Towards a unified method of synthesising scenarios and solvers in combinatorial optimisation via graph-based approaches

Christopher L. Stone

Doctor of Philosophy

2020

Abstract

Hyper-heuristics is a collection of search methods for selecting, combining and generating heuristics used to solve combinatorial optimisation problems. The primary objective of hyper-heuristics research is to develop more generally applicable search procedures that can be easily applied to a wide variety of problems. However, current hyper-heuristic architectures assume the existence of a domain barrier that does not allow low-level heuristics or operators to be applied outside their designed problem domain. Additionally the representation used to encode solvers differs from the one used to encode solutions. This means that hyper-heuristic internal components can not be optimised by the system itself. In this thesis we address these issues by using graph reformulations of selected problems and search in the space of operators by using Grammatical Evolution techniques to evolve new perturbative and constructive heuristics. The low-level heuristics (representing graph transformations) are evolved using a single grammar which is capable of adapting to multiple domains. We test our generators of heuristics on instances of the Travelling Salesman Problem, Knapsack Problem and Load Balancing Problem and show that the best evolved heuristics can compete with human written heuristics and representations designed for each problem domain. Further we propose a conceptual framework for the production and combination of graph structures. We show how these concepts can be used to describe and provide a representation for problems in combinatorics and the inner mechanics of hyper-heuristic systems. The final contribution is a new benchmark that can generate problem instances for multiple problem domains that can be used for the assessment of multi-domain problem solvers.

Acknowledgements

My deepest gratitude goes to my advisor Professor Emma Hart for her guidance throughout this research. Also to Professor Ben Paechter for his support and insights. To the Edinburgh Napier University for funding and hosting this research. Many thanks to my colleagues at the School of Computing with a special mention to those I have share my time and space in room C51 that made the journey pleasant and enlightening. Thanks to my colleagues at the University of St Andrews for their patience and support. Finally thanks to all my friends inside and outside the academic world.

To my mother Rosanna

Author's declaration

This thesis is a presentation of my original research work. Whenever contributions of others are involved, every effort is made to indicate this clearly, with due reference to the literature, and acknowledgement of collaborative research and discussions.

-Christopher L. Stone

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Combinatorial problems pervade human activities: from scheduling classes for schools (from primary to higher education), scheduling staff in businesses (from restaurants to hospitals, shops and factories), organising deliveries, structuring the topology and running of networks (electric, gas, water and data), to budgeting, stock trading and management of portfolios, supply chain management and more [104]. Although new methods are continuously developed and the computational power we have access to has increased exponentially for several decades, we still struggle to apply computational methods to optimise these problems efficiently. Going from identification of the problem to implementation of a computer mediated solution takes significant amount of time and effort, and this can be exacerbated if the condition of the problem changes or new problems appear.

A common approach used to solve these types of problems is to implement heuristic methods that can trade the solution's optimality for resources (such as time and/or memory) required to compute the solution. This allows the creation of less than optimal solutions in "reasonable" time for human applications. What a "reasonable" amount of time is depends on the specifics of each problem and may vary from milliseconds to months. The advantage behind this idea is that heuristic methods allow us to go from unfeasible to feasible strategies (in terms of physical or budgetary requirements rather than mathematical limits). However, the creation of heuristics for real-world applications is still in large an art that requires human intuition and ingenuity [79, 35].

On this matter, the field of hyper-heuristics [23] has devoted the last two decades to increasing the generality and applicability of heuristic methods to combinatorial optimisation problems. At the highest level hyper-heuristics have already developed a number of techniques in the spirit of cross-domain applicability. This has taken various forms such as selecting heuristics from a pool of available heuristics [46], generating new heuristics from components [25] or combining multiple existing heuristics that become tailored to solve specific problem classes [132]. These heuristics are normally called “high-level heuristics”. They normally manage “low-level heuristics”, which are heuristics that directly manipulate the solution of a given problem. One of the main issues of low-level heuristics is that they operate only on a specific representation of a problem (i.e. list of booleans, integers or matrices). They are only applicable to the original representation they were designed to operate on and not on any other representation. This limits the generality of solver systems and makes it difficult to solve problems from new domains that require the manual creation of new low-level heuristics or representations.

In this thesis we embrace the hyper-heuristic philosophy and investigate various approaches that could help advance the generality and applicability of this methodology. We investigate methodologies that allow hyper-heuristic systems to generate low level heuristics, applicable to combinatorial optimisation problems belonging to different domains that are as generic as possible. While the question is often “how do we solve this specific problem to optimality?” our desire is to investigate “how can we solve this new problem that we currently do not have methods to deal with?”. This is a much broader question that guides the choice of use cases described in this thesis. Sim [145], who developed techniques able to tackle problems from different classes within a domain efficiently, previously speculated that hyper-heuristic methods could transcend domain boundaries. This is something that the field of meta-heuristics have been attempting to achieve for about half a century now, yet the bulk of the research in the field tends to focus on specific problem domains and specific strategies that help the algorithm reach new levels of optimality within a single domain.

The work presented here can be divided into three parts.

The first part, composed of Chapter 2 (Fundamentals) and 3 (Theoretical and Practical Tools), sets the scene by introducing the fundamental ideas about combinatorial optimisation, hyper-heuristics and graph based representation. Respectively, the problems we wish to solve, the solvers we apply and the abstraction used to represent them. This is followed by a review of Grammatical Evolution, which is a methodology inspired by evolutionary processes used to produce snippets of code in an arbitrary language, order theory and geometry which are disciplines from which we borrow ideas and use them to develop the heuristic generators from Chapter 4 and 5.

The second part is composed of Chapter 4 (Synthesising Constructive Heuristics) and 5 (Synthesising Perturbative Heuristics) that has strong empirical component and is dedicated to the evaluation of whether we can automate the production of heuristics where a single grammar based approach and representation are used to produce heuristics for multiple domains. These generators of heuristics are tasked with producing heuristics for the Travelling Salesman Problem and the Multidimensional Knapsack Problem. In particular, contrary to common approaches, we study how heuristics can be evolved so that they can efficiently be applied on representations that are not "natural" to the problem, forcing different problem domains to share the same representation. This allows operators to become applicable on both problem domains and the bulk of the computational effort is spent searching for variants of the operators that work well for each specific problem domain. We show how different heuristic operators, produced by a single system, can be evolved to solve different problems encoded over the same representation and we compare them with other methods taken from the literature.

