECJ	No	Yes	No	Yes	Yes
HyperSpark	No	No	No	Yes	Yes
JAMES	No	No	No	Yes, limited	No
EvoloPy	No	Yes	No	No	Yes
jMetalSP	Yes	Yes	No	Yes	No
jMetalPy	Yes	Yes	No	Yes	No
OptPlatform (this work)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

2.3. Optimization Problems

This section introduces the optimization problems solved throughout the thesis. It is divided by benchmark problems – problems available in academic literature and more theoretical nature. The second part is real-world optimisation problems – optimisation models based on physical geographical locations and distribution networks.

2.3.1. Benchmark problems

2.3.1.1. Multiple Knapsack Problem (MKP)

The Multidimensional Knapsack Problem (MKP) is a well-known constrained optimisation problem, that has multiple real-world engineering applications, such as cutting stock [157], distributed computing resource allocation [158], cargo loading [159], satellite management [160], project selection [161] and capital budgeting [162]. The MKP is an extension of the 0-1 knapsack problem, where items have weight

vectors in multiple dimensions. The goal is to maximise the total profit by putting items into knapsacks while satisfying weight capacity constraints across all dimensions. MKP is formulated in (1) [163].

$$\begin{aligned} \text{max:} \quad & \sum_{i=1}^{n} (profit_i \times sel_i) & & \\ \text{subject to:} \quad & \sum_{i=1}^{n} (weight_{ji} \times sel_i) \leq W_j & \forall j \in \{1, \dots, m\} & \\ & sel_i \in \{0,1\} & \forall i \in \{1, \dots, n\} & \end{aligned} \tag{1}$$

where every item i in the list of n items ($y = 1 \dots n$) has a profit $profit_i$ and weight $weight_{ji}$ associated with an m-dimensional weight vector ($j = 1 \dots m$), that tries to satisfy a weight capacity constraint W_j in that dimension. Variable sel_i indicates whether the item is selected and included in the solution. Capacities, weights and profits are assumed to be positive.

Being an NP-hard problem with practical applications, many different approaches have been proposed for solving MKP, which can be divided into two groups – exact, deterministic, single-solution based algorithms and stochastic population/meta-heuristic based algorithms, with this thesis focusing on the latter approach.

2.3.1.2. *Multi Depot Vehicle Routing (MDVRP)*

The Vehicle Routing Problem (VRP), first described in 1959 [164], is an extension of the Traveling Salesman Problem (TSP) [165]. Compared to TSP, where an agent has only to visit all cities once, VRP introduces demands for each customer or stop. Demands need to be satisfied by routing vehicles such that they start and finish their paths at the same depot. Many real-life problems can be modelled as a form of VRP, for example, picking up and delivering mail, packages or any other goods or services. Due to the wide range of practical applications, many variations of VRP have since been explored. For instance, capacitated VRP introduces capacity constraints on the vehicles; VRP with Time Windows (VRPTW) requires delivery to happen within a specific time window; VRP with maximum vehicle distance constraints (DVRP) and many others [166].

A common VRP derivation is the Multi-Depot Vehicle Routing Problem (MDVRP). MDVRP is an extension of classical VRP by the introduction of multiple depots. Vehicles in the MDVRP are subject to capacity constraints (how much cargo can be

carried on board) and the route's maximum duration before the vehicle needs to return to the original depot. The MDVRP resembles a lot of everyday transportation, logistics and distribution problems and, therefore, has been a common research area [167]. Furthermore, the MDVRP is also an NP-hard combinatorial optimisation problem; thus, optimal solutions are hard to find [168]. Although exact algorithms for solving these problems exist, they are limited to small problem instances [169]. A wide range of metaheuristics and population-based algorithms have been used [167] to solve larger instances of the MDVRP.

The main aim of the MDVRP is to route a fleet of vehicles from multiple depots to multiple customers requiring goods or services. Figure 6 shows an example of a simple MDVRP solution with ten customers (as circles) and two depots (as rectangles). Although multi-objective approaches exist for solving MDVRP [170], the most common goal is to minimise the total cost.

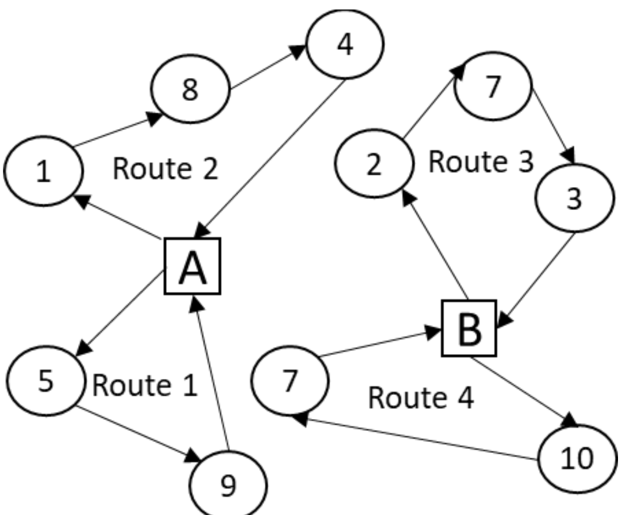

Figure 6. *Example of an MDVRP with ten customers (as circles) and two depots (A and B as rectangles)*

The MDVRP can be formalised in a mathematical model based on [171] and [172]. Given a direct graph $G = (A, \mathrm{B})$ where $A = H \cup D$ is a set of customers $H = \{H_1, H_2, \dots, H_N\}$ and depots $I = \{I_{\ 1}, I_{\ 2}, \dots, I_{\ M}\}$ and B is a set of edges between all the nodes in the graph. In a fully connected graph, every edge B_{ij} between nodes A_i and A_j $(i \neq j)$ has associated positive cost $cost_{ij}$ - distance or time, for example. Each customer has a positive demand d_i $(i \in \mathrm{H})$. Furthermore, there is also a fleet of K identical vehicles available at each depot $I_{\ k} \epsilon\, I$ (that are not allowed to exceed

capacity Q_{max} and duration R_{max}). The goal is to minimise the total cost across all vehicles (2).

$$min \sum_{i \in A} \sum_{j \in A} (cost_{ij} \times trav_{ij}) \quad (2)$$

where $trav_{ij}$ equals to 1 if i comes after j in the customer sequence on any route of all vehicles and 0 otherwise. The problem is subject to the following constraints a) each vehicle route starts and ends at the same depot; b) the total demand on each route does not exceed vehicle capacity Q_{max}; c) the maximum route duration R_{max} is not exceeded; e) each customer is served by exactly one vehicle.

Since the first formulation in [164], many exact and heuristic algorithms have been explored for vehicle routing problems. Most notably, [173] proposed a heuristic approach based on the cost savings algorithm that has since been used in some form in many other algorithms [174]. Another popular heuristics approach was introduced in [175] that allowed problems divided into sub-problems based on vehicles and then solved separately, combining results into a single solution afterwards. Although heuristic approaches such as integer programming [176] and variable neighbourhood search [177] have the potential to find optimal solutions every time, they generally do not scale well with the problem size and are limited to smaller MDVRP instances or are very time-consuming [169].

Meta-heuristic algorithms offer a stochastic approach for solving highly complex combinatorial problems with near-optimal or optimal solutions. They have been a growing interest in many areas [11], and MDVRP is no exception. A recent survey of metaheuristic algorithms [167] suggests that two of the most common algorithms used for solving MDVRP are Ant Colony Optimization (ACO) and Genetic Algorithm (GA). However, other algorithms like Particle Swarm Optimization (PSO) [178] and Ant Lion Optimization (ALO) [179] have also been successfully applied. GA is a nature-inspired algorithm that is based on the natural selection process. A comprehensive summary of methods and approaches used for solving MDVRP with GA is presented in [166]. ACO is another popular approach for solving VRP class problems as it mimics ants travelling and searching for food while creating paths for other ants to follow. Many ACO implementations for MDVRP exist in the literature; the most recent work includes [180] who applied the ACO algorithm for fresh seafood delivery routing problems.

2.3.2. Real-world problems

2.3.2.1. Aerial Surveying Problem (ASP)

Aerial surveys also referred to as drone surveys, Unmanned Aerial System (UAS) surveys or Unmanned Aerial Vehicle (UAV) surveys are becoming popular for surveying from the air. This inspection method offers a faster, safer and more cost-effective way to scan infrastructure objects, such as bridges, roads, wind turbines and rooftops. Furthermore, inspection from the air allows access to remote locations for forestry and agriculture plantations, and fast response for disaster management, such as oil spills, forest fires and earthquakes.

A recent survey in [181] looked at more than 200 articles related to aerial drones used for civil (non-military) applications. In particular, the survey focused on research that formulates an optimisation problem within the UAV domain. One of the most significant areas covered by previous research is UAV routing for a set of locations – these include applications such as surveillance [182] and deliveries [183], in the context of agriculture, infrastructure, transport and disaster management [181]. These aerial surveying problems can be modelled based on simpler routing problems such as the Travelling Salesman Problem or Vehicle Routing Problem.

This section takes a look at one such Aerial Surveying Problem (ASP), that can be modelled as multiple depots, mixed vehicle routing problem with multiple trips, where each of the vehicles can start and return to a different depot, or use a depot for refuelling/charging. A simplified example is provided in Figure 7, where each of the rectangles (A-C) represent a base station (depot), each of the circles (1-9) pose a task/location that needs to be surveyed (visited) once. Furthermore, the routes are colour coded for each of the aircraft.

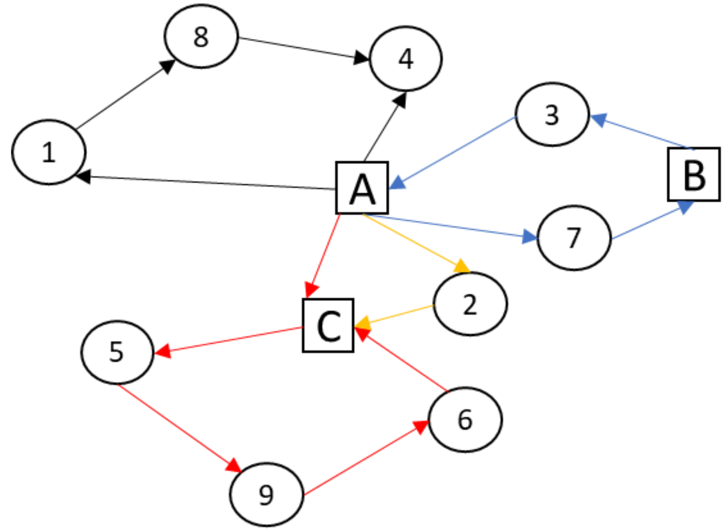

Figure 7. *Graphical representation of simplified Aerial Surveying Problem. Each rectangle (A, B, C) represent a base station and each of the circles (1-9) pose a task/location that needs to be visited. In this example, there are four different routes, route A-1-8-4-A in black, route A-7-B-3-A in blue, route A-C-5-9-6 in red and finally route A-2-C in yellow.*

Path in red in Figure 7 shows an aircraft that fly from base A to C, refuels and proceeds to visit tasks 5, 9 and 6 before returning to base C. The ASP's goal is to survey each of the tasks with the available fleet of aircraft while occurring the least amount of cost. The cost of an aircraft a is a function of the flight time and hourly rate. The total flight time for a path i between two edges is calculated as total distance over the cruise speed of the aircraft a. Therefore, the total cost is a sum of all path flown PF for each aircraft a, calculated in (2).

$$Total\ cost = \sum_{a=1}^{TA} \sum_{i=1}^{PF} (\frac{distance_i}{cruiseSpeed_a} \times costPerHour_a) \qquad (3)$$

where TA is the total number of aircraft used.

Like the vehicle routing problem, each aircraft is subject to the maximum range before it needs to refuel/recharge. Moreover, due to the aircraft's size or type, not all base stations can safely support all aircraft types, so additional aircraft type constraints are applied for each base station. The ASP dataset (made available in [184]) consists of 11 base stations, 10 types of aircraft and 12 locations that need to be surveyed, based on real-world locations and aircraft. This problem is an adopted version of a real-world Intelligence Surveillance and Reconnaissance (ISR) problem as part of Multi-Domain Operations (MDO) challenge [185].

2.3.2.2. *Outbound supply chain problem*

Supply chain optimisation has become an integral part of any global company with a complex manufacturing and distribution network. For many companies, inefficient distribution plan can make a significant difference to the bottom line. Modelling a complete distribution network from the initial materials to the customer's delivery is very computationally intensive. With increasing supply chain modelling complexity in ever-changing global geo-political environment, fast adaptability is an edge. A company can model the impact of currency exchange rate changes, import tax policy reforms, oil price fluctuations and political events such as Brexit, Covid-19 before they happen.

This section looks at a real-world dataset of an outbound logistics network is provided by a global microchip producer. The company provided demand data for 9,216 orders that need to be routed via their outbound supply chain network of 15 warehouses, 11 origin ports and one destination port (see Figure 8). Warehouses are limited to a specific set of products that they stock, furthermore, some warehouses are dedicated for supporting only a particular set of customers. Moreover, warehouses are limited by the number of orders that can be processed in a single day. A customer making an order decides what sort of service level they require – DTD (Door to Door), DTP (Door to Port) or CRF (Customer Referred Freight). In the case of CRF, the customer arranges the freight and company only incur the warehouse cost. In most instances, an order can be shipped via one of 9 couriers offering different rates for different weight bands and service levels. Although most of the shipments are made via air transport, some orders are shipped via ground – by trucks. The majority of couriers offer discounted rates as the total shipping weight increases based on different weight bands. However, a minimum charge for shipment still applies. Furthermore, faster shipping tends to be more expensive, but offer better customer satisfaction. Customer service level is out of the scope of this research.

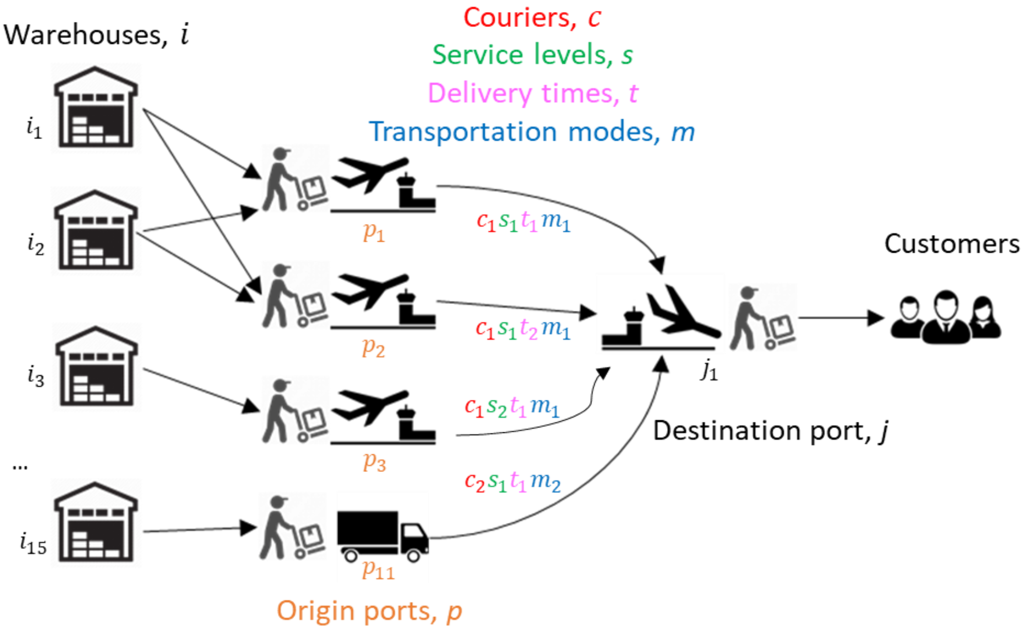

Figure 8. *Graphical representation of the outbound supply chain. Each warehouse i is connected to one or many origin ports p. The shipping lane between origin port p and destination port j is a combination of courier c, service level s, delivery time t and transportation mode m.*

Figure 8 shows a simplified example case of the supply chain model. Warehouses i_1 and i_2 can be supplied by either origin ports p_1 or p_2. In contrast, warehouse i_3 can only be supplied via origin port p_3 and warehouse i_{15} can be only supplied by origin port p_{11}. In the example shipping lane $p_1 j_1 c_1 s_1 t_1 m_1$ is chosen between p_1 and destination port j_1 with courier c_1, service level s_1, delivery time t_1 and transportation mode m_1.

Dataset [186] is divided into seven tables, one table for all orders that need to be assigned a route – OrderList table, and six additional files specifying the problem and restrictions. For instance, the FreightRates table describes all available couriers, the weight gaps for each lane and rates associated. The shipping lane refers to courier-transportation mode-service level combination between two shipping ports. The PlantPorts table describes the allowed links between the warehouses and shipping ports in the real world. Furthermore, the ProductsPerPlant table lists all supported warehouse-product combinations. The VmiCustomers contains all edge cases, where the warehouse is only allowed to support specific customer, while any other non-listed warehouse can supply any customer. Moreover, the WhCapacities lists warehouse capacities measured in the number of orders per day and the WhCosts specifies the

cost associated in storing the products in a given warehouse measured in dollars per unit.

The optimisation's main goal is to find a set of warehouses, shipping lanes, and couriers to use for the most cost-effective supply chain. Therefore the fitness function is derived from two incurred costs – warehouse cost WC_{ki} and transportation cost TC_{kpj} in equation (4). The totalling cost is then calculated across all orders o in the dataset.

$$min \sum_{k=1}^{TO} (WC_{ki} + TC_{\mathrm{kpj}}) \quad (4)$$

Where WC_{ki} is warehouse cost for order k at warehouse i and TC_{kpj} is transportation cost for order k between warehouse port p and customer port j; the total number of orders TO.

$$WC_{ki} = g_k \times P_i \quad (5)$$

Where warehouse cost WC_{ki} for order k at warehouse i is calculated in (5), by the number of units in order g_k multiplied by the warehouse storage rate P_i (WhCosts table).

Furthermore, transportation cost TC_{kpj} for a given order k and chosen line between origin port p and destination port j is calculated by the algorithm in Figure 9:

Transportation cost (TC_{kpj})

1. **if** $s_k = CRF$
2. $\quad TC_{kpj} = 0$
3. **else**
4. $\quad$ **if** $m = GROUND$
5. $\quad\quad TC_{kpj} = \frac{FreightRate_{pjcstm}}{\sum_{k=1}^{TO} weight_{kpjcstm}} \times weight_{kpjcstm}$
6. $\quad$ **else**
7. $\quad\quad TC_{kpj} = FreightRate_{pjcstm} \times weight_{kpjcstm}$
8. $\quad\quad$ **if** $TC_{kpj} < MinCharge_{pjcstm}$
9. $\quad\quad\quad TC_{kpj} = MinCharge_{pjcstm}$
10. $\quad\quad$ **end if**
11. $\quad$ **end if**
12. **end if**

Figure 9. *Pseudocode for calculating order transportation cost*

where s_k is the service level for order k, p – origin port, j – destination port, c – courier, s – service level, t – delivery time, m – transportation mode. Furthermore, $MinCharge_{pjcstm}$ is the minimum charge for given line $pjcstm$, $weight_{kpjcstm}$ is the weight in kilograms for order $k,$ TO – total number of orders; $FreightRate_{pjcstm}$ is the freight rate (dollars per kilogram) for given weight gap based on the total weight for the line $pjcstm$ (FreightRates table).

The transportation cost logic in Figure 9 first checks what kind of service level the order requires; if the service level s_k is equal to CRF (Customer Referred Freight) – transportation cost is 0. Furthermore, if order transportation mode m is equal to GROUND (order transported via truck), order transportation cost is proportional to the weight consumed by the order ($weight_{kpjcstm}$) in respect of the total weight for given line $pjcstm$ and the rate charged by a courier for full track $FreightRate_{pjcstm}$. In all other cases, the transportation cost is calculated based on order weight $weight_{kpjcstm}$ and the freight rate $FreightRate_{pjcstm}$. The freight rate is determined based on total weight on any given line $pjcstm$ and the corresponding weight band in the freight rate table. Furthermore, a minimum charge $MinCharge_{pjcstm}$ is applied in cases where the air transportation cost is less than the minimum charge.

The problem being solved complies with the following constraints:

$$\sum_{k=1}^{TO} o_{ki} \leq OrderLimit_i \tag{6}$$

where o_{ki} = 1 if order k was shipped from warehouse i and 0 otherwise. $OrderLimit_i$ is the order limit per day for warehouse i (WhCapacities table).

$$\sum_{k=1}^{TO} w_{kpjcstm} \leq \max\{Z_{pjcstm}\} \tag{7}$$

where w_{kpjcst} is the weight in kilograms for order k shipped from warehouse port p to customer port j via courier c using service level s, delivery time t and transportation mode m. Z_{pjcstm} is the upper weight gap limit for line $pjcstm$ (FreightRates table).

$$k_z \in i_z \tag{8}$$

where product z for order k belongs to supported products at warehouse i (ProductsPerPlant table). Warehouses can only support given customer in the VmiCustomers table, while all other warehouses that are not in the table can supply

any customer. Moreover, the warehouse can only ship orders via supported origin port, defined in PlantPorts table.

The outbound supply chain problem discussed above represents a real-world model, where the products need to be routed from various warehouses to the customers via different modes of transport. However, the problem only considers the flow and distribution of goods and omits the logistics of scheduling and managing the courier vehicle fleet. The Transcom scheduling and routing problem (discussed in the next section) models even more complex supply chains. Not only are goods delivered to their destinations, but vehicle availability, scheduling, and refiling are also considered.

2.3.2.3. Transcom scheduling and routing problem

These days we rely on complicated global supply chains for everyday shopping from the pasta imported from Italy and distributed across UK grocery stores; to the car we drive, whose components were sourced across multiple countries and continents. This section presents a cross-continent supply chain in the US air force called Transcom. The supply chain is modelled based on the distribution of quotidian goods – food, medicine, and other consumables – across multiple base stations located around the world.

Transcom problem considers a complex logistics network that includes multiple base stations that can both request and supply number of goods, usually on pallets. The demand can be satisfied either directly by the organisation or by outsourcing it to a third party – commercial partners. The cargo can either be supplied by a different kind of aircraft or by ground via trucks – each with different speed and carrying capacities. Furthermore, both aircraft and trucks require personal to be scheduled and supporting personnel for loading and unloading cargo. This creates a multi-dimensional optimisation problem, where both the best routes between the edges need to be found, as well as the best route sequence for delivering the cargo. A simplified example is given in **Figure 10**. A more realistic model with a numeric examples are provided in the Appendix.

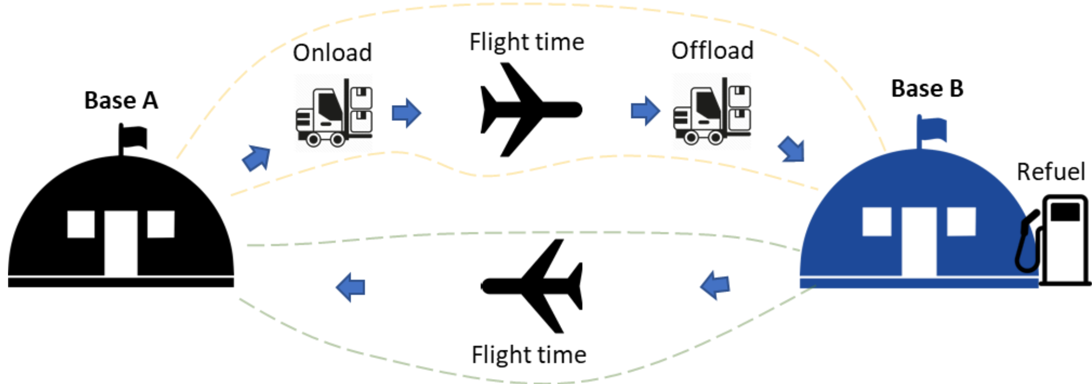

Figure 10. *Simplified Transcom supply chain example.*

There are two main objectives to optimize for in Transcom scheduling and routing problem:

- **Least time solution**

Total time required to fly a path i between two edges with given aircraft type a is calculated based on path distance and aircraft cruise speed. Furthermore, in cases where aircraft a has in-air refuel capability, additional time is added based on the number of in-air refuels executed (NR).

$$FlightTime_{ia} = \frac{distance_i}{cruiseSpeed_a} + NR * inAirRefuelTime_a \quad (9)$$

In addition to a flight time, the cargo pallets of material need to be loaded and offloaded of the plane or truck. This model assumes that each pallet takes 10 minutes to be loaded and 10 minutes to be unloaded. Therefore, an aircraft with a capacity of 36 pallets takes 6 hours to be fully loaded and an additional 6 hours to be fully offloaded. Furthermore, for aeroplanes that do not support in-air refuel, refuel time is done on the ground and it takes a duration specified in the *Aircraft data* table. It is assumed that all aeroplanes are fully fuelled at Time zero, partial refuels (fuelling up half the tank) are not allowed in this model. Therefore, the total time for a solution is the timespan required to satisfy all demand and land military aircraft back to military bases.

- **Lowest cost solution**

Similarly, the lowest cost objective tries to minimize the total cost occurred while satisfying the demand. Transcom problem consists of three transportation types of transportation for supplying the goods, each with its cost calculation:

- Military Aircraft (MA)

Total cost for given aircraft type a is calculated as a sum of all paths flown (PF) and the total number of aircraft (TA) of such type used on path i.

$$Cost(MA)_a = \sum_{i=1}^{PF} (FlightTime_{ia} \times costPerHour_a \times TA) \tag{10}$$

- Commercial Aircraft (CA)

Total cost for given aircraft type a is calculated as a sum of all paths flown (PN) and the total number of pallets shipped through the path i.

$$Cost(CA)_a = \sum_{i=1}^{PN} (costPerPallet_i \times nPallets_i) \tag{11}$$

- Commercial truck (CT)

Total cost for commercial truck is calculated as a sum of all paths driven (PN) and the total number of trucks used on the path i.

$$Cost(CT) = \sum_{i=1}^{PN} (costPerTruck_i \times nTrucks_i) \tag{12}$$

Total cost for the solution is a sum of total military aircraft cost, total commercial aircraft cost and total commercial truck cost. Expressed in the equation below, where N_{MA} is the total number of types of military aircraft, N_{CA} is the total number of types of commercial aircraft.

$$Total\ cost = \sum_{a=1}^{N_{MA}} Cost(MA)_a + \sum_{a=1}^{N_{CA}} Cost(CA)_a + Cost(CT) \tag{13}$$

The two above objectives are subject to the following constraints:

1. The total number of cargo pallets shipped $nPallets$ on aircraft a cannot exceed the maximum carry capacity of the aircraft $maxPallets$.

$$nPallets_a \leq maxPallets_a \tag{14}$$

2. For any in-flight refuelling incapable aircraft a, aircraft can only fly to paths i that are in $range$ on one full fuel tank.

$$distance_i \leq range_a \tag{15}$$

3. Furthermore, no ground refuels are allowed at Humanitarian destinations. And all military aeroplanes need to terminate their route at one of the military bases.
4. All commercial trucks t also need to comply with maximum carry capacity $maxPallets$ of the truck.

$$nPallets_t \leq maxPallets_t \tag{16}$$

5. Moreover, one can only transfer as many pallets as available at base/commercial partner location b at any given time.

$$palletsShipped_b \leq pallets_b \tag{17}$$

6. Demand at destination $dest$ can be supplied from multiple sources/routes; however, the total quantity of pallets shipped to the destination must be equal to $demand$ once the search is terminated.

$$pallets_{dest} = demand_{dest} \tag{18}$$

7. Total number of aircraft $a(d)$ departing from base b must be less or equal to the total number of available aircraft of such type in base/commercial partner location b.

$$a(d)_b \leq \sum a_b \tag{19}$$

In this model, there is no such constraint on the commercial trucks leaving any of the bases. Furthermore, not all aircraft are supported on all bases; the dependencies are defined in Aircraft Compatibility table. Also, deadhead[17] links between commercial partner locations (CPs) and military bases are allowed only in the direction from an army base to CP using commercial aircraft. Aircraft flying to the humanitarian destination must always carry onboard less or equal quantity of pallets than the demand at said destination.

The Transcom problem must both, route the cargo from source to destination and schedule the aircrafts such that they are positioned in the right locations before the cargo is delivered to the base. Furthermore, as the base can be both the supplier and the demand, some of the routes are re-used recursively, further increasing the model's

[17] Deadhead - One leg of a move without a paying cargo load.

complexity. The Transcom problem is the most complex model implemented and optimized in this thesis.

2.4. Summary

This chapter presented a general description of the most common metaheuristics and metaheuristic optimization frameworks (MOFs). In particular, this chapter focussed on Ant Colony Optimisation, Evolutionary Strategy and Imperialist Competitive Algorithm. Next, two benchmark and three real-world problems were introduced and formalised. Finally, sixteen MOFs were analysed and limitations of each compared. The following section will briefly explain how this thesis addresses the limitations of current MOFs.

Under a more fundamental level, most MOFs are limited to evolutionary computing type of algorithms and their encodings, where a solution is built on top of an existing solution; however, only few MOFs support algorithms building solutions from scratch in the ACO algorithm. Thus, a more generic optimization platform called OptPlatform is presented in Chapter 3. Furthermore, a new, improved algorithm based on ICA is developed in Chapter 4 within the platform.

With the majority of computers supporting multi-core processing, parallelism is another vital aspect of MOF development. As discussed, current MOFs are mainly developed for academic research and for solving benchmark problems. The parallelism dynamics of these benchmark problems does not necessarily apply to more complex real-life problems. Thus, Chapter 5 is an in-depth investigation of ACO scaling across different hardware types using the developed OptPlatform.

Another limitation discussed is the lack of supporting tools needed for these frameworks to be effectively used outside academia. In particular, tools such as automatic algorithm and hyperparameter selection are essential for adaption and the ease of use. Chapter 6 proposes and analyses algorithms to solve this problem. Furthermore, the solution visualization and recommender systems are implemented as part of OptPlatform in Chapter 3, which helps users to implement the theoretical solution into a real-life solution in the most optimum way.

3. OPTIMIZATION PLATFORM (OPTPLATFORM)

The current chapter describes the design and implementation of a metaheuristic optimization framework, called OptPlatform, which overcomes most of the current MOFs' limitations, analysed in the previous chapter. First, the motivation and requirements are formulated in section 3.1. Next, the OptPlatform's architecture and technology stack is explained in sections 3.2-3.4. Furthermore, section 3.5 lays out step by step process for implementing optimization problems in OptPlatform. Metaheuristic algorithm implementations and parallelism is described in section 3.6. A brief overview of supported visualization tools is provided in section 3.7, while an in-depth explanation of solution transition optimisation in section 3.8. Finally, multiple existing optimization platforms are compared against OptPlatform in section 3.9 and chapter summarized in section 3.10.

3.1. Target users and requirements

Creating a generic software that accommodates all possible users is difficult if not impossible task. It is especially true when complex systems such as metaheuristics are involved, where hundreds of optimization methods can be applied to infinite variations of application domains. Furthermore, the users also range in their skillset and demands – some users have little to no knowledge of the heuristic optimization, while others might have years of experience. Therefore, it is essential to understand the target user requirements.

Both [131] and [151] classified possible metaheuristic optimization software users into three overlapping categories: practitioners that are using the software for real-world applications; researchers - heuristic optimization experts analyse, hybridize and develop new algorithms and finally, students that are just starting out and still learning about heuristics and optimization. This work focuses on the arguably largest and most impactful group – practitioners, though most of the features and requirements also apply to researchers, less so to students' teaching.

Practitioners are people with a challenging optimization problem to solve (usually NP-hard), that have a problem domain knowledge but are not necessarily experts in optimization methods. Almost all domains such as engineering, medicine, economics, logistics and computer science have such challenging optimization problems, that would not be feasible to solve by hand, without automated strategies. This presents an infinite amount of problems to solve; therefore, the optimization software must be generic enough to accommodate all of them. Furthermore, most practitioners work in a domain unrelated to heuristic optimization or even software engineering in general. However, they have a deep understanding of the problem itself, its domain, restrictions and objectives and therefore, the optimization tools are purely black-box solvers to obtain the solution.

Moreover, in business, time is money, and a quick near-optimal solution is often valued versus an optimal solution that takes ten times as long to compute. Thus, every second spent in a sub-optimal state in an ever-changing environment is an unnecessary cost that can be avoided. Fast turnaround to solution also allows more sophisticated modelling of what-if scenarios essential in business planning. As the world gets more and more connected, the responsiveness to the ever-changing geopolitical environment is an edge, examples of such events include Brexit and Covid-19.

Efficient use of computing resources is another aspect valued by practitioners, as company computing resources are usually shared and in high demand. Computing resources that are not utilized are a lousy return on investment. Furthermore, it is expected that more computing power should either improve the results, solve larger problems, or consider more what-if scenarios.

Another essential factor of any software is the ease of use. Rarely if ever black-box optimization is used as a standalone tool, more commonly it needs to be integrated into existing systems and IT infrastructure. Consequently, the optimization software needs to be modular and portable, with clearly defined inputs and outputs. Similarly, the practitioners should only be focusing on the problem and not require an understanding of the internal algorithms or their parameters. Moreover, examples are usually an excellent starting point for any software system and therefore, a variety of easy to understand optimization problem examples are essential.

Therefore, optimization platform requirements for the practitioners can be summarized by the following (in alphabetic order, based on [131] and [151]):

- ***Applicability*** – the output of the software should be easy to understand and applicable to the real world. The platform should produce detailed suggestions on how a user can implement the new solution with the least distruptions to the existing real-world solution.
- ***Genericity*** – platform needs to be able to support a variety of optimization problems, their constraints and application domains. It should not be limited to any specific metaheuristic algorithm or solution representation.
- ***Interoperability*** – the software should be modular and easily integrable into existing systems and IT infrastructure. A generic communication protocol is required for supplying the software with new data and getting the resulting solution.
- ***Multi-algorithm support*** – it should be possible to use already implemented metaheuristic algorithms seamlessly and switch between them, while requiring no prior user knowledge.
- ***Learning effort*** – users should start using the platform for their optimization problems quickly with little programming or software development knowledge. The interface and user workflow, therefore, should be intuitive and easy to understand. The problem domain should be clearly decoupled and abstracted away from the underlying algorithms.
- ***Parallelism*** – the platform should be scalable and efficient at utilizing computing resources. Additionally, parallelism and scaling should be effortless, without the user's need to understand how the underlying parallelism is implemented. The user should control the computing resource utilization through parallelism level (number of workers/threads).
- ***Parameter Management*** – metaheuristic algorithms are subject to multiple parameters that influence their performance for a given problem. Furthermore, as metaheuristics are probabilistic and can produce different results for the same data inputs, tuning and evaluation tools are necessary. The parameter selection should be either automatic or guided by the user.
- ***Performance*** – most real-world problems are computationally intensive, and in time-critical applications, the turnaround to a solution is essential. Thus, the optimization platform should offer computationally efficient implementations of the underlying algorithms.

- ***Problem examples –*** examples of optimization problems are essential to guide and familiarize the user with the platform. They can also be used as building blocks for custom optimization problems.

3.2. Technologies used

When designing any software system, programming languages, tools, and target platforms need to be considered. Each programming language has its advantages and disadvantages, some claim to improve on existing languages, but lack the developer mind share. There are two main strategies used for existing metaheuristic frameworks – C++ for performance-oriented MOFs ([137], [133], [140], [135]) and Java for user-focused, interface-driven MOFs ([144], [141], [146], [138], [139], [142], [143]).

Use of low-level C/C++ has been associated with high-performance computing, as the computing resources can be accessed at a much lower level than any of the high-level interpreter languages. However, things such as interfacing, and GUIs are not trivial in C++ and usually a high-level language such as Java, C# or Python is preferred. Java is a modern programming language, as it is cross-platform and easy to learn. Furthermore, open-source nature and rich-set of APIs attracts a lot of researchers and practitioners to Java. Although modern Java implementations and compilation offers highly efficient code, low-level C/C++ code is preferred for high-performance applications. Thus, practitioners selecting existing MOFs must compromise between performance or ease of use/integration.

This work tries to bridge the gap between high-performance metaheuristics and their accessibility, the ease of use. For low-level search algorithms - C arrays and pointers are used for memory management, while C++ is abstracted for problem definition. Search algorithms are designed to be parallel (via OpenMP[18]) ground up and not as an after-the-fact. Moreover, this high-performance part of the platform is compiled as a dynamic link library (DLL) to be accessed by any high-level interface. Although in theory, the majority of programming languages can invoke and use the compiled DLL, OptPlatform uses C# for its high-level interfaces.

Compared to Java, C# is mainly focused on .NET framework or more recently, .NET Core and is targeted to Windows, though cross-platform adaptations such as Mono[19]

---

[18] OpenMP parallel API website. https://www.openmp.org/
[19] Mono project. https://www.mono-project.com/

exists. This, however, is not an issue for the user-focused platform as more than 76%[20] of desktop computer users use Windows as their operating system. Just like Java, C# has a rich set of existing libraries, APIs and tools. Additionally, both C++ and C# can be compiled, debugged and run under the same toolchain in Visual Studio IDE, making development more straightforward.

3.3. Fundamental concepts

The following section introduces the concepts of optimization problems and intends to explain the building blocks of the OptPlatform implementation.

- ***Problem*** – user defined inputs and supporting logic that clearly defines parameters and constraints for problem to be solved. The implementation is structured based on the concepts of O*rders* and E*lements*. An Order has a demand that needs to be satisfied with one or multiple *Elements.*
- ***Solution*** – a solution to a problem is defined as a vector of soliton pairs (*SolutionPair*). Solution pair is derived from both the order index and the element index. Additionally, the pair can also contain a quantity of the satisfied demand by choosing the *SolutionPair*. In most cases, the solution needs to be decoded back into problem-specific data before further processing.
- ***Search space*** – solution space, candidate set or feasible region - is a set of possible element and order indices (*PossibleElement*) that satisfies given problem constraints. Only valid solutions that meet all problem constraints are evaluated for performance score. Additionally, *PossibleElement* can also contain heuristic information about the element.
- ***Algorithm*** – search algorithm, search core - a methodological approach or procedure that solves a challenging problem. It is usually resource-intensive and has its own set of parameters and memory, independent of the problem.
- ***Seed –*** a seed is usually an integer value used as a starting point for Pseudo-Random Number Generator (PRNG). As OptPlatform focuses on probabilistic metaheuristic algorithms, some form of randomness is needed. PRNGs are good for this purpose as it offers both pre-defined randomness and reproducible results.

[20] Desktop Operating System Market Share Worldwide Desktop Operating System Market Share Worldwide - April 2020. Accessible https://gs.statcounter.com/os-market-share/desktop/worldwide

- **Config –** is a set of parameters that defines a configuration of the search algorithm and the problem. Search parameters such as termination criteria, logging level and computation resource utilization are common across all search algorithms.
- **Fitness –** a solution score (such as cost, time or profit), that is assigned to a full valid solution. Fitness scores are compared to obtain the best out of two or more solutions.

3.4. Architecture

Like many other existing MOFs, in OptPlatform problem-specific logic is separated from problem independent logic – such as search algorithms and supporting tools. Majority of existing MOFs focuses on the ease of new algorithm development and hybridization, aimed at researchers with expert knowledge. OptPlatform main aim is to target the practitioners with little to no understanding of metaheuristics and allow a more black-box approach for solving their industry problems. Although prior knowledge of underlying algorithms is beneficial, it is not necessary.

The high-level architecture of OptPlatform is demonstrated in **Figure 11**. It contains a User domain and a Platform, that is abstracted away from the user. Moreover, architecture is structured as a form of building blocks – modules. Modules are implemented in either C++ or C#. As the OptPlatform uses both C++ and C#, the data sharing and transfer between modules can be both via P-invoke of DLL or via the flat file system. C++ and C# communication is also abstracted away from the user in the *Search Wrapper* module.

User starts by specifying the problem-specific data structures in *Problem Manager* and implementing problem-specific functions (such as restrictions and fitness evolution) in *Opt Problem* module (section 3.5). Problem specific logic is then compiled with the search cores (ACO, ES or ICA) into a DLL (section 3.6). From *Problem Manager*, user can analyse the search process such as iteration performance and/or algorithmic specific data in *Search Visualizer* module (section 3.7). Similarly, the user can choose to auto-select and tune the search algorithms config for the implemented problem (Chapter 5). For real-world problems, where the transition between current existing state and the newly optimized state is unclear, Transition Opt module can help generate a step-by-step report (section 3.8). Finally, for optimization problems represented as geographical locations and/or links, the *Global Grid* module can both

generate paths between any two points in the map and create animated visualization (section 3.7).

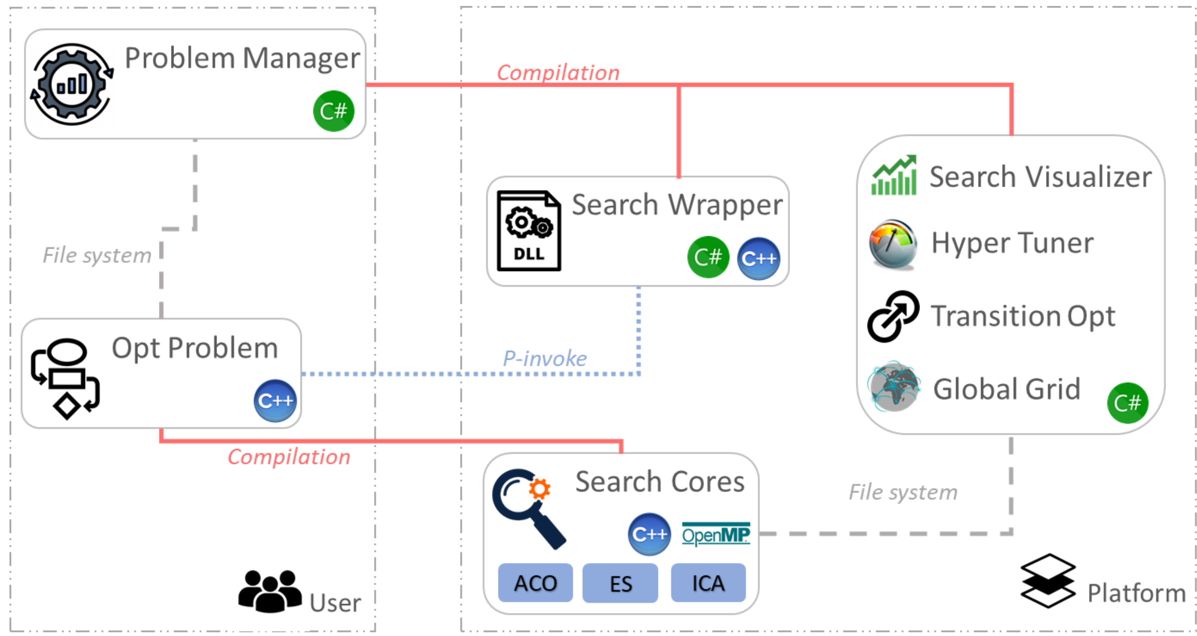

__Figure 11.__ A high-level overview of modules in OptPlatform. Optimization platform uses two languages – C++ for low-level high-performance search and C# for user interfacing and other accessory tools. Split into user domain, where only problem details are specified and the abstracted backend - platform.