The third part of the thesis is more theoretical in nature. Following the results of the experiment on generative hyper-heuristics we develop a representation that could suit future experiments with the goal of establishing methods that can be applied to as many domains as possible. In particular we developed a representation that could also be used to model the various part of hyper-heuristic systems themselves so they could be rewritten and optimised automatically.

In Chapter 6 we show that the mechanics normally present in hyper-heuristic systems can be described with the language and notation we have suggested. Finally we implement a scenario generator that makes use of the proposed formalism and produces scenarios for 3 different domains. The scenarios are combinatorial optimisation problems related to logistics. The purpose of the generator is multi-fold. It shows that the formal description can be translated into an empirical model and at the same time it is developed to facilitate the evaluation of multi-domain problem solvers. As algorithms tend to be evaluated over specific problem domains, there is a lack of tools for the evaluation of algorithms over a range of different problem domains. This is the case even though real problems are notoriously multi-domain i.e. factories have to manage staff shifts, deliveries, assign jobs to machines and schedule maintenance all in the same environment, classical meta-heuristics would require a user to work with multiple representation and algorithms. Even from a purely academic research standpoint this would be inconvenient due to the fact that evaluations based on comparative approaches would require the use of different benchmarks with different encoding. Testing solvers on them would require a new interface for each new domain which become unfeasible as the number of domains grows.

1.1 Research Questions

In this thesis we explore the idea that one representation can be used for a large variety of tasks given a number of trade offs. This is in contrast to the approach of using specific representations for each specific problem domain or subdomain. This is investigated as a possible complementary pathway that could increase the generality of multi-domain problem solving systems.

The thesis addresses the following questions:

1. To what extent can a *single* hyper-heuristic generator be used to synthesise heuristics for multiple combinatorial domains, assuming a common representation?

2. To what extent is a graph-based formalism able to express combinatorial problems and the constituent parts of complex hyper-heuristic algorithms so that more general hyper-heuristic optimisers could be developed?

Question 1 is answered in Chapter 4 and 5 where two grammars are proposed and developed in order to automatically synthesises heuristics for different combinatorial optimisation problems. The heuristics evolved with this approach are tested on well established benchmarks for each problem domain and compared with other techniques taken from the literature.

Question 2 is answered in Chapter 6 where first we propose a formalism for the definition and combination of graph structures and then implement some of its elements in Chapter 7 where it is shown empirically that it can be used to generate problem instances from a variety of domains constructed using data from real world road networks and synthetic data.

1.2 Thesis Contributions

- A novel algorithm and grammar for the generation of *constructive* heuristics that can evolve domain specific heuristics for the Travelling Salesman Problem *and* Multidimensional Knapsack Problem (Chapter 4). Empirical evaluations of the system have shown that it is feasible to use a single representation for more than one domain as long as an appropriate operator is applied to it. In contrast to previous results that used custom human heuristics, in these experiments heuristics for both domains are automatically generated.

- A novel algorithm and an alternative grammar for the generation of *perturbative* heuristics that can evolve domain specific heuristic for the Travelling Salesman Problem, Multidimensional Knapsack Problem *and* Load Balancing Problem (Chapter 5). The results of the experiments reconfirmed that, given sufficient computation, an appropriate operator can be evolved from building blocks that permute and invert elements of a sequence.

- The development of a meta-modelling language for the description of high level properties of graphs and their specific low level structure (Chapter 6). It is shown that the language developed can be used to easily describe the fundamental components of hyper-heuristic systems and their mechanics.

- The definition and implementation a problem specification describing a collection of combinatorial optimisation problems from different domains. The implementation makes use of the language to describe the problem's high level constraints. Real world data are fed in the problem synthesiser to create a novel set of new benchmark problem instances (Chapter 7)

1.3 Layout of the thesis

- Chapter 2 Fundamentals: a review of the fields that the thesis builds upon. It introduces the idea of Combinatorial Optimisation as the main category of problems we wish to address, hyper-heuristics in their various forms as the main tool set to address the problems and finally Graphs and Graph Transformations that will be our representation instruments describing problems and solvers.

- Chapter 3 Theoretical and Practical Tools: a review of the most important tools and methods used in the thesis, starting with Grammatical Evolution that is the engine that produces and evolves heuristics, Order theory that provide notions vital to the approaches developed such as ranking, sorting, inversions and permutations. Finally Geometry that gives the bases of measures and the properties used in the grammar developed in Chapter 4
- Chapter 4 - Synthesising Constructive Heuristics: Contains an empirical study on generating constructive heuristics for multiple domains applied to the same representation
- Chapter 5 - Synthesising Perturbative Heuristics: Contains an empirical study on generating perturbative heuristics for multiple domains
- Chapter 6 - A Conceptual Framework for the production of Graphs: presents a meta-modelling language for the high level description and manipulation of graph structures and a description of common hyper-heuristic mechanics
- Chapter 7 - Graph Based Problem Scenario Synthesis - presents the definition and implementation of problem domains and instance generators modelled using the proposed framework
- Chapter 8 - Conclusions: this is the final Chapter of the thesis which summarises the main outputs of the thesis and pathways for future research

1.4 Publications

- Stone, Christopher, Emma Hart, and Ben Paechter. “Automatic generation of constructive heuristics for multiple types of combinatorial optimisation problems with grammatical evolution and geometric graphs.” Applications of Evolutionary Computation - 21st International Conference, EvoApplications 2018, Parma, Italy, April 4-6, 2018, Proceedings. Volume 10784. Pages 578–593. Springer, Cham, 2018.
- Stone, Christopher, Emma Hart, and Ben Paechter. “On the Synthesis of Perturbative Heuristics for Multiple Combinatorial Optimisation Domains.” Parallel Problem Solving from Nature - PPSN XV - 15th International Conference, Coimbra, Portugal, September 8-12, 2018, Proceedings, Part I. Volume 11101. Pages 170-182. Springer, Cham, 2018.

Chapter 2

Fundamentals

This thesis revolves around three main disciplines that are broad fields of study: Combinatorial optimisation problems (*the problem*), hyper-heuristics methodologies (*the solver*) and graphs (*their representation*).

In this chapter they will be briefly introduced. In particular the ubiquity of combinatorial optimisation problems which form a giant family of related, yet extremely diverse, problems will be stressed. This is followed by a review of hyper-heuristics that is the chosen approach to deal with the issue of having to solve many different problems from different domains.

2.1 Combinatorial Optimisation

Combinatorial Optimisation (CO) is concerned with finding an optimal configuration among a set of discrete and finite amount of objects or structures. If in combinatorics the foundational question is “how many combinations?” in combinatorial optimisation it is often “what is the best combination of them all?”. This is assuming that we are able to measure in some way the quality of the possible combinations. Combinatorial optimisation problems arise in many fields, from operations research, network design of all kinds, to telecommunication and computer science [83]. Many sub-problems within these fields are part of a grand family of problems all related to each other as they belong to the complexity class of NP-complete problems. While an important characteristic of NP-complete problems is that one can reformulate one problem into any another (i.e. a routing problem into a boolean satisfiability problem or a bin packing into a routing problem), this operation is not trivial [118] and may add significant computational costs, more precisely in polynomial time. An even bigger issue is that different problems, even belonging to the same class, may require different solving strategies to be solved efficiently. Below we describe a small sample of these problems to convey an idea of their diversity and pervasiveness.

It should be clarified that the word "solve" and "solved" have different meanings in different fields. In mathematics, a problem in optimisation is considered "solved" when the point of *global maximum or minimum* has been found. While in management engineering, an optimisation problem is considered "solved" when a sufficiently good solution has been found even if better solutions may exist. In this manuscript we will use the latter interpretation of the word. Further, the use of the word "solving" will mean "searching for a solution" regardless of the outcomes of the search.

A common example of a CO problem is the Travelling Salesman Problem (TSP) where an agent has to visit n cities and has to visit every city and return home and must minimise the amount of distance travelled [97]. The number of possible routes available is given by the formula $x = (n-1)!$, a TSP where an agent has to visit 100 cities contains already more than 10^{155} possible routes and it is not trivial to prove that any given solution is better than all the others. This example does not take into consideration time, traffic or routes that are incorrect such as the agent visiting the same city multiple times or not arriving at home as the last stop. Similar examples can be made for counting the number of ways a circuit can be wired or a timetable organised [65].

Many routing problems are combinatorial in nature. For example delivering multiple packages, picking up people or packages, organising repairman routes or nurses visits [159]. This extends to the routing of data within a network. It is easy to see that sub-optimal solutions to these problems lead to an increase in costs, time of execution, resources used and potentially amount of pollution caused with large scale economic and social effects.

Packing problems are another large class of combinatorial problems [38] where a number of objects have to be packed in some container. This is not just a problem of objects to be placed in a container for delivery but the problem of choosing a subset from a larger set while trying to maximise some objective, i.e. buying assets within a budget in order to resell them and maximise profits, loading cargo in a balanced manner or fitting tiles within a surface.

Scheduling problems are the class of mathematical optimisation problems that formalise assigning tasks to resources at a particular time and/or in a particular order [81]. The problem is usually formulated by declaring two collections : n jobs J_1,J_2...J_n to be processed and m resources R_1,R_2,...R_m in which these tasks may be completed. The problem is to minimise the makespan C, where C is the total amount of time required to complete all the jobs. Timetabling problems are a common example of resource allocation and scheduling within education establishments. In particular they normally come with a large amount of constraints e.g.

- no students can be scheduled at more than one event at a time
- the room meets all features required by the event
- room capacity is respected
- no more than one event is allowed per room and per timeslot

Special cases and extensions of this problem can take various forms: jobs can be programs to be executed and the resources can be collections of computers, jobs can be objects to be built and workers can be human employees. Alternatively the amount to minimise could be both time and the energy consumed to complete all the jobs, jobs can be lecturers and the resources classrooms and the goal could be minimising student and time constraints.

Configuration of networks of all kinds leads to combinatorial optimisation problems. For example, in the construction of water networks the type and size of pipes have to be decided, as well as the number of valves, size of the valves and the specific state of the valves [93]. This problem reappears in a variety of other network, such as electric networks, where the type and size of wires, number and size of electric switches and again their state have to be decided. The same could be stated for gas networks, oil networks, or networks for any other type of fluid up to road networks and vehicle traffic. This can also be extended to data networks where valves are substituted with relays and optic switches and connections can be optic, electric and wireless. These decisions ultimately determine the configuration of the network, its cost, how long is going to take to build and maintain it, and what is the flow that the network will be able to sustain. It is important to highlight that the economic consequences of these decisions can influence the budgets of nations for decades.

Furthermore, it is important to highlight how in real scenarios different types of network are tightly connected, for example as a spill of water can interrupt a road network and may interrupt electric and communication networks. To an even greater extent this can be said for oil and gas networks. Car accidents (road network spills) can damage other networks and this happens globally with considerable frequency.

However this is rarely true for models as each network exists in its own silo and breaking a constraint usually means the solution receives a single penalty (often fixed extra cost or time) [30]. This is a known issue and examples already exist in the attempt to tackle this matter such as the travelling thief, that nest the problem of optimising routes and packing [19]. This is not just a problem of a reality gap between the model and the real world but a modelling problem too. In the sense that the most common graph theoretical models used in combinatorial optimisation literature cannot describe multi-typed networks in which chain reactions that modify the configuration and topology of the networks across types take place. In the majority of cases only single type static networks are considered [93].

In order to solve these types of problems, or at least simplified models of the problems, a large collection of methodologies have been developed in the past century. In particular a surge in effort sprung during the Second World War as more principled and mathematical approaches become dedicated to issues such as logistics and resource allocation. In particular Kantorovich developed mathematical methods for the optimal allocation of resources when he was consulting with the Soviet government in 1939 [84] that led to him winning a Nobel prize in 1975; and independently Dantzig developed the ground breaking *simplex method* while working for the U.S. Air Force which was published in 1947 [155]. These results paved the way to a variety of approaches for combinatorial optimisation that relied on geometric and algebraic methods [67].

An important property of NP-Complete problems is that they can be "reduced" to other problems of equivalent difficulty. Reductions are the most common tool used to verify the computational complexity of new problem classes. This is also exploited so that specific solvers such as SAT-solvers can be applied to a variety of problems as long as the appropriate reduction or reformulation is applied [68]. Like other methods this has a number of advantages and disadvantages that are class and instance dependent. The main advantage being that one solver can be used for several domains and the main disadvantage is that problems, once turned into SAT problems, may take a lot longer to find solutions or use vastly more memory [103]. This is often the case for satisfiability modulo theories (SMT) turned into SAT. Also SAT solvers performance may vary greatly on the same problem depending on the internal heuristics that are implemented [131].