3.5. User workflow

This section covers the user workflow for implementing a new optimization problem. The high-level overview is shown in **Figure 12**. The icons next to each of the steps represent the OptPlatform module used in the corresponding action, based on **Figure 11**. User is only required to interact with two modules – Problem Manager and Opt Problem; all other modules are optional and/or abstracted away from the user.

__Figure 12.__ User workflow for implementing an optimization problem. Icons represent the modules used (in __Figure 11__) during the process, some of them can be optional.

- **Step 1: Solution encoding**

The first step of implementation is to structure the problem such that it can be used within the platform – the search space definition. User needs to list all possible solution elements of size E_{max} for any given order o in the list of orders of size O_{max}. This is then encoded as a two-dimensional matrix, with each of the cells corresponding to the possible element-order combination that can be added to the final solution. The user needs to map the problems search space to the two-dimensional order-solution matrix. There need to be at least two elements for each order and at least one order in total. Furthermore, each of the orders has an integer value of demand, that needs to be satisfied during the solution creation. **Figure 13** shows a search space representation with how the solution is mapped from the 2D encoded matrix. For each corresponding order, some elements are selected to create a SolutionPair, in the example of **Figure 13**, order o_0 get assigned two elements - e_2 and e_6, thus generating two solution pairs – (o_0, e_2) and (o_0, e_6). Similarly, order o_1 gets assigned element e_5, generating a *SolutionPair* (o_1, e_5). The combination of all solution pairs creates the final encoded solution - (o_0, e_2); (o_0, e_6); (o_1, e_5).

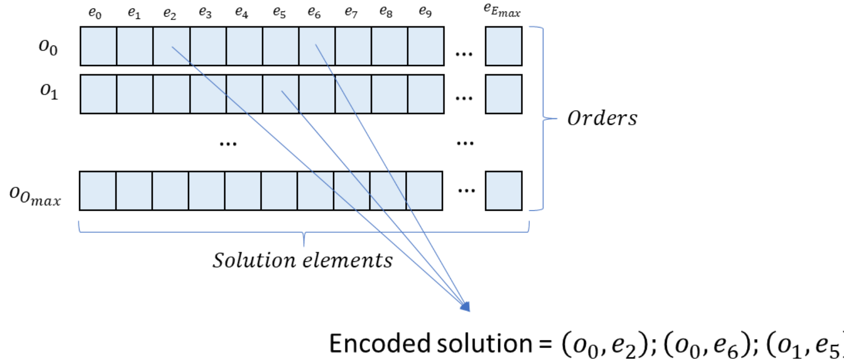

Figure 13. *Search space representation and solution element encoding. Constructed as a 2D matrix with sizes E_{max} and O_{max}. Search algorithm selects one or multiple cells to be added to the final solution.*

To illustrate the flexible mapping process between encoded solution and real-life model, two simple example problems are considered in both **Figure 14** and **Figure 16**. In **Figure 14**, a simple bin packing problem is considered, where the goal is to fit all the items in the bins without exceeding their capacity. The encoded solution is presented as an array of order-element pairs. Suppose one considers each of the bins as order and each of the items as elements. In that case, the solution can be easily

decoded by grouping all elements per order and mapping them to their corresponding bins.

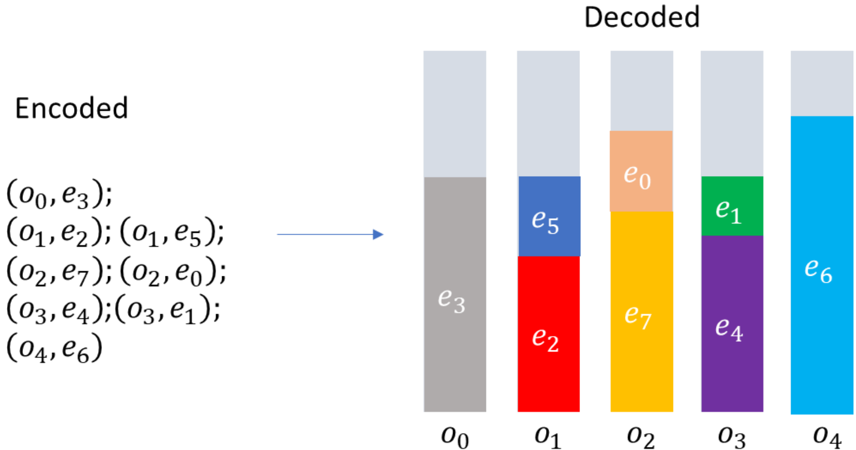

Figure 14. *Simple bin packing problem encoding and decoding with five identical bins (represented as orders) and eight items (elements).*

- **Step 2: Data type definition and assignment**

The next step in the process is to define any problem-specific data required for the problem. The problem-specific data is represented as a list of *ProblemAttribute,* where each ProblemAttribute represents the data with the corresponding name, data type and size. Problem-specific data, such as item weights and profits in Knapsack problem, travel distances between two cities in TSP, can either be read-only or dynamic. As the name suggests, read-only data are static data expected to not change during the search process. In contrast, dynamic data can be read and written during the search process. In Knapsack, for example, item profits are static and can be thought as read-only; however, the total weight usage in knapsack is changing as items get added and therefore – dynamic. At the start of each iteration, all dynamic data is reset to the initially defined value. The list of *ProblemAttribute* is then used and compiled as part of the search and problem logic definition.

- **Step 3: Generation of Opt Problem template**

Based on Step 2, all pre-defined problem-specific data types are used to compile a C++ template project for implementing the problem logic (the Opt Problem module). The problem-specific data access is abstracted and simplified using C++ definitions. The automatically created project has two files – *OptProblem.h* and *OptProblem.cpp* with all the necessary headers and pre-defined function implementations, and

examples of the problem-specific data access. Furthermore, the VS++ project is pre-configured with all the other modules in OptPlatform automatically.

- **Step 4: Problem logic implementation**

Based on the automatically generated project in the previous step, a user needs to implement at least three of the pre-defined logic methods based on the problem domain:

- ***canElementBeAdded(Element, data)*** – a required method that returns either true or false for the provided method. If the method returns true, the element will be added as part of the solution and not, if false. The user is expected to use only static data.
- ***addElementToSolution(Element, data)*** – *a* required method that returns the quantity of demand satisfied by adding this element to the solution. Furthermore, before the element gets added to the solution, the user can update any of the constraints and problem-specific data. The user is expected to write to dynamic data, if applicable.
- ***getSolutionPerformance(Solution, data) –*** a required method that evaluates the provided solution quality and returns a performance value. The solution is provided as a list of *SolutionPair*, built from the previously added elements. Therefore, the solution is expected to be within constraints, does not require additional checks, nor a penalty cost.
- ***isBetterPerformance(double, double)*** – an optional method that returns true if the first provided double is a better performance value that the second double. By default, all problems in OptPlatform are minimization problems, and therefore, it returns true if the first double is lower than the second double.
- ***userSyncAfterIteration(data)*** – an optional method that returns true if the search needs to be terminated and false otherwise (for problem-specific termination criteria). In this method, a lot of problem-specific data is exposed to the user after each iteration and allows for further customization. Customization such as problem-specific local search, statistical analysis between iterations and other, are possible.

All implemented search algorithms as part of *Search Cores* module follow the iterative process of solution construction and evaluation. The interface between the user and the platform can be seen in **Figure 15**.

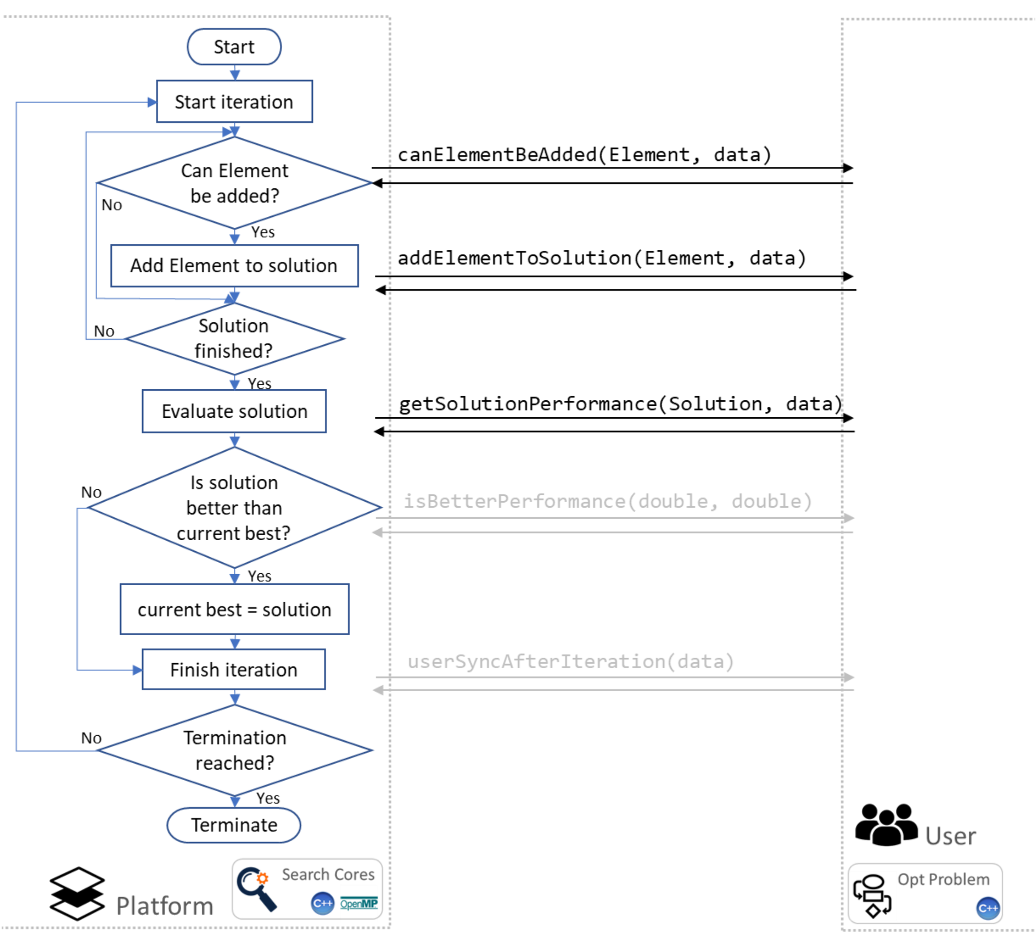

Figure 15. *The interface between the search algorithms in the Search Core module and user-defined problem in Opt Problem. Flowchart on the left is a generic model that all search algorithms in the Search Cores follow. Methods isBetterPerformance and userSyncAfterIteration are optional and therefore greyed out.*

- **Step 5: Search algorithm selection**

At this step, all problem-specific definitions and logic are already implemented, and the user just needs to pick one of the algorithms available in Search Cores module, such ACO, ES or ICA and define the search configuration. Some of the configurations are shared across all search algorithms, such as the termination criteria, the seed, degree of parallelism, number of parallel instances in the search and logging information. Furthermore, some problem-specific configurations, like the maximum number of solution pairs in the solution and whether incomplete solutions should be accepted for evaluation, can also be defined. Alternatively, user can run algorithmic parameter tuning in Hyper Tuner module to obtain the best configuration automatically.

- **Step 6: Run the search and analyse the solution performance**

Once the optimization algorithm is selected and configured, a search process can be started, and the performance evaluated. To make this process simpler and more user-friendly, a GUI interface allows users to start, pause and stop the search and adjust the level of parallelism dynamically during the search process. Furthermore, analysis tools are implemented in the GUI that allows to perform simple statistical analysis and graphically plot the convergence of the search across multiple experiments.

- **Step 7: Decode and implement the solution**

Once the search process is finished and the final solution exported, the user needs to decode the encoded solution back to a real-life representation. A simple Travelling Salesman Problem with five cities (A-E) is considered in **Figure 16**. There are two ways that TSP can be represented as part of the solution, either as a sequence of cities visited (Sequence encoding) or as a graph where the 2D order-element matrix, where the nodes (cities) are represented as orders and their interconnections (links) as elements (Graph encoding). In sequence encoding, it is assumed that there is just a single order; the sequence that elements are added to the solution to determine the chronology of the visited cities. In contrast, in the graph encoding, each city is an order, and the corresponding element is the next city to be visited. Thus, SolutionPair with order index 0 and element index 2, moves from city A to city C.

For problems that represent geographical locations, the Global Grid module can be used to animate the links in the map across the globe graphically. Furthermore, Transition Opt module is designed for models representing a real-life system with long-term contracts, facilities, and employees and cannot migrate to the new optimized solution overnight. The Transition Opt generates a step by step suggestions on transitioning from any given existing model to the newly optimized model with the least distruptions.

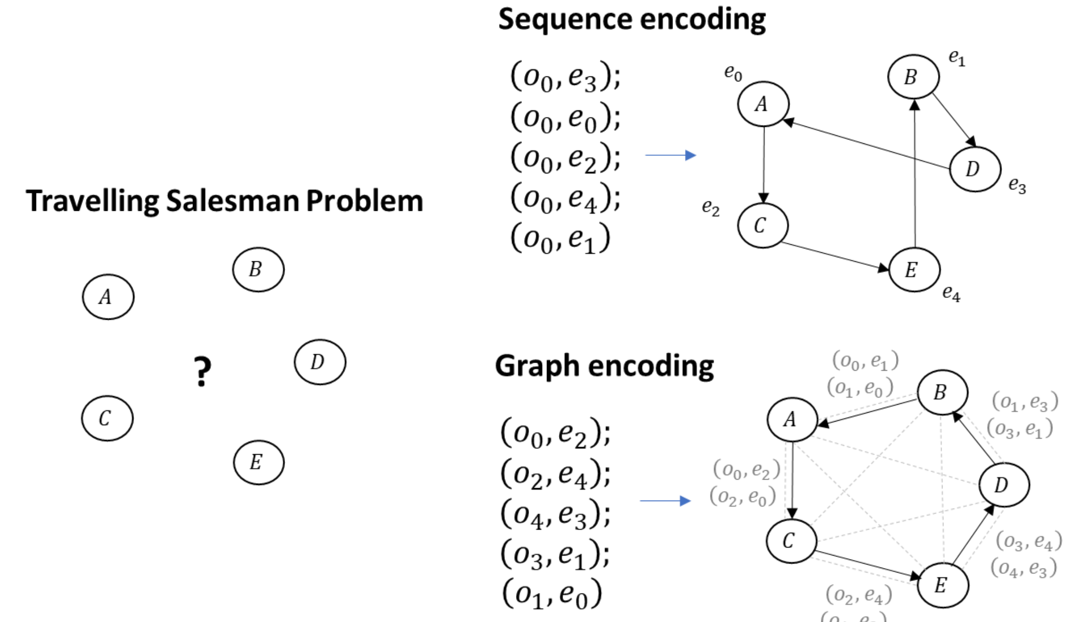

Figure 16. *Two example encodings for Travelling Salesman Problem (TSP). In Sequence encoding, only the selected element sequence in the solution is needed for encoding. Graph encoding represents nodes as a 2D graph, where the nodes themselves are represented as orders and the inter-connections as elements. Therefore, cells* o_0e_0 *and* o_1e_1 *would be invalid in TSP as it is a connection to itself.*

3.6. Search cores module

As discussed in the requirement analysis in section 3.1, the underlying algorithms in any MOF have to be very generic to accommodate a wide range of problems. Users must be able to implement their problem-specific logic without the need of previous knowledge of metaheuristics. Moreover, with ever-increasing computation power, many of the problems that were infeasible to be solved just a decade ago are now in the reach of practitioners. Most of the advances in computing have been derived from multi-core processor architectures, and thus any MOF must utilise these resources effectively. In fact, optimization software systems should be designed with parallelism and concurrent computing in mind. Compared to other optimization frameworks that offer parallelism as a plugin and an afterthought, OptPlatform is designed for performance ground up. All metaheuristic algorithms in OptPlatform follows the parallel master-slave model, where the master process manages the global information across iterations, while each of the slave processes builds and evaluates solution, as demonstrated in **Figure 17**.

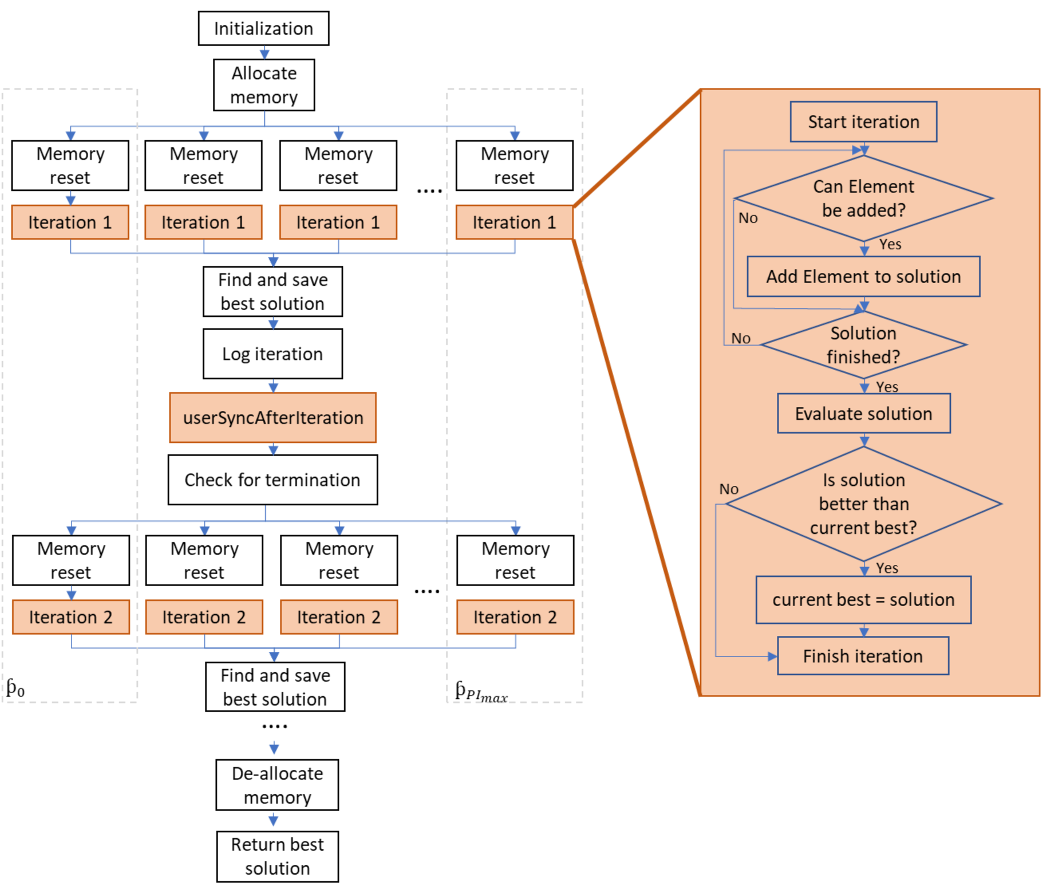

Figure 17. *Memory allocation and parallelism in Search Cores architecture. Areas of the process, where problem-specific methods are called, are in orange. Iterations are executed in sequence, however, in each iteration, multiple solutions are constructed and evaluated up to the maximum number of parallel instances -* PI_{max}.

In **Figure 17**, each of the OptPlatform search algorithms starts by allocating memory for PI_{max} Parallel Instances. Then, the master process starts every iteration by launching several slave processes (OpenMP threads, pre-defined in search config). Each of the slave process ƥ resets memory to their pre-set default values, builds and evaluates solution, as shown in **Figure 15**. After each iteration, the master process saves the best solution across all slaves and performs logging, as well as performs checks for search termination. The process is repeated iteratively till termination criteria is reached, at which point all the allocated memory is freed. This architecture minimizes memory allocation, as no new memory is allocated during the search, only once - at the start. Furthermore, to reduce the thread overhead, all OpenMP threads are re-used across iterations.

Furthermore, MOF needs to support a wide variety of metaheuristics that can be applied to the same user problem without adaptations or customizations. Although there are dozens of different kinds of metaheuristics, as shown in section 2.1.2, three

were selected to be implemented in OptPlatform. Ant Colony Optimization algorithm (in section 2.1.2.4) was chosen due to a long history of efficiency for solving routing and scheduling problems. Furthermore, the Evolutionary Strategy algorithm (in 2.1.2.5) was selected as it is the simplest algorithm from Evolutionary algorithm family. Finally, a more recent metaheuristic called Imperialist Competitive Algorithm was chosen for its promising performance for broad application areas, as examined in section 2.1.2.6). All three algorithm definitions and formulations are presented in more detail in the following subsections.

3.6.1. Ant Colony Optimization (ACO)

Ant Colony System was initially implemented for solving graph routing problem, where pheromone is deposited on the links between two nodes [106]. In OptPlatform, ACS nodes can be considered as Orders and the routes between the nodes – Elements. Graph encoding in **Figure 16** demonstrates the relationship. Furthermore, ACS differs from the other two metaheuristic algorithms because it relies on heuristic information for efficient search. Heuristic information is problem-specific data associated with each Element. For example, in TSP case, the distance between two nodes can be considered heuristic, allowing ACS to prioritize shorter routes between two nodes. OptPlatform's ACO implementation (**Figure 18**) differs from standard ACS [104]. First, it is designed to be parallel. Therefore, in each iteration, multiple ants are constructing and evaluating the solution in parallel. Secondly, ACO within OptPlatform introduces the concept of *heuristic priority*. The purpose of *heuristic priority* is to prioritize orders that have the highest impact if they were to be solved first. The process has been implemented as follows and is calculated at runtime:

1) All elements associated with an order are sorted based on numeric heuristic information, ascending.
2) The difference between the best and second-best heuristic across all order's elements is evaluated for each order.
3) Orders with the highest heuristic gap (largest difference) are given priority over other orders.

And finally, OptPlatform's ACO also implements the idea of cunning ants, based on [187]. In cunning ant ACO, each ant generates a solution by borrowing part of a solution from the best solution in the previous iteration, instead of building a solution

based on the pheromone. This approach has proven to increase the efficiency and convergence speed of the search.

Furthermore, the solution creation, evaluation and pheromone update is implemented based on the standard ACS formulated in [106]:

- The state transition rule is used to drive the search of the ants
- The global pheromone update rule is used to focus the search on the solution space's most promising areas.
- The local pheromone update rule is used to force ants to explore a more extensive solution space area.

In short, in every iteration, a colony creates a number of sub-colonies for each of the Parallel Instances (PI_{max}), where each of the sub-colonies releases several Local Ants (LA_{max}). Each ant ɐ builds a complete solution, if feasible. The ants are guided in the search by both the pheromone and heuristic information at each Element cell. The use of pheromone helps ants to choose lucrative routes (Elements). With the use of state transition rule, the ant can either exploit the best-known route or explore a new route by random.

Furthermore, the local pheromone update ensures that ants do not keep visiting the same routes repeatedly. Once all ants within the sub-colony have finished creating the search, each sub-colony's best solution is compared against all other sub-colonies. The best solution in the iteration is then used to update the global pheromone. The process continues till the termination condition is met.

From the probability distribution given in equation (20) [106], the state transition rule $stateTransit$ is:

$$stateTransit = \begin{cases} \underset{u \in J_k(e)}{\arg\max} \{[\tau(e, \text{o})]^{\alpha} \cdot [\eta(e, \text{o})]^{\beta}\} & if\ q \le q_0 \\ S & biased\ exploration \end{cases}, \quad (20)$$

where q is a random number uniformly distributed in [0…1], q_0 is a parameter $(0 \le q_0 \le 1)$ indicating the relative weighting of exploitation versus exploration, and S is a random variable selected according to the probability distribution given in equation (20).

```
initialize ACO parameters
calculate heuristic priority
allocate memory for PI_max instances
do
    for ƥ= 0 to PI_max parallel do
      local pheromone = global pheromone
      for ɐ= 0 to LA_max do
         construct solution
         evaluate solution
         local pheromone update
       end for
     end for
     keep best solution
     update global pheromone based on the best solution
while stopping condition not met
de-allocate memory
return solution
```

Figure 18. *High-level pseudo code for Ant Colony Optimization algorithm in OptPlatform.*

Only the ant with the best solution across all parallel instances ƥ deposits global pheromone. Let $F(ɐ)$ be a measure of ant ɐ's solution performance based on the objective function. Let ρ be the pheromone decay parameter in the range: $0 < \rho < 1$. Given the best solution found so far F^*, the global pheromone updating rule is defined as follow [106]:

$$\tau(e,o) = (1-\rho) \cdot \tau(e,o) + \rho \cdot \Delta\tau(e,o), \tag{21}$$

where $\Delta\tau_{ɐ}(e,o)$ is defined as [106]:

$$\Delta\tau_{ɐ}(e,o) = \begin{cases} F(ɐ) & if\ (e,o) \in F^* \\ 0 & otherwise \end{cases}. \tag{22}$$

As the ant constructs the tour, the pheromone level on visited *PossibleElement*s is changed by applying the local pheromone updating rule [106]:

$$\tau(e,o) = (1-\rho) \cdot \tau(e,o) + \rho \cdot \tau_0, \tag{23}$$

where ρ is the pheromone decay parameter in the range: $0 < \rho < 1$ and τ_0 is the initial pheromone level. The local pheromone update rule is designed to decrease the pheromone level on the visited PossibleElements such that they become less desirable for the next local ant. The effect of the local update is to decrease the pheromone level on visited edges which make them less desirable to subsequent ants.

This subsequently allows ants to explore more search space within the same iteration [106].

3.6.2. Evolutionary Strategy (ES)

Evolutionary Strategy (ES) is one of the simplest metaheuristics in terms of implementation, as it relies only on selection and mutation, as discussed in section 2.1.2.5. OptPlatform implements a simple (μ+1)-ES, where the parents μ is equal to the number of parallel instances PI_{max}. Furthermore, compared to standard (μ+1)-ES, ES in OptPlatform also implements a local search, where the mutation and evaluation is repeated for LI_{max} local iterations. The high-level pseudo-code is presented in **Figure 19**.

```
initialize ES parameters
allocate memory for PI_max instances
for þ = 0 to PI_max parallel do
    create a random solution
    evaluate solution
end for
keep best
do
    for þ = 0 to PI_max parallel do
      for r = 0 to LI_max do
        mutate
        evaluate solution
        keep best
      end for
    end for
    keep best
while stopping condition not met
de-allocate memory
return solution
```

Figure 19. *High-level pseudo-code for Evolutionary Strategy algorithm in OptPlatform*

The algorithm starts by creating a random population of size PI_{max}, by selecting an Order and associated Element at random, while satisfying the problem constraints. Next, the random population's best solution is chosen as a starting point for the search process. Each of the chromosome in the population is mutated and evaluated iteratively. In OptPlatform, a chromosome is represented as an encoded solution (see example in **Figure 20**). The mutation is performed by first removing order-element pairs from the solution, then adding new ones.

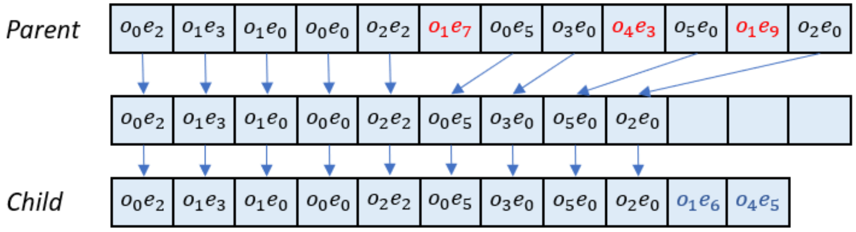

Figure 20. *Example of the mutation process of Evolutionary Strategy in OptPlatform. PossibleElement pairs with red are removed and replaced with PossibleElement pairs in blue.*

The number of elements to be removed is derived from mutation rate ʍ, expressed as a percentage. In the example of **Figure 20**, mutation rate ʍ is 0.25 or 25%; therefore, out of the 12 cells, three are removed. In the next step, the mutation process iterates over all *PossibleElement* pairs and adds the feasible ones to the solution. It is worth noting that the parent and child chromosome sizes can differ, depending on the problem constraints. In knapsack example, once some of the heavier items get removed, more smaller items can fit and vice versa.

The best chromosome in the iteration is kept as a parent for the next iteration. This process continues till a termination condition is reached. The variable size chromosome representation allows accommodating multiple problem encodings while maintaining the dynamics of the mutation. However, one of the drawbacks of this approach is that the mutation rate is dependent on the encoding. For example, problem encodings with small chromosome sizes (below 100 elements) would need proportionally larger mutation rate than the encoding with 1000 elements, to be able to maintain diversity in the population.

3.6.3. Imperialist Competitive Algorithm (ICA)

Although initially Imperialist Competitive Algorithm was introduced for continuous optimization problems, as discussed previously in section 2.1.2.6, in OptPlatform ICA is implemented for discrete optimization problems. Just like ES, ICA starts with the creation of a random population of size PI_{max}. Once the population is created, it proceeds with an empire initialization. Empire is a group of imperialist and at least one colony. In contrast to classic ICA, ICA in OptPlatform also implements a local search, where the solution creation via assimilation operator is repeated LS_{max} times. If at any point any of the colonies have a better cost than its imperialist, the imperialist and colony positions are swapped. Finally, iteration concludes with empire competition,

where the weakest empires are eliminated, while the strongest gain more power. The high-level process is shown in **Figure 21**. The following section formalises the ICA based on [124].

```
initialize ICA parameters
allocate memory for PI_max instances
for ƥ = 0 to PI_max parallel do
    create a random solution
    evaluate solution
end for
empire initialization
do
    for ƥ = 0 to PI_max parallel do
      for Ϭ = 0 to LS_max do
          assimilate
          evaluate solution
          keep best
       end for
     end for
     keep best
     empire competition
while stopping condition not met
de-allocate memory
return solution
```

Figure 21. *High-level pseudo code for Imperialist Competitive Algorithm in OptPlatform*

- **Empire initialization**

The ICA algorithm in OptPlatform starts by creating a random population and dividing them into colonies and imperialists based on the country's cost function. Furthermore, a country's cost is calculated in the same way as the provided solution's objective function. Therefore,

$$Cost = f(country) = f(solution) \quad (24)$$

At country initialisation, a random population of size N_{pop} is created and evaluated. The best countries of size N_{imp} are selected from the population and set as imperialists. Rest of the countries are set to be colonies N_{col}. In OptPlatform, ICA population N_{pop} is equal to the number of parallel instances PI_{max}.

$$N_{col} = N_{pop} - N_{imp} = PI_{max} - N_{imp} \quad (25)$$

Next, colonies are split amongst imperialists countries according to the power of the imperialists. The normalized cost of each imperialist country is determined by,

$$C_n = \max_i\{c_i\} - cost_n \quad (26)$$

where, $cost_n$ is the nth imperialist's cost, C_n is the normalized cost of nth imperialist. Weaker imperialist country (i.e. imperialist with higher cost) has a smaller normalized cost. Thus, the power of nth imperialist PO_n is calculated based on the normalized cost:

$$PO_n = \left| \frac{cost_n}{\sum_{N_{imp}}^{i=1} cost_i} \right| \tag{27}$$

The normalized power of nth imperialist is the number of colonies that are possessed by that imperialist, calculated by:

$$NC_n = round(PO_n \cdot N_{col}) \tag{28}$$

where NC_n is the number of initial colonies possessed by nth imperialist and $round$ is a function that gives the nearest integer of a fractional number.

- **Assimilation**

The classic ICA assimilation process is modified to accommodate discrete problems in OptPlatform. Each colony builds a new solution (country) by assimilating closer to its imperialist, based on assimilation rate θ ($0 \leq \theta \leq 1$). Assimilation rate determines how many entries in the solution is modified (assimilated) to create the new country. In the example of **Figure 22**, θ is set to 0.25, therefore 25% of all colony's solution is replaced by the imperialist's.

***Figure 22**. Example of Imperialist Competitive Algorithm assimilation process in OptPlatform. PossibleElements in red indicating the cells that are merged to create a new country.*

In the example, three cells out of twelve are replaced (marked in red) to create a new country that combines both colony and the imperialist, like combination operator in GA. The newly generated country must satisfy all problem constraints.

- **Empire competition**

Once each colony has finished building and evaluating solutions, empires compete amongst themselves to colonize each other's colonies. The empire competition is based on probabilistic empire power, where the strongest empires have the highest likelihood of possessing the weakest colonies. Total power of an empire is computed based on its imperialist power and a proportion of the power of its colonies [124].

$$TC_n = Cost(imperialist) + \zeta \cdot mean(Cost(colonies\ of\ empire_n)) \quad (29)$$

where TC_n is the total cost of nth empire, ζ is an empire influence coefficient ($0 \leq \zeta \leq 1$). Smaller values of ζ indicate a larger influence of the imperialist cost versus the mean of empire cost.

During the empire competition, weaker empires gradually collapse as they are left with no single colony. This means that the weaker imperialists lose their colonies and therefore the power to more powerful empires and consequently, increasing the power of the strongest imperialists. The competition process is modelled by computing the normalized cost of nth empire NTC_n [124]:

$$NTC_n = \max_i\{TC_i\} - TC_n \quad (30)$$

Then, the probability to possess a colony is computed by [124],

$$p_n = \left|\frac{NTC_n}{\sum_{N_{imp}}^{i=1} NTC_i}\right|, \text{ where } \sum_{i=1}^{N_{imp}} p_i = 1 \quad (31)$$

Let vector P of size N_{imp} contain the possession probabilities of a colony by empires as follows:

$$P = [p_1, p_2, \dots, p_{N_{imp}}] \quad (32)$$

Then, vector R with the same size is generated based on uniform distribution between 0 and 1.

$$R = [r_1, r_2, \dots, r_{N_{imp}}], \text{ where } r_i \sim U(0,1) \quad (33)$$

Next, vector K is calculated by subtracting P from R.

$$K = P - R = \left[K_1, K_2, \dots, K_{N_{imp}}\right] = [p_1 - r_1, p_2 - r_2, \dots, p_{N_{imp}} - r_{N_{imp}}] \quad (34)$$

Once the vector K is calculated, the weakest colony is assigned to the empire with the largest index.

3.7. Visualisation tools

Visualization tools are very important for understanding the produced outputs of the black-box optimizers. Thus, multiple visualization tools are developed in OptPlatform that help the users understand how the search process progresses and how to implement the produced output into an existing real-life system. This section gives a brief description of the tools and gives some examples.

- **Metaheuristic inner-state visualization**

A large proportion of metaheuristic algorithms have memory, that is being used to guide the search. To easier debug and understand how the search works, it is beneficial to look at the inner states of the memory and how it progresses.

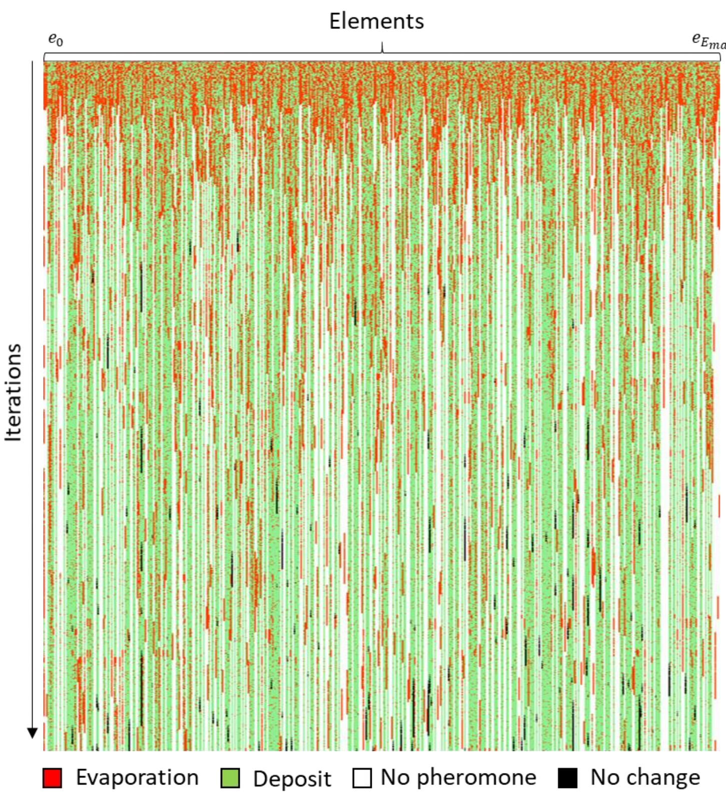

Figure 23. *The output of the global pheromone visualization tool. Each pixel represents a pheromone change for given element across multiple iterations. With red pixels indicating when evaporation happens, green – pheromone deposit, white – no pheromone left for the specific element and in black – no change between the iterations.*

Figure 23 gives an example of ant colony global pheromone matrix and how it evolves during the search process – in this case, MKP gk01 instance. On the horizontal axis are all the different possible elements (items) that are in the search

space plotted as transitions between iterations. In this case, each pixel represents a pheromone deposit or evaporation (green and red respectably). Once the pheromone is fully evaporated, it is coloured in white, while if there is no change between the iterations – pixel is coloured in black.

An inner-state visualization is a useful tool as it may quickly highlight problems with the search, such as insufficient diversification in the population or over intensification that leads to being stuck to a local optimum.

- **Search convergence and statistics**

Another essential property for any MOF is the ability to visualize and analyse different algorithm search performances quickly. For that reason, little utility is developed that allows user to import any simulation results and then compare the convergence graphs and averages across multiple strategies. As metaheuristics are non-deterministic, usually several runs with the same configuration but different seed are performed, also referred as *simulation*. **Figure 24** gives the GUI example, where various methods of ES are compared.

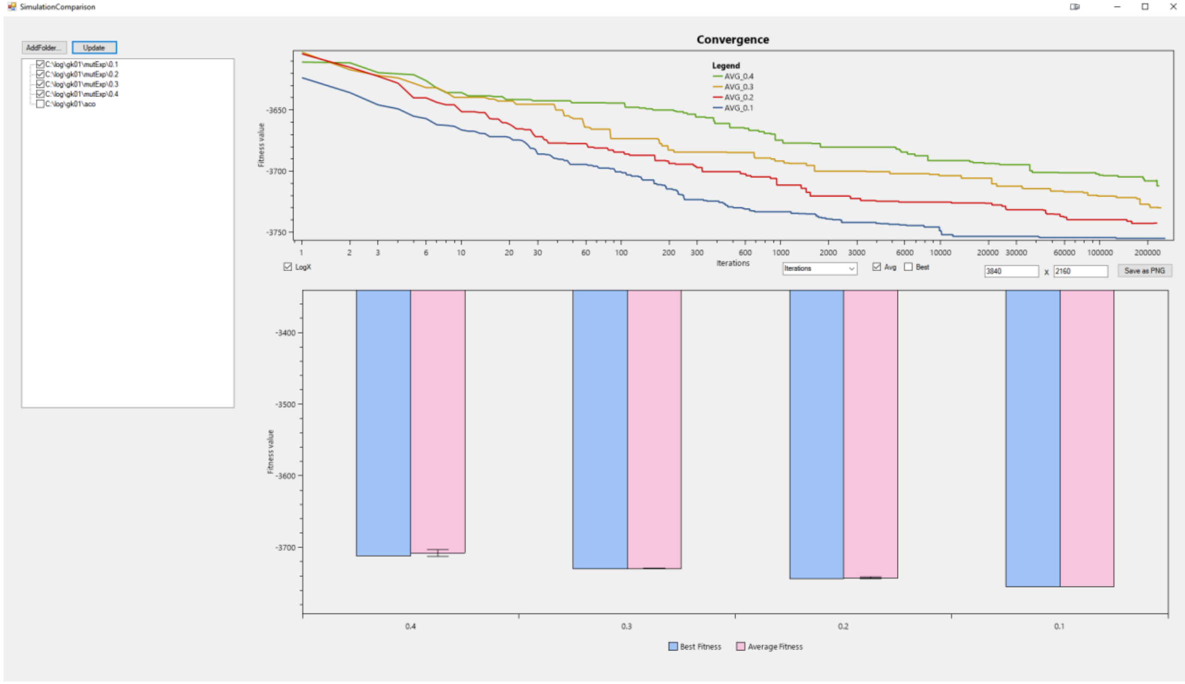

Figure 24. *OptPlatform's search visualization tool.*

Similarly, the user may want to investigate a specific result, and thus, double-clicking on any simulation opens a simulation summary, where minimum, maximum and average fitness across iterations are charted and additional statistical data provided about the simulation (GUI example in **Figure 25**).