Another approach relies on the use of ‘heuristics’ that do not guarantee optimal results but may be sufficiently good for the immediate needs. This often involves a trade off between the optimality of the results and the time taken to generate a solution to the problem [37]. Along this ideas the field of meta-heuristics started using higher level procedures that make little assumption about the underlying problem in order to be applicable to a larger variety of problems [16]. Genetic algorithms [101] and ant colony optimisation algorithms[44] are some of the most famous examples.

One of the most recent approaches to solve combinatorial optimisation problems combines both collections of heuristics and meta-heuristics under the name of hyper-heuristics.

2.2 Hyper-Heuristics

Hyper-heuristics is a broad approach to problem solving. It uses combinations of heuristics in order to leverage the strengths of multiple methodologies to increase the generality of problem solvers [23].

Burke *et al.* define the term hyper-heuristics as: “ an automated methodology for selecting or generating heuristics to solve hard computational search problems.”

One of the first uses of this kind of approach, even if under another name, was implemented by Denzinger et al in 1996 [43] where they created an automatic theorem proving engine based on a collection of AI techniques that would prove simple theorems and use them to create more complex proofs. In their work they call this methodology “TEAMWORK”. Their principle was based on the idea of having a collection of agents specialised in different methods that work concurrently and cooperatively. Some agents look for new solutions while others look at old problems and assess how similar they are to the new problem and may suggest them for consideration. But the methodology of using collections of heuristics, in the context of automated problem solving, can be traced back to 1959 where Gelernter [59], in his work on “The Geometry Machine”, used it to prove theorems in Euclidean geometry (incidentally using diagrams and graphs to construct solutions) and also used to investigate the potential of heuristic methods for mathematical discovery. His prototype was able to find proof for problems in geometry normally given to high-school sophomores.

The term “hyper-heuristic” was coined by Cowling et al in 2000 [39], in order to highlight the higher level of abstraction compared to meta-heuristic approaches. Over time the meaning of the term has evolved from “heuristics to choose heuristics” to encompass a number of diverse methods which include the generation of new heuristics and learning mechanisms. An extensive survey about the inception of hyper-heuristics approaches has been written by Burke et al in [23], which also gives an account of the various domains tackled using these approaches including scheduling [75], timetabling [29], packing [132] and vehicle routing [57].

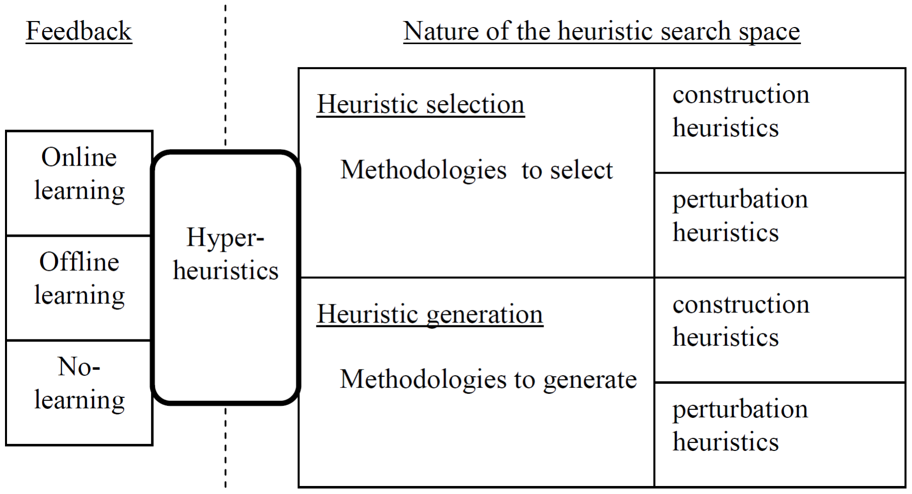


Figure 2.1: Classification of hyper-heuristics approaches taken from [24]

Figure 2.1 [24] shows a diagram of the most modern interpretation of hyper-heuristic methods where a main hyper-heuristic system orchestrates the use of lower level heuristics, by choosing available heuristics or generating new ones. Heuristics can be constructive, which start from an empty solution and construct a complete solution one step at a time, or perturbative, which means that they take a complete solution as an input and modify it according to some heuristic method often called a move operator. The system may have different kinds of learning methodologies or none at all. Learning is usually divided in two main styles: *Online Learning* and *Offline Learning.*

Online Learning is the ability of a hyper-heuristic system to evaluate information gathered during the current search and utilise it to guide the successive steps of the search in the heuristic landscape. The information is normally discarded at the end of the search [147]

Offline Learning: Learning is achieved by using previous searches as a training set. It is usually done before the new problem is tackled and independently from it [107].

Perturbative Heuristics are heuristics that start from an already complete solution and attempt to improve it by perturbing some property of the solution while maintaining its validity [127]. This could affect the order of the items in a solution or the values of the variables in a solution and depends on the specific problem class. This is usually achieved using operators borrowed from classical meta-heuristics algorithm such as mutation, hill climbing, annealing operators or what are generally called neighbourhood operators. Perturbative heuristics can be applied an arbitrary number of times to the solution, usually determined by number of iterations, time or fitness attained by the solution.

Construction Heuristics are those heuristics that can build a solution “from scratch” [1]. This means that they are able to start from an empty set and iteratively add elements until a valid solution is completed. An example of this heuristic is the nearest neighbour algorithm that is used to solve instances of the travelling salesman problem [86]. At each iteration a new edge is connected simply by selecting the vertex that is closest to the previously connected vertex until all vertices are connected.

Generative hyper-heuristic techniques involve the production of new heuristics using some other heuristic or meta-heuristic approach. The majority of modern hyper-heuristic frameworks utilise Genetic Programming(GP) for the production of heuristics. Genetic Programming is a collection of techniques that explore the space of programs using Darwinian principles similar to Genetic Algorithms. In its classical implementation, developed by Koza [91], programs are represented as trees. Depending on the domain of the problem the resulting output of the program could be anything from a collection of objects to be packed, a list of cities to be visited or the program itself could be a mathematical formula that is used to determine some decision. For example the chances of visiting any given city next.

Selection hyper-heuristics are methods with which the system chooses heuristics from an available pool of heuristics [27]. The heuristic selected can be either constructive or perturbative. Normally the heuristics that can be selected are called low-level heuristics and their applicability depends on their compatibility with the solution’s representation.

These techniques, combined with suitable low-level heuristics, are able to tackle a wide variety of problem classes and have been applied successfully to combinatorial problems outperforming current state-of-the-art approaches [148] [15]. Projects such as HyFlex are built with the goal of being applicable to as many problem as possible [106]. In HyFlex, high level heuristics manage a collection of problem specific heuristics in a modular way. These problem specific heuristics can each be applied to one domain and together can tackle a diverse set of problem domains. In its present implementation, six different hard combinatorial optimisation problems are implemented. Nevertheless low-level heuristics are tied to their representation in their applicability to a solution. This characteristic is embedded in hyper-heuristics practices and all systems assume that there is a "domain barrier" that make operators inapplicable across domains. Differently from classical practices, this thesis builds around the idea that the barrier is only between specific operators and representations but not between problem domains *per se*. In Section 2.2 a typical diagram of a hyper-heuristic system [154] can be seen referring to the domain barrier which prevents operators from being applied. It is the belief of the author that it would be more correct to call this limitation a ***representation barrier***. This is due to the fact that it is the representation that ultimately precludes the applicability of a heuristic and if two different domains are modelled using the same representation then a given heuristic could be applied to both.

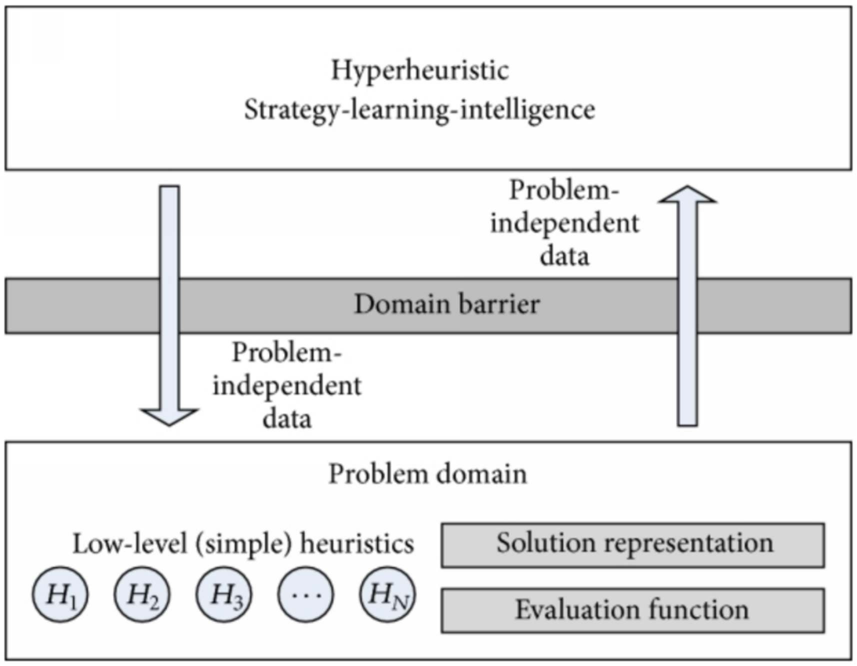


Figure 2.2: Common hyper-heuristic diagram taken from [154]

One of the goals of this work is increasing the generality of hyper-heuristic methods. This is achieved using a single representation for multiple different problem domains and synthesising collections of operators that are specialised to be effective on the specific domains. This is in contrast to the classical approach of designing domain specific representations and tailing operators around the specific combination of problems and representations. The specialisation happens using general methods of evolving low-level heuristics that are applicable to graph based representations. The low-level heuristics evolved in such a way that they are tied to the constraints of the problem instead of the specific representation chosen for the problem. In the following section we will take into consideration how and why graphs can be used to fulfil these needs.

2.3 Graphs and Graph Rewriting

One of the arguments put forward in this thesis is based on the impracticality of hand-designing specific representations for each single domain, which for large multi problem systems is equivalent to saying that each subsystem that deals with a different domain should be represented differently in some "optimal" way. This task is almost entirely left to the expertise of the developer deciding on encodings and mappings. In the majority of cases the recommendations that can be found in the literature, based on empirical results, are only applicable to the specific benchmarks on which the representation was tested. In real scenarios the actual problem will be different from any given benchmark and hence results may be vastly different.

Here we consider the approach of using graphs as higher-level representations that describe problem structures, but can also be utilised for describing solver mechanics as well. This is important because hyper-heuristics are not only a tool for solving CO problems, but hyper-heuristic systems themselves contain CO problems within their own internal mechanics (i.e. selecting a subset of heuristics, choosing in which order to apply a sequence of heuristics and how many times each operator should be applied). Therefore it is desirable to be able to describe both using the same type of representation.

A graph is a mathematical structure formed by vertices and edges connecting the vertices. Depending on the author, vertices can be called nodes or points and edges may be called arcs. More formally a graph G = (V,E) consists of a set of objects V = v_1,v_2, ...,v_n called vertices and a set of objects E = e_1,e_2,...,e_n called edges which are a subset of VxV.

Any object that involves points and connections between them may be called a graph. Configuration of vertices and edges occur in a great diversity of applications. They may represent physical networks, such as electrical circuits, roadways, or organic molecules. They are also used in representing less tangible interactions such as ecosystems, sociological relationships, databases, or in the flow of control in a computer program. There is no unique way of drawing a graph; the relative positions of points representing vertices and lines representing edges have no significance (with exceptions such as in Euclidean graphs).

Various forms of graph theory have been used in the context of combinatorics to model problems in a formal manner. In particular, graph based models have been used to encode constraints within problems such as in [22] by Bujstas *et al* which models a bin-packing problem where some item types cannot be placed in the same bin, thus creating a conflict graph where the items are encoded as nodes and conflict between items as edges between those nodes. Travelling salesman problems are often two dimensional and modelled as planar graphs forming a Hamiltonian path in which the goal is to minimize the total length of all the edges [14]. Scheduling problems are normally turned into graph colouring problems where the problem can be represented by a conflict graph where the vertices are the jobs and the edges represent a conflict between two jobs sharing the same resources. Each time slot (or resource) is represented by a colour (or an integer) that is assigned to the vertices of the graph. This becomes equivalent to finding the chromatic number of the graph (which is still NP-hard) [94]. Scheduling problems can take different forms such as the Nurse Scheduling Problem [63], Exam Timetabling [139], the Job Shop Scheduling Problem [90], aircraft scheduling or TV program scheduling [117].