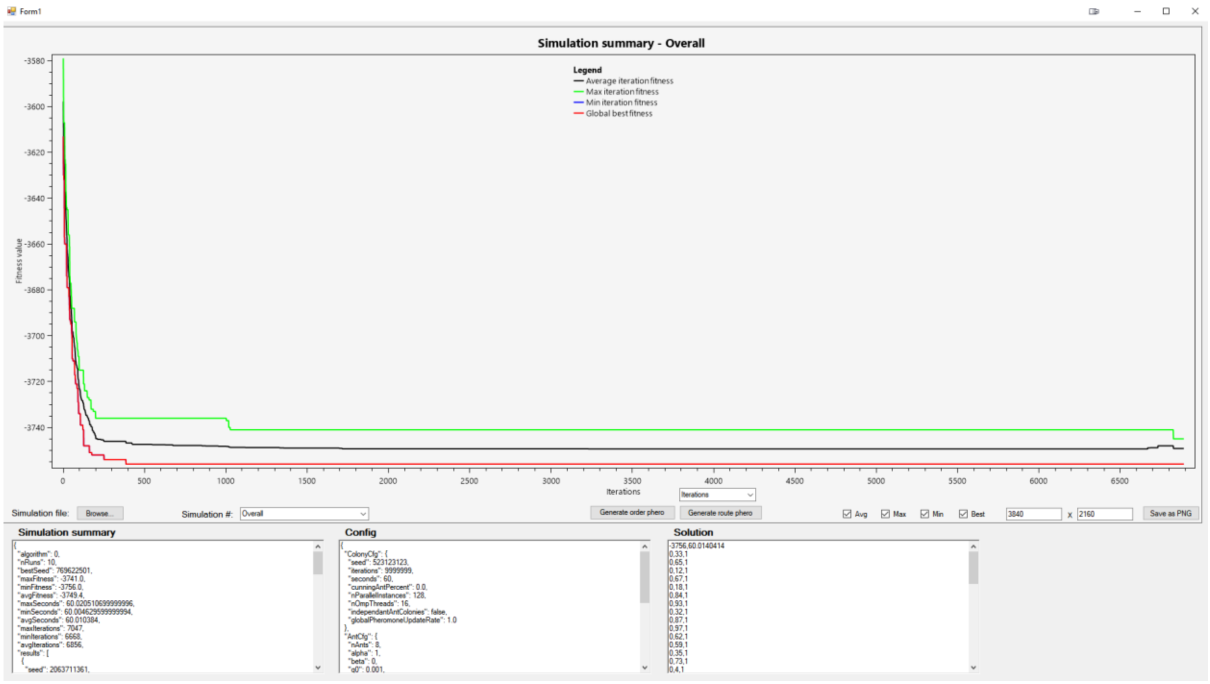

Figure 25. *Simulation summary graphical interface.*

- **Global grid**

Any routing optimization problem modelled based on real locations in the map requires a visualization tool to understand how the locations are linked and interact with each other. For that reason, Global grid has been developed, where a simple animation is generated automatically with the provided edge coordinates. If applicable, animation considers how fast the given route is executed and assigns an icon for the mode of transport – a truck, plane, or a ship.

Output visualization example of the Transcom problem (section 2.3.2.3) is shown in **Figure 26**.

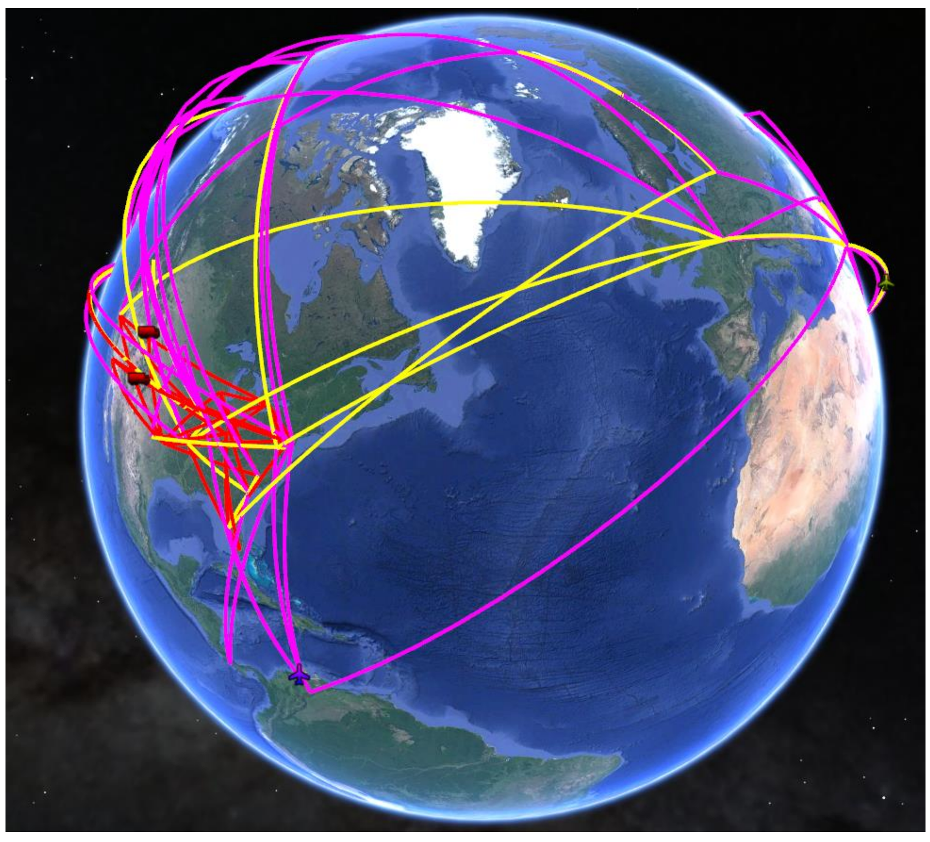

Figure 26. *Automatically generated solution animation of Transcom problem using Google Earth.*

3.8. Solution transition optimisation

In many complex real-world optimisation models, getting optimum or close to the optimum solution is only part of the problem. It is often impossible to adopt the optimum solution overnight, as it would cause too much distribution and negate any savings. A good example of such dilemma is often seen in the global supply chain, where the contracts are signed for months and years in advance and just changing the courier might add additional costs, such as penalties and legal costs. Similarly, in manufacturing plant, migrating to entirely different equipment or workflow all at once may cause a disruption in itself. For that reason, a more gradual transition from the sub-optimal current state to the optimized state is required. The question then becomes which changes take priority over the others; they might be specified by expert knowledge, or by an automated greedy search. This section looks at transition optimisation problem – how to transition from the current sub-optimal solution to the optimal solution, within a limited number of steps (Stages).

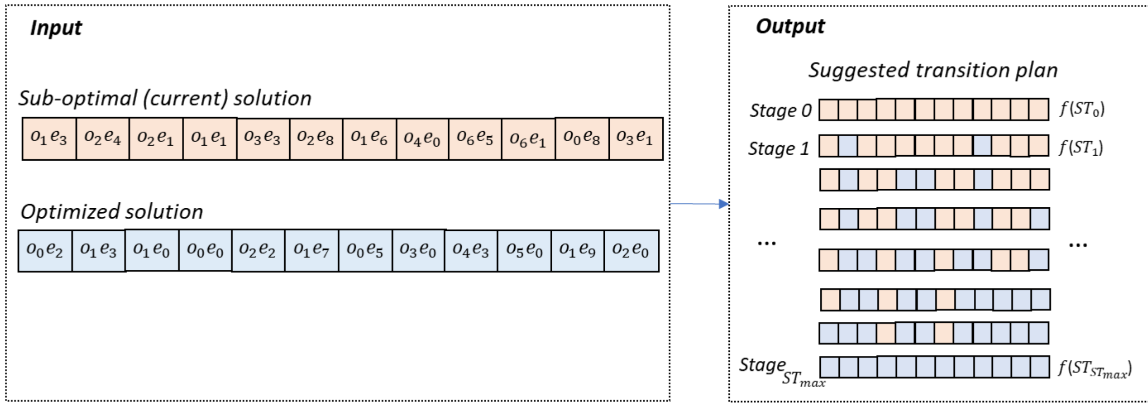

Figure 27. *A high-level overview of transition optimisation, where two solutions (sub-optimal and optimized) are used as inputs to generate a transition plan based on the provided goal. In this example, seven stages are generated starting from the sub-optimal solution at Stage 0 to optimized solution at Stage 7.*

The transition optimisation problem can be structured as a combinatorial problem, where the goal is to minimize the sum of solution scores (fitness's) f across all stages ST_i ,where ST_{max} is the target number of stages:

$$min \sum_{i=1}^{ST_{max}} f\,(ST_i) \tag{35}$$

Each stage represents a valid solution with fitness $f(ST)$ that is a combination of the two inputs – the current suboptimal solution and the optimized solution. A high-level overview is shown in **Figure 27**, where the current sub-optimal solution is represented in orange and optimized solution in blue.

In order to construct transition optimisation as a combinatorial problem, the in-between stage solutions need to be created, such that they both are valid and encompass the required number of stages ST_{max}. For this, a simple state generation algorithm is implemented. The algorithm starts by first, calculating the total number of non-overlapping element-order pairs N_{SL}. Each order-element pair that differ between the two solutions are put on the *swap list,* with the size of N_{SL}. The *swap list* represents the differences between the two solutions. Next, the target number of swaps per stage TN is calculated as:

$$TN = \frac{N_{SL}}{ST_{max} - 1} \tag{36}$$

Then an exhaustive bucket S_{ST} containing all potential swaps of the size TN is generated. This set is used as an input for the optimization algorithm.

Optimization algorithm uses the exhaustive bucket S_{ST} to select entries one by one to assemble a transition plan, where each addition to the solution represents a Stage in the transition plan. If the order-element pair already exists in the previous Stage, it is not added again. If there is no feasible transition that can be achieved by single addition, two or more additions are used per Stage. This is now structured as a combinatorial problem, where elements from a set are selected to assemble the solution – the final transition plan. Furthermore, OptPlatform's search cores are re-used for the optimization of large models (with N_{SL} above 50), though for smaller models search is done exhaustively. Moreover, the algorithm also reuses the optimisation problem constraints and fitness calculation specified by the user; thus, only the current state (sub-optimal) solution is needed as the input to generate suggestive transition plan automatically.

3.8.1. Numerical examples

To demonstrate this technique, first the transition optimization is applied to simple benchmark MKP gk01 [188] instance with 100 items, see section 2.3.1.1 for problem definition. Each selected item in the MKP represents an element in the solution. An exhaustive search was used for optimisation, and the summary of the best transition plan with five stages between the sub-optimal solution of 3553 to the optimal solution of 3766 is shown in **Table 3**.

Table 3. *Solution transition plan for MKP gk01 with maximizing profit as objective $f(ST)$. N_{SL} represents the number of element-order pairs that differ from the final solution.*

Stage	$f(ST)$	N_{SL}
0	3553	43
1	3636	32
2	3712	21
3	3752	8
4	3766	0

The initial sub-optimal solution at Stage 0 differ by 43 items in the knapsack; thus, the target number of swaps TN is 11. It is worth noting that it was impossible to transition between Stage 2 and Stage 3 by using exactly 11 swaps; thus, constraints were relaxed up until 13 swaps, when feasible solution state for the Stage 3 was generated with state fitness of 3752.

Next, practical, real-world model based on Transcom optimisation problem (section 2.3.2.3) was used to optimize the transition plan for the lowest total cost objective. The network's current state costs $2543 million in total, while the optimized network costs only $929 million (reduction of 63%). Similar to above MKP example, transition plan targets five stages with four transitions. Initial Stage differs by 66 swap pairs from the final Stage ($N_{SL} = 66$), thus the target number of swaps TN is 17. As this is a large model, three metaheuristics were used as optimization algorithms for the transition plan generation, the best results are summarized and compared in **Table 4**.

Table 4. *Solution transition plan for Transcom scheduling and routing problem with minimizing total cost (in million $) as objective $f(ST)$. N_{SL} represents the number of element-order pairs that differ from the final solution.*

	$f(ST)$, cost in million $			N_{SL}		
Stage	**ACO**	**ES**	**ICA**	**ACO**	**ES**	**ICA**
0	2543	2543	2543	66	66	66
1	1796	1732	1774	49	49	49
2	1421	1618	1355	32	30	31
3	1032	1132	1012	15	13	10
4	929	929	929	0	0	0
Total	7721	7954	7613			

Out of the three transition plans in **Table 4**, the most cost-effective solution is the transition plan generated by ICA, where the total fitness across all stages f is lower than both ACO and ES. Furthermore, if we assume each stage represents a calendar month, ICA proposed transition plan costs more than ES ($1774 million vs $1732 million) in the first month. It still offers significant cost reduction in the second and third month before the final optimum solution is reached in the fourth month. These examples clearly illustrate the importance of transition planning problem and the corresponding optimisation for cost savings.

3.9. MOF comparisons

This section compares the implemented OptPlatform discussed in this chapter with other metaheuristic optimization frameworks available in the literature (discussed in section 2.2).

Nine out of sixteen frameworks containing knapsack problem example code were considered for comparison with OptPlatform, however, due to the combination of lack of documentation, missing source code or broken dependencies, only three MOFs

could be compiled and run successfully. The provided example code of knapsack problem was extended to Multiple Knapsack Problem (MKP, specified in section 2.3.1.1) for three MOFs - HeuristicLab, JAMES and JCOP. Furthermore, commercial tools based on Google OR-tools library[21] are included for reference.

All platforms considered were using default parameters available in the knapsack examples provided and the number of iterations for each run closely matched to similar algorithms. For example, all GA instances were run with 25,000 generations; similarly, ES generations were set to 50,000. The algorithm was terminated either by reaching the optimum solution, the maximum number of iterations or maximum computation time of 180 seconds. The configurations used are summarized in **Table 5**.

Table 5. *Parameters used for an experiment on a various algorithm on different MOFs*

Platform	Algorithm	Comment
Google OR-tools	B&B	Branch and Bound solver, termination set to 180 seconds
	CBC	Integer Programming Solver CBC
HeuristicLab	GA	MultiBinaryVectorCrossover, 25,000 iterations, 5% mutation rate
	ES	SomePositionsBitflipManipulator, 50,000 iterations, 5% mutation rate
JAMES	RD	Random Descent, termination set to 180 seconds
	PT	Parallel Tempering, 64 nReplicas, termination set to 180 seconds
JCOP	GA	25,000 iterations, 5% mutation, termination set to 180 seconds
OptPlatform (this work)	ACO	10,000 iterations
	ES	50,000 iterations, 5% mutation rate
	ICA	Termination set to 20 stagnant iterations

One small MKP instance from OR benchmark library (OR5x100-0.25_01) [189] and three medium-hard MKP instances of GK benchmark library (gk01, gk02 and gk03) [188] were selected for the comparison. Each algorithm was run ten times to establish best and average error percentage from the optimum solution, as well as standard deviation and average computation time, in seconds. Where applicable, parallelism was enabled in the MOF. Results are summarized in **Table 6**. All experiments were conducted on Windows 10 pro workstation with AMD Ryzen Threadripper 3970X 32c-64t processor and 64GB of RAM.

It is worth noting that PSO failed to run on HeuristicLab using the MKP – indicating the shortcomings of the platform's generalizability. Furthermore, JAMES documentation claim to support tabu search, however, could not be applied to MKP. Similarly, JCOP failed to run OR-100 example, even though had no problems with more complex MKP instances, demonstrating some stability issues with the platform.

[21] Google OR-tools library: https://developers.google.com/optimization

Table 6. *Metaheuristic Optimization Framework comparisons. Best and average expressed as error per cent from an optimal solution, colour coded from the best error (in green) to the worse (in red). Google OR-tools is added for reference only and is not considered a MOF.*

		Best error				Average error				Standard deviation				Computation time (s)			
		OR-100	gk01	gk02	gk03	OR-100	gk01	gk02	gk03	OR-100	gk01	gk02	gk03	OR-100	gk01	gk02	gk03
Google OR-tools	B&B	29.10%	44.40%	51.69%	66.99%	29.10%	44.40%	51.69%	66.99%	0.0	0.0	0.0	0.0	180.0	180.0	180.0	180.0
	CBC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.0	0.0	0.0	0.0	1.1	8.5	74.3	9777.3
HeuristicLab	GA	1.40%	1.78%	1.89%	1.93%	2.75%	2.37%	2.24%	2.52%	236.0	16.3	10.1	21.1	107.7	111.6	113.6	114.0
	ES	7.51%	1.96%	2.12%	2.32%	8.45%	2.22%	2.13%	2.37%	326.2	4.9	1.2	2.8	118.6	128.3	133.4	138.8
JAMES	RD	13.10%	5.76%	1.82%	5.50%	10.03%	6.57%	3.96%	5.52%	534.4	32.9	42.4	1.2	180.0	180.0	180.0	180.0
	PT	0.00%	0.19%	0.03%	0.28%	0.00%	0.20%	0.11%	0.40%	0.0	0.5	2.2	3.4	51.2	180.0	180.0	180.0
JCOP	GA	-	0.29%	0.25%	0.34%	-	0.35%	0.36%	0.38%	-	2.5	2.7	1.9	-	180.0	180.0	180.0
OptPlatform (this work)	ACO	0.00%	0.00%	0.18%	0.25%	0.22%	0.41%	0.36%	0.43%	48.3	8.6	3.5	8.0	20.0	50.6	66.3	144.2
	ES	0.00%	0.19%	0.08%	0.32%	0.16%	0.36%	0.31%	0.46%	34.8	4.0	5.2	6.4	36.4	52.2	54.2	90.4
	ICA	0.00%	0.00%	0.03%	0.11%	0.00%	0.00%	0.04%	0.11%	0.0	0.0	0.9	0.9	4.6	12.5	56.0	106.1
Optimum		*24381*	*3766*	*3958*	*5656*												

Results in **Table 6** demonstrate the wide range of performance of optimization methods. On the one hand, you have a simple Branch and Bound (B&B) algorithm that cannot find even adequate solution within 180 seconds. On the other hand, you have a linear solver (CBC) that is guaranteed to find an optimal solution, but on complex MKP instances take exponentially more time. For example, gk03 took on average 9777 seconds or 2.7 hours to produce a solution. In comparison, most metaheuristic algorithms were completed within three minutes, with few reaching near-optimal solutions before that.

It is hard to draw impartial comparisons between structurally different MOFs and their corresponding algorithms. However, when MOFs are compared to similar family algorithms, like ES, GA and ICA, it can be clearly seen that OptPlatform implementations outperform all other MOFs in terms of solution quality and computation time. Out of the four compared MOFs, ICA on OptPlatform performed the best, followed by PT on JAMES. Furthermore, a third-place shared by ACO/ES on OptPlatform and GA on JCOP.

Results demonstrate that OptPlatform is generic and supports a wide range of metaheuristic implementations. It also benefits from the hybrid C++/C# architecture, where the high-performance low-level search cores produce a good quality solution in a fraction of the time compared to competing MOFs.

3.10. Summary

In this chapter, the requirements, essential for adaptation of MOFs by practitioners in the industry, have been set out and fulfilled. Furthermore, the chapter lays out the reasoning and implementation of the architecture in OptPlatform. This section summarises how the OptPlatform satisfies and improves on all the requirements laid out in section 3.1.

In terms of **applicability**, the optimization problem implementation is well documented with examples. Furthermore, to further assist the user, program code templates are automatically generated, such that the problem constraints and fitness metrics can be easily implemented without prior knowledge of the underlying metaheuristics. Finally, the solution transition optimisation automatically generates a step by step guide on applying the optimal solution in real-world.

Another critical requirement was the platform's **genericity**, which OptPlatform achieves by separating the problem-specific user domain from the underlying platform's search algorithms. The solution encoding is generic so that multiple different kinds of problems can be implemented and successfully optimized for. The OptPlatform is capable enough to optimize five problems with different complexity and application domains, discussed as part of section 2.3. The three benchmark problems can also be used as **problem code examples** to familiarize the user with the platform.

An **interoperability** is essential for the platform to be integrated into the existing infrastructure. OptPlatform deploys hybrid C++/C# architecture, where the low-level high-performance search cores are compiled as C++ DLL library and thus can be used by any existing software. A higher-level language such as C# allows the whole platform to be interfaced with existing APIs, databases or data streams easily without losing the performance.

The OptPlatform was designed with **parallelism** as its core feature. The concurrency is abstracted away from the user with efficient use of static and dynamic memory in the problem definitions. This leads to very efficient and **high-performance** search algorithm implementations, that allows for reasonable quality solutions to be generated quickly. OptPlatform shows better and faster results than competing MOFs, as shown in section 3.9. Furthermore, as the platform is intended mainly for industry, the parallelism dynamics of real-world problems are studied in detail in Chapter 5.

Parameter management that allows an unsophisticated user to get most of the metaheuristics algorithms is one of the major improvements of OptPlatform that are missing on the existing MOFs. Thus, this feature is covered separately in Chapter 6.

Furthermore, OptPlatform improves on most existing MOFs, which can only support evolutionary algorithm-like encoding, where a solution is built on top of an existing solution. In OptPlatform, there is no such limitation, and metaheuristic algorithms that build the solution from scratch (like ant colony optimization) are also supported. Thus, OptPlatform **supports multiple algorithms** and even a new metaheuristic algorithm design. Moreover, the separation between the problem domain and the search domain allows a carefree implementation of the user problem. This, in return, lowers the learning effort required to start using the software.

Although the OptPlatform is not designed for new metaheuristic algorithm research, it can be successfully used also for that purpose, as shown in the next chapter, Chapter 4. The framework is flexible enough that new metaheuristic algorithms can be developed independently while maintaining existing operational models and algorithms intact.

4. THE IMPERIALIST COMPETITIVE ALGORITHM WITH INDEPENDENCE AND CONSTRAINED ASSIMILATION (ICAwICA)

This chapter is based on the results published in [3].

Any metaheuristic optimisation framework search results are limited to the underlying metaheuristic algorithms. Although metaheuristics are not problem-specific, some are better at solving the problem at hand than others. As discussed in section 3.6, the Imperialist Competitive Algorithm was chosen for OptPlatform due to the wide range of applications and improved search convergence compared to the genetic algorithm.

This chapter develops methods to improve existing ICA for combinatorial problems, called ICA with Independence and Constrained Assimilation (ICAwICA). The proposed algorithm introduces the concept of colony independence – a free will to choose between classic ICA assimilation to the empire's imperialist or any other imperialist in the population. Furthermore, a constrained assimilation process has been implemented that combines classical ICA assimilation and revolution operators, while maintaining population diversity. In order to evaluate the performance and generalisation aspects of the proposed approach, two different kinds of combinatorial benchmark problems were selected – subset selection and routing, Multiple Knapsack Problem (section 2.3.1.1) and Multiple Depot Vehicle Routing Problem (section 2.3.1.2), respectively. The performance is evaluated against competing metaheuristics in the literature using the implementation within OptPlatform (described in Chapter 3).

4.1. Motivation and related work

Imperialist Competitive Algorithm (ICA), described in detail in both sections 2.1.2.6 and 3.6.3, was first developed for solving continuous math equations. Since then, there have been various attempts on improving the standard ICA search performance. For example, authors in [190] proposed an adaptive ICA (AICA) that uses a

probabilistic model based on colony positions to escape local optimum. Similarly, [191] improved the convergence speed of the algorithm by adding additional value to an unfeasible solution, based on its distance from the relative imperialist. Both [192] and [193] enhanced ICA by implementing an attraction and repulsion concept during the search for better solutions. The less researched area is the use of local search in ICA. Local search has been used to improve convergence on other metaheuristics, such as in Ant Colony System [106] by local pheromone update rules, or small swarm division in PSO [194]. The standard ICA does not implement any form of local search and therefore, may get stuck in local optima before converging to the global best solution [195]. Only a few approaches for solving this problem have been proposed in the literature, such as simulated annealing-like processes in [196], where the local search process is applied for machine-selection part and the operation-sequence part in Flexible Job-Shop Problem (FJSP). The 2-opt is another popular local-search operator for routing problems, such as Travelling Salesman Problem (TSP). For example, work in [197] uses 2-opt with ICA to improve the imperialists. For continuous optimization problems, local search operator such as random line search has been explored in [198], where authors applied the problem-specific local search for the imperialist solutions.

However, many of these local search implementations rely on problem-specific operators or assimilation. These operators exploit the underlying problem dynamics and are an effective way to improve the convergence. Although some can be transferrable across similar class problems, they are rarely generic enough to be applied for a wide range of problems. For example, a 2-opt local search would be of no use for a knapsack problem. In attempt to overcome this issue, this chapter proposes a modified ICA, where the local search process is performed in terms of both an Independence operator and a Constrained Assimilation (ICAwICA). Compared to existing ICA local search approaches, ICAwICA proposes a more generic implementation that does not require problem-specific operators.

Thus, in this chapter, a more generic algorithm with a local search is presented. It expands on the classic ICA, with the use of novel Independence operator and Constrained Assimilation, called ICAwICA. The contributions can be summarized into the following to aspects:

- A novel generic ICA is proposed, where the standard assimilation and revolution process is replaced with constrained assimilation and the novel independence operator used for local search.
- The performance of the ICAwICA algorithm is comprehensively evaluated via well-known Multiple Knapsack Problem (MKP) and Multi Depot Vehicle Routing Problem (MDVRP) benchmark instances. The experimental results demonstrate the superiority over classic ICA and universality of the local search.

4.2. Methods and implementation

The following section introduces the classic ICA and the novel ICA with Independence and Constrained Assimilation (ICAwICA) algorithm. It discusses the changes and advantages of constrained assimilation. Finally, ICAwICA application to two different example problems is considered.

4.2.1. Classic ICA

Like many other population algorithms, ICA starts its search by generating a random initial population where each individual of the population represents a country. Countries within ICA can be thought of as chromosomes in a genetic algorithm. The initial population is separated into multiple groups (so-called empires). Most influential countries become imperialist within the empire and weakest - their colonies. Each colony within empire moves closer to their imperialist in the form of assimilation operator. In order to provide diversity amongst countries, a revolution operator (mutation in GA) is implemented. If at any point a colony becomes stronger than its imperialist, then the two countries are swapped, such that imperialist is the strongest country in the empire. The search follows an iterative process, where after each iteration, the weakest colony within the weakest empire is assigned to one of the stronger empires – following the imperialist competition process. An empire is eliminated once it contains no more colonies. The search usually continues until the termination criteria are met. Ideally, the search is terminated once all empires are eliminated and only one, the best, empire remaining.

4.2.2. ICAwICA

The proposed ICAwICA follows the classic ICA [125] principles for both empire initialisation and empire competition; however, assimilation and revolution operators are replaced with a constrained assimilation and repair mechanism. Furthermore, in the classic ICA, each colony within an empire is moving closer to the imperialist within that empire. In contrast, in ICAwICA all colonies are given a free choice to move closer to any of the imperialists of other empires (independence), as long as it improves the country's well-being (associated cost). Therefore, at each iteration, a colony k has a probability based on a uniform distribution $(rand)$ of either move closer to their own empire's imperialist or to move closer to any other imperialist j, determined by $iRate$ (0-1.0). Moreover, this process is repeated δ times for each colony to explore more search space around its position in the form of local search. Pseudocode of the ICAwICA is shown in **Figure 29**. The flowchart for both classic ICA and ICAwICA is shown in **Figure 28**, with red indicating the changes.

4.2.3. Constrained assimilation

Classic ICA was first developed for continuous math's problem with simple assimilation processes [125], ICA has since been applied to multiple binary problems, such as feature selection [199][200], content-based-image retrieval (CBIR) [201] and single-dimensional 0-1 knapsack problems [202]. However, binary assimilation approaches cannot always be extended to other discrete, non-binary problems.

Furthermore, most ICA discrete assimilation implementations follow simple genetic-algorithm-like crossover operations, where the chromosomes are expected to be of equal size [203] [204]. The proposed Constrained Assimilation (CA) process does not require equal chromosome/solution size and is extendable to other constrained discrete problems. CA exploits the fact that two solutions cannot always be merged without violating constraints. Therefore, CA builds a new incomplete solution from the two donor solutions/countries (colony and imperialist) according to the assimilation rate and finishes the solution by a repair mechanism.

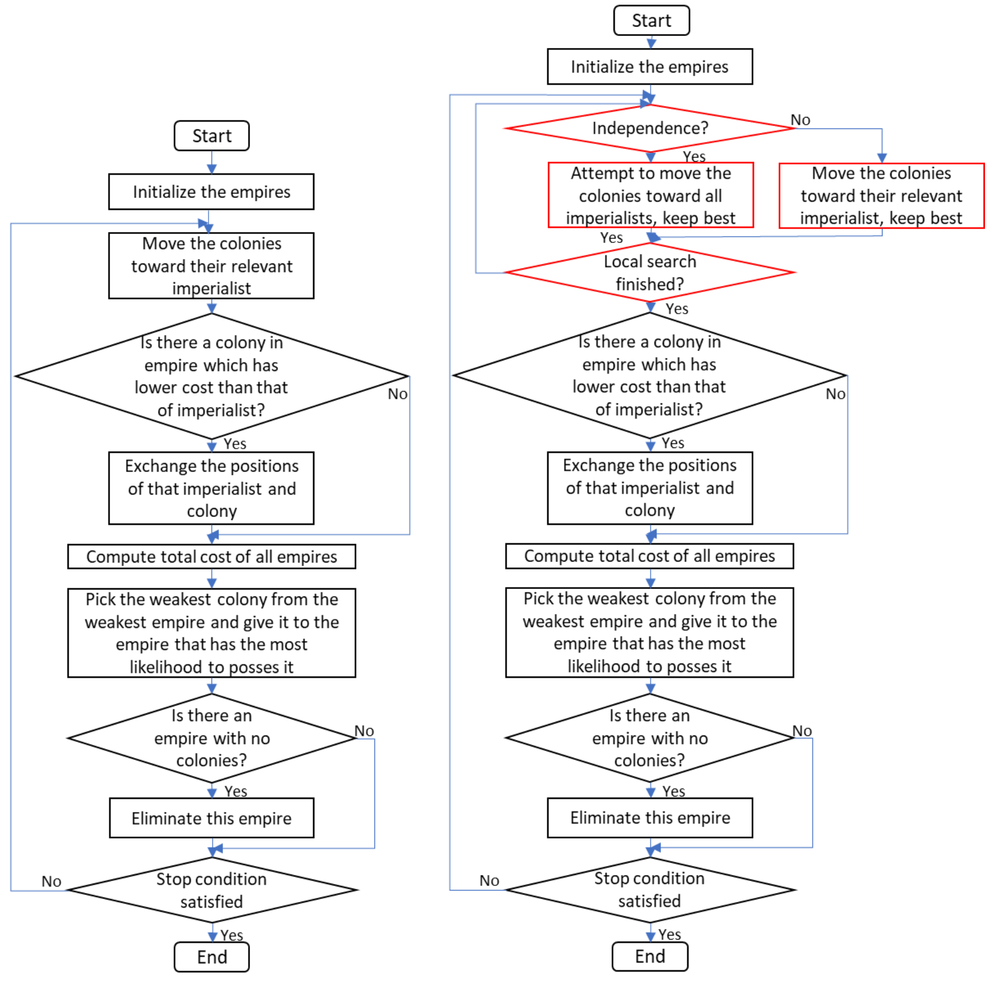

Figure 28. *Flowchart of classic ICA [125] (to the left) and the proposed ICAwICA (on the right), with red indicating the changes.*

There are multiple ways to implement the solution repair mechanism - based on heuristics, existing solution population, sequence-based [205] etc. The most straightforward repair mechanism is - scanning through all possible entries and trying to add them to the solution without violating constraints (used in the OptPlatform's ICA implementation). Furthermore, this incomplete solution repair enables diversity without an explicit revolution operator like classic ICA. Although more computationally expensive than simple assimilation, this approach has potential for broad applications and generalisation. It does not depend on two solutions having the same size or problem-specific assimilation or repair mechanism. Furthermore, CA's generated solutions are always within constraints and do not require any penalty cost definition at evaluation.

```
1. Initialize ICA parameters.
2. Create the population randomly.
3. Initialize empires:
   for $i = 1$ to $N_{pop}$
      Compute the cost function $C_i$;
      Sort the computed cost $C_i$ in descending order for the entire population;
      Select $N_{imp}$ out of $N_{pop}$;
      Normalize the cost of each imperialist $C_n$;
      Calculate the normalized power of each imperialist $PO_n$;
      Assign remaining countries $N_{col}$ to the imperialists;
   end loop
do
   4. Assimilation and local search process for ICAwICA:
      for k $= 1$ to $N_{col}$
         for $l = 1$ to 6
            if $rand < iRate$
           for $j = 1$ to $N_{imp}$
               assimilate colony $k$ closer to $j$
               if cost for new position is less than original position
                  keep assimilated position
               else
                  discard and move back to original position
               endif
           end loop
            else
      assimilate colony $k$ closer to empire's Imperialist
            endif
        end loop
      end loop
      for $j = 1$ to $N_{imp}$
       if the cost of any colony is less than cost of imperialist
             exchange the position of the colony and imperialist;
       endif
      end loop
   Pick the weakest colony (colonies) from the weakest empire and assign it to the
   empire with highest probability to possess it;
  5. Elimination process:
      If there is imperialist with no colonies
      eliminate the imperialist;
      endif
while stopping condition not met;
```

Figure 29. *The pseudocode for new assimilation and local search method for ICAwICA*

A CA example is provided in **Figure 30**. Both colony and imperialist are assimilated, with bold integer values corresponding to solution entries (item indices in MKP case, or depo indices in MDVRP case) are passed to the new country, determined by assimilation rate. In this simple example, a 50% assimilation rate of $N_{solutionSize}$ is used to build the new country. Due to constraints, not all solution entries can be added to the new country and hence the solution is in an incomplete state. The repair process iterates over all possible solution entries and fills the gaps while complying with constraints. Let us consider in detail the assimilation process shown in **Figure 30**. The colony solution is shown in blue and the imperialist in yellow, with the newly generated

country nc, new entries (index 1 and index 3) were introduced to the solution after repair that were not in any of the donor countries

$N_{solutionSize}$

Colony	2	9	8	6	7	13	5	
Imperialist	12	19	5	12	2	6	15	8
New country nc	2	19	5	6	7			8
nc after repair	2	19	5	6	7	1	3	8

Figure 30. *Imperialist and colony constrained assimilation process with solution repair. With integer values corresponding to solution entries (item indices in MKP case or depo indices in MDVRP case).*

4.2.4. ICAwICA solution encoding for MKP and MDVRP

The ICAwICA is generic and does not rely on any specific solution structure or problem-specific assimilation operators and, therefore, can be applied to various kinds of discrete optimisation problems. Two different types of combinatorial problems have been explored – a subset selection problem in MKP and a routing problem in MDVRP. In the MKP case, each element in the solution represents an item index that has been added in the knapsacks. Thus, the performance of the solution is evaluated by iterating over all entries and matching indices to the item profits.

For the MDVRP, first, customer-depot relationships are encoded as a country. Each country is represented as a vector of the size of the number of customers, where each customer is assigned a depot index. An example of new country creation via assimilation for the MDVRP is shown in **Figure 31**, where the initial colony has encoded the following grouping: Customer 2 and 8 will be routed from Depot 1; Customers 1, 3 and 6 will be routed from Depot 2; Customers 5,7,9 and 10 will be routed from Depot 3, and finally, Customer 4 will be routed from Depot 4. Each time a new country is created as part of the ICAwICA assimilation process, capacity constraints are considered such that the total demand for all customers assigned to the depot does not exceed the maximum capacity available across all vehicles to the given depot.

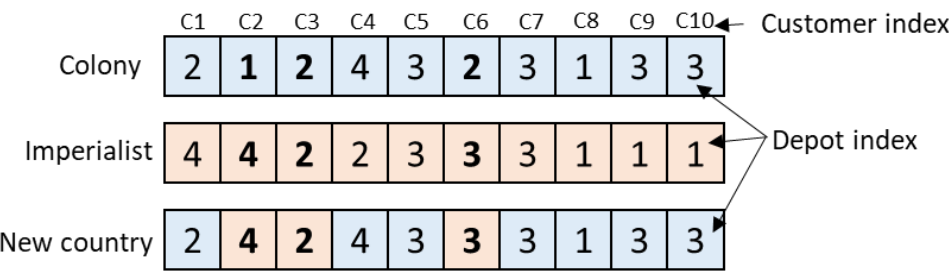

Figure 31. *Customer assignment to depots in MDVRP using ICAwICA assimilation. Where C1-C10 are customer indices and the encoded integers are depot indices that are assigned to a given customer, with bold representing assimilated changes.*

Furthermore, the example in **Figure 31** also shows an assimilation process for the colony and imperialist; considers ten customers that are grouped into four depots. Bold type represents assimilated changes. For example, Customer 2 (C2) demand was previously supplied by Depot 1 but now is supplied by Depot 4. Similarly, Customer 6 (C6) demand was previously supplied by Depot 2 but now is supplied by Depot 3.

Finally, solution performance is evaluated by first grouping all depot indices in the solution, then constructing routes based on the sequence it was added to the solution (from left to right). Thus, in the example in **Figure 31**, the new country solution would be Depot 1 supplying customer 8, Depot 2 supplying customers 1 and 3, Depot 3 supplying customer sequence 5-6-7-9-10, and finally, Depot 4 supplying customers 2 and 4.

4.3. Experiments

In this section, the proposed ICAwICA algorithm performance is compared to classic ICA. Next, the dynamics of independence operator are analysed. Finally, extensive computational experiments on classical MKP and MDVRP benchmark instances are conducted and compared to the current state-of-the-art algorithms.

4.3.1. Benchmark instances

Multidimensional knapsack problem instances were chosen because of their availability, ease of implementation and the frequent use as benchmarks across the research community. ICAwICA was tested across 41 accessible benchmark instances, all available from the compiled library in [189].

The simplest benchmarks are derived from the WEISH dataset, containing 30 problems with the number of items ranging from 30 to 90 and with five knapsacks

each. Furthermore, to explore the performance of the proposed algorithm across a range of datasets, large MKP instances, generated by Glover and Kochenberger (GK) [188], were also selected. The GK dataset contains 11 instances with the number of items ranging from 100 to 2500 with 15 to 100 knapsacks each and provides a broad spectrum of complexity.

Moreover, the ICAwICA was also tested on the 23 Cordeau's MDVRP benchmark instances obtained from [206]. The benchmark dataset offers a wide range of complexity, from the number of customers ranging from 50 to 360 and the number of depots from 2 to 9; and specifies the current Best-Known Solution (BKS).

4.3.2. Experimental setup

The proposed ICAwICA algorithm was implemented in C++ using the Visual Studio 2019 (v142) compiler. The computation was performed on a workstation with AMD Threadripper 2990WX processor (3.0 GHz, 64GB RAM), running Windows 10 Pro operating system.

Like classic ICA, ICAwICA also has multiple algorithmic hyper-parameters that were empirically set and are as follows for all tested instances unless specified otherwise: MKP - total number of countries N_{pop} is set to 4096 for all instances with the number of items $n < 500$ and value of 512 for all instances with $n \geq 500$. Out of all countries, 40% are initialised as imperialists N_{imp}. Local iterations ϭ are set to 3. Assimilation rate θ set to 0.5; the coefficient associated with an average power of the empire's colonies ζ set to 0.05; $iRate$ set to 0.7 (70% probability of independence). Due to constrained computing resources, limited time and a large problem set, termination criteria of stagnation were implemented, where the search terminates if no improvement has been made to the best solution for ε number of iterations. For problem instances with $n < 500$, ε is set to $0.1n$, and for MKP instances with $n \geq 500$, $\varepsilon = n$.

MDVRP - the total number of countries N_{pop} is set to 4096 for all instances. Out of all countries, 40% are initialised as imperialists N_{imp}. Local iterations ϭ are set to 16. Assimilation rate θ set to 0.05; coefficient associated with an average power of empire's colonies ζ set to 0.05; i set to 0.7 (70% probability of independence). Finally, stagnation iterations ε set to 10.

Due to the stochastic nature of the algorithm, 30 independent runs were computed for each problem instance. Best and average solution performance, as well as the average time in seconds $t_{avg}(s)$ (average time in minutes $t_{avg}(m)$) required to reach such performance value, were recorded for all problem instances.

4.3.3. Comparison to classic ICA

Novel ICAwICA was first compared to classic ICA based on [125]. Three problem instances from both MKP (gk01, gk03, gk06) and MDVRP (p01, p03, p06) were selected for comparison, and the results are summarised in **Table 7**.

Table 7. *Comparison of best and average scores between Classic ICA and ICAwICA across six test problem instances. Average and best out of 10 runs with standard deviation (std), BKS – Best Known Solution.*

Dataset	Goal	BKS	Classic ICA			ICAwICA		
			Average	Best	Std	Average	Best	Std
MKP-gk01	Max	**3766**	3753.8	**3766**	8.11	**3766.0**	**3766**	0.00
MKP-gk03	Max	**5656**	5631.5	5638	5.12	5649.2	5650	0.90
MKP-gk06	Max	**7680**	7629.7	7639	8.16	7669.7	7671	1.19
MDVRP-p01	Min	**576.87**	587.20	580.70	8.92	576.87	**576.87**	0.00
MDVRP-p03	Min	**641.19**	658.10	645.16	7.55	655.29	**641.19**	3.25
MDVRP-p06	Min	**876.5**	893.80	885.84	10.83	887.71	**876.50**	3.93

Results show a significant improvement in the best scores obtained - ICAwICA reaching best-known solution (BKS) in four out of six instances, while classic ICA only once. Furthermore, average scores are consistently higher, and the standard deviation suggests that ICAwICA results are also more consistent. It is worth noting that MKP objective is to maximise profit, while MDVRP is to minimise the total route cost. Therefore, the average error gap (see equation (37)) against the best-known solution is used for easier comparisons and are summarised in **Figure 32**. The average error for ICAwICA is consistently smaller than classic ICA across all six test instances.

$$Average\ error\ gap\ (\%) = \frac{1}{n}\sum_{i=1}^{n}\frac{o_i - p_i}{o_i} * 100\% \qquad (37)$$

where o_i is the optimal score for the instance i, and p_i – achieved best or average score on the instance.

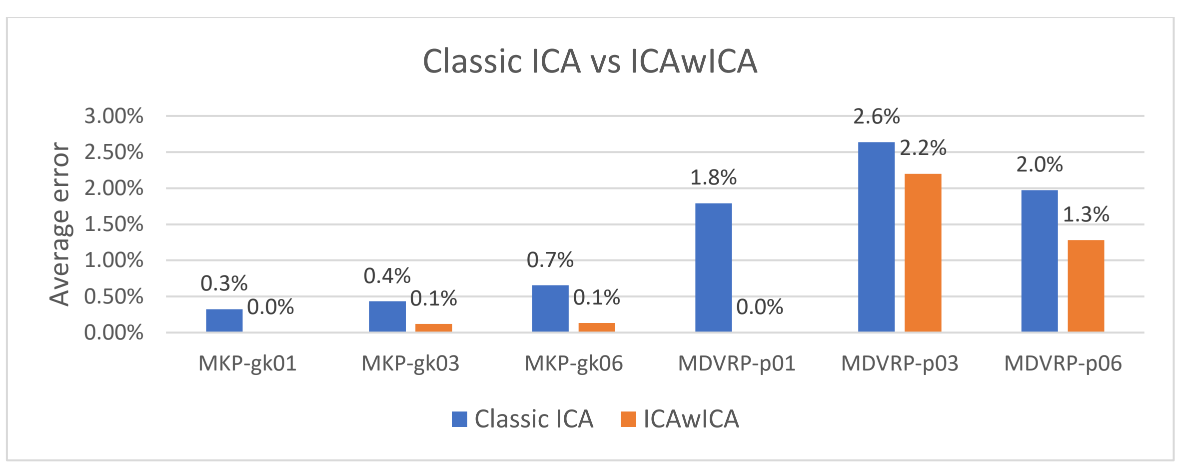

***Figure 32**. Comparison between Classic ICA [125] and ICAwICA for six test problem instances. Expressed as average error percentage to the best know solution. The graph demonstrates ICAwICA achieves average error of 0.62% while Classic ICA achieves 1.3%, relative improvement of over two times.*

4.3.4. Sensitivity analysis of independence rate

The newly implemented mechanism of colony independence was tested by altering the $independanceRate$ parameter from 0 to 1, with 0.2 increments and the average error gap (see equation (37)) as well as execution time $t_{avg}(s)$ recorded. The experimental results are summarised in **Table 8**.

***Table 8.** Sensitivity analysis of Independence rate as an average error per cent gap for six test problem instances. With 0 representing ICA with no independence operator, $t_{avg}(s)$ representing the average time in seconds to converge to the best solution, BKS – Best Known Solution*

			Independence rate					
Dataset	**Goal**	**BKS**	**0**	**0.2**	**0.4**	**0.6**	**0.8**	**1**
MKP-gk01	Max	3766	3.02%	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
MKP-gk03	Max	5656	2.75%	0.13%	**0.12%**	**0.12%**	**0.12%**	**0.12%**
MKP-gk06	Max	7680	2.36%	0.34%	0.20%	0.14%	**0.12%**	0.13%
MDVRP-p01	Min	576.87	4.79%	0.61%	0.04%	**0.00%**	**0.00%**	**0.00%**
MDVRP-p03	Min	641.19	10.98%	3.07%	2.67%	**2.12%**	**2.12%**	2.17%
MDVRP-p06	Min	876.5	7.79%	2.72%	1.53%	**1.25%**	1.36%	1.46%
		Average error	5.28%	1.14%	0.76%	**0.61%**	0.62%	0.65%
		$t_{avg}(s)$	40	570	836	1033	1315	1524

Results in **Table 8** show a definite improvement in the introduction of the Independence operator within ICA. Compared to ICA with no independence (independence rate of 0) and ICA with independence rate higher than 0, the average error across all test instances reduced by a factor of 4.6 (5.28% and 1.14% respectively). However, there is also a time penalty associated with doing the extra

work of assimilating to all imperialists compared to a single imperialist, with an average time to reach the final solution increasing from seconds to minutes. The best average error was achieved with the Independence rate between 0.6 and 0.8, and therefore independence rate at 0.7 was adopted for use throughout all further experiments.

4.3.5. Comparison to the state-of-the-art metaheuristics for MKP

To evaluate the proposed ICAwICA algorithm's performance, 12 state-of-the-art population-based/heuristic algorithms were compared across 41 common MKP instances.

First, a comparison was performed on simple WEISH instances, where most algorithms in the literature can achieve the optimum solution. Therefore, performance is measured in terms of the success rate (how many times the algorithm was able to achieve optimum) or in terms of the average error percentage error (see equation (37)) across all instances. For the comparison, the six best-performing algorithms were selected from the literature, which includes Ant Colony Optimization with Dynamic impact (ACOwD) described in [207], Improved Whale Optimization Algorithm (IWOA) [208], two variations of binary differential search TE-BDS and TR-BDS proposed in [209], and two implementations of Particle Swarm Optimization (PSO) with self-adaptive check and repair - SACRO-CBPSOTVAC and SACRO-BPSOTVAC [210].

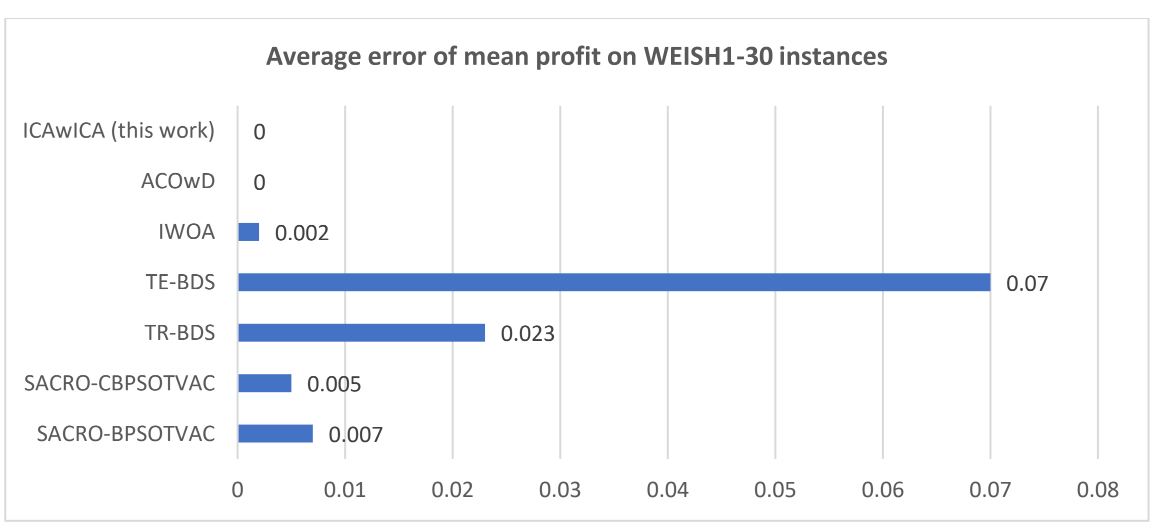

Figure 33. *The average error of the mean profit across all WEISH (1-30) instances. Average of 30 independent runs.*

Results in **Figure 33** show that all compared algorithms can reach the optimal solution in most cases. However, only 2 of them ICAwICA and ACOwD can do it

consistently across 30 runs with 100% success rate. ICAwICA achieved the optimal solution every time (100% success rate), at the first iteration, and on average took 1.5 seconds.

Next, large Glover and Kochenberger (GK) instances were solved and compared to eight heuristic algorithms from the literature in terms of average error per cent (see equation (37)) gap against best-known profit (BKS) from the literature. Compared algorithms include ACOwD, IWOA, Two-phase tabu-evolutionary algorithm (TPTEA) [211], harmony search based algorithm NBHS2 proposed in [212], an evolutionary algorithm with logic gates LGEA [213], shuffled complex evolution algorithm SCEcr [214], hyper-heuristic inspired CF-LAS [215] and BCSA – binary cuckoo search algorithm [216].

Table 9. *Algorithm comparison across large Glover and Kochenberger (GK) knapsack instances. Results are expressed as average error percentage gap % against best-known profit. Colour coded from the best gap (green) to worst gap (red) for any given dataset. With dash (-) representing results that are not available. BKS – Best Known Solution, Std – Standard Deviation of the absolute value.*

Data set	Problem size (n x m)	BKS	ACOwD [207]	NBHS2 [212]	IWOA [208]	LGEA [213]	TPTEA [211]	SCEcr [214]	CF-LAS [215]	BCSA [216]	ICAwICA (this work)			
											Average	Best	Std	$t_{avg}(s)$
gk01	100x15	3766	0.14%	0.29%	0.68%	0.66%	0.00%	0.76%	0.31%	0.23%	0.00%	0.00%	0.00	16.8
gk02	100x25	3958	0.05%	0.30%	-	0.55%	0.00%	1.06%	0.36%	0.27%	0.05%	0.03%	0.99	19.4
gk03	150x25	5656	0.26%	0.55%	0.85%	0.97%	0.06%	0.91%	0.37%	0.17%	0.12%	0.11%	0.90	62.5
gk04	150x50	5767	0.17%	0.45%	0.89%	1.02%	0.01%	1.48%	0.45%	0.15%	0.07%	0.05%	0.93	84.4
gk05	200x25	7561	0.21%	0.44%	0.94%	1.32%	0.01%	0.73%	0.24%	0.18%	0.09%	0.04%	1.54	145.7
gk06	200x50	7680	0.26%	0.52%	0.77%	1.05%	0.08%	1.14%	0.46%	3.54%	0.13%	0.12%	1.19	247.7
gk07	500x25	19221	0.20%	0.26%	1.09%	1.08%	0.04%	0.46%	0.13%	0.70%	0.11%	0.07%	5.89	280.3
gk08	500x50	18806	0.22%	0.56%	0.85%	-	0.06%	0.67%	0.20%	0.77%	0.12%	0.08%	2.98	357.8
gk09	1500x25	58091	0.18%	0.27%	1.54%	1.08%	0.02%	1.78%	1.77%	0.98%	0.14%	0.09%	14.61	1611.0
gk10	1500x50	57295	0.20%	0.54%	0.80%	1.01%	0.04%	0.36%	0.10%	-	0.18%	0.12%	13.67	2219.1
gk11	2500x100	95238	0.32%	0.64%	1.07%	1.13%	0.07%	0.30%	0.09%	-	0.31%	0.24%	61.54	7200.6

Table 9 is colour coded from red (worst average error %) to the best average error per cent, in green, for each problem instance with dashes (-) representing scores that were not available. Compared to 8 other algorithms in the literature, ICAwICA shows competitive results, coming in second place for gk01-gk09 and in top three for gk10 and in fourth place for the largest gk11 instance. The best achieved error percentage along with the average time $t_{avg}(s)$ and standard deviation (Std) have been included for reference. The proposed algorithm performs well on medium to large MKP instances, however, struggles on very large instances (gk11). Further investigation needs to be conducted to improve performance on the most complex benchmarks.

4.3.6. Comparison to the state-of-the-art metaheuristics for MDVRP

The ICAwICA algorithm was next evaluated for the MDVRP compared to other state-of-the-art approaches. Although many algorithms have been applied to the MDVRP, the most recent literature techniques were selected and summarised in **Table 10**. A cooperative coevolutionary algorithm called CoES [217], Improved Ant Colony Optimization (IACO) [180], Tabu Search Heuristic (TSH) in [218], as well as hybrid Ant Colony with simulated annealing and local search algorithm called ACO+ [172] were selected for the comparison. The ICAwICA algorithm was also compared to the best-known solutions (BKS) in [206]; it is worth mentioning that these solutions are outdated as better results are reported in the literature. Nevertheless, the best-known solutions of [206] are included for reference.

Compared with other algorithms in **Table 10**, ICAwICA obtained the same best score in 11 out of 23 instances and outperformed the four rival algorithms on p08 instance. On average error percentage in respect to BKS, ICAwICA fell short compared to ACO+ (0.13% vs 0.28% error), however, outperformed other compared approaches.

***Table 10**. Best solution obtained by ICAwICA compared to other algorithms in the literature across Cordeau's MDVRP benchmark instances and the best-known solution (BKS). The best scores represented in bold, N representing the number of customers, M – the number of depots. Average error percentage calculated using BKS as a reference, $t_{avg}(m)$ – average time to converge to a solution, in minutes, Std – Standard Deviation*

Data set	N	M	BKS	CoES, 2016 [217]	IACO, 2017 [180]	TSH, 2019 [218]	ACO+, 2020 [172]	ICAwICA (this work) Best	Average	Std	$t_{avg}(m)$
p01	50	4	**576.87**	**576.87**	**576.87**	**576.87**	**576.87**	**576.87**	576.87	0.00	4.2
p02	50	4	**473.53**	**473.87**	**473.53**	**473.53**	**473.53**	**473.53**	481.24	3.00	6.2
p03	75	5	**641.19**	**641.19**	**641.19**	**641.19**	**641.19**	**641.19**	655.29	3.25	7.9
p04	100	2	1001.59	1007.40	**1001.49**	1008.47	1003.52	1006.66	1015.11	3.97	12.4
p05	100	2	**750.03**	**750.11**	750.26	758.87	751.90	753.40	789.15	4.39	20.3
p06	100	3	**876.50**	**876.50**	**876.50**	881.76	881.60	**876.50**	887.71	3.93	14.7
p07	100	4	885.80	888.41	885.69	896.96	**884.66**	895.53	916.79	8.12	11.5
p08	249	2	**4420.94**	4445.37	4482.44	4430.36	4428.00	**4420.94**	4493.66	17.87	65.2
p09	249	3	3900.22	3895.70	3912.23	3971.59	**3897.33**	3900.22	3975.29	23.52	67.6
p10	249	4	3663.02	3666.35	3663.00	3779.10	**3657.03**	3666.35	3696.71	10.88	82.2
p11	249	5	3554.18	3569.68	3648.95	3652.01	**3549.99**	3554.18	3604.88	22.75	71.0
p12	80	2	**1318.95**	**1318.95**	**1318.95**	**1318.95**	**1318.95**	**1318.95**	1359.49	4.88	10.0
p13	80	2	**1318.95**	**1318.95**	**1318.95**	**1318.95**	**1318.95**	**1318.95**	1320.79	0.82	8.9
p14	80	2	**1360.12**	**1360.12**	1365.68	1365.69	**1360.12**	1365.68	1394.01	6.71	6.7
p15	160	4	**2505.42**	2526.06	2505.29	2552.79	**2505.42**	2565.67	2644.14	6.13	25.5
p16	160	4	**2572.23**	**2572.23**	2587.87	**2572.23**	**2572.23**	**2572.23**	2577.66	1.6	16.0
p17	160	4	2709.09	2709.09	**2708.99**	2731.37	2709.09	**2709.09**	2742.93	5.51	12.3
p18	240	6	**3702.85**	3771.35	3781.04	3802.29	3710.49	3710.49	3756.70	20.83	73.2
p19	240	6	**3827.06**	**3827.06**	**3827.06**	3831.71	**3827.06**	**3827.06**	3857.36	5.21	42.3
p20	240	6	**4058.07**	**4058.07**	**4058.07**	4097.06	4091.78	**4058.07**	4134.88	21.06	73.6
p21	360	9	**5474.84**	5608.26	**5474.84**	5617.53	5505.39	5495.54	5564.61	24.90	81.9
p22	360	9	**5702.16**	5702.16	**5702.06**	5706.81	5702.16	5702.16	5753.71	25.14	86.0
p23	360	9	**6095.46**	6129.99	**6095.46**	6145.58	6140.53	6145.58	6205.46	24.05	83.7
			Average error gap	0.33%	0.33%	0.96%	0.13%	0.28%			

4.4. Summary

This chapter proposed a novel generic Imperialist Competitive Algorithm (ICA) based algorithm for solving constrained combinatorial problems called ICA with Independence and Constrained Assimilation (ICAwICA). The algorithm implements a new Independence operator for ICA, where each of the colonies has a free will to choose between assimilating to its imperialist or any other imperialist in the population. Additionally, a generic constrained assimilation process is proposed as part of the local search. The constrained assimilation exploits the fact that two solutions cannot be merged without violating constraints. Furthermore, it combines the classic ICA assimilation and revolution operators in one, in a generic manner.

To evaluate the performance and versatility of the ICAwICA algorithm, two different kinds of combinatorial benchmark problems were selected – subset selection and routing, Multiple Knapsack Problem (MKP) and Multiple Depot Vehicle Routing

Problem (MDVRP), respectively. First, the ICAwICA was compared to classic ICA, and results showed a definite improvement in all benchmark test instances. Next, the sensitivity of the Independence operator was (evaluated?). Analysis shows that independence probability of greater than zero improves the results at the expense of computing time. Finally, the ICAwICA was compared to the current state-of-the-art population-based algorithms for both MKP and MDVRP. The proposed algorithm outperformed the majority of the competition on both types of problems across multiple instances, indicating the generic, universal nature of the ICAwICA within the OptPlatform.

Generic metaheuristic support is an important aspect of any MOF, as it allows the user to focus on the problem specifics and modelling and avoid spending time understanding the suitability of the underlying metaheuristics algorithms. Instead, the most suitable and efficient algorithm selection is performed in the background automatically. An essential aspect of metaheuristic efficiency is how well they are utilising the available computing resources. This is the focus of the next chapter, where the efficiency of ACO is evaluated across multiple hardware platforms.

5. ACCELERATING SUPPLY CHAINS WITH ANT COLONY OPTIMIZATION ACROSS A RANGE OF HARDWARE SOLUTIONS

This chapter is based on the results published in [4].

As discussed in section 2.2.1, parallelism and scaling of metaheuristics are important aspects of any MOF. This is especially true for large real-world models that are computationally intensive. This chapter explores how Ant Colony Algorithm scales within the platform for solving global supply chain (described in section 2.3.2.2) and compares the dynamics to a simpler benchmark problem.

Ant Colony algorithm has been applied to various optimisation problems; however, most of the previous work on scaling and parallelism focuses on Travelling Salesman Problems (TSPs). Although useful for benchmarks and new idea comparison, the algorithmic dynamics do not always transfer to complex real-life problems, where additional meta-data is required during solution construction. This chapter explores how the benchmark performance differs from real-world problems in the context of Ant Colony Optimization (ACO) and demonstrates that in order to generalise the findings, the algorithms have to be tested on both standard benchmarks and real-world applications.

The chapter starts by analysis of the various hardware architectures and the related work in the domain of ACO scaling. Next, a brief overview of the technology used is provided in section 5.2. The two parallel ACO architectures – Independent Ant Colonies (IAC) and Parallel Ants (PA) are described in section 5.3 and an in-depth empirical study provided in section 5.4.

5.1. Motivation and related work

Supply chain optimisation has become an integral part of any global company with a complex manufacturing and distribution network. For many companies, inefficient distribution plan can make a significant difference to the bottom line. Modelling a

complete distribution network from the initial materials to the delivery to the customer is very computationally intensive. With increasing supply chain modelling complexity in ever-changing global geo-political environment, fast adaptability is an edge. A company can model the impact of currency exchange rate changes, import tax policy reforms, oil price fluctuations and political events such as Brexit, Covid-19 before they happen. Such modelling requires fast optimisation algorithms.

Mixed Integer Linear Programming (MILP) tools such as Cplex are commonly used to optimise various supply chain networks [219]. Although MILP tools can obtain an optimum solution for many linear models, not all real-world supply chain models are linear. Furthermore, MILP is computationally expensive and on large instances can fail to produce an optimal solution. For that reason, many alternative algorithmic approaches (heuristics, meta-heuristics, fuzzy methods) have been explored to solve large-complex SC models [219]. One of these algorithms is the Ant Colony Optimization (ACO), which can be well mapped to real-world problems such as routing [220] and scheduling [221]. Supply Chain Optimization Problem (SCOP) includes both, finding the best route to ship a specific order and finding the most optimal time to ship it, such that it reaches expected customer satisfaction while minimising the total cost occurred. Although other metaheuristics algorithms exist in the literature for solving SCOPs, such as Genetic Algorithm (GA) [222][223] and Simulated Annealing (SA) [224][225], ACO was chosen due to the long history of the algorithm applied to various vehicle routing [226][227] and supply chain [228][229] problems with great solution quality and speed. Also, a recent study in [230] concluded that compared to GA and SA, the ACO performs the best for routing problems such as the Travelling Salesman Problem (TSP).

Researchers in [231] compared an industrial optimisation-based tool – IBM ILOG Cplex with their proposed ACO algorithm. It was concluded that the proposed algorithm covered 94% of optimal solutions on small problems and 88% for large-size problems while consuming significantly less computation time. Similarly, [232] compared ACO and Cplex performance on multi-product and multi-period Inventory Routing Problem. On small instances, ACO reached 95% of the optimal solution while on large instances performed better than time-constrained Cplex solver. Furthermore, ACO implementations of Closed-Loop Supply Chain (CLSC) have been proposed; CLSC contains two parts of the supply chain – forward supply and reverse/return. [233] solved CLSC models, where the ACO implementation outperformed commercial MILP

(Cplex) on nonlinear instances and obtained 98% optimal solution with 40% less computation time on linear instances.

Academic literature suggests that Graphical Processing Units (GPUs) are very suitable for solving benchmark routing problems such as Travelling Salesman Problem (TSP), with speedups of up to 60x [234] and even 172x [235] when compared to the sequential CPU implementation. This chapter aims to explore if the same ACO architectures that are so well suited for TSP can be applied for a real-world supply chain optimisation problem. Furthermore, investigate what hardware architectures are the best suited for the supply chain problem solved.

5.1.1. Parallel Ant Colony Optimization

Since the introduction of ACO in 1992, numerous ACO algorithms have been applied to many different problems, and many different parallel architectures have been explored previously. [236] specifies 5 of such architectures:

- Parallel Independent Ant Colonies – each ant colony develop their solutions in parallel without any communication in-between;
- Parallel Interacting Ant Colonies – each colony creates a solution in parallel and some information is shared between the colonies;
- Parallel Ants – each ant builds solution independently, then all the resulting pheromones are shared for the next iteration;
- Parallel Evaluation of Solution Elements – for problems where fitness function calculations take considerably more time than the solution creation;
- Parallel Combination of Ants and Evaluation of Solution Elements – a combination of any of the above.

Researchers have tried to exploit the parallelism offered from recent multi-core CPUs [237], along with clusters of CPUs ([238][239]) and most recently GPUs [240] and Intel's many-core architectures such as Xeon Phi [241]. Breakdown of the strategies and problems solved are shown in **Table 11**.

***Table 11**. ACO architecture and hardware configurations explored. LAC - Longest Common Subsequence Problem, MKP - Multidimensional Knapsack Problem, TSP - Travelling Salesman problem. IAC – Independent Ant Colonies, IntAC – Interactive Ant Colonies, PA – Parallel Ants.*

Platform	Task parallelism, IAC	Task parallelism, IntAC	Task parallelism, PA	Data parallelism, PA
CPU	Scheduling [242]	Scheduling [242]	TSP [243] [244] Scheduling [242] **Supply chain [this work]**	TSP [245] **Supply chain [this work]**
GPU	n/a	n/a	Protein folding [246] TSP [243] MKP [247] LAC [248]	TSP [245][249][250] Edge detection [251] **Supply chain [this work]**
CPU cluster	Scheduling [252]	TSP [236]	TSP [239]	n/a
Xeon Phi	n/a	n/a	**Supply chain [this work]**	TSP [253] [254] [255] **Supply chain [this work]**

During the search, an Ant has to keep track of the existing state meta-data, for instance Travelling Salesman Problem only need to keep the record of what cities have been visited as part of problem constraint. However, real-life problems have many more constraints and therefore require a lot of meta-data storage during solution creation. This chapter explores such a problem in the supply chain domain. **Table 12** shows the most common problems solved by ACO and their corresponding associated constraints / meta-data required during solution creation.

***Table 12.** Meta-data required during solution creation based on problem type*

Problem	Meta-data required during solution creation	Comment
Scheduling	2	Resource and precedence constraints
TSP	1	Has the city been visited
Protein Folding	1	Has the sequence been visited
MKP	1	Total weight per knapsack
LAC	1	Tracking of the current position in a string
Edge detection	1	Has edge already been visited
Supply chain (this work)	**3**	**Capacity, daily order, freight weight constraints**

5.1.2. CPU

Parallel ACO CPU architectures have been applied to various tasks – for example, [242] applied ACO for supply chain scheduling problem in mining domain. Authors managed to reduce the execution time from one hour (serial) to around 7 minutes. Both [256] and [257] used ACO for image edge detection with varying results, [256] achieved a speedup of 3-5 times while [257] managed to reduce sequential runtime by 30%. Most commonly, ACO has been applied to the Travelling Salesman Problem (TSP) benchmarks. For instance, [244] proposed an ACO approach with randomly synchronised ants; the strategy showed a faster convergence than other TSP approaches. Moreover, authors in [245] proposed a new multi-core Single Instruction Multiple Data (SIMD) model for solving TSPs. Similarly, both [258] and [259] tries to solve large instances of TSP (up to 200k and 20k cities, respectively) where the architectures are limited to the size of the pheromone matrix. [259] discusses such limitations and proposes a new pheromone sharing for local search – effective heuristics ACO (ESACO), which was able to compute TSP instances of 20k. In contrast, authors in [258] eliminate the need for pheromone matrix and store only the best solutions similar to the Population ACO. Furthermore, researchers implement a Partial Ant, also known as the cunning ant, where ant takes an existing partial solution and builds on top of it. Speedups of as much as 1200x are achieved compared to sequential Population ACO.

Generally, CPU parallel architecture implementations come down to three programming approaches - Message Passing Interface (MPI) parallelism, OpenMP parallelism [260] and data parallelism with the vectorisation of SIMD. For instance, [261] explored both master-slave and coarse-grained strategies for ACO parallelisation using MPI. It was concluded that fine-grained master-slave policy performed the best. [262] used MPI with ACO to accelerate Maximum Weight Clique Problem (MWCP). The proposed algorithm was comparable to the ones in literature and outperformed Cplex solver in both – time and performance. Moreover, authors in [252] implemented parallel ACO for solving Flow shop scheduling problem with restrictions using MPI. Compared to the sequential version of the algorithm, 93 node cluster achieved a speedup of 16x. [263] compared ACO parallel implementation on MPI and OpenMP on small vector estimation problem. It was found that maximum speedup of OpenMP was 24x while MPI – 16x. Furthermore, [245] explored the multi-

core SIMD CPU with OpenCL and compared it to the performance of the GPU. It was found that optimised parallel CPU-SIMD version can achieve similar solution quality and computation time than the state of art GPU implementation solving TSP.

5.1.3. Xeon Phi

Intel's Xeon Phi Many Integrated Core (MIC) architecture offers many cores on the CPU (60-72 cores per node) while offering lower clock frequency. Few researchers have had the opportunity to research ACO on the Xeon Phi architecture. For instance, [253] showed how utilising L1 and L2 cache on Xeon Phi coprocessor allowed a speedup of 42x solving TSP compared to sequential execution. Due to the nature of SIMD features such as AVX-512 on Xeon Phi, researchers in both [254] and [255] proposed a vectorisation model for roulette wheel selection in TSP. In the case of [255], a 16.6x speedup was achieved compared to sequential execution. To the best of the author's knowledge, Xeon Phi and ACO parallelism have not been explored to any other problem except TSP.

5.1.4. GPUs

General Purpose GPU (GPGPU) programming is a growing field in computer science and machine learning. Many researchers have tried exploiting latest GPU architectures to speed optimise the convergence of ACO. ACO GPU implementation expands to many fields, such as edge detection ([251][264]), protein folding [246], solving Multidimensional Knapsack Problems (MKPs) [247] and Vertex colouring problems [265]. Moreover, researchers have used GPU implementations of ACO for classification ([266] [267]) and scheduling ([268][269]) with various speedups compared to the sequential execution.

However, the majority of publications are solving Travelling Salesman Problems [270], although useful for benchmarking and comparison, little characteristics transfer to other application areas. For instance, highly optimised local memory on GPU (Compute Unified Device Architecture - CUDA) can significantly speed up TSP's execution. However, when applied to real-life problems where additional restrictions and metadata is required to build a solution, most of the data needs to be stored on much slower global memory. In [244], the authors did extensive research comparing server, desktop and laptop hardware solving TSP instances on both CUDA and

OpenCL. Although there are a couple of ACO OpenCL implementations on GPU ([248][271]), the majority of studies use CUDA. For instance, [272] implemented a GPU-based ACO and achieved a speedup of 40x compared to sequential ACS. Similarly, a 22x speedup was obtained in [273] solving pr1002 TSP and 44x on fnl4461 TSP instance in [274]. However, there are also various hybrid approaches for solving TSP - [275] uses parallel Cultural ACO (pCACO) (a hybrid of genetic algorithm and ACO). Research showed that pCACO outperformed sequential and parallel ACO implementations in terms of solution quality. Furthermore, [276] solved TSP instances using ACO-PSO hybrid and authors in [277] explored heterogeneous computing with multiple GPU architectures for TSP. Finally, authors in [250] explored six different min-max ACO architectures on GPU and their TSP performance.

Although task parallelism has potential for a speedup, [278] showed how data parallelism (vectorisation) on GPU could achieve better performance by proposed Independent Roulette wheel (I-Roulette). Same authors then expanded the I-Roulette implementation in [249], where SS-Roulette wheel was introduced. SS-Roulette stands for Scan and Stencil Roulette wheel. It mimics a sequential roulette wheel while allowing higher throughput due to parallelism. First, the Tabu list is multiplied by the probabilities and the results stored in a choice vector (*scan*). A stencil pattern is then applied to the choice vector based on a random number to select an individual (*stencil*). Further, [235] implements a G-Roulette – a grouped roulette wheel selection based on I-Roulette, where cities in TSP selection are grouped in CUDA warps[22]. An impressive speedup of 172x was achieved compared to the sequential counterpart.

5.1.5. Comparing hardware performances

Fairly comparing parallel performances of different hardware architectures is by no means trivial. Most research compares a sequential CPU ACO implementation to one of the parallel GPUs, which is hardly fair [279]. Also, unoptimized sequential code is compared to highly optimised GPU code. Such comparisons result in misleading and inflated speedups [240]. Furthermore, [248] argues that the parameter settings chosen for the sequential implementation are often biased in favour of GPU. [240] proposes criteria to calculate the real-world efficiency of two different hardware architectures by

[22] Groups of 32 threads, are known as CUDA warps. For information refer to: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/

comparing the theoretical peak performances of GPU and CPU. While the proposed method is more appropriate, it still does not account for real-life scenarios where memory latency/speed, cache size, compilers and operating systems all play a role of the final execution time. Therefore, two different systems with similar theoretical floating-point operations per second running the same executable can have significantly different execution times.

Furthermore, in some instances, only execution time or solution quality is compared, rarely both are considered when analysing results.

5.2. Background

This section briefly covers the tools and hardware-specific languages used in the implementation.

5.2.1. Parallel processing with OpenMP

OpenMP[23] is a set of directives to a compiler that allows a programmer to create parallel tasks as well as vectorisation (Single Instruction Multiple Data - SIMD) to speed up execution of a program. A program containing parallel OpenMP directives starts as a single thread. Once directive such as *#pragma omp parallel* is reached, the main thread will create a thread pool and all methods within the *#pragma* region will be executed in parallel by each thread in the thread group. Moreover, once the thread reaches the end of the region, it will wait for all other threads to finish before dissolving the thread group and only the main thread will continue.

Furthermore, OpenMP also supports nesting, meaning a thread in a thread-group can create its own individual thread-group and become the master thread for the newly created thread-group. However, thread-group creation and elimination can have significant overhead and therefore, thread-group re-use is highly recommended [280]. Both *omp parallel* and *omp simd* directives are used in this study.

5.2.2. CUDA programming model

Compute Unified Device Architecture (CUDA) is a General-purpose computing model on GPU developed by Nvidia in 2006. Since then, this proprietary framework

[23] OpenMP API website and documentation https://www.openmp.org/

has been utilised in the high-performance computing space via multiple Artificial Intelligence (AI) and Machine Learning (ML) interfaces and libraries/APIs. CUDA allows writing C programs that take advantage of any recent Nvidia GPU found in laptops, workstations and data centres.

Each GPU contains multiple Streaming Multiprocessors (SM) that are designed to execute hundreds of threads concurrently. To achieve that, CUDA implements SIMT (Single Instruction Multiple-Threads) architecture, where instructions are pipelined for instruction-level parallelism. Threads are grouped in sets of 32 – called *warps*. Each warp executes one instruction at a time on each thread. Furthermore, CUDA threads can access multiple memory spaces – global memory (large size, slower), texture memory (read only), shared memory (shared across threads in the same SM, lower latency) and local memory (limited set of registers within each thread, fastest)[24].

A batch of threads is grouped into a *thread-block*. Multiple thread-blocks create a *grid of thread blocks*. The programmer specifies the grid dimensionality at kernel launch time, by providing the number of thread-blocks and the number of threads per thread-block. Kernel launch fails if the program exceeds the hardware resource boundaries.

5.2.3. Xeon Phi Knights Landing architecture

Knights Landing is a product code name for Intel's second-generation Intel Xeon Phi processors. First-generation of Xeon Phi, named Knights Corner, was a PCI-e coprocessor card based on many Intel Atom processor cores and support for Vector Processing Units (VPUs). The main advancement over Knights Corner was the standalone processor that can boot stock operating systems, along with improved power efficiency and vector performance. Furthermore, it also introduced a new high bandwidth Multi-Channel DRAM (MCDRAM) memory. Xeon phi support for standard x86 and x86-64 instructions, allows majority CPU compiled binaries to run without any modification. Moreover, support for 512-bit Advanced Vector Extensions (AVX-512) allows high throughput vector manipulations.

[24] CUDA documentation https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

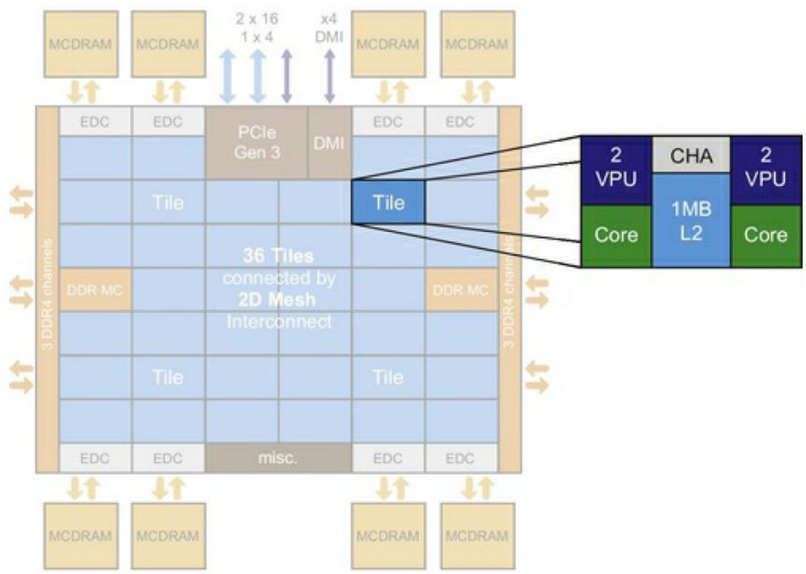

Figure 34. *Knights Landing tile with a larger processor die [281]*

The Knights Landing cores are divided into tiles (typically between 32 and 36 tiles in total). Each tile contains two processor cores and each core is connected to two vector processing units (VPUs). Utilising AVX-512 and two VPUs, each core can deliver 32 dual-precision (DP) or 64 single-precision (SP) operations each cycle [281]. Furthermore, each core supports up to four threads of execution – hyper threads where instructions are pipelined.

Another introduction with the Knights Landing is the cluster modes and MCDRAM/DRAM management. The processor offers three primary cluster modes[25] – All to all mode, Quadrant mode and Sub-Numa Cluster (SNC) mode and three memory modes – cache mode, flat mode and hybrid mode. For a detailed description of the Knights Landing Xeon Phi architecture refer to [281].

5.3. Methods and implementation

To solve the transportation network optimisation problem, an Ant Colony System algorithm (first proposed by [106]) has been implemented. Because ACO is an iterative algorithm, it does require sequential execution. Therefore, the most naïve approach for parallel ACO is running multiple Independent Ant Colonies (IAC) with a unique seed for the pseudo-random number generator for each colony (high-level pseudocode in

[25] Detailed description of Xeon Phi memory and cache modes available at: https://software.intel.com/en-us/articles/intel-xeon-phi-x200-processor-memory-modes-and-cluster-modes-configuration-and-use-cases

Figure 35). Due to the stochastic nature of solution creation, it is, therefore, more probabilistic to reach a better solution than a single colony. This approach has the advantage of low overhead as it requires no synchronisation between the parallel instances during the search. At the very end of the search, the best solution of all parallel colonies is chosen as the final solution. The main disadvantage of IAC is that if one of the colonies finds a better solution, there is no way to improve all the other colony's fitness values.

Independent Ant Colonies (IAC)

1. **for** all parallel instances *m* **parallel *do***
2.   **for** all iterations *iter* **do**
3.   **for** all local ants *a* **do**
4.   local pheromone = global pheromone
5.   construct solution
6.   local pheromone update
7.   **end for**
8.   update global pheromone update based on the best solution
9.   **end for**
10. **end for**
11. find the best solution across parallel instances

Figure 35. *High-level pseudocode for Independent Ant Colonies (IAC) search algorithm*

Alternatively, the ACO search algorithm could also be letting the artificial ant colonies synchronise after every iteration. Therefore, all parallel instances are aware of the best solution and can share pheromones accordingly. High-level pseudocode of such Parallel Ant (PA) implementation is shown in **Figure 36**. The main advantage of this architecture is that it allows efficient pheromone sharing, therefore converging faster. However, there is a high risk of getting stuck into local optima as all ants start iteration with the same pheromone matrix. Furthermore, synchronisation of all parallel instances after every iteration is costly.

Parallel Ants (PA)

```
1.  for all iterations iter do
2.      for all parallel instances m parallel do
3.          for all local ants a do
4.              local pheromone = global pheromone
5.              construct solution
6.              local pheromone update
7.          end for
8.      end for
9.      find the best solution across parallel instances
10.     update global pheromone update based on the best solution
11. end for
```

Figure 36. *High-level pseudocode Parallel Ants (PA) search algorithm*

Both IAC and PA implementations are exploiting task parallelism – each parallel instance (thread) gets a set of tasks to complete. An alternative approach would be to look at data parallelism and vectorisation. In such a strategy, each thread processes a specific section of the data and cooperatively complete the given task. Due to the highly sequential parts of ACO, it would not be practical to only use vectorisation alone. A more desirable path would be to implement vectorisation in conjugate to the task parallelism. In case of CPU, task parallelism can be done by the threads, while vectorisation is done by Vector Processing Units (VPUs) based on Advanced Vector Extensions 2 (AVX2) or AVX512. Moreover, in the case of GPU and CUDA – task parallelism would be done at a thread-block level while data parallelism would exploit WARP structures. Parallel Ants with Vectorisation (PAwV) expands on the Parallel Ants architecture by introducing data-parallelism of solution creation and an alternative roulette wheel implementation – SS-Roulette, first proposed in [282]. Local search in **Figure 37** expands on the implementation in **Figure 36** (lines 3-7). First, the *choiceMatrix* is calculated by multiplying the probability of the route to be chosen with the t*abuList* – a list of still available routes (where 0 represents not available and 1 – route still can be selected). A random number between 0 and 1 is generated to determine if a given route will be chosen based on exploitation or exploration. In the case of exploitation, the *choiceMatrix* is reduced to obtain the maximum and the corresponding route index. Furthermore, in the case of exploration, the route is chosen based on the SS-Roulette wheel described by [282].

Parallel Ants with Vectorization (PAwV)

```
1.  for all local ants a do
2.      local pheromone = global pheromone
3.      for all orders o do
4.          for all routes r for order do SIMD
5.              choiceMatrix[r] = probability[r] * tabuList[r]
6.          end for
7.          if rand() <= q0 then
8.              SIMD reduce max (choiceMatrix)
9.          else
10.             SS-Roulette wheel [282]
11.         end if
12.     end for
13.     local pheromone update
14. end for
```

Figure 37. *High-level pseudocode for Parallel Ants with Vectorization (PAwV) search algorithm. Expanding on **Figure 36**' lines 3-7.*

The main advantage of IAC is that it requires to synchronise between threads only at the start of the search and at the very end of the search, therefore keeping synchronisation overhead low. However, as there is no pheromone sharing, new better solutions cannot be shared across the parallel instances. In contrast, both PA and PAwV offers sharing of the best performing ants' pheromone before the next iteration begins. The potential drawback is that search might get stuck in local optimum as all parallel instances share the same pheromone starting point. Furthermore, pheromone sharing and therefore, synchronisation between threads is costly overhead, especially if performed after each iteration. The PAwV architecture exploits the use of SIMD instructions for further data parallelism inside the Ant's solution construction. **Table 13** summarises these architectural features.

Table 13. *Comparison of Independent Ant Colonies (IAC), Parallel Ants (PA) and parallel Ants with Vectorisation (PAwV) architectures.*

	IAC	PA	PAwV
Synchronisation between threads during search	No	Yes	Yes
Pheromone sharing between parallel instances	No	Yes	Yes
Data parallelism	No	No	Yes

5.4. Experiments

A sequential implementation of ACO described in [106] is adapted from [283] by altering the heuristic information calculation for a given route – defined as a proportion of order's weight and the maximum weight gap (see equation (5)). Furthermore, the ACO set of parameters were obtained from both work in [283] and empirical experimentation. **Table 14** summarises these algorithm hyperparameters. Moreover, three different Parallel ACO architectures were implemented – Independent Ant Colonies (IAC), Parallel Ants (PA) and Parallel Ants with Vectorisation (PAwV) in C++ and CUDA C.

Experiments were conducted on three different hardware configurations – CPU, GPU and Xeon Phi.

Table 14. *Ant Colony System set of parameters for all configurations and architectures*

Parameter	Value
Pheromone evaporation rate (rho)	0.1
Weight on pheromone information (α)	1
Weight on heuristic information (β)	8
Exploitation to exploration ratio (q0)	0.9

<u>Hardware A - *CPU*</u>

- CPU: AMD Ryzen™ Threadripper™ 1950X (16 cores, 32 threads), running at 3.85GHz.
- RAM: 64GB 2400MHz DDR4, 4 channels.
- OS: Windows 10 Pro, version 1703
- Toolchain: Intel C++ 18.0 toolset, Windows SDK version 8.1, x64

<u>Hardware B - *Xeon Phi*</u>

- CPU: Intel® Xeon Phi™ Processor 7250F (68 cores, 272 hyper-threads), running at 1.4GHz. Clustering mode set to *Quadrant* and memory mode set to *Cache mode*.
- RAM: 16GB on-chip MCDRAM and 96GB 2400MHz DDR4 ECC.
- OS: Windows Server 2016, version 1607
- Toolchain: Intel C++ 18.0 toolset, Windows SDK version 8.1, x64, KMP_AFFINITY=scatter

<u>Hardware C - *GPU*</u>

- CPU/RAM/OS – see host Hardware A.
- GPUs: 4x Nvidia GTX1070, 8GB GDDR5 per GPU, 1.9GHz core, 4.1GHz memory. PCIe with 16x/8x/16x/8x.