Graph theoretical approaches have been used to provide insight and basis for models in the area of Operations Research [66]. In [157] Wagner and Neshat measure and manage supply chain vulnerabilities using graph modelling. Graph based descriptions are used in the catalogue of global constraints in an effort to accelerate and improve the areas of constraint programming and related operations research activities [13]. An example of adaptive workflow scheduling that considers resource allocation constraints and network dynamics that are built upon a graph theoretical model can be found in [10].

In electrical engineering, graph theory provides an elegant formulation of Kirchhoff's laws [80], allows the modelling of electronic circuits, facilitates the synthesis of circuit paths [156], describes signal flows, assists in fault diagnosis [153] and finally it has been used to provide an underlying model for the automated design of efficient digital circuits by mean of evolutionary computation [100]. Graph theory is used for scheduling tasks in multi-core processors in order to simultaneously optimise performance and energy consumption [31]. In industrial engineering it is used to model the production planning and control, and assist with the design of the layout of physical facilities [143].

Graph theory is used as an abstraction tool in many successful industrial applications, where efficiency is a vital element, such as: Google's page rank algorithm [116], Amazon and Netflix recommendation systems[18], Drug Discovery [5], General Electric's power distribution system[53] and GSM mobile phone network frequency assignments[56]. While the success of a company is dictated by many other variables, it is encouraging that they choose graphs to model their framework. Nonetheless aligning the modelling standards of hyper-heuristics to one used in the industry could be valuable in its own right. The results of this research will provide an empirical evaluation to establish if these speculations are correct and to what degree.

Graph rewriting or graph transformation systems [77] are collections of substitution rules that define languages that modify graphs in order to produce new graphs. This type of system can be used to describe the mechanics of algorithms that aim at modifying or constructing graphs where graphs are solutions to combinatorial optimisation problems. Thanks to the work of Ehrig [52] on graph grammars and typed graphs, it was shown that graph transformation systems can be used as models of computation: further this model of computation has been shown to be Turing complete [71]. This means that, potentially, any program that can be computed by a Turing machine can also be simulated by a Graph Transformation system.

The notion of Graph Transformation comprises the concepts of graph grammars and graph rewriting. It is a discipline in Computer Science which dates back to the early 1970s. Its core idea is the rule-based modification of graphs, where each application of a graph rule leads to a graph transformation step [134]. Methods, techniques and results from the area of graph transformation have already been studied and applied in several fields of computer science, including compiler construction, software engineering, modelling of concurrent and distributed systems, database design and theory, logical and functional programming and artificial intelligence and visual modelling [7] [49].

Graph Transformation has three different roots [51]:

- from Chomsky grammars on strings to graph grammars,
- from term rewriting to graph rewriting,
- from textual description to visual modelling.

One of the main drivers for the creation of the concept of *Computing by Graph Transformation* was to move away from computation based on term rewriting and tree transformation, because trees do not allow sharing of common substructures which is one of the main reasons for efficiency problems in functional and logical programs. This computational model is applicable at low levels and is able to provide specification and implementation of systems by graph transformation. Graphs and their transformations provide an expressive way to represent entities via vertices and relationships between them as edges and in this model computation is achieved by describing the changes in the relations between the entities, the entities themselves or by changes in the attributes of the graph's elements.

Graph transformations are implemented via rule-based modification of graphs. The core of the rule or production $p = (L,R)$ is a pair of graphs L,R, called left hand side L and right hand side R. Applying the rule p means finding a match of L in a source graph and replacing L with R and connecting R with the context of the graph [50]. Where the context are all those elements connected to L. This can also be interpreted as a rule where L contains the pre-condition necessary for the application of the rule and R the post-conditions that are implemented after the application of the rule. This simple approach provides a mathematical formalism that gives a way to prove that a condition of the system is met and to modify the system while ensuring that the specified properties are preserved after some change.

It should be highlighted that while graph transformations and hyper-heuristics are both part of Computer Science they barely interact within or outside academia. The former is used to develop precise formalisms for graph databases, formal languages, model driven development, code generation and compiler optimisation with well defined policies [52], while the latter is mostly driven by heuristic rules and a lack of theoretical foundations which, in most cases, is developed in an attempt to understand how these algorithms can be so successful [122].

2.4 Summary

In this chapter we covered three broad fields presenting the usefulness of hyper-heuristics methods. We highlighted that the field focuses on combining strategies and often assumes that a set of low-level heuristics is available. Further, we noted that there is a lack of methods for generating low-level heuristics that can produce heuristics for more than one specific domain. Finally we considered the versatility of graph based representation that seem an ideal candidate for increasing the number of domains a representation can cover. We will now introduce some specific tools that play an important role in the proposed research.

Chapter 3

Theoretical and Practical Tools

In this chapter we introduce theoretical and practical tools that are employed to model key elements of the systems developed in this manuscript and used in the implementations of our experiments. It begins by introducing grammatical evolution that is a methodology inspired by evolutionary processes which can construct sequences in an arbitrary language. This methodology is used for the synthesis of both constructive and perturbative heuristics. Finally some elements of geometry are reviewed; a field precursor of graph theory, that is interested with points, lines, angles, distances, shapes and proportions. These notions are integral to the grammar developed for the synthesis of constructive heuristics in chapter 3 and to the idea of transforming a set of parameters into a point. This allows the use of spatial properties between points as a source of information which can be used to make a heuristic decision. The chapter concludes with a section on order and ordering heuristics which are pivotal in the mechanics of both constructive and perturbative heuristics.

3.1 Grammatical Evolution

Grammatical Evolution (GE) is a population based evolutionary computation approach used to construct sequences of symbols in an arbitrary language defined by a grammar. Michael O'Neill proposed the first version of a GE based algorithm in 1998 [136]. The main contribution of GE with respect to previous methods such as Genetic Programming (GP) [91] is the introduction of an intermediate mapping between genotype and phenotype.