- Toolchain: Visual Studio v140 toolset, Windows SDK version 8.1, x64, CUDA 9.0, compute_35, sm_35

Hardware architecture C shares the same host CPU as Hardware A.

5.4.1. Benchmarks

It is crucial to consider both elapsed time and solution quality when referring to speed optimisation of optimisation algorithms. One could get superior convergence within iteration but, take twice as long to compute. Similarly, one could claim that the algorithm is much faster at completing a defined number of iterations but sacrifice solution quality. Furthermore, there is little point comparing sequential execution of one hardware platform to parallel implementation of another. A comparison should take into consideration all platform strengths and weaknesses and set up the most suitable configuration for a given platform.

To obtain a baseline fitness convergence rate at a various number of parallel instances, a matrix of Iterations vs Parallel Instances are created for all architectures. An example of such matrix for Parallel Ants is shown in **Table 15**. The matrix is derived by averaging the resulting fitness obtained from 10 independent simulations with a unique seed value for each given Parallel Instances configuration. All configurations are run for *x* number of iterations, where *x* is based on the total number of solutions explored and is a function of the number of Parallel Instances. The total number of solutions explored is set to 768k. The number of Parallel Instances is varied by 2^{n-1} with maximum n of 11, i.e. 1024 parallel instances. The best value after every 5 iterations is also recorded.

The number of iterations required to reach a specific solution quality for different ACO architectures are computed in **Table 16**, expressed as proximity to the best-known optimal solution. For the particular problem and dataset, the best solution is the total cost of 2,701,367.58. There are six checkpoints of solution quality ranging from 99% to 99.9%. Although at first 1% gain might not seem significant, one must remember that global supply chain costs are measured in hundreds of millions, and even 1% savings do affect the bottom line. Empty fields (-) represent instances where the ACO was not able to converge to given solution quality.

On all experiments, IAC was able to obtain solution quality only below 99.6%. In contrast, PA and PA with 5 ant local search were able to achieve above 99.9% solution

quality with 512 and 1024 parallel instances. Furthermore, IAC did not see any significant benefit of adding more parallel instances for 99% and 99.25% checkpoints.

***Table 15**. Parallel Ants fitness value baseline for different configurations of the number of parallel instances and the number of iterations. Each Parallel Instance data point is an average of 10 individual runs (table derived from 11*10 =110 runs). Expressed as a percentage of the proximity of the best-known solution (2,701,367.58). Colour-coded from worse – in red, to the best – in green.*

Baseline for Parallel Ants (PA)

The number of Iterations	The number of Parallel Instances										
	1	2	4	8	16	32	64	128	256	512	1024
5	98.646%	98.701%	98.740%	98.713%	98.813%	98.825%	98.857%	98.859%	98.881%	98.931%	98.923%
20	98.921%	98.935%	98.973%	98.987%	98.980%	99.063%	99.053%	99.082%	99.102%	99.133%	99.150%
40	99.165%	99.265%	99.315%	99.300%	99.343%	99.355%	99.366%	99.413%	99.410%	99.427%	99.443%
60	99.354%	99.413%	99.466%	99.503%	99.530%	99.536%	99.541%	99.562%	99.573%	99.592%	99.598%
80	99.438%	99.459%	99.547%	99.547%	99.585%	99.585%	99.582%	99.630%	99.638%	99.660%	99.667%
100	99.444%	99.459%	99.548%	99.559%	99.589%	99.592%	99.584%	99.646%	99.641%	99.672%	99.674%
200	99.452%	99.461%	99.551%	99.569%	99.601%	99.605%	99.599%	99.724%	99.717%	99.846%	99.844%
300	99.452%	99.461%	99.558%	99.574%	99.615%	99.615%	99.606%	99.734%	99.743%	99.869%	99.878%
400	99.456%	99.464%	99.559%	99.577%	99.615%	99.628%	99.631%	99.739%	99.763%	99.877%	99.885%
500	99.456%	99.465%	99.560%	99.584%	99.624%	99.637%	99.641%	99.739%	99.772%	99.884%	99.891%
600	99.456%	99.471%	99.560%	99.584%	99.624%	99.641%	99.643%	99.740%	99.772%	99.891%	99.898%
750	99.458%	99.474%	99.560%	99.588%	99.634%	99.647%	99.645%	99.753%	99.778%	99.896%	99.901%
1500	99.462%	99.494%	99.572%	99.590%	99.638%	99.662%	99.656%	99.764%	99.792%	99.917%	
3000	99.471%	99.504%	99.582%	99.601%	99.651%	99.672%	99.666%	99.779%	99.812%		
6000	99.486%	99.506%	99.596%	99.616%	99.659%	99.675%	99.675%	99.787%			
12000	99.494%	99.517%	99.604%	99.626%	99.666%	99.681%	99.692%				
24000	99.498%	99.540%	99.611%	99.629%	99.681%	99.693%					
48000	99.508%	99.546%	99.622%	99.638%	99.685%						
96000	99.514%	99.555%	99.622%	99.643%							
192000	99.527%	99.563%	99.622%								
384000	99.538%	99.569%									
768000	99.551%										

In contrast, PA does benefit from the increase in the number of parallel instances. For instance, PA can obtain the same solution quality in half the number of iterations at 99% checkpoint (scaling of 2x for sequential vs 1024 parallel instances). Scaling of 633.7x in case of 99.5% checkpoint for sequential counterpart. Similarly, PA with 5 ant sequential local search has the same dynamics, with scaling of 4x at 99% checkpoint compared to sequential and 140x at 99.6% checkpoint compared to 2 and 1024 parallel instances. One can also note that at increased solution quality and a little number of parallel instances, PA with 5 ant local search also offers improved efficiency in terms of total solutions explored. For example, at the 99.5% checkpoint with 2 parallel instances, PA takes 2590 iterations, while PA with 5 ant local search only

requires 65 (decrease of 40x iterations or 8x total solutions explored). However, in most instances, PA without any local search is more efficient.

Table 16. *The number of iterations required to reach a specific solution quality. Each data point in the table is an average of 10 individual runs. Empty fields (-) represent instances where ACO did not obtain specified solution quality in 768k solutions explored. The solution quality is expressed as a percentage of the proximity of the best-know solution (2,701,367.58).*

The number of iterations required to reach specific solution quality												
Architecture	Checkpoint of optimal solution	The number of parallel instances										
		1	2	4	8	16	32	64	128	256	512	1024
Independent Ant Colonies	99.00%	30	30	35	30	30	35	30	30	25	25	25
	99.25%	45	45	40	40	45	40	40	35	35	35	35
	99.50%	31685	31055	29550	28895	29075	15910	10950	-	-	-	-
	99.60%	-	-	-	-	-	-	-	-	-	-	-
	99.75%	-	-	-	-	-	-	-	-	-	-	-
	99.90%	-	-	-	-	-	-	-	-	-	-	-
Parallel Ants	99.00%	30	25	25	25	25	25	20	15	15	15	15
	99.25%	45	40	40	35	35	35	35	35	30	30	30
	99.50%	31685	2590	65	60	60	55	55	55	55	50	50
	99.60%	-	-	9190	2640	195	170	230	70	70	65	65
	99.75%	-	-	-	-	-	-	-	685	310	140	135
	99.90%	-	-	-	-	-	-	-	-	-	800	675
Parallel Ants with 5 sequential ant local search	99.00%	20	15	15	15	15	10	10	10	10	10	5
	99.25%	30	30	30	30	30	25	30	25	20	25	20
	99.50%	400	65	55	55	50	50	50	50	45	45	45
	99.60%	-	7715	160	135	90	65	60	65	60	55	55
	99.75%	-	-	-	-	6630	205	150	155	130	125	125
	99.90%	-	-	-	-	-	-	-	-	460	255	160

5.4.2. Speed performance

To evaluate speed performance, each given configuration and parallel architecture were ran for 500 iterations or 10 minutes wall-clock time (whichever happens first) and recorded the total number of iterations and wall-clock time for three independent runs. Then, average wall-clock time per iteration was calculated. It is essential to measure the execution time correctly, just purely comparing computation per kernel/method may not show the real-life impact. For that reason, total time is measured from the start of the memory allocation to the freeing of the allocated memory, however it does not include the time required to load the dataset into memory. This allows us to estimate, with reasonable accuracy, what is the wall-clock time needed to run a specific architecture and configuration to converge to a given fitness quality. Although running each given architecture and configuration 10 times would produce more accurate convergence rate estimates, it would also require significantly more computation time. Furthermore, all vectorised implementations went through iterative profiling and optimisation process to obtain the fastest execution time. To the best of

the author's knowledge, all vectorised implementations have been fully optimised for the given hardware.

- **CPU**

ACO implementation of IAC, PA and PAwV was implemented in C++ and multiple experiments of the configurations are shown in **Table 17**. Intel C++ 18.0 with OpenMP 4.0 was used to compile the implementation. KMP[26] (an extension of OpenMP) config was varied based on total hardware core and logical core count (16c,2t = 32 OpenMP threads).

Very similar results were obtained for both IAC double precision and PA double precision, with PA having around 5% overhead compared to IAC. In both instances, running 32 OpenMP threads offered around 24% speed reduction compared to 16 threads. Furthermore, PAwV with double precision vectorisation using AVX2 offered speed reduction of 26%, while scaling from 16 OpenMP threads to 32 offered almost no scaling at 256 parallel instances upwards.

The nature of ACO pheromone sharing and probability calculations does not require double precision and therefore can be substituted with single-precision calculations.

AVX2 offers 256-bit manipulations, therefore increasing theoretical throughput by a factor of 2, compared to double precision. 36% decrease in execution time was obtained, as not all parts of the code can take advantage of SIMD.

Furthermore, doing 5 ant sequential local search within each parallel instance increases time linearly and produces little time savings in terms of solutions explored. The overall scaling factor at 1024 parallel instances compared to sequential execution at PAwV (single precision with AVX2 and 16c2t) is therefore 25.4x.

[26] OpenMP Thread Affinity Control https://software.intel.com/en-us/articles/openmp-thread-affinity-control

Table 17. *Hardware A wall-clock time per iteration, in seconds. KMP config is environment variable set as part of KMP_PLACE_THREADS, for all instances KMP_AFFINITY=scatter, optimisation level /O3, favour speed /Ot.*

Hardware A - CPU computation time per iteration (in seconds)												
Configuration		The number of Parallel Instances										
	KMP config	1	2	4	8	16	32	64	128	256	512	1024
IAC, double precision	16c,1t	0.078	0.081	0.083	0.085	0.112	0.196	0.372	0.691	1.368	2.661	5.263
	16c,2t						0.148	0.277	0.517	1.002	2.014	4.093
PA, double precision	16c,1t	0.082	0.084	0.085	0.090	0.115	0.205	0.383	0.705	1.411	2.743	5.483
	16c,2t						0.153	0.288	0.539	1.044	2.088	4.220
PAwV, double precision, AVX2	16c,1t	0.050	0.053	0.057	0.058	0.075	0.131	0.233	0.426	0.805	1.547	3.101
	16c,2t						0.107	0.189	0.351	0.749	1.536	3.095
PAwV, single precision, AVX2	16c,1t	0.049	0.050	0.052	0.055	0.066	0.111	0.206	0.367	0.699	1.355	2.664
	16c,2t						0.088	0.152	0.275	0.501	1.006	1.975
PAwV, single precision, AVX2, with 5 sequential ant local search	16c,1t	0.212	0.218	0.227	0.241	0.264	0.484	0.918	1.722	3.380	6.759	13.461
	16c,2t						0.347	0.645	1.222	2.369	4.659	9.704

- **Xeon Phi**

Similar experiments were also conducted on the Xeon Phi hardware, **Table 18**. Due to the poor convergence rate and search capability, the execution time for IAC was not measured. Xeon Phi differs from Hardware A with the ability to utilise up to 4 hyper-threads per core and AVX512 instruction set. Although Hardware B has 68 physical cores, for more straightforward comparison on base 2, only 64 were used in experiments. At 1024 parallel instances on double-precision PA, having 2 threads and 4 threads per core does offer speedup of 30% and 42% respectively, compared to 1 thread per core. Moving to the vectorised implementation of 256-bit AVX2, gains additional speedup of around 37% across all parallel instances, however, did not benefit from 4 hyper-threads. Furthermore, exploiting the AVX512 instruction set offers a further 24% speedup compared to AVX2. In this configuration having 4 hyper threads per core worsens the speed performance (3.644 seconds vs 3 seconds). Like Hardware A, PAwV was explored with single precision and offered near-perfect scaling on 1024 parallel instances with 4 hyper-threads per core, or 40% overall speed improvement compared to PAwV with double precision (3 seconds vs 1.804 seconds). Alike Hardware A, having 5 sequential local ants does not provide any time savings and time increases linearly. The overall scaling factor at 1024 parallel instances compared to sequential execution at PAwV (single precision with AVX512 and 64c4t) is therefore 148x.

Table 18. *Hardware B wall-clock time per iteration, in seconds. KMP config is environment variable set as part of KM_PLACE_THREADS, for all instances KMP_AFFINITY=scatter, optimisation level /O3, favour speed /Ot.*

Hardware B - Xeon Phi computation time per iteration (in seconds)												
Configuration		The number of Parallel Instances										
	KMP config	1	2	4	8	16	32	64	128	256	512	1024
	64c,1t								1.417	2.787	5.941	11.089
PA, double precision	64c,2t	0.687	0.687	0.725	0.726	0.726	0.729	0.734	1.014	1.974	3.845	7.669
	64c,4t								1.087	1.606	3.226	6.438
	64c,1t								0.818	1.578	3.094	6.114
PAwV, double precision, AVX2	64c,2t	0.408	0.411	0.430	0.431	0.433	0.434	0.438	0.563	1.047	2.022	3.964
	64c,4t								0.625	1.101	2.072	4.082
	64c,1t								0.608	1.152	2.242	4.404
PAwV, double precision, AVX512	64c,2t	0.304	0.309	0.326	0.326	0.327	0.332	0.335	0.446	0.809	1.535	3.000
	64c,4t								0.494	0.982	1.913	3.644
	64c,1t								0.521	0.970	1.900	3.806
PAwV, single precision, AVX512	64c,2t	0.261	0.266	0.282	0.284	0.284	0.287	0.288	0.359	0.646	1.210	2.361
	64c,4t								0.412	0.542	0.957	1.804
	64c,1t								2.342	4.601	9.136	18.844
PAwV, single precision, AVX512, with 5 sequential ant local search	64c,2t	1.105	1.123	1.195	1.200	1.205	1.205	1.215	1.489	2.915	5.743	11.815
	64c,4t								1.553	2.225	4.428	9.054

- **GPUs**

A further set of experiments were also conducted for GPU, **Table 19**. The implementation with no vectorisation (Blocks x1), uses 1 thread per CUDA block to compute one solution, therefore 1024 parallel instances require 1024 blocks. Similarly, for (Blocks x32), 32 threads are used per block, each thread computing its own solution independently. For parallel instances of 32, only 1 block would be used with 32 threads. The implementation of no vectorisation utilises no shared memory; however, all static problem metadata is stored as textures. A single kernel is launched, and the best solution across all parallel instances is returned.

Vectorized version implements architecture described in [282], storing the route choice matrix in shared memory and utilising local warp reduction for sum and max operations. Each thread-block builds its solution, while the extra 32 threads assist with the reduction operations, memory copies and fitness evaluation. **Table 19** shows a comparison between the two implementations. Implementation without vectorisation performs on average two times slower compared to the vectorised version. Furthermore, 64 threads per block (Blocks x64) performs slower than 32 threads per block (Block x32).

Next, scaling across multiple GPUs were explored. Each device takes a proportion of 1024 instances with unique seed values and after each iteration, the best overall solution is reduced. In the case of 2 GPUs and 1024 parallel instances, each device

will compute 512 parallel instances concurrently. Scaling across 2 (2x) and 4 GPUs (4x) did not provide any significant speedup (only 10%). This is due to the fact that each iteration consumes at least 50 seconds and scaling across multiple GPUs adds almost no overhead. The maximum number of parallel instances might need to be increased to fully utilise all 4 GPUs to the point where all Streaming Multiprocessors (SMs) are saturated and increasing block count increases the computation time linearly.

GPU implementation is, therefore, one magnitude of order slower than that of CPU. However, this could be explained by the nature of the problem and not be specific to ACO architecture, as there have been a lot of success on GPUs solving simple, low memory footprint TSP instances [273][282][284]. However, the supply chain problem requires a lot of random global memory access to check for all restrictions such as order limits, capacity constraints and weight limits, which are too big to be stored on the shared memory.

Table 19. *Hardware C wall-clock time per iteration, in seconds. The total number of parallel instances are adjusted for the thread-block dimensions. Compiled with CUDA 9.0. 1x, 2x and 4x correspond to the number of devices used to compute.*

Hardware C - GPU computation time per iteration (in seconds)											
	The number of Parallel Instances										
Configuration	1	2	4	8	16	32	64	128	256	512	1024
1x GPU no vectorisation (Blocks x 1)	46.7	47.6	47.6	47.4	47.4	48.9	50.8	53.4	60.8	126.8	229.0
1x GPU no vectorisation (Blocks x 32)	-	-	-	-	-	108.3	110.5	112.5	113.2	114.5	115.2
1x GPU with vectorisation (Blocks x32)	-	-	-	-	-	49.8	52.4	54.1	55.4	58.8	64.5
1x GPU with vectorisation (Blocks x64)	-	-	-	-	-	-	57.1	58.5	59.6	61.0	65.8
2x GPU with vectorisation (Blocks x32)	-	-	-	-	-	-	50.0	52.6	55.4	55.5	60.8
4x GPU with vectorisation (Blocks x32)	-	-	-	-	-	-	-	50.0	52.7	54.4	55.8

5.4.3. Hardware Comparison and speed of convergence

If both convergence rate of the architecture and the speed of the hardware is considered, an estimate can be made on what would be the average wall-clock time to converge to specific solution quality. The fastest configuration for both Hardware A (**Table 17**) and Hardware B (**Table 18**) was chosen and then multiplied by the number of iterations required to reach a specific solution quality (**Table 16**) to obtain an estimate of the compute time required (**Table 20**). Therefore, a fairer real-life impact can be derived.

If one only considers the best time to converge to 99% solution quality, Hardware A can do that in 1.24 seconds on average while Hardware B would take 6.66 seconds.

Furthermore, if we look at 99.5% solution quality, Hardware A would take 3.33 seconds while Hardware B - 17.01 seconds. Faster clock speed for Hardware A gives an advantage over Hardware B at lower solution quality checkpoints. In contrast, at 99.75% and 99.9% solution quality, Hardware B outperforms. More experimentation is required to determine if exploring more than 768k solutions at lower Parallel Instance count affects the dynamics at the 99.75-99.9% range. In addition, best computation time to achieve specific solution quality was also compared in **Figure 38**, where the estimated best computation time required (in logarithmic) is plotted against three tested architectures across various solution quality checkpoints. **Figure 38** clearly shows that GPU results (Hardware C) were considerably slower and therefore, author conclude that GPUs are not suitable for the supply chain problem solved.

Table 20. *Estimated time (in seconds) required to converge to specific solution quality. Calculated by multiplying the number of iterations by the time taken for iteration for individual best performing hardware configuration. Solution quality is expressed as a percentage of the proximity of the best-know solution (2,701,367.58).*

Estimated time required (in seconds) to reach specific solution quality												
Architecture	Checkpoint of optimal solution	The number of parallel instances										
		1	2	4	8	16	32	64	128	256	512	1024
Hardware A - TR1950x	99.00%	1.46	1.24	1.30	1.39	1.64	2.19	3.04	4.13	7.52	15.10	29.63
	99.25%	2.19	1.99	2.07	1.94	2.29	3.06	5.31	9.64	15.03	30.19	59.25
	99.50%	1539.02	128.82	3.37	3.33	3.93	4.81	8.35	15.14	27.56	50.32	98.75
	99.60%			476.40	146.33	12.78	14.88	34.92	19.27	35.07	65.42	128.38
	99.75%								188.60	155.33	140.91	266.63
	99.90%										805.20	1333.13
Hardware B - Xeon Phi 7250F	99.00%	7.84	6.66	7.04	7.09	7.10	7.18	5.76	6.18	8.13	14.36	27.06
	99.25%	11.76	10.65	11.27	9.92	9.94	10.05	10.08	14.42	16.26	28.71	54.12
	99.50%	8282.30	689.67	18.31	17.01	17.04	15.79	15.84	22.66	29.81	47.85	90.20
	99.60%			2588.73	748.49	55.39	48.80	66.26	28.84	37.94	62.21	117.26
	99.75%								282.22	168.02	133.98	243.54
	99.90%										765.60	1217.70
Hardware C - GPU	99.00%	1404	1191	1190	1187	1186	1223	1001	751	791	816	838
	99.25%	2106	1905	1904	1662	1661	1712	1752	1752	1581	1632	1676
	99.50%	1482595	123373	3095	2850	2847	2690	2753	2753	2899	2720	2794
	99.60%			437536	125398	9254	8315	11511	3504	3689	3536	3632
	99.75%									16338	7617	7544
	99.90%										43525	37719

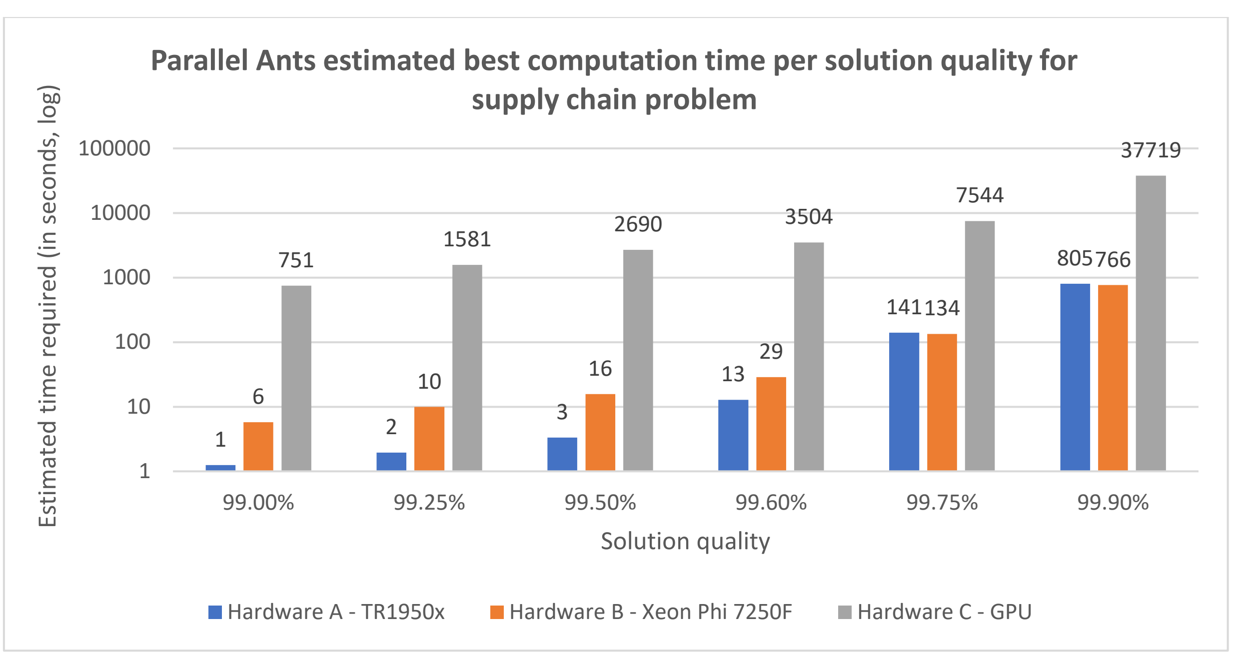

Figure 38. *Parallel Ants best estimated computation time per solution quality for supply chain problem to converge to specific solution quality. Solution quality is expressed as a percentage of the proximity of the best-know solution (2,701,367.58).*

5.4.4. Comparisons using the Travelling Salesman Problem

In addition to the real-world supply chain problem, a single TSP instance with 318 cities (lin318) is selected for comparison. The lin318 instance is small enough such that all experiments can be computed quickly but large enough to see measurable differences between hardware architectures explored. Like in the supply chain problem, solution quality checkpoints against optimal fitness value of 42029 were recorded during the convergence process. Moreover, just like in supply chain problem, PA outperformed IAC architecture for solving lin318. The lin318 computation time was plotted against various hardware solutions and solution quality checkpoints in **Figure 39**.

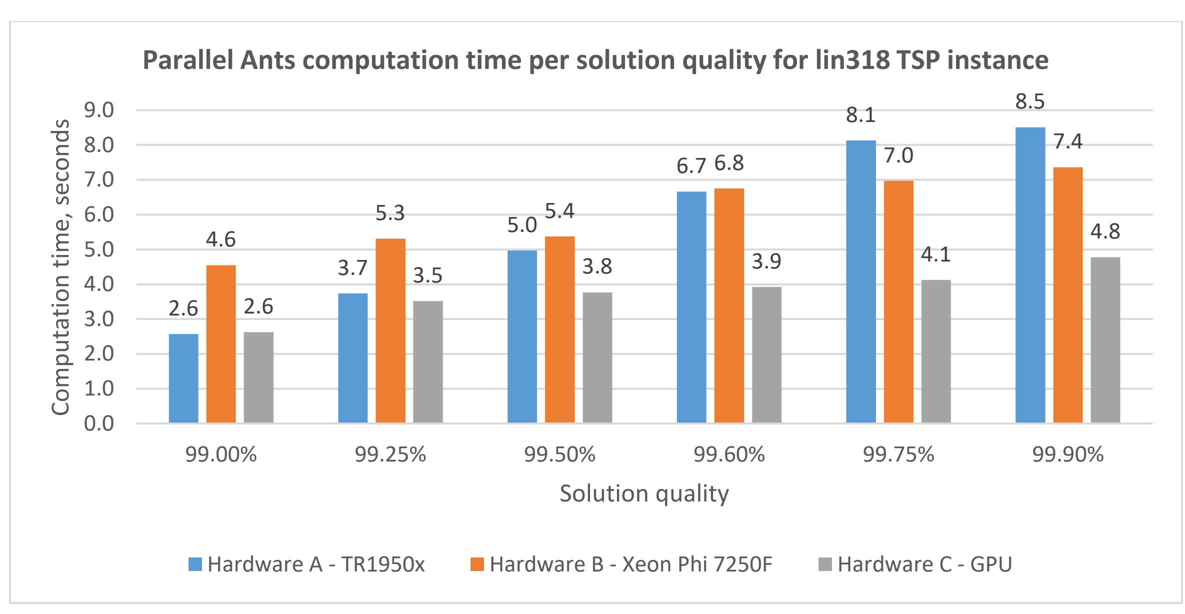

Figure 39. *Parallel Ants computation time per solution quality for lin318 TSP to converge to specific solution quality. Solution quality is expressed as a percentage of the proximity of the best-know solution (a distance of 42029).*

When solving the lin318 TSP instance, Hardware A performs faster than Hardware B for solution quality between 99.0% and 99.6% and slower for higher solution quality, similar to the supply chain problem results in **Figure 38**. Although Hardware C - GPU performed magnitudes slower in supply chain problem, for the TSP instance it was able to converge faster than Hardware A and Hardware B. Therefore, author can confirm the findings of [273][282][284], that suggest that GPUs offer speedup over CPU counterpart when routing simple TSPs. However, author also acknowledge that these dynamics do not apply for a more complex real-world routing problem where GPU is magnitudes slower than CPU counterparts (Hardware A or Hardware B) due to the additional meta-data required to be stored during solution creation.

5.5. Summary

Nature-inspired meta-heuristic algorithms such as Ant Colony Optimization (ACO) have been successfully applied to multiple different optimisation problems. Most work focuses on the Travelling Salesman Problem (TSP). While TSPs are a good benchmark for new idea comparison, the dynamics of the proposed algorithms for benchmarks do not always match real-world performance where the problem has more constraints (more meta-data during solution creation). Furthermore, speed and

fitness performance comparisons are not always completely fair when compared to a sequential implementation.

As the OptPlatform is designed for industry problem optimisation, this chapter explored the dynamics of different ACO architectures applied to benchmark and real-world problems. The experimental results demonstrate that the TSP benchmarks' results cannot be generalised to real-world applications, especially in terms of hardware performance and usage. Therefore, the findings demonstrate that in order to achieve the generalisable conclusions, the experimental work has to be completed on both: standard benchmarks and real-world applications.

Furthermore, the work solves a real-world outbound supply chain network optimisation problem and compares two different ACO architectures – Independent Ant Colonies (IAC) and Parallel Ants (PA). It was concluded that PA outperformed IAC in all instances, as IAC failed to find any better solution than 99.5% of optimal. In comparison, PA was able to find a near-optimal solution (99.9%) in fewer iterations due to effective pheromone sharing across ants after each iteration. Furthermore, PA shows that it consistently finds a better solution with the same number of iterations as the number of parallel instances increase.

Moreover, a detailed speed performance was measured for three different hardware architectures – 16 core 32 thread workstation CPU, 68 core server-grade Xeon Phi and general-purpose Nvidia GPUs. Results showed that although GPUs can scale when solving simple TSP (as confirmed by multiple other studies), those scaling dynamics do not transfer to more complex real-world problems. The memory access footprint required to check capacity limits and weight constraints did not fit on the small shared memory on GPU. Thus, it performed 29 times slower than the other two hardware solutions even when running 4 GPUs in parallel. Therefore, this finding is considered to be a new knowledge with surprise value.

When compared to a real-life impact on the time required to reach a specific solution quality, both CPU and Xeon Phi optimised-vectorised implementations showed comparable speed performance; with CPU taking the lead with lower Parallel Instances count due to the much higher clock frequency. At near-optimal solution (99.75%+) and 1024 parallel instances, Xeon Phi was able to take full advantage of AVX512 instruction set and outperformed CPU in terms of speed. Therefore, compared to an equivalent sequential implementation at 1024 parallel instances, CPU was able to scale 25.4x while Xeon Phi achieved a speedup of 148x.

Due to the findings of this study, OptPlatform targets mainly CPU architecture for the metaheuristic algorithm implementation. This has multiple benefits; first – all computers contain a CPU, though not all are guaranteed to contain a GPU. Furthermore, Xeon Phi is specialised hardware that has now been discontinued, thus not future proof. Next, CPU implementations are less complex and are not suspect to specific hardware vendor (CUDA is Nvidia proprietary software, for example).

Moreover, fast and efficient optimisation algorithms on CPU have multiple advantages. First, the limited computing cycles are utilised efficiently and not wasted; second, faster optimization allows to compute more what-if scenarios or optimize more networks/models. Finally, a quicker turnaround allows more agile problem modelling with quick feedback. The rapid feedback is critical when decisions need to be made quickly in case of disruptions, such as a global pandemic closing shipping ports and borders. Although computing is a considerable part of the optimisation process, the problem with implementation and metaheuristic tuning are usually the more time and labour intensive parts of optimisation. Fortunately, at least one part of that process can be further automated – the next chapter investigates automated ways to both select the best metaheuristic for the problem and fine-tune it for the best performance.

6. SIMPLE GENERATE-EVALUATE STRATEGY FOR TIGHT-BUDGET PARAMETER TUNING PROBLEMS

This chapter is based on the results published in [6].

Good hyperparameter selection is essential for metaheuristic algorithm performance. Tuning is usually a time-consuming and tedious task that requires user expertise for the best results. Automated tuning algorithms can help speed up this process and even lead to better parameter configurations; however, it requires vast computing resources. This is especially true for complex real-world problems where a single evaluation of a configuration can take minutes, hours or even days.

To overcome the problem, the eTuner and eTunerAlgo have been proposed as part of the OptPlatform. The distinctive feature of eTunerAlgo is that both algorithm selection and parameter tuning is performed automatically. Proposed algorithms were evaluated using three metaheuristics introduced in section 3.6 – ACO, ES, ICA and two NP-hard problems – Aerial Surveying Problem (ASP) and Multiple Knapsack Problem (MKP), section 2.3.2.1 and section 2.3.1.1 respectively. Furthermore, a metaheuristic tuning benchmark containing 18,760 configurations is generated for efficient method evaluation and published in [5] to encourage further research in this area.

6.1. Motivation

Most of the metaheuristics contain stochastic components and often have settings – set of hyperparameters – that can be defined by the user to solve the problem at hand. The metaheuristic setting (parameter setting) has a direct impact on the performance and efficiency of the metaheuristic [285]. Although, most metaheuristic algorithm implementations provide a default set of parameters (also referred to as *configuration*), tuning the algorithm's parameters for the problem at hand can lead to significant performance improvement. This is due to the fact that the default settings

are usually tuned for a different class of problems and may not be suitable for the problem at hand. Moreover, the process of parameter tuning up until the end of last century was done “by hand”, i.e. typical workflow would include running multiple experiments with a different set of parameters or using expertise knowledge [285] for both algorithm selection and parameter tuning. The rise of ease of access and reduced cost of computing has provided the means for a more systematic and automated approach for parameter setting problem, see **Figure 40** for a workflow comparison.

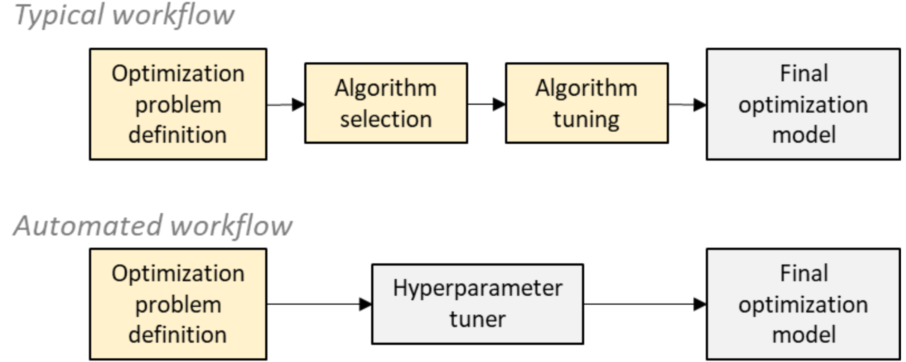

Figure 40. *Comparison between typical user workflow and automated workflow. Yellow boxes are indicating areas where user expertise is necessary for optimal results. In the automated workflow, algorithm selection and tuning are performed automatically.*

Parameter setting problems can be divided into two categories [286]: a) *parameter tuning* (also referred as *off-line tuning*), where all parameter settings are defined before applying an algorithm to solve problems at hand; b) *parameter control* (also referred as *on-line tuning*); where algorithmic parameters are managed and tuned during the execution of the algorithm. In this chapter, the focus is on the area of off-line parameter tuning problem.

There are clear benefits of parameter tuning; however, the process can be very time consuming and require user expertise and hence the algorithm parameter settings in most research is still performed by hand or the default settings used. Automated tuning methods have the advantage of not requiring users to know how parameters of the algorithm impact the performance. Furthermore, they can offer time savings and potentially result in better algorithm configuration than manual methods. However, for some real-world complex optimisation problems, where one algorithm parameter evaluation can take minutes, hours or days, a sophisticated tuning method may not be viable due to how many evaluations of configurations are required for a good

performance. In such cases, to obtain a quality configuration in a constrained tuning budget, a more straightforward method might be beneficial.

Motivated by such complex problems, a simple generate-evaluate method has been develop for both algorithm selection and parameter tuning. The contributions can be summarised as follows: a) a simple generate-evaluate tuning method is proposed based on elitism strategy for problems with low compute budget; b) novel algorithm selection method is described; c) metaheuristic benchmark of three optimisation algorithms with combined 18,760 configurations (with 10 evaluations each) for solving Aerial Surveying Problem (ASP) is generated and made available in [5].

6.1.1. Parameter tuning problem

In the parameter tuning problem, the main goal is to find a configuration that maximises the performance of an algorithm over the given problem instance(s), formally stated by [287]:

Given:

1) A parameterised algorithm A with free parameters that affect its behaviour.
2) A configuration space (or parameter space) C, which defines possible configurations (i.e., parameter settings).
3) A set of problem instances I.
4) A performance metric m that measures the performance of A across I for a given configuration c ($c \in C$)

Find: A configuration $c^* \in C$ that optimises the performance of A on I according to metric m.

The following glossary is introduced to facilitate ease of reading:

- ***Configuration*** – parameter values, parameter setting, hyperparameter setting, that are defined before applying the algorithm to solve problems at hand.
- ***Tuner –*** tuning algorithm, the automatic parameter tuning method used for finding optimal configuration.
- ***Evaluation*** – also referred to as an algorithm *run*, is a single compute using a metaheuristic algorithm with a configuration to solve the optimisation problem. The result is a solution for the optimisation problem with a given metric.

- ***Simulation*** – simulation refers to the compute of the tuner to obtain the best parameter configuration for the metaheuristic. The result is an average metric score of the metaheuristic evaluations using the best configuration.

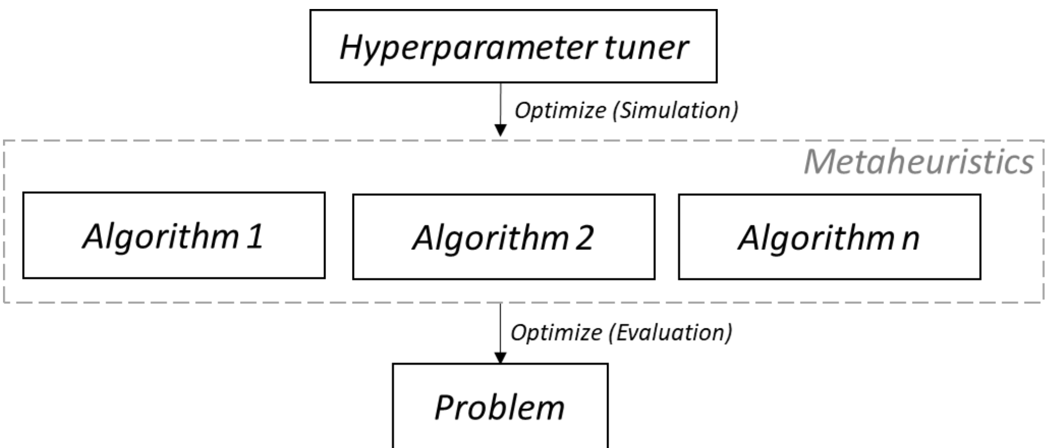

Figure 41*. A high-level overview of the process flow. The underlying problem is optimised by one or several metaheuristic algorithms. The metaheuristic algorithm(s)' parameters are optimised by the hyperparameter tuner.*

The high-level overview of parameter tuning is shown in **Figure 41**. The hyperparameter tuner is used for optimising the metaheuristic algorithm(s), while the metaheuristic algorithm(s) are optimising the problem at hand.

6.2. Related work

Multiple different tuning methods have been proposed over the last two decades to determine the best configuration of algorithms when solving the problem at hand. A recent survey in [285] discusses the full range of tuning algorithms deployed so far in great detail and classifies the approaches into three categories: simple generate-evaluate methods, Iterative generate-evaluate methods and high-level generate-evaluate methods. This section only reviews the most popular and relevant tuning approaches found in the literature.

The simple generate-evaluate methods are noniterative tuners that first generates a set of candidate configurations and only then evaluates them to find the best performing configuration. Techniques such as naïve brute force method as well as F-race algorithm fits this category. F-race is inspired from the Hoeffding race [288] initially used for machine learning model selection, later adopted for tuning metaheuristics in [289]. The basic idea of F-race is to sequentially evaluate candidate configurations and eliminate bad configurations as soon as sufficient statistical

evidence is present. F-race uses the compute power more efficiency compared to repeated evaluations in brute force, however, if the target algorithm has an ample parameter space, a large number of configurations needs to be evaluated before a good-performing result is found [290].

To overcome the drawbacks of F-race, authors in [291] proposed iterative application of F-race, called iterated F-Race (I/F-Race). Its promise was successfully demonstrated in tuning the MAX-MIN ant system and Simulated Annealing (SA) algorithm. Iterative F-Race, as the name suggests, follows an iterative tuning process, where at each iteration, a set of candidate configurations are generated based on the probabilistic model, then standard F-Race is performed. The survived candidate configurations are then used to update the probabilistic model for the next iteration [292]. Iterated F-Race is one of the more popular tuning approaches to date, used for automatic parameter tuning in [293] and [294]. One of the limitations of Iterative F-race is that it requires a sufficient number of iterations to be performed to obtain acceptable results. If the tuning budget is too small, the resulting configuration might be bad performing [285].

Another attractive iterative generate-evaluate approach is ParamILS [295], which uses a well-established stochastic local search method [296] as its core. It starts the search by variation of the default configuration and several randomly generated configurations. It then iteratively creates a new candidate which differs by an exactly single parameter – only one parameter value is changed at the time. Once the local best configuration is found, it performs stochastic local search procedure to determine which of the two candidate configurations is better. Other variations of the tuners are examined in ParamILS framework, most notably, tuning with the variable neighbourhood search [297]. Although ParamILS can support both categorical and numeric parameters, it requires for them to be discretised, such that each neighbouring candidate can be defined. Furthermore, this approach also relies on the default configuration to be accurately identified and be somewhat suitable for the problem at hand for best results.

Both, the Iterative F-Race and ParamILS are proven to be good tuners for small benchmark instances, where each configuration evaluation takes milliseconds/seconds. For such problems, a good configuration out of hundreds of thousands of evaluations can be found within a reasonable time frame. However, many real-world problems are more complex and require minutes, hours or even days

to evaluate. In such instances, hundreds of thousands of configuration evaluations are just not feasible, and the simpler generate-evaluate methods can help. For example, with a computing budget of one day, a problem that takes 1 minute to optimise would only offer 1440 evaluations within a 24h period. Although with the rise of modern computers, many of these tasks can be parallelised on clusters, efficient ways of parameter tuning for such large-scale problems are needed. Furthermore, with the ever-increasing speed of information, the latency of model creation and deployment is shrinking – thus time to market is more critical than ever. And because parameter tuning is an important aspect of increasing efficiency of metaheuristics, the tuning time should not be the bottleneck delaying the deployment.

6.3. Proposed methods

The purpose of a tuning algorithm is to determine both – the most suitable metaheuristic algorithm to be used for the problem, as well as to offer insides of the best hyperparameter configurations for the chosen metaheuristic. As discussed in the previous section, many approaches can be deployed for parameter tuning problem. One method is a naïve brute-force strategy, where an adequate number of configurations are evaluated over a sufficient number of evaluations, and the best overall average score is the final configuration. However, this approach requires some expert knowledge to determine the right size of configurations – a too small sample size leads to missed useful configurations. At the same time, too many configurations lead to wasted computation time.

Furthermore, it is also up to the user to determine how many evaluations for each configuration are necessary to cope with the stochastic nature of the metaheuristics. These drawbacks lead brute-force strategy in rigorously evaluating both good and bad configuration equally, further wasting computation resources. In some complex real-world optimisation problems, brute force method for tuning hyperparameters is prohibitively expensive as each configuration evaluation can take minutes, hours or even days. Thus, this section describes two strategies that overcome these drawbacks. The first approach, called eTuner tries to find good configurations across all metaheuristic algorithm configurations. The second method, called eTunerAlgo, starts by estimating the best metaheuristic first, and only then focus on metaheuristic parameters within the reduced set.

6.3.1. Elitist Tuner - eTuner

Elitist Tuner, eTuner for short, conceptually follows the elitism strategy found in genetic algorithms – best candidates in the population survive and reproduce. The eTuner starts with a random sample of candidate configurations and iteratively reduces the candidate configuration set based on the accumulated best averages achieved. The number of configurations remaining for the next iteration is determined by elitism rate ER. Furthermore, the number of iterations n in the eTuner is determined by maximising equation in (38):

$$\begin{aligned} \text{max:} \quad & TCT = RT * \sum_{n=1}^{\infty} \left\lfloor ʍ * \left(\frac{1}{ER}\right)^{n-1} \right\rfloor \\ \text{subject to:} \quad & TCT \leq TB \\ & ER \in (0,1) \\ & \forall ʍ \in \{1, \ldots, 10\} \end{aligned} \qquad (38)$$

where TCT is total compute time, RT is the average time in seconds to compute a single configuration, ER is the elitism rate, and TB is the total tuning budget, in seconds. Finally, $ʍ$ is an integer starting value.

Once the number of iterations n and the integer starting value $ʍ$ is determined, the starting number of random candidate configurations SNC is calculated by (39):

$$SNC = ʍ * \left(\frac{1}{ER}\right)^{n-1} \qquad (39)$$

At every iteration, first, each individual configuration performances in the configuration set CS are averaged, and the averages sorted to determine the elitists. Next, the candidate configuration set is reduced by eliminating the worst performing configurations, based on the individual configuration performance so far. The number of elitists NE are kept for the next iteration i, based on (40):

$$NE_i = ʍ * \left(\frac{1}{ER}\right)^{n-i} \qquad (40)$$

Figure 42 visualises the different ways computing resources can be allocated, where the compute budget TB is set to 100 hours and each configuration evaluation RT is assumed to be 60 seconds, see section 6.1.1 for terminology. After maximising equation (38) for three levels of elitism rate ER, we can obtain the number of iterations n and the integer starting value $ʍ$. ({ER=0.25, $ʍ$=1, n=7}; {ER=0.50, $ʍ$=5, n=10}; {ER=0.75, $ʍ$=2, n=24}). Then, the starting number of configurations SNC is derived by equation (39) and are 4096 for ER=0.25, 2560 for ER=0.50, 1495 for ER=0.75.

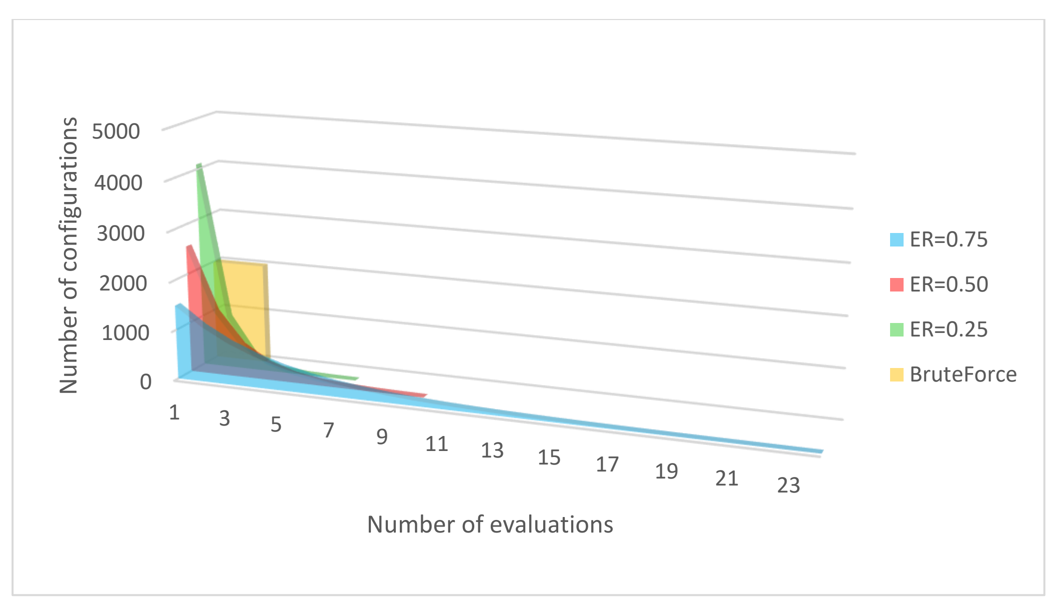

Figure 42. *Graphical representation of the allocation of configuration evaluations by variations of Elitism Rate and a brute force method for reference. All approaches are allowed to perform the same number of total experiments (100-hour tuning budget, with 60 second compute time for each configuration); thus, all three figures cover the same surface area.*

The above-described procedure discards weak configurations quickly while thoroughly evaluating more promising configurations. The Elitism Rate ER controls the trade-off between exploration of configurations against the repeated evaluations of configurations for more reliable estimates of their behaviour, pseudocode shown in **Figure 43**.

> Calculate the number of iterations n.
> Calculate the number of random starting configurations SNC.
> Generate random configuration set CS of size SNC.
> **for** $i = 0$ to n
>   Evaluate each configuration in the set CS one time.
>   Calculate evaluation averages for each configuration in CS.
>   Sort CS based on average performances.
>   Keep the best NE_i configurations in CS, discard the rest.
> **end for**
> return the best configuration in CS.

Figure 43. *Pseudocode of the proposed Etilist Tuner - eTuner algorithm*

6.3.2. Elitist tuning with pre algorithm selection – eTunerAlgo

One of the great features of metaheuristics is that they are generic and not problem-specific. This attribute allows the same metaheuristic algorithm to be applied to multiple different problem domains. Similarly, the same problem can be solved by numerous metaheuristic algorithms. The metaheuristic selection is usually done

manually based on some prior expert knowledge. Alternatively, the algorithm selection can be formulated as another categorical parameter and solved automatically by a hyperparameter tuner – approach discussed in the previous section, *eTuner*. However, this results in even higher parameter space to be tuned. If one of the metaheuristic X is more suited for the problem than metaheuristic Y, it would make sense to only focus on tuning metaheuristic X and discard the metaheuristic Y. The following section describes a simple method used to estimate the most suitable metaheuristic algorithm within a given set, called *eTunerAlgo*.

Given a set of metaheuristic algorithms A = $\{A_1, A_2, \dots A_M\}$, every metaheuristic algorithm has a set of hyperparameters associated with it – $P = \{P_1, P_2, \dots P_K\}$. Algorithm selection is performed as follows: 1) the average point between upper and lower bounds is calculated for each parameter in the set (required parameter to be a numerical value) to obtain overall "average" configuration across all parameters for given metaheuristic; 2) for each parameter in the list P, the "average" configuration is modified with upper and lower bounds value of the parameter, to create two new candidate configurations; 3) the "average" as well as two candidate configurations for each parameter is evaluated once, and the scores averaged; 4) The best overall averaged score is used to select the best metaheuristic algorithm from the list A. Example of this procedure is demonstrated in **Figure 44**.

$P_1 \in [0,1]$
$P_2 \in [2,10]$
$P_3 \in [10,100]$

$`average` \Rightarrow P_1=0.5, P_2=6, P_3 = 55$

$\{P_1=0, P_2=6, P_3 = 55\}; \{P_1=1, P_2=6, P_3 = 55\}$
$\{P_1=0.5, P_2=2, P_3 = 55\}; \{P_1=0.5, P_2=100, P_3 = 55\}$
$\{P_1=0.5, P_2=6, P_3 = 10\}; \{P_1=0.5, P_2=6, P_3 = 100\}$

Figure 44. *Example of new candidate configuration generation for metaheuristic algorithm selection. Where given three parameters P, one "average" configuration is generated and six other candidate configurations.*

The described metaheuristic selection approach aims to quickly estimate if one metaheuristic is better than other, on average. The downside of this approach is that it requires all hyperparameters to be a numeric value. This approach can also be adopted for categorical parameters with low dimensions.

Once the metaheuristic algorithm is selected, it follows the same elitism strategy as eTuner. The only additions are the extra configurations computed for the metaheuristic selection (AST), calculated in equation (41):

$$AST = \sum_{a=1}^{M} (1 + 2K_a) \tag{41}$$

where M is the total number of metaheuristic algorithms in the list and K is the total number of parameters for the metaheuristic a.

Thus, the total number of iterations in eTunerAlgo are determined by maximising equation (42) and following the same iterative elimination process as described in the previous section – eTuner.

$$\begin{aligned} \textbf{max:}\quad & TCT = RT * \left(AST + \sum_{n=1}^{\infty} \left\lfloor ʍ * \left(\frac{1}{ER}\right)^{n-1} \right\rceil\right) \\ \textbf{subject to:}\quad & TCT \leq TB \\ & ER \in (0,1) \\ & \forall ʍ \in \{1, \ldots, 10\} \end{aligned} \tag{42}$$

where TCT is total compute time, RT is the average time in seconds to compute a single configuration, ER is the elitism rate, and TB is the total tuning budget, in seconds. Finally, $ʍ$ is an integer starting value.

6.4. Experiments

In this study, first, a metaheuristic benchmark dataset for Aerial Surveying Problem (ASP) (section 2.3.2.1) was created to effectively evaluate the dynamics of various tuning approaches. Then, to validate and remove any biases, the best method is used to tune metaheuristics for Multi Knapsack Problem (section 2.3.1.1). Both problems are hard to solve, have practical applications and are fundamentally different from one another.

6.4.1. Experimental setup

- **Metaheuristics benchmark**

Three metaheuristic algorithms were selected for solving the optimisation problems – Ant Colony Optimization (ACO) based on the implementation in section Ant Colony Optimization 3.6.1, Evolutionary Strategy (ES) based on (μ+1)-ES section 3.6.2 and Imperialist Competitive Algorithm (ICA) based on section 4.2.2. All three metaheuristic

algorithms contain multiple numerical parameters, that can be tuned to increase the efficiency of the search. These parameters are summarised in **Table 21**. Candidate configurations are generated by dividing each parameter into discrete sets - Full Fractional Design (FFD) approach. This creates a total of 12,000 candidate configurations for ACO, 1000 for ES and 5,760 for ICA, a total of 18,760 across all algorithms. Although FDD is used for benchmark creation, the proposed tuning methods are not limited to a discrete set of parameters.

Table 21. *The algorithms and hyperparameters used for tuning. Each of the parameters has a discrete set of values that can be used for the candidate. The total number of candidate configurations for Ant Colony Optimization is 12,000, for Evolutionary Strategy – 1000 and Imperialist Competitive Algorithm – 5760. Thus, the total number of candidate configurations is 18,760.*

Parameter	**Discrete Set**	**Size of the set**
	Ant Colony Optimisation	
Parallel instances, PI_{max}	{32, 128, 512, 2048, 8192}	5
Number of ants, n_a	{1, 5, 9, 13}	4
Relative pheromone strength, α	{0, 2, 4, 8, 16}	5
Relative heuristic information strength, β	{0, 2, 4, 8, 16}	5
Exploitation to exploration ration, $q0$	{0, 0.3, 0.6, 0.9}	4
Cunning rate, CR	{0, 0.2, 0.4, 0.6, 0.8. 1}	6
	Evolutionary Strategy	
Population size, $N_{population}$	{32, 128, 512, 2048, 8192}	5
Mutation rate, M	{0.01, 0.06, 0.11, 0.16, 0.26, 0.31, 0.36, 0.4}	8
Local iterations, $N_{localIter}$	{1, 3, 5, 7, 9}	5
Swap ratio, Ƹ	{0.1, 0.3, 0.5, 0.7, 0.9}	5
	Imperialist Competitive Algorithm	
Number of countries, $N_{population}$	{32, 128, 512, 2048, 8192}	5
Imperialist ratio, N_{imp}	{0.1, 0.4, 0.7}	3
Local iterations, $N_{localIter}$	{1, 4, 7, 10}	4
Assimilation rate, θ	{0.1, 0.3, 0.5, 0.7}	4
Average power of empire's colonies, ξ	{0.05, 0.15, 0.25, 0.35}	4
Independence rate, $iRate$	{0, 0.2, 0.4, 0.6, 0.8, 1}	6

Considering that a single candidate configuration on Aircraft Surveying Problem takes around one minute to complete, it would be impractical to test and efficiently compare tuning algorithms. For that reason, a baseline was created by running all 18,760 configurations 10 times in a computer cluster, using 60 seconds elapsed compute time per configuration as the termination condition. This allows the creation of a benchmark dataset for all further tuning algorithm evaluations, as results can be sampled from memory at random (following uniform distribution), instead of requiring to be computed every time. The dataset have been made public in [5] to encourage

further research in this area. The dataset contains a list of all configurations with associated finesses for individual evaluations.

- **Implementation platform**

All metaheuristic algorithms were implemented in C++ using the Visual Studio 2019 (v142) compiler. Tuning algorithms, as well as metaheuristic benchmark queries, were deployed in C# using .NET framework 4.6.1. The computation was performed on a workstation cluster containing five AMD Threadripper 2990WX processors (3.0GHz, 64GB RAM), running Windows 10 Pro operating system.

6.4.2. Experimental results

As mentioned in the previous section, it would be impractical to compare and evaluate tuning methods efficiently on the Aerial Surveying Problem. For that reason, a tuning benchmark was created and made available in [5]. The benchmark creation totalled in around 130 days of computing time. After benchmark generation, a naïve brute force approach was simulated by altering the number of evaluations to establish a baseline for further experiments and results are shown in **Figure 45**. A *simulation* is referred to as a complete run of the tuner algorithm that produces a single best parameter configuration. The average cost of 10 evaluations of the best configuration in simulation is then retrieved from the memory.

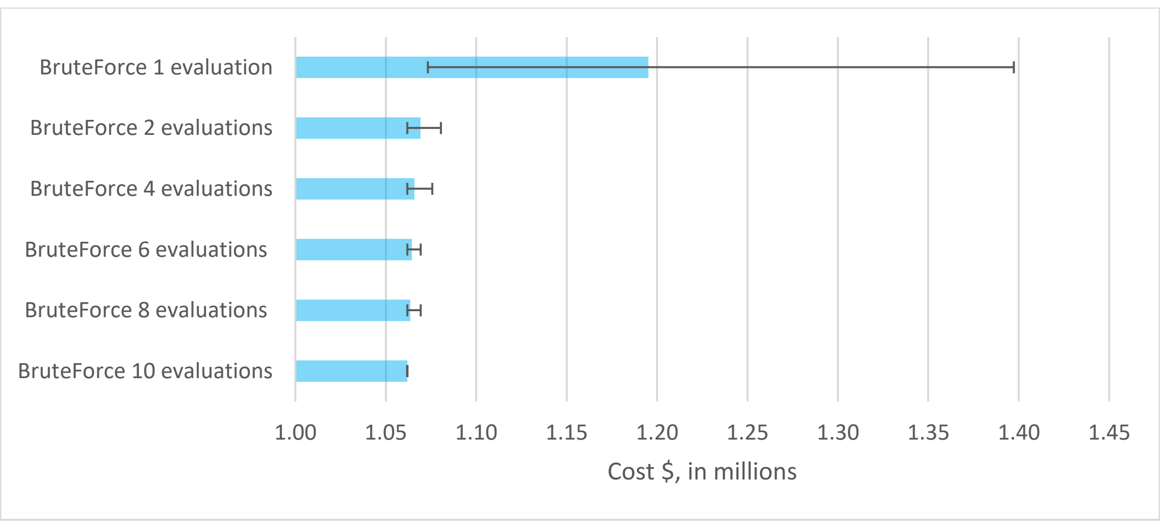

Figure 45. *The baseline for Aerial Surveying Problem (ASP) with a simple exhaustive search (brute force) approach, where each evaluation represents a single run for each of the 18,760 configurations. Error bars represent the minimum and maximum values achieved during 10 simulations—average total cost, in a million dollars (minimisation problem, lower costs are better).*

Results in **Figure 45** demonstrate that it requires at least two evaluations per configuration to achieve a reliable final configuration. Increasing evaluation count further just incrementally improves and stabilises the final result. Unfortunately, even two evaluations of exhaustive search across all configurations require 625 compute hours (26 days) to complete, which is one of the reasons brute force methods are to be avoided in large configuration space. Finding a good configuration in a reasonable computing time budget is one of the main goals of a hyperparameter tuner. What accounts as a reasonable computing time is very much at the discretion of the researcher or practitioner and depends on the underlying problem, metaheuristics and compute resources available. Seven levels of tuning budget are defined, ranging from 2 to 100 hours for the ASP.

Next, the dynamics of Elitism Rate (ER) impact on the proposed tuning algorithms were analysed, by setting ER at three levels – 0.25, 0.5 and 0.75. Each tuner was then simulated 10 times for seven tuning budgets to evaluate how the stochastic nature of metaheuristics impact the tuner performance and consistency.

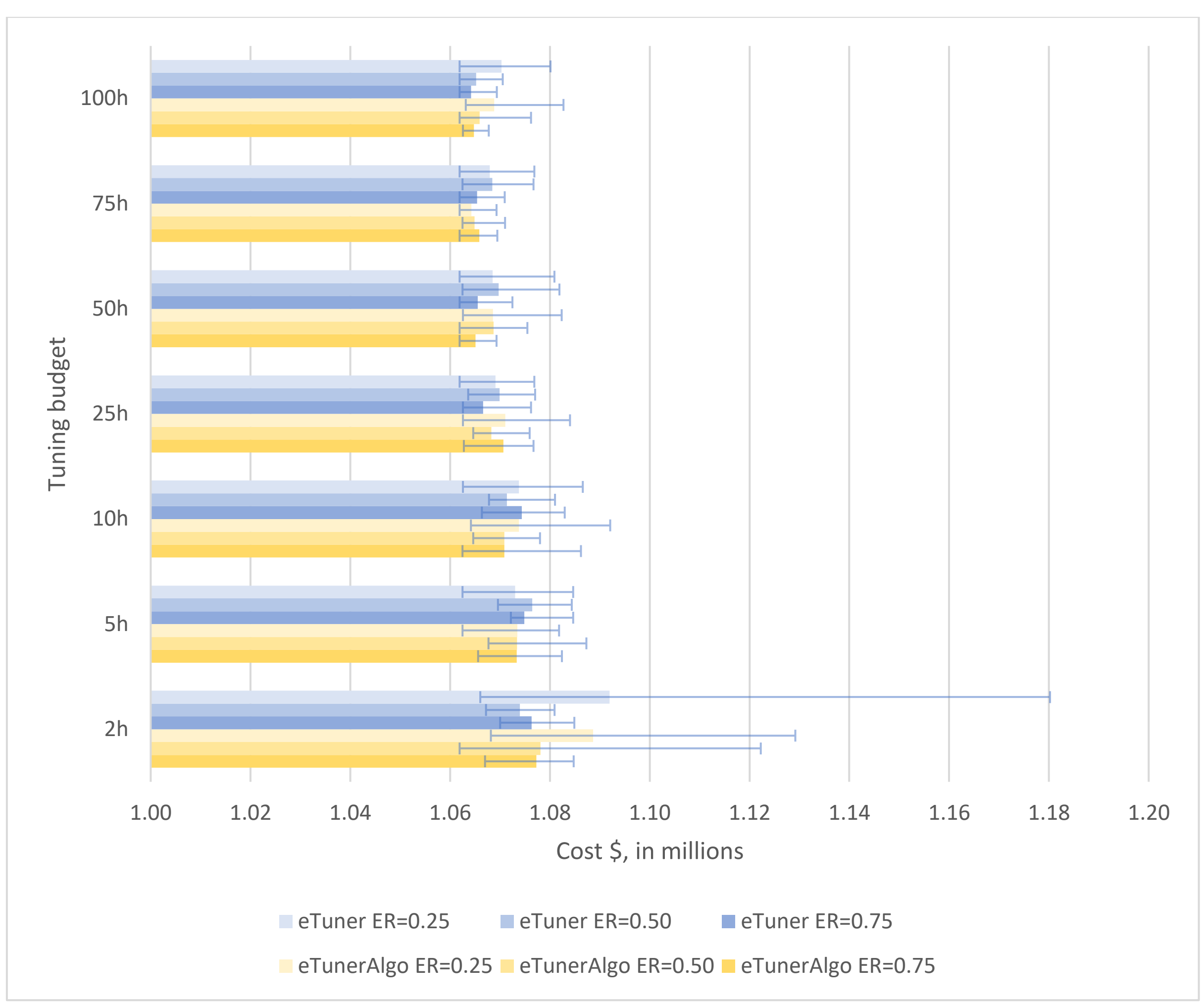

Figure 46. *Comparison of eTuner and eTunerAlgo approaches for Aerial Surveying Problem. Error bars represent the minimum and maximum values achieved during 10 simulations—average total cost, in a million dollars (minimisation problem, lower costs are better).*

Results are summarised in **Figure 46**, with the eTuner in blue and eTunerAlgo in yellow. There is a definite improvement in both tuner consistency when the tuning budget is increased from 2 hours to 5 hours; however, from 5 hours to 100 hours, the gain is less explicit. Results also suggest that higher Elitism Rate (ER) of 0.75 is beneficial for both tuners, however only marginally. Finally, both eTuner and eTunerAlgo results are comparable, though eTunerAlgo on average performs better.

For the next experiment, both proposed tuners were compared to a simple random approach, where the configurations are sampled at random and the best scores used for final configuration; as well as popular tuning algorithm called Iterative F-Race [292]. The *irace* package in R [298] was used for the I/R-Race implementation, where each metaheuristic parameter was set as a category, with conditions filtrating the

appropriate parameters for each metaheuristic. The default settings were used, and the compute budget enforced with the “--max-time” attribute.

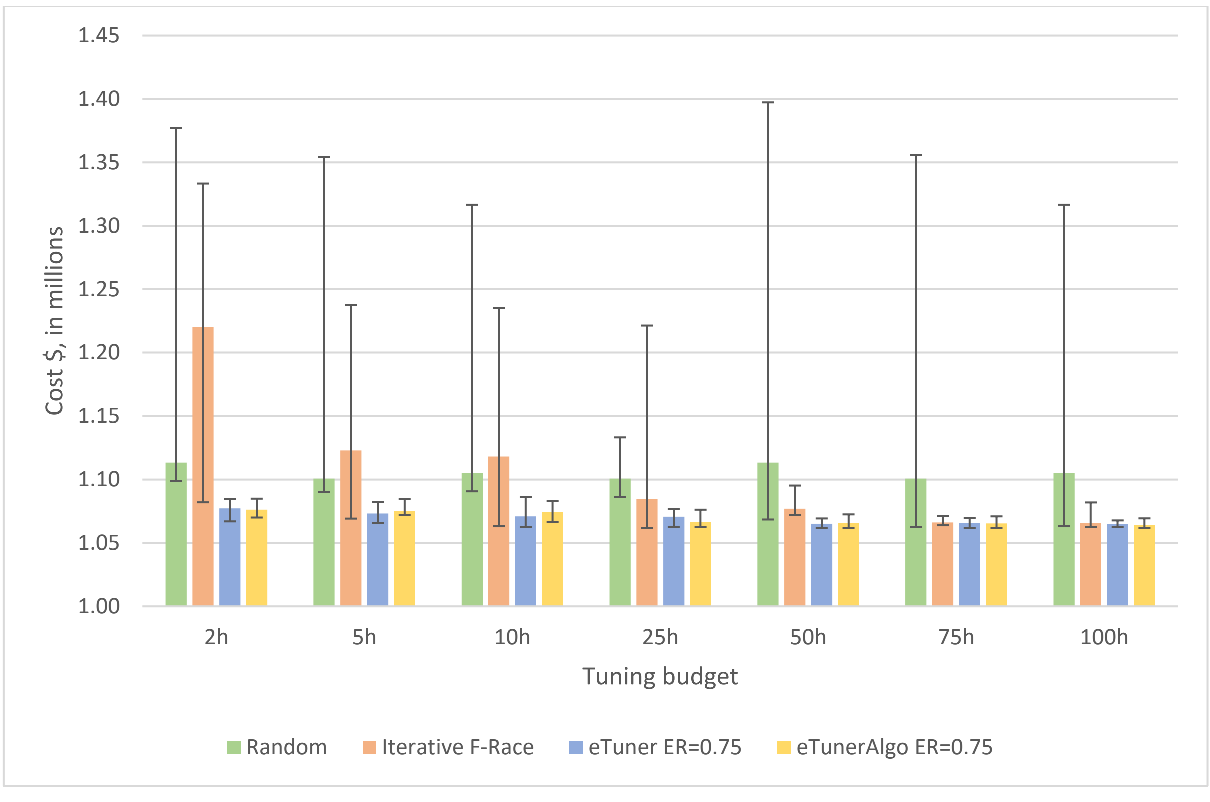

Figure 47. *Tuning algorithm comparison for Aerial Surveying Problem. Error bars represent the minimum and maximum values achieved during 10 simulations—average total cost, in a million dollars (minimisation problem, lower costs are better). eTuner and eTunerAlgo are the proposed methods, Iterative F-Race is the implementation of [298].*

Although trivial to implement, Random sampling approach alone is not suitable for finding an acceptable metaheuristic configuration reliably, as shown from the wide variance in found configuration scores across all time budgets in **Figure 47**. Furthermore, Iterative F-Race also has a high variation between 2- and 25-hour tuning budget, settling down to more stable solutions only from 50 hours onwards. As suggested in [285], Iterative F-race requires a sufficient number of candidate configurations to be sample and evaluated, otherwise if the tuning budget is too small, resulting configuration might be weak. This can be seen in both **Figure 47** and **Figure 48**, where with limited timing budget, Iterative F-Race does not have sufficient statistical evidence to pick the best configurations, but becomes stable and well-performing once at least 75-hour tuning budget is allowed. Moreover, both eTuner methods outperform the other approaches for tuning budgets up to 50h in both average scores and the consistency of the resulting configurations.

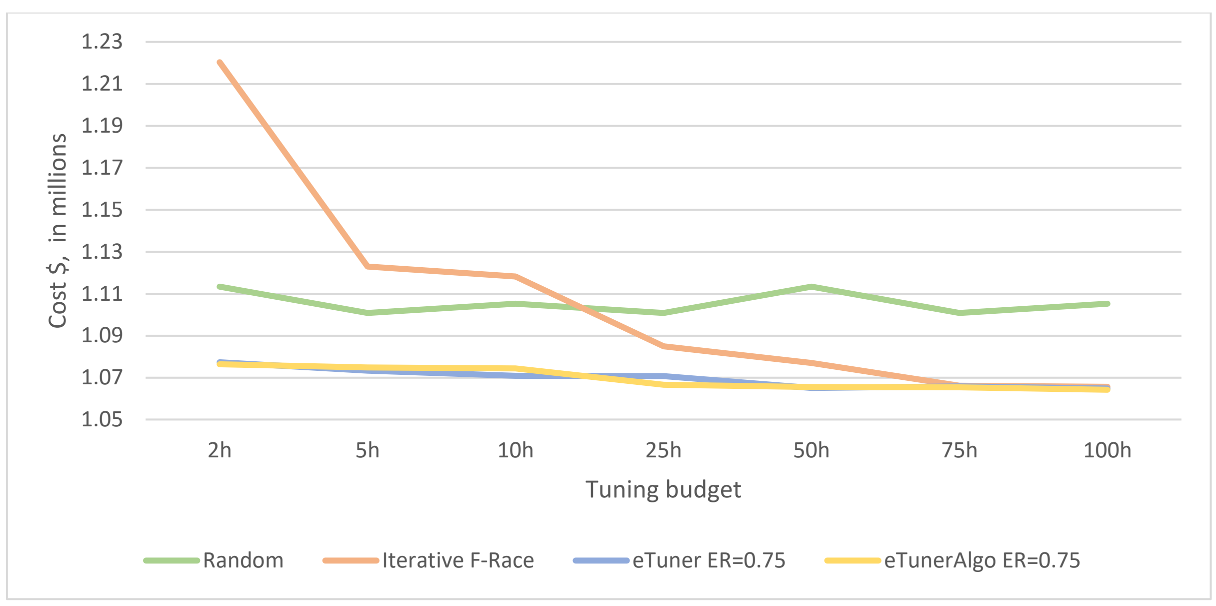

Figure 48. *Tuning algorithm comparison for Aerial Surveying Problem. Average total cost in a million dollars of 10 simulations (minimisation problem, lower costs are better). eTuner and eTunerAlgo are the proposed methods, Iterative F-Race is the implementation of [298].*

To remove any potential biases created by the generated metaheuristics benchmark for ASP, eTunerAlgo was also evaluated the for an entirely different NP-hard problem – Multiple Knapsack Problem using a complex gk10 instance. The same metaheuristics and parameter configuration sets in **Table 21** were used. Both Iterative F-Race and eTunerAlgo was simulated 10 times, and for every resulting configuration, the average of 10 evaluations computed. Tuning budget was set to 2, 5, 10 and 25 hours, with each configuration evaluation limited to 60 seconds.

Results in **Figure 49** show the improved configuration scores for the proposed method compared to the popular Iterative F-Race tuning algorithm. The average scores are not only better for eTunerAlgo, but also the resulting configurations are more consistent across all tuning budgets. Thus, the proposed method shows a high potential for tuning metaheuristics with a high dimensionality of parameters within a very tight tuning budget.

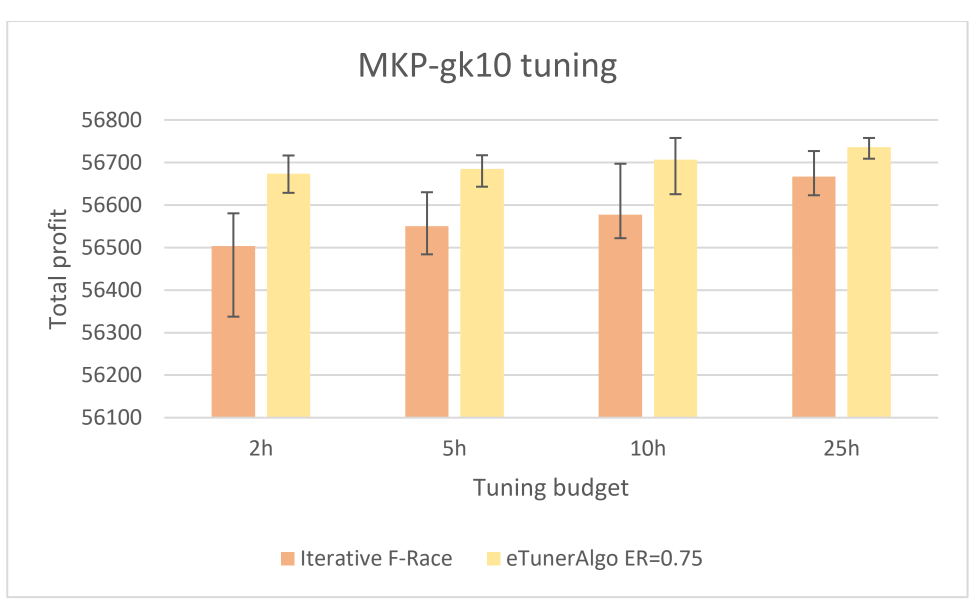

Figure 49. *Tuner performance comparison for MKP-gk10 instance. Error bars represent the minimum and maximum values achieved during 10 simulations—average profit (maximisation problem, higher profits are better). eTunerAlgo is the proposed method, Iterative F-Race is the implementation of [298].*

6.5. Summary

In this chapter, two new simple generate-evaluate tuning methods based on elitism strategy are presented. One of the methods, called eTuner uses the elitism strategy to select the best configuration out of a pool of metaheuristic algorithm' configurations. Furthermore, the second strategy – eTunerAlgo, first estimates the best and most suitable algorithm out of all available algorithms, before starting the tuning process on the reduced set of configurations.

The novel strategy is evaluated by first, generating a metaheuristic tuning benchmark containing three metaheuristics – ACO, ES, ICA and 18,760 configurations for easier method evaluation and comparison. Then, the popular Iterative F-Race strategy is used as a baseline. Results show that on a limited tuning budget, the developed approach can find better configurations with more consistency compared to the competition. Finally, to remove any potential biases, both eTunerAlgo and the baseline Iterative F-Race are used for other NP-hard problem. Similarly, eTunerAlgo outperforms the competition, indicating the superiority over the rival tuner on tight tuning budgets.

The tight tuning budgets are significant for the targeted user base of OptPlatform, where the real-world models take minutes to hours to compute. Therefore, both

proposed tuning methods have been implemented as part of an optional module in OptPlatform and seamlessly integrated into user workflow, as all fitness evaluations and problem specifics are already defined in previous steps of the optimisation. Furthermore, tuning methods' coherent integration allows further abstraction away from the user, thus lowering the expert knowledge required to produce close to optimum results within the platform.

7. CONCLUSIONS AND FUTURE WORK

7.1. Conclusions

This thesis's first and foremost contribution focuses on creating a new metaheuristic optimisation framework (MOF) called OptPlatform that improves on the limitations of existing MOFs laid out in section 2.2.1. One of the main limiting factors of existing MOFs is the lack of genericity of the supported metaheuristics algorithms. The majority of MOFs only support evolutionary computing based encodings and are thus limited in their applications. The proposed and implemented architecture in Chapter 3 overcomes this, by flexible solution encoding and static-dynamic memory model. Compared to other MOFs, OptPlatform outperforms the competition in both solution performance and the time required to achieve the solution. Furthermore, this work also implemented supporting tools lacking on the other MOFs, such as transition optimisation. Transition optimisation is beneficial when the new solution integration into real-world is not trivial and a multi-step process is necessary.

As the implemented OptPlatform mainly targets practitioners, the dynamic of scaling Ant Colony Optimization (ACO) algorithm across various hardware solutions was studied in-depth using a complex real-world problem, forming the contribution as part of Chapter 5. Unlike existing literature that only focuses on simple travelling salesman instances (TSP), this study analysed parallel ACO scaling on a real-world supply. Results showed that although these ACO architectures can scale for small benchmark problems such as TSP when applied to complex real-world problems with extra meta-data, platforms such as GPUs are not suited and get outperformed by CPUs.

The next contribution proposes a new, improved metaheuristic based on Imperialist Competitive Algorithm (ICA) called ICA with Independence and Constrained Assimilation (ICAwICA). The ICAwICA was implemented within OptPlatform and was used to solve multiple benchmark problems to demonstrate the algorithm's generic nature; this work formed Chapter 4. The Constrained Assimilation combines classical ICA assimilation and revolution operator while independence operator works as a local search to accelerate the convergence. Compared to other, problem-tuned algorithms

in the literature, the proposed ICAwICA showed very competitive results for both MKP and MDVRP instances.

In Chapter 6, the final contribution implements automatic algorithm selection and tuning to ease the development and improve metaheuristics performance within OptPlatform. Automated algorithm selection and tuning are significant to OptPlatform's target users – practitioners, as they are not expected to have an in-depth knowledge of metaheuristics or their hyperparameters. This feature is unique to OptPlatform, as no other analysed MOF offered automated algorithm or parameter selection. Furthermore, existing tuning methods are again, targeted to small benchmark problems, where results can be obtained in fractions of the second. When these existing tuning methods are applied to more complex real-world problems, they can be sub-optimal. For that reason, two alternative automatic tuning methods are proposed – eTuner and eTunerAlgo. Results show that within low tuning budget, both eTuner and eTunerAlgo outperform the more established tuning method in the literature.

7.2. Future work

Although the implemented OptPlatform provides multiple advancements on existing metaheuristic optimisation frameworks, numerous improvements would be beneficial.

First, currently, OptPlatform implements only three metaheuristics. Although the selected three algorithms are a good representation of overall metaheuristics, some are sufficiently different and thus might be better performing for some problems. Therefore, one of the areas of future focus would be implementing additional metaheuristics to improve further the benefits of using OptPlatform.

Although after an in-depth analysis in Chapter 5 it was concluded that CPU is the more suitable hardware platform for solving real-world supply chain; it would be beneficial to offer a GPU accelerated metaheuristics alongside the CPU option in the hyperparameter tuning module. That way, the computing hardware platform could be automatically selected as part of automated algorithm selection and tuning. Thus, accelerating the smaller benchmark instances on GPU, while the more complex problems would be assigned to the CPU-based implementations.

The automated tuning methods proposed could be further improved by assigning each parameter configuration to its own CPU in a cluster, thus clustered

implementation of OptPlatform is part of future research. Furthermore, the proposed tuning methods were only compared to the most established generate-evaluate method called iterative F-Race due to the time constraints. More throughout comparison between the dozen other tuning methods found in literature would be insightful and is part of future work.

Finally, the current implementation of OptPlatform is intended to be used as a stand-alone application that integrates into a larger existing IT infrastructure. As the OptPlatform is structured as an input-output black box, this could be further abstracted as part of a cloud service API that can be easier integrated into any system. A simple block diagram interface could be created for encoding and decoding of the problem. Such a cloud service could become an independent commercial product that can be used by multiple companies across the globe with little maintenance.

8. REFERENCES

[1] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 67–82, Apr. 1997, doi: 10.1109/4235.585893.

[2] I. Dzalbs and T. Kalganova, "Forecasting Price Movements in Betting Exchanges Using Cartesian Genetic Programming and ANN," *Big Data Res.*, vol. 14, pp. 112–120, 2018, doi: 10.1016/j.bdr.2018.10.001.

[3] I. Dzalbs, T. Kalganova, and I. Dear, "Imperialist Competitive Algorithm with Independence and Constrained Assimilation," in *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, 2020, pp. 1–11, doi: 10.1109/HORA49412.2020.9152916.

[4] I. Dzalbs and T. Kalganova, "Accelerating supply chains with Ant Colony Optimization across a range of hardware solutions," *Comput. Ind. Eng.*, vol. 147, p. 106610, Sep. 2020, doi: 10.1016/j.cie.2020.106610.

[5] I. Dzalbs and T. Kalganova, "Metaheuristic Parameter Tuning dataset," *Figshare*. 2020, doi: 10.6084/m9.figshare.12770201.

[6] I. Dzalbs and T. Kalganova, "Simple generate-evaluate strategy for tight-budget parameter tuning problems," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2020, pp. 783–790, doi: 10.1109/SSCI47803.2020.9308348.

[7] S. S. Rao, *Engineering Optimization*, 5th ed. Hoboken: John Wiley & Sons, Ltd, 2019.

[8] M. Karim and S. Hossein, "A Survey on the Combined Use of Optimization Methods and Game Theory," *Arch. Comput. Methods Eng.*, vol. 27, no. 1, pp. 59–80, 2020, doi: 10.1007/s11831-018-9300-5.

[9] G. J. Woeginger, "Exact Algorithms for NP-Hard Problems: A Survey," 2003, pp. 185–207.

[10] S. Desale, A. Rasool, S. Andhale, and P. Rane, "Heuristic and Meta-Heuristic Algorithms and Their Relevance to the Real World: A Survey," *Int. J. Comput. Eng. Res. Trends*, vol. 351, no. 5, pp. 2349–7084, 2015.

[11] T. Dokeroglu, E. Sevinc, T. Kucukyilmaz, and A. Cosar, "A survey on new generation metaheuristic algorithms," *Comput. Ind. Eng.*, vol. 137, no. September, p. 106040, Nov. 2019, doi: 10.1016/j.cie.2019.106040.

[12] C. Blum, M. J. B. Aguilera, A. Roli, and M. Sampels, Eds., *Hybrid Metaheuristics*, vol. 114. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[13] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, Sep. 2003, doi: 10.1145/937503.937505.

[14] M. Abdel-Basset, L. Abdel-Fatah, and A. K. Sangaiah, *Metaheuristic Algorithms: A Comprehensive Review*. Elsevier Inc., 2018.

[15] D. Molina, J. Poyatos, J. Del Ser, S. García, A. Hussain, and F. Herrera, "Comprehensive Taxonomies of Nature- and Bio-inspired Optimization: Inspiration versus Algorithmic Behavior, Critical Analysis and Recommendations," pp. 1–76, Feb. 2020.

[16] M. Gendreau and J.-Y. Potvin, "Metaheuristics in Combinatorial Optimization," *Ann. Oper. Res.*, vol. 140, no. 1, pp. 189–213, Nov. 2005, doi: 10.1007/s10479-005-3971-7.

[17] I. Boussaïd, J. Lepagnot, and P. Siarry, "A survey on optimization metaheuristics," *Inf. Sci. (Ny).*, vol. 237, pp. 82–117, Jul. 2013, doi: 10.1016/j.ins.2013.02.041.