Grammatical Evolution is taken in consideration, among other reasons such as its proven applicability over a large variety of domains [135] and the availability of software libraries that facilitate prototyping, due to the fact that the mapping mechanic uses the same principle, called production or rewriting rules, used by Graph Transformation Systems. These production rules substitute the symbols in the left-hand side of a rule and replace them with the symbols in its right-hand side. It is possible that this equivalent mechanic will facilitate the unification of hyper-heuristic processes under one model.

The mapping between genotype and phenotype is defined in a grammar by a collection of production rules which are selected by the values within a genotype. Each specific derivation sequence produces a specific phenotype which, in most cases, is an executable computer program. The result of the execution of the phenotype/program is then used to determine its fitness as in GP or other evolutionary computing methods. GE uses a search mechanism, most often a Genetic Algorithm (GA), to modify the genotype and explore its space of possible configuration. The most common way to specify a grammar is the use of a Backus–Naur form (BNF) notation. A BNF Grammar consists of a set of *production rules* composed of terminal and non-terminal symbols. The production rules are used to substitute the non-terminal symbols with other symbols, which can be both non-terminal or terminal symbols, repeatedly until a whole sequence of terminal symbols is composed. Each non terminal symbol has its own set of production rules. *Codons* (Our implementations represented as a single integer) specify which specific production rule should be chosen at each step.

BNF Grammar example:

```
<expr> ::= <expr> <op> <expr>
           | ( <expr> <op> <expr> )
           | <var>
<op> ::= +
       | -
       | /
       | *
<var> ::= X
```

In this example, the non-terminal symbol <expr> can produce three possible results. The specific choice will depends on the codon value within the genome. Similarly for the terminal node <op> a codon will select one of the four available options. The complete transcription will produce an arithmetic formula in one variable X.

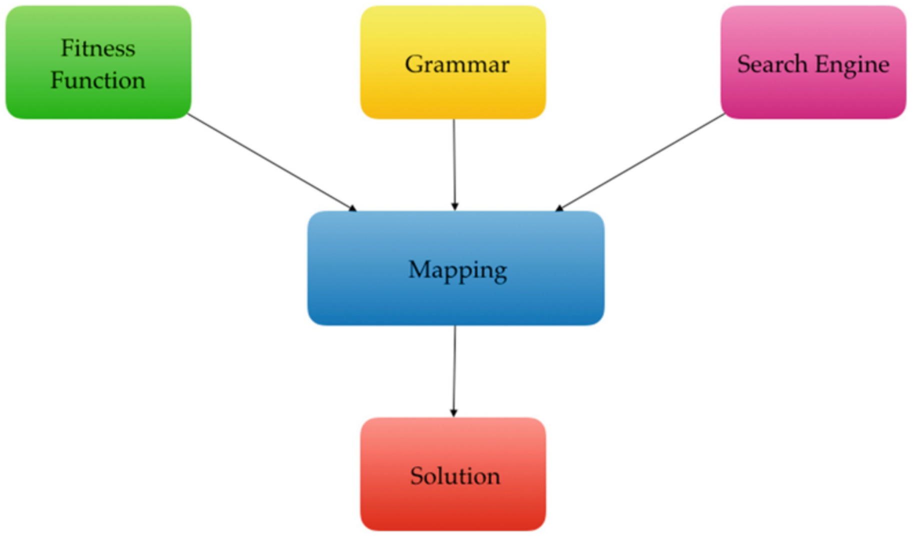


Figure 3.1: GE modular architecture

Figure 3.1 shows a basic diagram of the modules that are present in GE systems.

Fitness Function. The function, sometime called reward/objective/goal, that associates a fitness value to each genome. These values will determine if the genome is selected and used in successive iteration.

Grammar. The collection of production rules that transform a genotype into a phenotype. Each production rule substitutes the symbols on the left hand-side of the rule that appear in a string with the symbols on the right hand-side of the rule.

Search Method. The type of algorithm used to search over the space of genomes. In classical Grammatical Evolution a genetic algorithm is used as search method.

The modular nature of GE allows to swap any of the fundamental components with ease and a vast number of studies have been published in the past two decades on the effects that specific techniques have on different applications. This method and its variants have been used for a large number of applications in the past 21 years such as finance [112], where it has been used to estimate the credit rating for bond-issuer in a financial context [20], bioinformatics [102], neural networks [152], engineering [108], game AI [119], cyber security [78], classification and several others [135].

Grammatical evolution has also been used in the context of hyper-heuristics to evolve new heuristics used in time-tabling problems [137], 1-D packing problems [26] and vehicle routing problems [45]. It has been used to evolve Multi-Objective Evolutionary Algorithms applied to the Test Ordering Problem [98] and decision trees for classification problems. [12]. We highlight that in all these cases the grammar used and the heuristic generated are always applied to one specific domain.

3.2 Geometry

Geometry is a field of studies that dates back to the second millenia BC and is concerned with the measure and construction of geometric objects with the use of points, lines and areas [92]. It was used, and still is, in construction, measurement of distant objects and astronomy [151]. Geometry has been incorporated by several fields in order to provide a spacial frame of reference. This has been particularly successful in physics, engineering and architecture where spacial properties are naturally present in almost all applications of these fields [69]. Geometry has also been usefully applied to more abstract domains such as statistics where the idea of 'distance' between two probability distributions can be calculated using generalised forms of geometric distances [4]. This has given rise to the subject of information geometry.

One of the recent advances in geometry regards the study of combinatorial properties that arise when combining geometric objects. This field is called "discrete geometry" [99]. From this field we inherit the idea of measurable combinatorial properties within geometric objects. These concepts have been used in applications such as: tessellation [161], sphere packing [128], computer vision [64] and fluid dynamics [58].

Modern geometry definitions of geometric objects have become much more advanced and abstract that simple points and lines, which seek to be as general as possible by including tensors, bundles, jets and manifolds [70] [140]. However this is not the one used to model the problems taken in consideration in this manuscript.

The combination of graph theoretical and geometric notions gave rise to the field of Geometric Graph Theory [115]. In a broad sense this field is concerned with graphs defined by geometric means.