[18] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *J. Chem. Phys.*, vol. 21, no. 6, pp. 1087–1092, Jun. 1953, doi: 10.1063/1.1699114.

[19] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science (80-.).*, vol. 220, no. 4598, pp. 671–680, May 1983, doi: 10.1126/science.220.4598.671.

[20] B. Suman and P. Kumar, "A survey of simulated annealing as a tool for single and multiobjective optimization," *J. Oper. Res. Soc.*, vol. 57, no. 10, pp. 1143–1160, Oct. 2006, doi: 10.1057/palgrave.jors.2602068.

[21] J.-P. Courat, G. Raynaud, I. Mrad, and P. Siarry, "Electronic component model minimization based on log simulated annealing," *IEEE Trans. Circuits Syst. I Fundam. Theory Appl.*, vol. 41, no. 12, pp. 790–795, 1994, doi: 10.1109/81.340841.

[22] M. Creutz, "Microcanonical Monte Carlo Simulation," *Phys. Rev. Lett.*, vol. 50, no. 19, pp. 1411–1414, May 1983, doi: 10.1103/PhysRevLett.50.1411.

[23] G. Dueck and T. Scheuer, "Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing," *J. Comput. Phys.*, vol. 90, no. 1, pp. 161–175, Sep. 1990, doi: 10.1016/0021-9991(90)90201-B.

[24] I. Charon and O. Hudry, "The noising method: a new method for combinatorial optimization," *Oper. Res. Lett.*, vol. 14, no. 3, pp. 133–137, Oct. 1993, doi: 10.1016/0167-6377(93)90023-A.

[25] T. A. Feo and M. G. . Resende, "A probabilistic heuristic for a computationally difficult set covering problem," *Oper. Res. Lett.*, vol. 8, no. 2, pp. 67–71, Apr. 1989, doi: 10.1016/0167-6377(89)90002-3.

[26] T. A. Feo and M. G. C. Resende, "Greedy Randomized Adaptive Search Procedures," *J. Glob. Optim.*, vol. 6, no. 2, pp. 109–133, Mar. 1995, doi: 10.1007/BF01096763.

[27] M. G. C. Resende and C. C. Ribeiro, "Greedy Randomized Adaptive Search Procedures: Advances, Hybridizations, and Applications," in *Handbook of Metaheuristics*, vol. 146, M. Gendreau and J.-Y. Potvin, Eds. Boston, MA: Springer US, 2010, pp. 283–319.

[28] M. G. C. Resende and C. C. Ribeiro, "Greedy randomized adaptive search procedures: Advances and extensions," *Int. Ser. Oper. Res. Manag. Sci.*, vol. 272, pp. 169–220, 2019, doi: 10.1007/978-3-319-91086-4_6.

[29] J. G. Villegas, C. Prins, C. Prodhon, A. L. Medaglia, and N. Velasco, "GRASP/VND and multi-start evolutionary local search for the single truck and trailer routing problem with satellite depots," *Eng. Appl. Artif. Intell.*, vol. 23, no. 5, pp. 780–794, Aug. 2010, doi: 10.1016/j.engappai.2010.01.013.

[30] A. Salehipour, K. Sörensen, P. Goos, and O. Bräysy, "Efficient GRASP+VND and GRASP+VNS metaheuristics for the traveling repairman problem," *4OR*, vol. 9, no. 2, pp. 189–209, Jun. 2011, doi: 10.1007/s10288-011-0153-0.

[31] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Comput. Oper. Res.*, vol. 13, no. 5, pp. 533–549, Jan. 1986, doi: 10.1016/0305-0548(86)90048-1.

[32] A. Amuthan and K. Deepa Thilak, "Survey on Tabu Search meta-heuristic optimization," in *2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES)*, 2016, vol. 6, no. 2, pp. 1539–1543, doi: 10.1109/SCOPES.2016.7955697.

[33] L. Piniganti, "A Survey of Tabu Search in Combinatorial Optimization," 2014.

[34] D. Cvijović and J. Klinowski, "Taboo Search: An Approach to the Multiple-Minima

Problem for Continuous Functions," in *Handbook of Global Optimization*, vol. 2, 2002, pp. 387–406.

[35] H. R. Lourenço, O. C. Martin, and T. Stützle, "Iterated Local Search," in *Handbook of Metaheuristics*, Boston: Kluwer Academic Publishers, 2003, pp. 320–353.

[36] H. R. Lourenço, O. C. Martin, and T. Stützle, "Iterated Local Search: Framework and Applications," in *Handbook of Metaheuristics*, vol. 146, M. Gendreau and J.-Y. Potvin, Eds. Boston, MA: Springer US, 2010, pp. 363–397.

[37] X. Li and M. Clerc, "Swarm Intelligence," in *Swarm Intelligence*, vol. II, CRC Press, 2019, pp. 353–384.

[38] W. T. Reeves, "Particle Systems—a Technique for Modeling a Class of Fuzzy Objects," *ACM Trans. Graph.*, vol. 2, no. 2, pp. 91–108, Apr. 1983, doi: 10.1145/357318.357320.

[39] S. Selvaraj and E. Choi, "Survey of swarm intelligence algorithms," *ACM Int. Conf. Proceeding Ser.*, pp. 69–73, 2020, doi: 10.1145/3378936.3378977.

[40] F. Fausto, A. Reyna-Orta, E. Cuevas, Á. G. Andrade, and M. Perez-Cisneros, "From ants to whales: metaheuristics for all tastes," *Artif. Intell. Rev.*, vol. 53, no. 1, pp. 753–810, Jan. 2020, doi: 10.1007/s10462-018-09676-2.

[41] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, 1995, vol. 4, no. 2, pp. 1942–1948, doi: 10.1109/ICNN.1995.488968.

[42] A. Banks, J. Vincent, and C. Anyakoha, "A review of particle swarm optimization. Part I: background and development," *Nat. Comput.*, vol. 6, no. 4, pp. 467–484, Oct. 2007, doi: 10.1007/s11047-007-9049-5.

[43] H. Firpi, "Handbook of Bioinspired Algorithms and Applications," *Brief. Bioinform.*, vol. 8, no. 4, pp. 275–276, Mar. 2007, doi: 10.1093/bib/bbm009.

[44] S. Sengupta, S. Basak, and R. Peters, "Particle Swarm Optimization: A Survey of Historical and Recent Developments with Hybridization Perspectives," *Mach. Learn. Knowl. Extr.*, vol. 1, no. 1, pp. 157–191, Oct. 2018, doi: 10.3390/make1010010.

[45] S. Lalwani, H. Sharma, S. C. Satapathy, K. Deep, and J. C. Bansal, "A Survey on Parallel Particle Swarm Optimization Algorithms," *Arab. J. Sci. Eng.*, vol. 44, no. 4, pp. 2899–2923, Apr. 2019, doi: 10.1007/s13369-018-03713-6.

[46] M. Habib, I. Aljarah, H. Faris, and S. Mirjalili, "Multi-objective Particle Swarm Optimization: Theory, Literature Review, and Application in Feature Selection for Medical Diagnosis," in *International Journal of Environmental Science and Technology*, vol. 16, no. 2, Springer Berlin Heidelberg, 2020, pp. 175–201.

[47] M. Cherrington, D. Airehrour, J. Lu, F. Thabtah, Q. Xu, and S. Madanian, "Particle Swarm Optimization for Feature Selection: A Review of Filter-based Classification to Identify Challenges and Opportunities," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2019, pp. 0523–0529, doi: 10.1109/IEMCON.2019.8936185.

[48] N. Nayar, S. Ahuja, and S. Jain, "Swarm Intelligence for Feature Selection: A Review of Literature and Reflection on Future Challenges," in *Lecture Notes in Networks and Systems*, vol. 39, 2019, pp. 211–221.

[49] P. Lučić and D. Teodorović, "Computing with Bees: Attacking Complex Transportation Engineering Problems," *Int. J. Artif. Intell. Tools*, vol. 12, no. 03, pp. 375–394, Sep. 2003, doi: 10.1142/S0218213003001289.

[50] X.-S. Yang, "Engineering Optimizations via Nature-Inspired Virtual Bee Algorithms," in *Lecture Notes in Computer Science*, vol. 3562, no. PART II, 2005, pp. 317–323.

[51] H. F. Wedde, M. Farooq, and Y. Zhang, "BeeHive: An Efficient Fault-Tolerant Routing Algorithm Inspired by Honey Bee Behavior," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3172 LNCS, 2004, pp. 83–94.

[52] N. Gordon, I. A. Wagner, and A. M. Bruckstein, "Discrete bee dance algorithm for pattern formation on a grid," in *IEEE/WIC International Conference on Intelligent Agent Technology, 2003. IAT 2003.*, 2003, pp. 545–549, doi: 10.1109/IAT.2003.1241141.

[53] D. Karaboga, "An Idea Based on Honey Bee Swarm for Numerical Optimization," 2005.

[54] S. K. Agarwal and S. Yadav, "A Comprehensive Survey on Artificial Bee Colony Algorithm as a Frontier in Swarm Intelligence," vol. 904, no. January, Springer Singapore, 2019, pp. 125–134.

[55] E. Hancer, "Artificial Bee Colony: Theory, Literature Review, and Application in Image Segmentation," 2020, pp. 47–67.

[56] E. E. Okoro, O. E. Agwu, D. Olatunji, and O. D. Orodu, "Artificial Bee Colony ABC a Potential for Optimizing Well Placement - A Review," in *SPE Nigeria Annual International Conference and Exhibition*, 2019, doi: 10.2118/198729-MS.

[57] A. Kaur and S. Goyal, "A survey on the applications of bee colony optimization techniques," *Int. J. Comput. Sci. Eng.*, vol. 3, no. 8, pp. 3037–3046, 2011.

[58] D. Karaboga, B. Gorkemli, C. Ozturk, and N. Karaboga, "A comprehensive survey: artificial bee colony (ABC) algorithm and applications," *Artif. Intell. Rev.*, vol. 42, no. 1, pp. 21–57, Jun. 2014, doi: 10.1007/s10462-012-9328-0.

[59] X. Yang and Suash Deb, "Cuckoo Search via Lévy flights," in *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*, 2009, pp. 210–214, doi: 10.1109/NABIC.2009.5393690.

[60] A. S. Joshi, O. Kulkarni, G. M. Kakandikar, and V. M. Nandedkar, "Cuckoo Search Optimization- A Review," *Mater. Today Proc.*, vol. 4, no. 8, pp. 7262–7269, 2017, doi: 10.1016/j.matpr.2017.07.055.

[61] M. Shehab, A. T. Khader, and M. A. Al-Betar, "A survey on applications and variants of the cuckoo search algorithm," *Appl. Soft Comput.*, vol. 61, no. April 2019, pp. 1041–1059, Dec. 2017, doi: 10.1016/j.asoc.2017.02.034.

[62] A. B. Mohamad, A. M. Zain, and N. E. Nazira Bazin, "Cuckoo Search Algorithm for Optimization Problems—A Literature Review and its Applications," *Appl. Artif. Intell.*, vol. 28, no. 5, pp. 419–448, May 2014, doi: 10.1080/08839514.2014.904599.

[63] H. Chiroma *et al.*, "Bio-inspired computation: Recent development on the modifications of the cuckoo search algorithm," *Appl. Soft Comput.*, vol. 61, pp. 149–173, Dec. 2017, doi: 10.1016/j.asoc.2017.07.053.

[64] X. Yang, *Nature-Inspired Metaheuristic Algorithms*. Luniver Press, 2010.

[65] S. L. Tilahun, J. M. T. Ngnotchouye, and N. N. Hamadneh, "Continuous versions of firefly algorithm: a review," *Artif. Intell. Rev.*, vol. 51, no. 3, pp. 445–492, Mar. 2019, doi: 10.1007/s10462-017-9568-0.

[66] S. L. Tilahun and J. M. T. Ngnotchouye, "Firefly algorithm for discrete optimization problems: A survey," *KSCE J. Civ. Eng.*, vol. 21, no. 2, pp. 535–545, Feb. 2017, doi: 10.1007/s12205-017-1501-1.

[67] W. A. Khan, N. N. Hamadneh, S. L. Tilahun, and J. M. T. Ngnotchouye, "A Review and Comparative Study of Firefly Algorithm and its Modified Versions," in *Optimization Algorithms - Methods and Applications*, InTech, 2016.

[68] I. Fister, I. Fister, X. Yang, and J. Brest, "A comprehensive review of firefly algorithms,"

Swarm Evol. Comput., vol. 13, pp. 34–46, Dec. 2013, doi: 10.1016/j.swevo.2013.06.001.
[69] N. Dey, J. Chaki, L. Moraru, S. Fong, and X.-S. Yang, “Firefly Algorithm and Its Variants in Digital Image Processing: A Comprehensive Review,” 2020, pp. 1–28.
[70] J. Nayak, B. Naik, D. Pelusi, and A. V. Krishna, “A Comprehensive Review and Performance Analysis of Firefly Algorithm for Artificial Neural Networks,” 2020, pp. 137–159.
[71] A. E. Eiben and M. Schoenauer, “Evolutionary Computing,” Nov. 2005.
[72] M. J. Fogel, L. J., Owens, A. J., & Walsh, *Artificial intelligence through simulated evolution*. John Wiley & Sons, 1966.
[73] I. Rechenberg, *Evolutionstrategie: Optimierung Technisher Systeme nach Prinzipien des Biologischen Evolution*. Stuttgart: Fromman-Hozlboog Verlag, 1973.
[74] H.-P. Schwefel, *Numerical Optimization of Computer Models*. New-York: John Wiley & Sons, 1981.
[75] J. Koza, “Genetic programming as a means for programming computers by natural selection,” *Stat. Comput.*, vol. 4, no. 2, Jun. 1994, doi: 10.1007/BF00175355.
[76] P. A. Vikhar, “Evolutionary algorithms: A critical review and its future prospects,” in *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*, 2016, pp. 261–265, doi: 10.1109/ICGTSPICC.2016.7955308.
[77] R. SARKER, J. KAMRUZZAMAN, and C. NEWTON, “EVOLUTIONARY OPTIMIZATION (EvOpt) : A BRIEF REVIEW AND ANALYSIS,” *Int. J. Comput. Intell. Appl.*, vol. 03, no. 04, pp. 311–330, Dec. 2003, doi: 10.1142/S1469026803001051.
[78] C. Blum *et al.*, “Evolutionary Optimization,” in *Variants of Evolutionary Algorithms for Real-World Applications*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–29.
[79] A. Shukla, H. M. Pandey, and D. Mehrotra, “Comparative review of selection techniques in genetic algorithm,” in *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, 2015, pp. 515–519, doi: 10.1109/ABLAZE.2015.7154916.
[80] P. Kora and P. Yadlapalli, “Crossover Operators in Genetic Algorithms: A Review,” *Int. J. Comput. Appl.*, vol. 162, no. 10, pp. 34–36, Mar. 2017, doi: 10.5120/ijca2017913370.
[81] U. A.J. and S. P.D., “CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A REVIEW,” *ICTACT J. Soft Comput.*, vol. 06, no. 01, pp. 1083–1092, Oct. 2015, doi: 10.21917/ijsc.2015.0150.
[82] S. Mirjalili, J. S. Dong, A. S. Sadiq, and H. Faris, *Nature-Inspired Optimizers*, vol. 811. Cham: Springer International Publishing, 2020.
[83] C. K. H. Lee, “A review of applications of genetic algorithms in operations management,” *Eng. Appl. Artif. Intell.*, vol. 76, no. May, pp. 1–12, Nov. 2018, doi: 10.1016/j.engappai.2018.08.011.
[84] S. K. Jauhar and M. Pant, “Genetic algorithms in supply chain management: A critical analysis of the literature,” *Sādhanā*, vol. 41, no. 9, pp. 993–1017, Sep. 2016, doi: 10.1007/s12046-016-0538-z.
[85] K. Höschel and V. Lakshminarayanan, “Genetic algorithms for lens design: a review,” *J. Opt.*, vol. 48, no. 1, pp. 134–144, Mar. 2019, doi: 10.1007/s12596-018-0497-3.
[86] Z. Wang and A. Sobey, “A comparative review between Genetic Algorithm use in composite optimisation and the state-of-the-art in evolutionary computation,” *Compos. Struct.*, vol. 233, p. 111739, Feb. 2020, doi:

10.1016/j.compstruct.2019.111739.
[87] B. M. Varghese and R. J. S. Raj, “A survey on variants of genetic algorithm for scheduling workflow of tasks,” in *2016 Second International Conference on Science Technology Engineering and Management (ICONSTEM)*, 2016, pp. 489–492, doi: 10.1109/ICONSTEM.2016.7560870.
[88] P. Krömer, J. Platoš, and V. Snášel, “Nature-inspired meta-heuristics on modern GPUs: State of the art and brief survey of selected algorithms,” *Int. J. Parallel Program.*, vol. 42, no. 5, pp. 681–709, 2014, doi: 10.1007/s10766-013-0292-3.
[89] L. Miguel Antonio and C. A. Coello Coello, “Coevolutionary Multiobjective Evolutionary Algorithms: Survey of the State-of-the-Art,” *IEEE Trans. Evol. Comput.*, vol. 22, no. 6, pp. 851–865, Dec. 2018, doi: 10.1109/TEVC.2017.2767023.
[90] M. A. Potter and K. A. Jong, “A cooperative coevolutionary approach to function optimization,” 1994, pp. 249–257.
[91] F. Hsieh, F.-M. Zhan, and Y.-H. Guo, “A solution methodology for carpooling systems based on double auctions and cooperative coevolutionary particle swarms,” *Appl. Intell.*, vol. 49, no. 2, pp. 741–763, Feb. 2019, doi: 10.1007/s10489-018-1288-x.
[92] Z. Li, M. N. Janardhanan, Q. Tang, and P. Nielsen, “Co-evolutionary particle swarm optimization algorithm for two-sided robotic assembly line balancing problem,” *Adv. Mech. Eng.*, vol. 8, no. 9, p. 168781401666790, Sep. 2016, doi: 10.1177/1687814016667907.
[93] C. Hu, P. Zhang, and H. Liu, “Cooperative Co-evolutionary Artificial Bee Colony Algorithm Based on Hierarchical Communication Model,” *Chinese J. Electron.*, vol. 25, no. 3, pp. 570–576, May 2016, doi: 10.1049/cje.2016.05.025.
[94] W. D. Hillis, “Co-evolving parasites improve simulated evolution as an optimization procedure,” *Phys. D Nonlinear Phenom.*, vol. 42, no. 1–3, pp. 228–234, Jun. 1990, doi: 10.1016/0167-2789(90)90076-2.
[95] X. Ma *et al.*, “A Survey on Cooperative Co-Evolutionary Algorithms,” *IEEE Trans. Evol. Comput.*, vol. 23, no. 3, pp. 421–441, Jun. 2019, doi: 10.1109/TEVC.2018.2868770.
[96] S. Nguyen, Y. Mei, and M. Zhang, “Genetic programming for production scheduling: a survey with a unified framework,” *Complex Intell. Syst.*, vol. 3, no. 1, pp. 41–66, Mar. 2017, doi: 10.1007/s40747-017-0036-x.
[97] M. T. Ahvanooey, Q. Li, M. Wu, and S. Wang, “A Survey of Genetic Programming and Its Applications,” *KSII Trans. Internet Inf. Syst.*, vol. 13, no. 4, Apr. 2019, doi: 10.3837/tiis.2019.04.002.
[98] A. De Lorenzo, A. Bartoli, M. Castelli, E. Medvet, and B. Xue, “Genetic programming in the twenty-first century: a bibliometric and content-based analysis from both sides of the fence,” *Genet. Program. Evolvable Mach.*, no. 0123456789, Jul. 2019, doi: 10.1007/s10710-019-09363-3.
[99] A. Khan, A. S. Qureshi, N. Wahab, M. Hussain, and M. Y. Hamza, “A Recent Survey on the Applications of Genetic Programming in Image Processing,” pp. 1–30, Jan. 2019.
[100] I. Dzalbs and T. Kalganova, “Multi-step Ahead Forecasting Using Cartesian Genetic Programming,” in *Inspired by Nature. Emergence, Complexity and Computation*, 2018, pp. 235–246.
[101] J. L. Deneubourg, S. Aron, S. Goss, and J. M. Pasteels, “The self-organizing exploratory pattern of the argentine ant,” *J. Insect Behav.*, vol. 3, no. 2, pp. 159–168, Mar. 1990, doi: 10.1007/BF01417909.
[102] M. Dorigo and T. Stützle, “Ant Colony Optimization: Overview and Recent Advances,”

in *International Series in Operations Research and Management Science*, vol. 272, 2019, pp. 311–351.

[103] M. Dorigo, V. Maniezzo, and a Colorni, “The ant system: An autocatalytic optimizing process,” *TR91-016, Politec. di Milano*, pp. 1–21, 1991.

[104] M. Dorigo, V. Maniezzo, and A. Colorni, “Ant system: optimization by a colony of cooperating agents,” *IEEE Trans. Syst. Man Cybern. Part B*, vol. 26, no. 1, pp. 29–41, 1996, doi: 10.1109/3477.484436.

[105] A. K. Mandal and S. Dehuri, “A Survey on Ant Colony Optimization for Solving Some of the Selected NP-Hard Problem,” 2020, pp. 85–100.

[106] M. Dorigo and L. M. Gambardella, “Ant colony system: a cooperative learning approach to the traveling salesman problem,” *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 53–66, Apr. 1997, doi: 10.1109/4235.585892.

[107] Y.-H. Huang, C. A. Blazquez, S.-H. Huang, G. Paredes-Belmar, and G. Latorre-Nuñez, “Solving the Feeder Vehicle Routing Problem using ant colony optimization,” *Comput. Ind. Eng.*, vol. 127, pp. 520–535, Jan. 2019, doi: 10.1016/j.cie.2018.10.037.

[108] I. D. I. D. Ariyasingha and T. G. I. Fernando, “A New Multi-Objective Ant Colony Optimisation Algorithm for Solving the Quadratic Assignment Problem,” *Vidyodaya J. Sci.*, vol. 22, no. 1, p. 1, Nov. 2019, doi: 10.4038/vjs.v22i1.6060.

[109] K. S. P. Deepalakshmi, “Role and Impacts of Ant Colony Optimization in Job Shop Scheduling Problems,” in *Evolutionary Computation in Scheduling*, I. R. Amir H. Gandomi, Ali Emrouznejad, Mo M. Jamshidi, Kalyanmoy Deb, Ed. John Wiley & Sons, 2020, pp. 11–34.

[110] I. Ben Mansour, I. Alaya, and M. Tagina, “A gradual weight-based ant colony approach for solving the multiobjective multidimensional knapsack problem,” *Evol. Intell.*, vol. 12, no. 2, pp. 253–272, 2019, doi: 10.1007/s12065-019-00222-9.

[111] F. E. B. Otero, A. A. Freitas, and C. G. Johnson, “cAnt-Miner: An Ant Colony Classification Algorithm to Cope with Continuous Attributes,” in *Ant Colony Optimization and Swarm Intelligence*, vol. 5217 LNCS, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 48–59.

[112] K. Kwarciak, M. Radom, and P. Formanowicz, “A multilevel ant colony optimization algorithm for classical and isothermic DNA sequencing by hybridization with multiplicity information available,” *Comput. Biol. Chem.*, vol. 61, pp. 109–120, Apr. 2016, doi: 10.1016/j.compbiolchem.2016.01.010.

[113] A. Akhtar, “Evolution of Ant Colony Optimization Algorithm -- A Brief Literature Review,” Aug. 2019.

[114] M. Pedemonte, S. Nesmachnow, and H. Cancela, “A survey on parallel ant colony optimization,” *Appl. Soft Comput.*, vol. 11, no. 8, pp. 5181–5197, Dec. 2011, doi: 10.1016/j.asoc.2011.05.042.

[115] T. Bäck, H. P. Schwefel, and F. Hoffmeister, “A survey of Evolutionary Strategies,” *Proc. Fourth Int. Conf. Genet. Algorithms*, vol. 9, 1991.

[116] H.-G. Beyer and H.-P. Schwefel, “Evolution strategies – A comprehensive introduction,” *Nat. Comput.*, vol. 1, no. 1, pp. 3–52, 2002, doi: 10.1023/A:1015059928466.

[117] J. A. Lozano, “An Introduction to Evolutionary Algorithms,” 2002, pp. 3–25.

[118] H.-P. Schwefel and G. Rudolph, “Contemporary evolution strategies,” 1995, pp. 891–907.

[119] N. Hansen, D. V. Arnold, and A. Auger, “Evolution Strategies,” in *Springer Handbook of Computational Intelligence*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp.

871–898.

[120] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, “Evolution Strategies as a Scalable Alternative to Reinforcement Learning,” pp. 1–13, Mar. 2017.

[121] E. Mezura-Montes and C. A. Coello Coello, “A simple multimembered evolution strategy to solve constrained optimization problems,” *IEEE Trans. Evol. Comput.*, vol. 9, no. 1, pp. 1–17, 2005, doi: 10.1109/TEVC.2004.836819.

[122] J. J. Korczak, P. Lipiński, and P. Roger, “Evolution Strategy in Portfolio Optimization,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2310, 2002, pp. 156–167.

[123] O. M. Shir, “Niching in Derandomized Evolution Strategies and its Applications in Quantum Control,” Leiden University, 2008.

[124] S. Hosseini and A. Al Khaled, “A survey on the Imperialist Competitive Algorithm metaheuristic: Implementation in engineering domain and directions for future research,” *Appl. Soft Comput. J.*, vol. 24, pp. 1078–1094, 2014, doi: 10.1016/j.asoc.2014.08.024.

[125] E. Atashpaz-Gargari and C. Lucas, “Imperialist competitive algorithm: An algorithm for optimization inspired by imperialistic competition,” *2007 IEEE Congr. Evol. Comput. CEC 2007*, pp. 4661–4667, 2007, doi: 10.1109/CEC.2007.4425083.

[126] Q. Fang, H. Nguyen, X.-N. Bui, and T. Nguyen-Thoi, “Prediction of Blast-Induced Ground Vibration in Open-Pit Mines Using a New Technique Based on Imperialist Competitive Algorithm and M5Rules,” *Nat. Resour. Res.*, vol. 29, no. 2, pp. 791–806, Apr. 2020, doi: 10.1007/s11053-019-09577-3.

[127] B. Tashayo, K. Behzadafshar, M. Soltani Tehrani, H. Afkhami Banayem, M. H. Hashemi, and S. S. Taghavi Nezhad, “Feasibility of imperialist competitive algorithm to predict the surface settlement induced by tunneling,” *Eng. Comput.*, vol. 35, no. 3, pp. 917–923, Jul. 2019, doi: 10.1007/s00366-018-0641-3.

[128] Z. Aliniya and S. A. Mirroshandel, “A novel combinatorial merge-split approach for automatic clustering using imperialist competitive algorithm,” *Expert Syst. Appl.*, vol. 117, pp. 243–266, 2019, doi: 10.1016/j.eswa.2018.09.050.

[129] Z. Aliniya and S. A. Mirroshandel, “A novel combinatorial merge-split approach for automatic clustering using imperialist competitive algorithm,” *Expert Syst. Appl.*, vol. 117, no. January 2018, pp. 243–266, Mar. 2019, doi: 10.1016/j.eswa.2018.09.050.

[130] Y. C. Ho and D. L. Pepyne, “Simple Explanation of the No-Free-Lunch Theorem and Its Implications,” *J. Optim. Theory Appl.*, vol. 115, no. 3, pp. 549–570, Dec. 2002, doi: 10.1023/A:1021251113462.

[131] J. A. Parejo, A. Ruiz-Cortés, S. Lozano, and P. Fernandez, “Metaheuristic optimization frameworks: a survey and benchmarking,” *Soft Comput.*, vol. 16, no. 3, pp. 527–561, Mar. 2012, doi: 10.1007/s00500-011-0754-8.

[132] M. A. Lopes Silva, S. R. de Souza, M. J. Freitas Souza, and M. F. de França Filho, “Hybrid metaheuristics and multi-agent systems for solving optimization problems: A review of frameworks and a comparative analysis,” *Appl. Soft Comput.*, vol. 71, pp. 433–459, Oct. 2018, doi: 10.1016/j.asoc.2018.06.050.

[133] E. Alba *et al.*, “MALLBA: A Library of Skeletons for Combinatorial Optimisation,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2400, 2002, pp. 927–932.

[134] E. Alba, G. Luque, J. G. Nieto, G. Ordonez, and G. Leguizamon, “MALLBA: a software library to design efficient optimisation algorithms,” *Int. J. Innov. Comput. Appl.*, vol. 1,

no. 1, p. 74, 2007, doi: 10.1504/IJICA.2007.013403.

[135] S. Cahon, N. Melab, and E.-G. Talbi, “ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics,” *J. Heuristics*, vol. 10, no. 3, pp. 357–380, May 2004, doi: 10.1023/B:HEUR.0000026900.92269.ec.

[136] S. Wagner and M. Affenzeller, “HeuristicLab: A Generic and Extensible Optimization Environment,” in *Adaptive and Natural Computing Algorithms*, Vienna: Springer-Verlag, 2005, pp. 538–541.

[137] C. Gagné and M. Parizeau, “Genericity in evolutionary computation software tools: Principles and case-study,” *Int. J. Artif. Intell. Tools*, vol. 15, no. 2, pp. 173–194, 2006, doi: 10.1142/S021821300600262X.

[138] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, “JCLEC: a Java framework for evolutionary computation,” *Soft Comput.*, vol. 12, no. 4, pp. 381–392, Oct. 2007, doi: 10.1007/s00500-007-0172-0.

[139] O. Skalicka, “Java Combinatorial Optimization Platform,” 2014. [Online]. Available: http://jcop.sourceforge.net/en/index.html.

[140] L. S. Coelho, I. M., Munhoz, P. L. A., Haddad, M. N., Coelho, V. N., Silva, M. M., Souza, M. J. F., Ochi, “OptFrame: A computational framework for combinatorial optimization problems,” *VII ALIO/EURO Work. Appl. Comb. Optim.*, no. May, 2011.

[141] M. Kronfeld, H. Planatscher, and A. Zell, “The EvA2 Optimization Framework,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6073 LNCS, 2010, pp. 247–250.

[142] J. J. Durillo and A. J. Nebro, “JMetal: A Java framework for multi-objective optimization,” *Adv. Eng. Softw.*, vol. 42, no. 10, pp. 760–771, 2011, doi: 10.1016/j.advengsoft.2011.05.014.

[143] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, “Opt4J,” in *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO ’11*, 2011, p. 1723, doi: 10.1145/2001576.2001808.

[144] D. R. White, “Software review: the ECJ toolkit,” *Genet. Program. Evolvable Mach.*, vol. 13, no. 1, pp. 65–67, Mar. 2012, doi: 10.1007/s10710-011-9148-z.

[145] P. D. I. Milano, “HyperSpark : A Framework for Scalable Execution of Computationally-Intensive Algorithms over Spark,” POLITECNICO DI MILANO, 2016.

[146] H. De Beukelaer, G. F. Davenport, G. De Meyer, and V. Fack, “JAMES: An object-oriented Java framework for discrete optimization using local search metaheuristics,” *Softw. Pract. Exp.*, vol. 47, no. 6, pp. 921–938, Jun. 2017, doi: 10.1002/spe.2459.

[147] H. Faris, I. Aljarah, S. Mirjalili, P. A. Castillo, and J. J. Merelo, “EvoloPy: An Open-source Nature-inspired Optimization Framework in Python,” in *Proceedings of the 8th International Joint Conference on Computational Intelligence*, 2016, vol. 1, no. Ijcci, pp. 171–177, doi: 10.5220/0006048201710177.

[148] C. Barba-González *et al.*, “jMetalSP: A framework for dynamic multi-objective big data optimization,” *Appl. Soft Comput.*, vol. 69, pp. 737–748, Aug. 2018, doi: 10.1016/j.asoc.2017.05.004.

[149] A. Benítez-Hidalgo, A. J. Nebro, J. García-Nieto, I. Oregi, and J. Del Ser, “jMetalPy: A Python framework for multi-objective optimization with metaheuristics,” *Swarm Evol. Comput.*, vol. 51, p. 100598, Dec. 2019, doi: 10.1016/j.swevo.2019.100598.

[150] N. Melab, T. Van Luong, K. Boufaras, and E.-G. Talbi, “ParadisEO-MO-GPU: a framework for parallel GPU-based local search metaheuristics,” in *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference - GECCO ’13*,

2013, p. 1189, doi: 10.1145/2463372.2465804.
[151] S. Wagner *et al.*, “Architecture and Design of the HeuristicLab Optimization Environment,” in *1st Australian Conference on the Applications of Systems Engineering ACASE’12*, 2014, pp. 197–261.
[152] A. Elyasaf and M. Sipper, “Software review: the HeuristicLab framework,” *Genet. Program. Evolvable Mach.*, vol. 15, no. 2, pp. 215–218, Jun. 2014, doi: 10.1007/s10710-014-9214-4.
[153] S. Luke, “ECJ then and now,” in *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO ’17*, 2017, pp. 1223–1230, doi: 10.1145/3067695.3082467.
[154] M. Ciavotta, S. Krstic, D. A. Tamburri, and W.-J. Van Den Heuvel, “HyperSpark: A Data-Intensive Programming Environment for Parallel Metaheuristics,” in *2019 IEEE International Congress on Big Data (BigDataCongress)*, 2019, no. July, pp. 85–92, doi: 10.1109/BigDataCongress.2019.00024.
[155] H. De Beukelaer, G. F. Davenport, G. De Meyer, and V. Fack, “JAMES: An object-oriented Java framework for discrete optimization using local search metaheuristics,” *Softw. Pract. Exp.*, vol. 47, no. 6, pp. 921–938, Jun. 2017, doi: 10.1002/spe.2459.
[156] C. Barba-González, A. J. Nebro, A. Benítez-Hidalgo, J. García-Nieto, and J. F. Aldana-Montes, “On the design of a framework integrating an optimization engine with streaming technologies,” *Futur. Gener. Comput. Syst.*, vol. 107, pp. 538–550, 2020, doi: 10.1016/j.future.2020.02.020.
[157] P. C. Gilmore and R. E. Gomory, “The Theory and Computation of Knapsack Functions,” *Oper. Res.*, vol. 14, no. 6, pp. 1045–1074, Dec. 1966, doi: 10.1287/opre.14.6.1045.
[158] B. Gavish and H. Pirkul, “Allocation of databases and processors in a distributed data processing,” in *Management of Distributed Data Processing*, J. Akola, Ed. Amsterdam: North-Holland, 1982, pp. 215–231.
[159] W. Shish, “A branch & bound method for the multiconstraint zero-one knapsack problem,” *J. Oper. Res. Soc.*, vol. 30, pp. 369–378, 1979.
[160] M. Vasquez and J. K. Hao, “A ‘logic-constrained’ knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite,” *Comput. Optim. Appl.*, vol. 20, no. 2, pp. 137–157, 2001, doi: 10.1023/A:1011203002719.
[161] C. C. Petersen, “Computational Experience with Variants of the Balas Algorithm Applied to the Selection of R&D Projects,” *Manage. Sci.*, vol. 13, no. 9, pp. 609–772, 1967.
[162] H. M. Weingartner and D. N. Ness, “Methods for the Solution of the Multidimensional 0/1 Knapsack Problem,” *Oper. Res.*, vol. 15, no. 1, pp. 83–103, Feb. 1967, doi: 10.1287/opre.15.1.83.
[163] T. Setzer and S. M. Blanc, “Empirical orthogonal constraint generation for Multidimensional 0/1 Knapsack Problems,” *Eur. J. Oper. Res.*, vol. 282, no. 1, pp. 58–70, 2020, doi: 10.1016/j.ejor.2019.09.016.
[164] G. B. Dantzig and J. H. Ramser, “The Truck Dispatching Problem,” *Manage. Sci.*, vol. 6, no. 1, pp. 80–91, Oct. 1959, doi: 10.1287/mnsc.6.1.80.
[165] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, “Erratum: The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization,” *J. Oper. Res. Soc.*, vol. 37, no. 6, pp. 655–655, Jun. 1986, doi: 10.1057/jors.1986.117.
[166] S. Karakatič and V. Podgorelec, “A survey of genetic algorithms for solving multi depot vehicle routing problem,” *Appl. Soft Comput. J.*, vol. 27, pp. 519–532, 2015, doi:

10.1016/j.asoc.2014.11.005.
[167] S. Samsuddin, M. S. Othman, and L. M. Yusuf, "a Review of Single and Population-Based Metaheuristic Algorithms Solving Multi Depot Vehicle Routing Problem," *Int. J. Softw. Eng. Comput. Syst.*, vol. 4, no. 2, pp. 80–93, 2018, doi: 10.15282/ijsecs.4.2.2018.6.0050.
[168] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freem. 1979.
[169] N. Sharma and M. Monika, "A Literature Survey on Multi Depot Vehicle Routing Problem," *IJSRD -International J. Sci. Res. Dev.*, vol. 3, no. 04online, pp. 2321–613, 2015.
[170] L. F. Galindres-Guancha, E. M. Toro-Ocampo, and R. A. Gallego-Rendón, "Multi-objective MDVRP solution considering route balance and cost using the ILS metaheuristic," *Int. J. Ind. Eng. Comput.*, vol. 9, no. 1, pp. 33–46, 2018, doi: 10.5267/j.ijiec.2017.5.002.
[171] J. Renaud, G. Laporte, and F. F. Boctor, "A tabu search heuristic for the multi-depot vehicle routing problem," *Comput. Oper. Res.*, vol. 23, no. 3, pp. 229–235, 1996, doi: 10.1016/0305-0548(95)O0026-P.
[172] P. Stodola, "Hybrid ant colony optimization algorithm applied to the multi-depot vehicle routing problem," *Nat. Comput.*, vol. 6, 2020, doi: 10.1007/s11047-020-09783-6.
[173] G. Clarke and J. W. Wright, "Scheduling of Vehicles from a Central Depot to a Number of Delivery Points," *Oper. Res.*, vol. 12, no. 4, pp. 568–581, Aug. 1964, doi: 10.1287/opre.12.4.568.
[174] P. Surekha and S. Sumathi, "Solution To Multi-Depot Vehicle Routing Problem Using Genetic Algorithms," *World Appl. Program.*, no. 13, pp. 118–131, 2011.
[175] B. E. Gillett and J. G. Johnson, "Multi-terminal vehicle-dispatch algorithm," *Omega*, vol. 4, no. 6, pp. 711–718, 1976, doi: 10.1016/0305-0483(76)90097-9.
[176] D. Gulczynski, B. Golden, and E. Wasil, "The multi-depot split delivery vehicle routing problem: An integer programming-based heuristic, new test problems, and computational results," *Comput. Ind. Eng.*, vol. 61, no. 3, pp. 794–804, 2011, doi: 10.1016/j.cie.2011.05.012.
[177] A. Imran, "A Variable Neighborhood Search-Based Heuristic for the Multi-Depot Vehicle Routing Problem," *J. Tek. Ind.*, vol. 15, no. 2, pp. 95–102, Dec. 2013, doi: 10.9744/jti.15.2.95-102.
[178] Y. M. Shen and R. M. Chen, "Optimal multi-depot location decision using particle swarm optimization," *Adv. Mech. Eng.*, vol. 9, no. 8, pp. 1–15, 2017, doi: 10.1177/1687814017717663.
[179] S. B. Sarathi Barma, J. Dutta, and A. Mukherjee, "A 2-opt guided discrete antlion optimization algorithm for multi-depot vehicle routing problem," *Decis. Mak. Appl. Manag. Eng.*, vol. 2, no. 2, pp. 112–125, 2019, doi: 10.31181/dmame1902089b.
[180] B. Yao, C. Chen, X. Song, and X. Yang, "Fresh seafood delivery routing problem using an improved ant colony optimization," *Ann. Oper. Res.*, vol. 273, no. 1–2, pp. 163–186, 2019, doi: 10.1007/s10479-017-2531-2.
[181] A. Otto, N. Agatz, J. Campbell, B. Golden, and E. Pesch, "Optimization approaches for civil applications of unmanned aerial vehicles (UAVs) or aerial drones: A survey," *Networks*, vol. 72, no. 4, pp. 411–458, Dec. 2018, doi: 10.1002/net.21818.
[182] G. Q. Li, X. G. Zhou, J. Yin, and Q. Y. Xiao, "An UAV scheduling and planning method for post-disaster survey," *ISPRS - Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci.*, vol. XL–2, no. 2, pp. 169–172, Nov. 2014, doi: 10.5194/isprsarchives-XL-2-169-2014.

[183] H. Zhang *et al.*, “Scheduling methods for unmanned aerial vehicle based delivery systems,” in *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, 2014, pp. 6C1-1-6C1-9, doi: 10.1109/DASC.2014.6979499.

[184] I. Dzalbs and T. Kalganova, “Aerial Surveying Problem dataset,” *Figshare*. 2020, doi: 10.6084/m9.figshare.12770177.

[185] “Afwerx challange home page.” [Online]. Available: https://afwerxchallenge.com/.

[186] T. K. Ivars Dzalbs, “Supply Chain Logistics Problem Dataset.” [Online]. Available: https://brunel.figshare.com/articles/Supply_Chain_Logistics_Problem_Dataset/7558679.

[187] S. Tsutsui, “cAS: Ant Colony Optimization with Cunning Ants,” in *System*, 2006, pp. 162–171.

[188] F. Glover and G. A. Kochenberger, “Critical Event Tabu Search for Multidimensional Knapsack Problems,” in *Meta-Heuristics*, Boston, MA: Springer US, 1996, pp. 407–427.

[189] J. H. Drake, “Benchmark instances for the Multidimensional Knapsack Problem,” 2020. [Online]. Available: https://www.researchgate.net/publication/271198281_Benchmark_instances_for_the_Multidimensional_Knapsack_Problem.

[190] M. Abdechiri, H. Bahrami, and K. Faez, “Adaptive Imperialist Competitive Algorithm (AICA),” *Proc. 9th IEEE Int. Conf. Cogn. Informatics, ICCI 2010*, pp. 940–945, 2010, doi: 10.1109/COGINF.2010.5599776.

[191] M. R. Maheri and M. Talezadeh, “An Enhanced Imperialist Competitive Algorithm for optimum design of skeletal structures,” *Swarm Evol. Comput.*, vol. 40, no. November, pp. 24–36, 2018, doi: 10.1016/j.swevo.2017.12.001.

[192] L. D. Afonso, V. C. Mariani, and L. Dos Santos Coelho, “Modified imperialist competitive algorithm based on attraction and repulsion concepts for reliability-redundancy optimization,” *Expert Syst. Appl.*, vol. 40, no. 9, pp. 3794–3802, 2013, doi: 10.1016/j.eswa.2012.12.093.

[193] A. Rabiee, M. Sadeghi, and J. Aghaei, “Modified imperialist competitive algorithm for environmental constrained energy management of microgrids,” *J. Clean. Prod.*, vol. 202, pp. 273–292, 2018, doi: 10.1016/j.jclepro.2018.08.129.

[194] J. J. Liang and P. N. Suganthan, “Dynamic Multi-Swarm Particle Swarm Optimizer with Local Search,” in *2005 IEEE Congress on Evolutionary Computation*, 2005, vol. 1, no. May 2014, pp. 522–528, doi: 10.1109/CEC.2005.1554727.

[195] M. Li, D. Lei, and H. Xiong, “An imperialist competitive algorithm with the diversified operators for many-objective scheduling in flexible job shop,” *IEEE Access*, vol. 7, pp. 29553–29562, 2019, doi: 10.1109/ACCESS.2019.2895348.

[196] S. Karimi, Z. Ardalan, B. Naderi, and M. Mohammadi, “Scheduling flexible job-shops with transportation times: Mathematical models and a hybrid imperialist competitive algorithm,” *Appl. Math. Model.*, vol. 41, pp. 667–682, 2017, doi: 10.1016/j.apm.2016.09.022.

[197] Z. Ardalan, S. Karimi, O. Poursabzi, and B. Naderi, “A novel imperialist competitive algorithm for generalized traveling salesman problems,” *Appl. Soft Comput. J.*, vol. 26, pp. 546–555, 2015, doi: 10.1016/j.asoc.2014.08.033.

[198] J. L. Lin, H. C. Chuan, Y. H. Tsai, and C. W. Cho, “Improving imperialist competitive algorithm with local search for global optimization,” *Proc. - Asia Model. Symp. 2013 7th Asia Int. Conf. Math. Model. Comput. Simulation, AMS 2013*, pp. 61–64, 2013, doi: 10.1109/AMS.2013.14.

[199] S. J.MousaviRad, F. Akhlaghian Tab, and K. Mollazade, “Application of Imperialist Competitive Algorithm for Feature Selection: A Case Study on Bulk Rice Classification,” *Int. J. Comput. Appl.*, vol. 40, no. 16, pp. 41–48, 2012, doi: 10.5120/5068-7485.
[200] D. S. Huang, K. Han, and A. Hussain, “Improved Binary Imperialist Competition Algorithm for Feature Selection from Gene Expression Data,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9773, p. V, 2016, doi: 10.1007/978-3-319-42297-8.
[201] M. Mirhosseini and H. Nezamabadi-pour, “BICA: a binary imperialist competitive algorithm and its application in CBIR systems,” *Int. J. Mach. Learn. Cybern.*, vol. 9, no. 12, pp. 2043–2057, 2018, doi: 10.1007/s13042-017-0686-4.
[202] S. Nozarian, H. Soltanpoor, and M. Vafaei, “A Binary Model on the Basis of Imperialist Competitive Algorithm in Order to Solve the Problem of Knapsack 1-0,” *Proc. third Int. Conf. Serv. Emerg. Mark. 2012*, vol. 34, pp. 67–71, 2012.
[203] S. Xu, Y. Wang, and A. Huang, “Application of imperialist competitive algorithm on solving the traveling salesman problem,” *Algorithms*, vol. 7, no. 2, pp. 229–242, 2014, doi: 10.3390/a7020229.
[204] S. H. Mirhoseini, S. M. Hosseini, M. Ghanbari, and M. Ahmadi, “A new improved adaptive imperialist competitive algorithm to solve the reconfiguration problem of distribution systems for loss reduction and voltage profile improvement,” *Int. J. Electr. Power Energy Syst.*, vol. 55, pp. 128–143, 2014, doi: 10.1016/j.ijepes.2013.08.028.
[205] S. S. Ray, S. Bandyopadhyay, and S. K. Pal, “Genetic operators for combinatorial optimization in TSP and microarray gene ordering,” *Appl. Intell.*, vol. 26, no. 3, pp. 183–195, 2007, doi: 10.1007/s10489-006-0018-y.
[206] “Multiple Depot VRP Instances,” *University of Malaga, Spain.* [Online]. Available: http://neo.lcc.uma.es/vrp/vrp-instances/multiple-depot-vrp-instances/.
[207] J. Skackauskas, T. Kalganova, I. Dear, and M. Janakram, “Dynamic Impact for Ant Colony Optimization algorithm,” Feb. 2020.
[208] M. Abdel-Basset, D. El-Shahat, and A. K. Sangaiah, “A modified nature inspired meta-heuristic whale optimization algorithm for solving 0–1 knapsack problem,” *Int. J. Mach. Learn. Cybern.*, vol. 10, no. 3, pp. 495–514, 2019, doi: 10.1007/s13042-017-0731-3.
[209] J. Liu, C. Wu, J. Cao, X. Wang, and K. L. Teo, “A Binary differential search algorithm for the 0–1 multidimensional knapsack problem,” *Appl. Math. Model.*, vol. 40, no. 23–24, pp. 9788–9805, 2016, doi: 10.1016/j.apm.2016.06.002.
[210] M. Chih, “Self-adaptive check and repair operator-based particle swarm optimization for the multidimensional knapsack problem,” *Appl. Soft Comput. J.*, vol. 26, pp. 378–389, 2015, doi: 10.1016/j.asoc.2014.10.030.
[211] X. Lai, J. K. Hao, F. Glover, and Z. Lü, “A two-phase tabu-evolutionary algorithm for the 0–1 multidimensional knapsack problem,” *Inf. Sci. (Ny).*, vol. 436–437, pp. 282–301, 2018, doi: 10.1016/j.ins.2018.01.026.
[212] X. Kong, L. Gao, H. Ouyang, and S. Li, “Solving large-scale multidimensional knapsack problems with a new binary harmony search algorithm,” *Comput. Oper. Res.*, vol. 63, pp. 7–22, 2015, doi: 10.1016/j.cor.2015.04.018.
[213] A. A. Ferjani and N. Liouane, “Logic gate-based evolutionary algorithm for the multidimensional knapsack problem,” *2017 Int. Conf. Control. Autom. Diagnosis, ICCAD 2017*, pp. 164–168, 2017, doi: 10.1109/CADIAG.2017.8075650.
[214] M. Daniel Valadao Baroni and F. M. Varejao, “A shuffled complex evolution algorithm for the multidimensional knapsack problem using core concept,” *2016 IEEE Congr. Evol.*

Comput. CEC 2016, pp. 2718–2723, 2016, doi: 10.1109/CEC.2016.7744131.

[215] J. H. Drake, E. Özcan, and E. K. Burke, "A case study of controlling crossover in a selection hyper-heuristic framework using the multidimensional Knapsack problem," *Evol. Comput.*, vol. 24, no. 1, pp. 113–141, 2016, doi: 10.1162/EVCO_a_00145.

[216] K. K. Bhattacharjee and S. P. Sarmah, "Modified swarm intelligence based techniques for the knapsack problem," *Appl. Intell.*, vol. 46, no. 1, pp. 158–179, 2017, doi: 10.1007/s10489-016-0822-y.

[217] F. B. De Oliveira, R. Enayatifar, H. J. Sadaei, F. G. Guimarães, and J. Y. Potvin, "A cooperative coevolutionary algorithm for the Multi-Depot Vehicle Routing Problem," *Expert Syst. Appl.*, vol. 43, pp. 117–130, 2016, doi: 10.1016/j.eswa.2015.08.030.

[218] M. E. H. Sadati, D. Aksen, and N. Aras, "The r-interdiction selective multi-depot vehicle routing problem," *Int. Trans. Oper. Res.*, vol. 27, no. 2, pp. 835–866, 2020, doi: 10.1111/itor.12669.

[219] M. Esmaeilikia, B. Fahimnia, J. Sarkis, K. Govindan, A. Kumar, and J. Mo, "Tactical supply chain planning models with inherent flexibility: definition and review," *Ann. Oper. Res.*, vol. 244, no. 2, pp. 407–427, Sep. 2016, doi: 10.1007/s10479-014-1544-3.

[220] M. Schyns, "An ant colony system for responsive dynamic vehicle routing," *Eur. J. Oper. Res.*, vol. 245, no. 3, pp. 704–718, Sep. 2015, doi: 10.1016/j.ejor.2015.04.009.

[221] Z. Zhang, N. Zhang, and Z. Feng, "Multi-satellite control resource scheduling based on ant colony optimization," *Expert Syst. Appl.*, vol. 41, no. 6, pp. 2816–2823, May 2014, doi: 10.1016/j.eswa.2013.10.014.

[222] N. Azad, A. Aazami, A. Papi, and A. Jabbarzadeh, "A two-phase genetic algorithm for incorporating environmental considerations with production, inventory and routing decisions in supply chain networks," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '19*, 2019, pp. 41–42, doi: 10.1145/3319619.3326781.

[223] W.-C. Yeh and M.-C. Chuang, "Using multi-objective genetic algorithm for partner selection in green supply chain problems," *Expert Syst. Appl.*, vol. 38, no. 4, pp. 4244–4253, Apr. 2011, doi: 10.1016/j.eswa.2010.09.091.

[224] A. M. Fathollahi-Fard, K. Govindan, M. Hajiaghaei-Keshteli, and A. Ahmadi, "A green home health care supply chain: New modified simulated annealing algorithms," *J. Clean. Prod.*, vol. 240, p. 118200, Dec. 2019, doi: 10.1016/j.jclepro.2019.118200.

[225] A. Mohammed and S. Duffuaa, "A Meta-Heuristic Algorithm Based on Simulated Annealing for Designing Multi-Objective Supply Chain Systems," in *2019 Industrial & Systems Engineering Conference (ISEC)*, 2019, pp. 1–6, doi: 10.1109/IASEC.2019.8686517.

[226] C. B. Kalayci and C. Kaya, "An ant colony system empowered variable neighborhood search algorithm for the vehicle routing problem with simultaneous pickup and delivery," *Expert Syst. Appl.*, vol. 66, pp. 163–175, 2016, doi: 10.1016/j.eswa.2016.09.017.

[227] S. Zhang, W. Zhang, Y. Gajpal, and S. S. Appadoo, "Ant Colony Algorithm for Routing Alternate Fuel Vehicles in Multi-depot Vehicle Routing Problem," Springer Singapore, 2019, pp. 251–260.

[228] E. Bottani, T. Murino, M. Schiavo, and R. Akkerman, "Resilient food supply chain design: Modelling framework and metaheuristic solution approach," *Comput. Ind. Eng.*, vol. 135, no. October 2018, pp. 177–198, Sep. 2019, doi: 10.1016/j.cie.2019.05.011.

[229] V. V. Panicker, M. V. Reddy, and R. Sridharan, "Development of an ant colony

optimisation-based heuristic for a location-routing problem in a two-stage supply chain," *Int. J. Value Chain Manag.*, vol. 9, no. 1, p. 38, 2018, doi: 10.1504/IJVCM.2018.091109.

[230] F. Valdez, F. Moreno, and P. Melin, "A Comparison of ACO, GA and SA for Solving the TSP Problem," 2020, pp. 181–189.

[231] K.-J. Wang and C.-H. Lee, "A revised ant algorithm for solving location–allocation problem with risky demand in a multi-echelon supply chain network," *Appl. Soft Comput.*, vol. 32, pp. 311–321, Jul. 2015, doi: 10.1016/j.asoc.2015.03.046.

[232] L. Wong and N. H. Moin, "Ant Colony Optimization For Split Delivery Inventory Routing Problem," *Malaysian J. Comput. Sci.*, vol. 30, no. 4, pp. 333–348, Dec. 2017, doi: 10.22452/mjcs.vol30no4.5.

[233] P. F. Vieira, S. M. Vieira, M. I. Gomes, A. P. Barbosa-Póvoa, and J. M. C. Sousa, "Designing closed-loop supply chains with nonlinear dimensioning factors using ant colony optimization," *Soft Comput.*, vol. 19, no. 8, pp. 2245–2264, Aug. 2015, doi: 10.1007/s00500-014-1405-7.

[234] P. Yelmewad, A. Kumar, and B. Talawar, "MMAS on GPU for Large TSP Instances," in *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2019, pp. 1–6, doi: 10.1109/ICCCNT45670.2019.8944770.

[235] W. Zhou, F. He, and Z. Zhang, "A GPU-based parallel MAX-MIN Ant System algorithm with grouped roulette wheel selection," in *2017 IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2017, pp. 360–365, doi: 10.1109/CSCWD.2017.8066721.

[236] M. Randall and A. Lewis, "A Parallel Implementation of Ant Colony Optimization," *J. Parallel Distrib. Comput.*, vol. 62, no. 9, pp. 1421–1432, Sep. 2002, doi: 10.1006/jpdc.2002.1854.

[237] A. Prakasam and N. Savarimuthu, "Metaheuristic algorithms and probabilistic behaviour: a comprehensive analysis of Ant Colony Optimization and its variants," *Artif. Intell. Rev.*, vol. 45, no. 1, pp. 97–130, 2016, doi: 10.1007/s10462-015-9441-y.

[238] Ş. Gülcü, M. Mahi, Ö. K. Baykan, and H. Kodaz, "A parallel cooperative hybrid method based on ant colony optimization and 3-Opt algorithm for solving traveling salesman problem," *Soft Comput.*, vol. 22, no. 5, pp. 1669–1685, Mar. 2018, doi: 10.1007/s00500-016-2432-3.

[239] G. Weidong, F. Jinqiao, W. Yazhou, Z. Hongjun, and H. Jidong, "Parallel Performance of an Ant Colony Optimization Algorithm for TSP," in *2015 8th International Conference on Intelligent Computation Technology and Automation (ICICTA)*, 2015, pp. 625–629, doi: 10.1109/ICICTA.2015.159.

[240] Y. Tan and K. Ding, "A Survey on GPU-Based Implementation of Swarm Intelligence Algorithms," *IEEE Trans. Cybern.*, vol. 46, no. 9, pp. 2028–2041, Sep. 2016, doi: 10.1109/TCYB.2015.2460261.

[241] M. Sato, S. Tsutsui, N. Fujimoto, Y. Sato, and M. Namiki, "First results of performance comparisons on many-core processors in solving QAP with ACO," pp. 1477–1478, 2014, doi: 10.1145/2598394.2602274.

[242] D. Thiruvady, A. T. Ernst, and G. Singh, "Parallel ant colony optimization for resource constrained job scheduling," *Ann. Oper. Res.*, vol. 242, no. 2, pp. 355–372, Jul. 2016, doi: 10.1007/s10479-014-1577-7.

[243] G. D. Guerrero, J. M. Cecilia, A. Llanes, J. M. García, M. Amos, and M. Ujaldón, "Comparative evaluation of platforms for parallel Ant Colony Optimization," *J.*

Supercomput., vol. 69, no. 1, pp. 318–329, Jul. 2014, doi: 10.1007/s11227-014-1154-5.
[244] Q. Yang, L. Fang, and X. Duan, “RMACO :a randomly matched parallel ant colony optimization,” *World Wide Web*, vol. 19, no. 6, pp. 1009–1022, Nov. 2016, doi: 10.1007/s11280-015-0369-6.
[245] Y. Zhou, F. He, N. Hou, and Y. Qiu, “Parallel ant colony optimization on multi-core SIMD CPUs,” *Futur. Gener. Comput. Syst.*, vol. 79, pp. 473–487, 2018, doi: 10.1016/j.future.2017.09.073.
[246] A. Llanes, C. Vélez, A. M. Sánchez, H. Pérez-Sánchez, and J. M. Cecilia, “Parallel Ant Colony Optimization for the HP Protein Folding Problem,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9656, F. Ortuño and I. Rojas, Eds. Cham: Springer International Publishing, 2016, pp. 615–626.
[247] H. Fingler, E. N. Cáceres, H. Mongelli, and S. W. Song, “A CUDA based Solution to the Multidimensional Knapsack Problem Using the Ant Colony Optimization,” *Procedia Comput. Sci.*, vol. 29, no. 30, pp. 84–94, 2014, doi: 10.1016/j.procs.2014.05.008.
[248] D. Markvica, C. Schauer, and G. R. Raidl, “CPU Versus GPU Parallelization of an Ant Colony Optimization for the Longest Common Subsequence Problem,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9520, R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, Eds. Cham: Springer International Publishing, 2015, pp. 401–408.
[249] J. M. Cecilia, A. Llanes, J. L. Abellán, J. Gómez-Luna, L. W. Chang, and W. M. W. Hwu, “High-throughput Ant Colony Optimization on graphics processing units,” *J. Parallel Distrib. Comput.*, vol. 113, pp. 261–274, 2018, doi: 10.1016/j.jpdc.2017.12.002.
[250] R. Skinderowicz, “Implementing a GPU-based parallel MAX–MIN Ant System,” *Futur. Gener. Comput. Syst.*, vol. 106, pp. 277–295, May 2020, doi: 10.1016/j.future.2020.01.011.
[251] L. Dawson and I. A. Stewart, “Accelerating ant colony optimization-based edge detection on the GPU using CUDA,” in *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 1736–1743, doi: 10.1109/CEC.2014.6900638.
[252] Y. Huo and J. X. Huang, “Parallel Ant Colony Optimization for Flow Shop Scheduling Subject to Limited Machine Availability,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 756–765, doi: 10.1109/IPDPSW.2016.151.
[253] F. Tirado, A. Urrutia, and R. J. Barrientos, “Using a coprocessor to solve the Ant Colony Optimization algorithm,” in *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, 2015, vol. 2016-Febru, pp. 1–6, doi: 10.1109/SCCC.2015.7416584.
[254] F. Tirado, R. J. Barrientos, P. González, and M. Mora, “Efficient exploitation of the Xeon Phi architecture for the Ant Colony Optimization (ACO) metaheuristic,” *J. Supercomput.*, vol. 73, no. 11, pp. 5053–5070, Nov. 2017, doi: 10.1007/s11227-017-2124-5.
[255] H. Lloyd and M. Amos, “A highly parallelized and vectorized implementation of Max-Min Ant System on Intel® Xeon Phi™,” *2016 IEEE Symp. Ser. Comput. Intell. SSCI 2016*, 2017, doi: 10.1109/SSCI.2016.7850085.
[256] C. S, H. S. Seshadri, and V. Lokesha, “An Effective Parallelism Topology in Ant Colony Optimization algorithm for Medical Image Edge Detection with Critical Path Methodology (PACO-CPM),” *Int. J. Recent Contrib. from Eng. Sci. IT*, vol. 3, no. 4, p. 12,

Dec. 2015, doi: 10.3991/ijes.v3i4.5139.

[257] A. Aslam, E. Khan, and M. M. S. Beg, "Multi-threading based implementation of Ant-Colony Optimization algorithm for image edge detection," in *2015 Annual IEEE India Conference (INDICON)*, 2015, vol. 151, no. 2005, pp. 1–6, doi: 10.1109/INDICON.2015.7443603.

[258] D. M. Chitty, "Applying ACO to Large Scale TSP Instances," *Adv. Intell. Syst. Comput.*, vol. 650, pp. 104–118, 2018, doi: 10.1007/978-3-319-66939-7_9.

[259] H. Ismkhan, "Effective heuristics for ant colony optimization to handle large-scale problems," *Swarm Evol. Comput.*, vol. 32, pp. 140–149, 2017, doi: 10.1016/j.swevo.2016.06.006.

[260] A. A. Abouelfarag, W. M. Aly, and A. G. Elbialy, "Performance analysis and tuning for parallelization of ant colony optimization by using openmp," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9339, pp. 73–85, 2015, doi: 10.1007/978-3-319-24369-6_6.

[261] B. H. Li, M. Lu, Y. G. Shan, and H. Zhang, "Parallel ant colony optimization for the determination of a point heat source position in a 2-D domain," *Appl. Therm. Eng.*, vol. 91, pp. 994–1002, 2015, doi: 10.1016/j.applthermaleng.2015.09.002.

[262] D. El Baz, M. Hifi, L. Wu, and X. Shi, "A Parallel Ant Colony Optimization for the Maximum-Weight Clique Problem," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 796–800, doi: 10.1109/IPDPSW.2016.111.

[263] H. H. Mehne, "Evaluation of Parallelism in Ant Colony Optimization Method for Numerical Solution of Optimal Control Problems," *J. Electr. Eng. Control Comput. Sci. JEEECCS*, vol. 1, no. 2, pp. 15–20, 2015.

[264] L. Dawson, "Generic Techniques in General Purpose Gpu Programming With Applications To Ant Colony and Image Processing Algorithms," 2015.

[265] R. Murooka, Y. Ito, and K. Nakano, "Accelerating Ant Colony Optimization for the Vertex Coloring Problem on the GPU," in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, 2016, pp. 469–475, doi: 10.1109/CANDAR.2016.0088.

[266] T. Tufteland, G. Ødesneltvedt, and M. Goodwin, "Optimizing PolyACO Training with GPU-Based Parallelization," in *International Series in Operations Research and Management Science*, vol. 272, 2016, pp. 233–240.

[267] J. Gao, Z. Chen, L. Gao, and B. Zhang, "GPU implementation of ant colony optimization-based band selections for hyperspectral data classification," in *2016 8th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*, 2016, pp. 1–4, doi: 10.1109/WHISPERS.2016.8071720.

[268] P. Wang, H. Li, and B. Zhang, "A GPU-based Parallel Ant Colony Algorithm for Scientific Workflow Scheduling," *Int. J. Grid Distrib. Comput.*, vol. 8, no. 4, pp. 37–46, 2015, doi: 10.14257/ijgdc.2015.8.4.04.

[269] N. A. Kallioras, K. Kepaptsoglou, and N. D. Lagaros, "Transit stop inspection and maintenance scheduling: A GPU accelerated metaheuristics approach," *Transp. Res. Part C Emerg. Technol.*, vol. 55, pp. 246–260, 2015, doi: 10.1016/j.trc.2015.02.013.

[270] K. Khatri and V. Kumar Gupta, "Research on Solving Travelling Salesman Problem using Rank Based Ant System on GPU," *Compusoft*, vol. 4, no. 5, p. 2320, 2015.

[271] S. NSharma and V. Garg, "Multi Colony Ant System based Solution to Travelling Salesman Problem using OpenCL," *Int. J. Comput. Appl.*, vol. 118, no. 23, pp. 1–3, May 2015, doi: 10.5120/20882-3637.

[272] A. Wagh and V. Nemade, “Query Optimization using Modified Ant Colony Algorithm,” *Int. J. Comput. Appl.*, vol. 167, no. 2, pp. 29–33, Jun. 2017, doi: 10.5120/ijca2017914185.

[273] A. Uchida, Y. Ito, and K. Nakano, “Accelerating ant colony optimisation for the travelling salesman problem on the GPU,” *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 4. Taylor & Francis, pp. 401–420, 2014, doi: 10.1080/17445760.2013.842568.

[274] Y. Zhou, F. He, and Y. Qiu, “Dynamic strategy based parallel ant colony optimization on GPUs for TSPs,” *Sci. China Inf. Sci.*, vol. 60, no. 6, p. 068102, Jun. 2017, doi: 10.1007/s11432-015-0594-2.

[275] O. U. B and R. Tarnawski, *Machine Learning, Optimization, and Big Data*, vol. 10710. Cham: Springer International Publishing, 2018.

[276] O. Bali, W. Elloumi, A. Abraham, and A. M. Alimi, “ACO-PSO optimization for solving TSP problem with GPU acceleration,” *Adv. Intell. Syst. Comput.*, vol. 557, pp. 559–569, 2017, doi: 10.1007/978-3-319-53480-0_55.

[277] A. Llanes, J. M. Cecilia, A. Sánchez, J. M. García, M. Amos, and M. Ujaldón, “Dynamic load balancing on heterogeneous clusters for parallel ant colony optimization,” *Cluster Comput.*, vol. 19, no. 1, pp. 1–11, Mar. 2016, doi: 10.1007/s10586-016-0534-4.

[278] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón, “Enhancing data parallelism for Ant Colony Optimization on GPUs,” *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 42–51, Jan. 2013, doi: 10.1016/j.jpdc.2012.01.002.

[279] R. Skinderowicz, “The GPU-based parallel Ant Colony System,” *J. Parallel Distrib. Comput.*, vol. 98, pp. 48–60, Dec. 2016, doi: 10.1016/j.jpdc.2016.04.014.

[280] R. Rohit Chandra, Leo Dagum, David Kohr, *Parallel Programming in OpenMP*. Elsevier, 2000.

[281] J. J. R. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition, Edition 2*. Morgan Kaufmann, 2016.

[282] J. M. Cecilia, A. Llanes, J. L. Abellán, J. Gómez-Luna, L.-W. Chang, and W.-M. W. Hwu, “High-throughput Ant Colony Optimization on graphics processing units,” *J. Parallel Distrib. Comput.*, vol. 113, pp. 261–274, Mar. 2018, doi: 10.1016/j.jpdc.2017.12.002.

[283] M. Veluscek, T. Kalganova, P. Broomhead, and A. Grichnik, “Composite goal methods for transportation network optimization,” *Expert Syst. Appl.*, vol. 42, no. 8, pp. 3852–3867, May 2015, doi: 10.1016/j.eswa.2014.12.017.

[284] F. Li, “GACO: A GPU-based High Performance Parallel Multi-ant Colony Optimization Algorithm,” *J. Inf. Comput. Sci.*, vol. 11, no. 6, pp. 1775–1784, 2014, doi: 10.12733/jics20103218.

[285] C. Huang, Y. Li, and X. Yao, “A Survey of Automatic Parameter Tuning Methods for Metaheuristics,” *IEEE Trans. Evol. Comput.*, vol. 24, no. 2, pp. 201–216, Apr. 2020, doi: 10.1109/TEVC.2019.2921598.

[286] A. E. Eiben, R. Hinterding, and Z. Michalewicz, “Parameter control in evolutionary algorithms,” *IEEE Trans. Evol. Comput.*, vol. 3, no. 2, pp. 124–141, Jul. 1999, doi: 10.1109/4235.771166.

[287] H. H. Hoos, “Automated Algorithm Configuration and Parameter Tuning,” in *Autonomous Search*, vol. 9783642214, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 37–71.

[288] O. Maron and A. Moore, “Hoeffding Races: Accelerating Model Selection Search for Classification and Function Approximation,” *Adv. Neural Inf. Process. Syst.*, pp. 59–66,

1993.
[289] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A Racing Algorithm for Configuring Metaheuristics," *Proc. Genet. Evol. Comput. Conf.*, no. May 2014, pp. 11–18, 2002.
[290] M. Birattari, *Tuning Metaheuristics*, vol. 197. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.
[291] P. Balaprakash, M. Birattari, and T. Stützle, "Improvement Strategies for the F-Race Algorithm: Sampling Design and Iterative Refinement," in *Hybrid Metaheuristics*, vol. 37, no. 3, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 108–122.
[292] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, "F-Race and Iterated F-Race: An Overview," in *Experimental Methods for the Analysis of Optimization Algorithms*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 311–336.
[293] T. Liao, M. A. M. de Oca, and T. Stützle, "Computational results for an automatically tuned CMA-ES with increasing population size on the CEC'05 benchmark set," *Soft Comput.*, vol. 17, no. 6, pp. 1031–1046, Jun. 2013, doi: 10.1007/s00500-012-0946-x.
[294] T. Liao, D. Molina, and T. Stützle, "Performance evaluation of automatically tuned continuous optimizers on different benchmark sets," *Appl. Soft Comput.*, vol. 27, pp. 490–503, Feb. 2015, doi: 10.1016/j.asoc.2014.11.006.
[295] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stuetzle, "ParamILS: An Automatic Algorithm Configuration Framework," *J. Artif. Intell. Res.*, vol. 36, pp. 267–306, Oct. 2009, doi: 10.1613/jair.2861.
[296] H. H. Hoos and T. Stützle, *Stochastic Local Search*. Elsevier, 2005.
[297] L. Pérez Cáceres and T. Stützle, "Exploring variable neighborhood search for automatic algorithm configuration," *Electron. Notes Discret. Math.*, vol. 58, pp. 167–174, Apr. 2017, doi: 10.1016/j.endm.2017.03.022.
[298] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, M. Birattari, and T. Stützle, "The irace package: Iterated racing for automatic algorithm configuration," *Oper. Res. Perspect.*, vol. 3, pp. 43–58, 2016, doi: 10.1016/j.orp.2016.09.002.

9. APPENDIX

The following section describes the Transcom problem in section 2.3.2.3. in more details and give numerical examples.

Dataset tables

- *Available Aircraft* – table containing the number of aircraft of given type at each of the military bases/airports at time zero.
- *Aircraft to Base Compatibility* – matrix representation of given aircraft type access to any of the military bases or commercial airport. For example, none of the military aircraft can land at commercial airports, however, some of the military bases share runway and allow commercial aircraft.
- *Aircraft data* – table of attributes of any given aircraft type. Some of the aircraft support in-air refuel, where the range of aircraft is reset. As military aircraft costs are calculated per hour, it is assumed that refuel costs are included as part of the refuel time delay. All ground refuel time is assumed to be 1 hour. Furthermore, table also contains *Cost per flight hour*, cruise speed (mph) for each aircraft type as well as the maximum number of pallets it can carry. Table also contains the crew required to fly the plane (*Flight Crew Required*), as well as the number of crew required for fuelling up and safety checks (*Ground Staff Required*). Moreover, both flight crew and ground staff must arrive defined number of hours before the flight (*Prefetch time*) and stay longer after the flight *(Post mission time*).
- *Material Sources* – table of locations and quantity of the pallets of material needed to fill the demand.
- *Material Demand Destinations* – table of locations that require the quantity of pallets to be delivered. There are two kind of destinations – Humanitarian and Resupply. Furthermore, table also lists the aircraft constraints for given destination. For example, none of the commercial aircraft can land at Humanitarian destination, however, some commercial aircraft can supply military bases.

- *Base to Base Edges* – table provides the starting point and end point coordinates for each of the base to base connections as well as the straight-line distance (in miles).
- *Base to Destination Edges* – table provides starting point and end point coordinates for each of the base to destination connections as well as the straight-line distance (in miles).
- *Destination to Destination Edges* – table provides starting point and end point coordinates for each of the destination to destination connections as the straight-line distance (in miles).
- *CP to CP Edges* – table provides starting point and end point coordinates for each of the commercial partner destinations as well as the straight-line distance (in miles). Furthermore, cost per pallet is also provided for each of the paths.
- *Base to CP Land Edges* – table provides ground links between military bases and commercial partner locations. Includes start and end point coordinates as well as straight-line distance in miles, total time on path (hours) and cost per truck. Furthermore, maximum number of pallets per truck is also provided.
- *CP to Base Air Edges* – table provides air links between commercial partner locations and military bases that share common runway. Provides start and end point coordinates, as well as the straight-line distance (in miles) and total cost per pallet.

Example scenario A:

In the given scenario A shown in **Figure 50**, military plane without in-air refuel capability is supplying 6 pallets of cargo to Base B that is 1000 miles away.

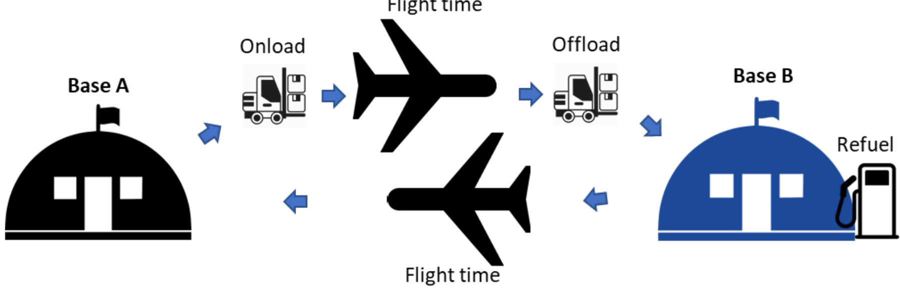

Figure 50. *Scenario A – Simplified Transcom supply chain example*

Plane costs $10,000 per hour to fly and it takes 1 hour to fully refuel. The total distance between the two bases is 1000 miles and airplane cruise speed is 200 miles per hour. Because range of the airplane is not enough to fly both directions, it needs to refuel at the destination base (Base B). Total timespan to satisfy Base B demand of 6 pallets of cargo therefore is 13 hours:

$$Timespan =$$
$$= onloadTime + flightTime + offloadTime + refuelTime + flightTime =$$
$$= \frac{6}{6} + \frac{1000}{200} + \frac{6}{6} + 1 + \frac{1000}{200} = 13\ hours$$

And the total cost for the given route:

$$Cost = Cost(MA) * 2 = (10 * \$10{,}000) * 2 = \$200{,}000$$

Example scenario B:

Although scenario A is easy to follow example, the supply chain solved in Transcom problem is not as simple. **Figure 52** and **Figure 52** shows more representative example of the Transcom supply chain complexity. It involves 5 military bases, 2 commercial partner (CP) locations, 2 military planes, 1 commercial plane and 4 trucks.

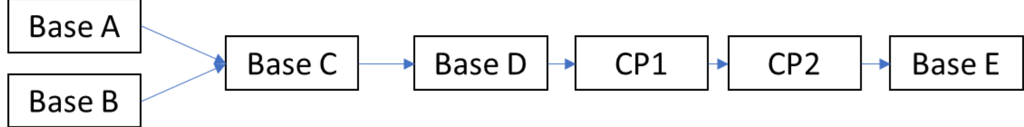

Figure 51*. Scenario B pallet flow between military bases and Commercial Partners (CPs).*

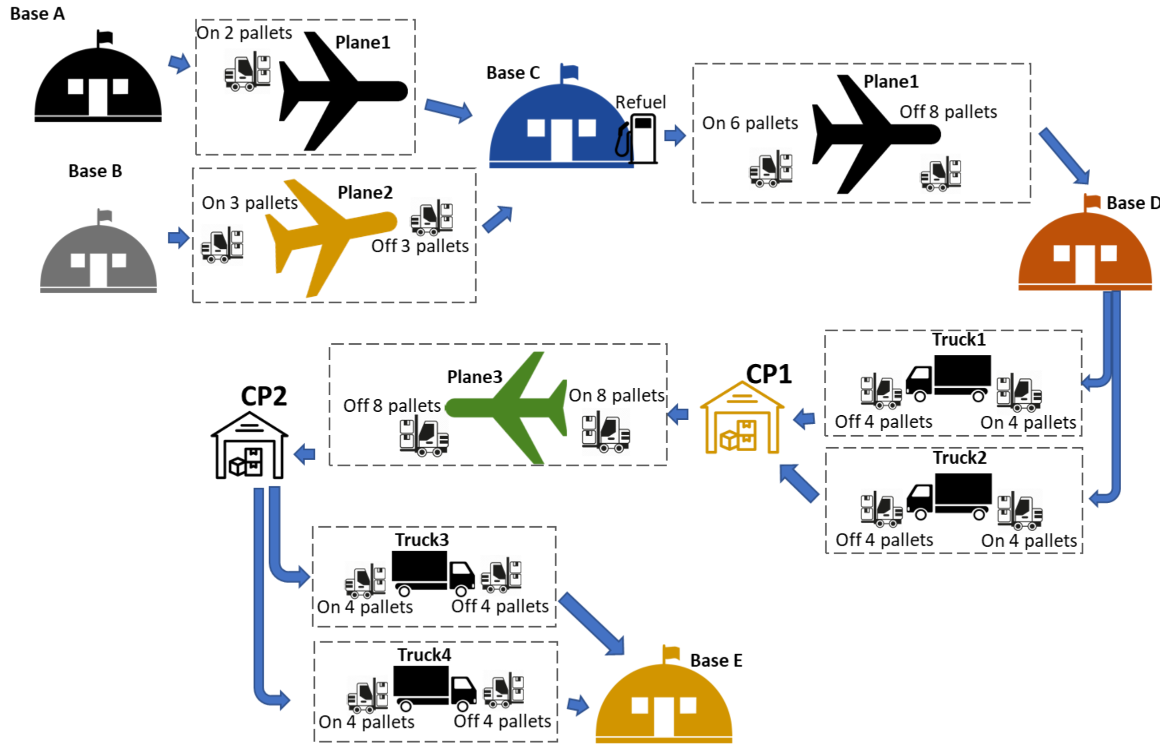

***Figure 52**. Scenario B: Realistic Transcom supply chain example*

In this example, the demand for Base E is 8 pallets. At Time zero, Base A has 2 pallets, Base B and C have 3 pallets each. Plane1 flight time between Base A and Base C is 4 hours and Plane2 flight time between Base B and Base C is 5 hours. Plane1 has to wait for Plane2 to offload its pallets before it can continue flight to Base D. Furthermore, Plane1 also need to refuel at Base C as the distance between Base A – Base B – Base C exceeds its range. Plane1 then continues its path to Base D for 6 hours, where it needs to offload 8 pallets. From there, pallets are shipped to CP location CP1. Due to capacity constraints each truck can only transfer 4 pallets at a time, hence 2 trucks drive to CP1 in parallel and takes 1 hour each. From CP1 8 pallets are onloaded to commercial plane Plane3 and transferred to CP2, which takes another 1 hour. Two trucks are again needed to transfer the 8 pallets to their final destination – Base E (1 hour).

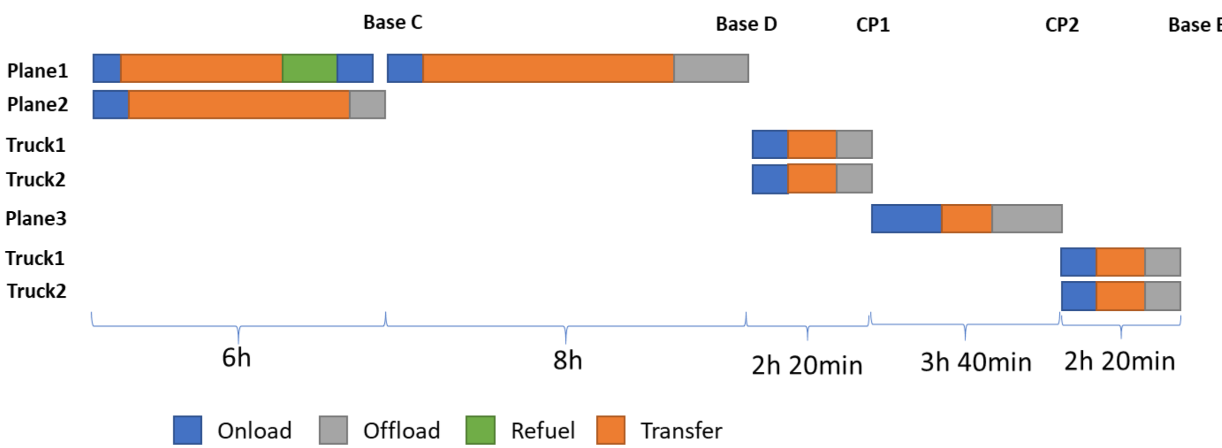

Figure 53. *Transcom Scenario B timeline*

In Transcom problem supply chain algorithm must work with multiple parallel timelines in order to model all demand accurately. **Figure 53** shows the Scenario B timeline, where there are two parallel military aircraft flying to the same destination from different starting points. Plane2 starts its journey by onloading 3 pallets (30 minutes) and flying to Base C (5 hours), once arrived at Base C, offloading the 3 pallets (30 minutes), totalling of 6 hours. Meantime, Plane1 starts its journey by onloading 2 pallets (20 minutes), flying to Base C (4 hours), but as it is still waiting for Plane2 to land, it refuels (1 hour) and onloads the 3 pallets located at Base C (30 minutes). Which leaves 10 minutes of idle time for Plane1 till Plane2 offloads its load.

Only at 6-hour mark can Plane1 start its next leg of the journey, by onloading the 3 pallets delivered from Plane2 (30 minutes) and flying to Base D (6 hours), where it needs to offload aircraft completely, which takes 80 minutes for 8 pallets. Total of 8 hours.

Once pallets are offloaded from airplane at Base D, two commercial trucks (Truck1 and Truck2) are onloaded in parallel taking 4 pallets each (40 minutes). Transferring cargo to Commercial Partner facility at CP1 (1h) where pallets are offloaded again (40 minutes). Total of 2 hours and 20 minutes.

From CP1 commercial Plane3 takes all 8 pallets onboard (80 minutes) and fly them to CP2 (1h flight time), where they are again offloaded (80 minutes). Total of 3 hours and 40 minutes.

At the last stage, at CP2 are again onloaded onto two trucks (Truck3 and Truck4) in parallel (40 minutes each) and transferred to their final destination BaseE (1 hour), where they are offloaded (40 minutes each). Total of 2 hours and 20 minutes.

Therefore, the total timespan required to satisfy demand of 8 pallets at Base E is 22 hours and 20 minutes.

Assuming cost per hour of Plane1 is $10,000 and $1,500 for Plane2, the total cost for military aircraft $Cost(MA)$ can be calculated

$$Cost(MA) = \$10{,}000 * (4 + 6) + \$1{,}500 * 5 = \$107{,}500$$

Assuming cost per pallet for Plane3 between CP1 and CP2 is $15,000, the total cost for commercial aircraft $Cost(CA)$ can be calculated

$$Cost(CA) = \$15{,}000 * 8 = \$120{,}000$$

Assuming cost per truck for path between Base D and CP1 is $4,000 and between CP2 and Base E is $5,000, the total cost for commercial trucks $Cost(CT)$ can be calculated

$$Cost(CT) = \$4{,}000 * 2 + \$5{,}000 * 2 = \$18{,}000$$

Therefore, the total cost of given solution is

$$Total\ Cost = \$107{,}500 + \$120{,}000 + \$18{,}000 = \$245{,}500$$