In the literature a geometric graph often refers to a graph embedded on a 2 dimensional Euclidean plane [114], however in this manuscript we will use the broader definition of geometric graphs where the graph can be embedded in any arbitrary space (both Euclidean and non-Euclidean). This idea will be used in the chapter dedicated to the synthesis of constructive heuristics (4) where the objects of the problems are considered vertices; this could be cities, items to be packed or processes to be executed. The vertices have coordinates associated with them which are the numerical values associated to the object in the problem, this could be the location of the city or constraints and value of the items. Once the vertices are placed in a space by some coordinate system their geometric features are used to make a decision on where to operate next. In our system we can change the space in which they are embedded as well, which alters the geometric features that can be measured over them.

3.3 Ordering Heuristics

Concepts related to order are ubiquitous in mathematics and computer science [42] [86] [123] [141]. Formally, a simply ordered set is a binary relation between elements of a set which is anti-symmetric, transitive and a connex relation.

We dedicate a section to it in the manuscript as it plays a vital role in the following 3 chapters. In both Chapters 4 and 5 the heuristics that we synthesise rely on the idea that problems that have no ordering requirements can be treated as ordering problems via special assumptions. For example in the knapsack problem a number of objects have to be chosen out of a set of objects. The order in which the objects are placed has no effect on its original formulation. However, we can heuristically pick one object at a time and subtract the constraints of this object from the total constraints available. This action alters what items are feasible to be picked next. This causes the ordering of the choices to have an effect on the solution that will be generated.

As it will be seen in Chapter 4 the technique developed is applicable to graphs by labelling their vertices via synthesised heuristic functions that give specific ranks to each single vertex. The problem becomes similar to other known combinatorial problems that revolve around ordering of vertices such as minimum linear arrangements [144] [6].

In chapter 5 we use a similar approach as each vertex of the graph is labelled by a unique integer imposing an ordering over the vertices of the graph. The perturbative heuristics synthesised by our system then are applied to the graph by re-ordering the vertices in some specific way that depends on the specific heuristic. In this particular case as each integer is unique they can be considered equivalent to permutations.

In chapter 6 we construct graphs with specific topological properties with the use of concepts drawn from order theory. In particular we consider each family of graphs as elements of a partially ordered set [48] in order to construct new families of graphs that are built by composing the properties of the lower level graphs similarly to the example in 3.2, also known as an Hasse Diagram.

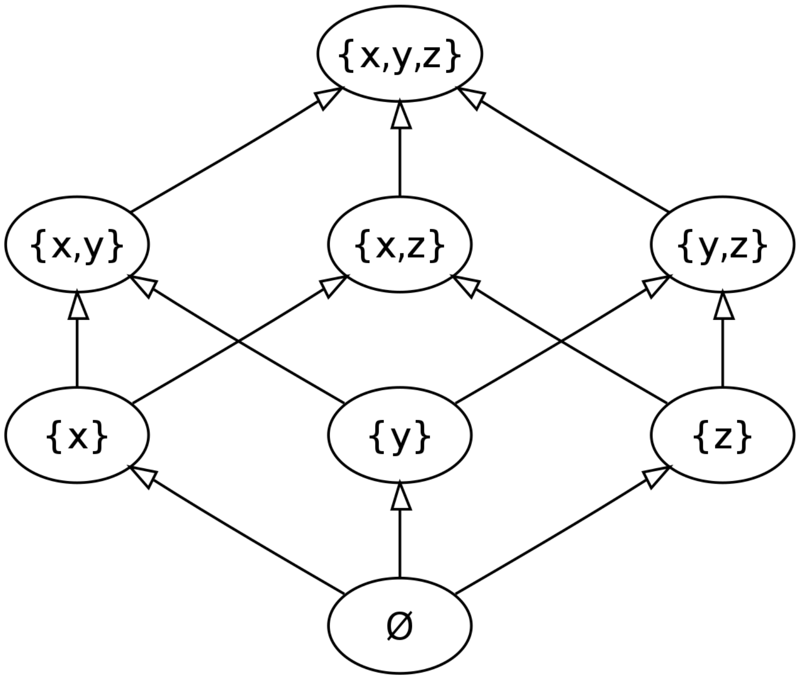


Figure 3.2: Example of a 3 elements Hasse diagram

In the following chapters we present a series of results on the production of heuristics using Grammatical Evolution. The results show how navigating the space of heuristics can lead to regions in which heuristics are good for one problem domain while other regions of the same heuristic space contains heuristics that are good for a different problem domain.

These engineering and empirical results are used to strengthen the idea of a necessity for a more abstract and higher level representation that can describe many elements and levels of the system.

Chapter 4

Synthesising Constructive Heuristics

4.1 Introduction

The canonical hyper-heuristic framework builds around the idea of algorithms that choose heuristics from a set of low-level heuristics separated by a concept known as a "domain barrier". The low-level heuristics are specific to a particular domain, and may be designed by hand, relying on intuition or human-expertise [23], or can be evolved by methods such as Genetic Programming [146]. The low-level heuristics can not be applied to domains apart from the one they were designed for as often they rely on different representations. The success of the high-level heuristic is strongly influenced by the number and the quality of the low-level heuristics available. Given a new problem domain that does not map to well-studied domains in the literature, it can be challenging to find a suitable set of low-level heuristics to utilise with a hyper-heuristic. Although this can be addressed by evolving new heuristics [11], this process requires in-depth understanding of the problem and effort in designing a specialist algorithm to evolve the heuristic. The aim of hyper-heuristic systems is to be more generally applicable, yet most of the recent developments revolve around domain specific solvers packaged into portfolios where solvers developed or synthesised for one domain cannot be applied to other domains. If a new domain has to be tackled, new representations and representation-specific operators usually need to be developed.

We propose to address this by introducing methods of creating new heuristics that are *cross-domain*. The methods can be used without modification to create heuristics in multiple domains, assuming a common problem representation. In this chapter we look at one of the major classes of hyper-heuristics: generative constructive heuristics, where some technique, in this case Grammatical Evolution, is used to produce new constructive heuristics. Constructive heuristics can be used to find an initial solution or as a fast way to produce a final output without further refinements. This approach works by starting with a completely empty solution and one is constructed by composing it one element at a time. It is fundamental for a problem solver to be able to go from the state of "no solution" to the one of "some solution". In this case, as it is usual in heuristic approaches, the solution is built one block at a time, chosen according to some evolved criterion.

In particular we study whether one hyper-heuristic generator can be used to produce constructive heuristics for more than one domain of combinatorial optimisation problems. The study should not be confused with an attempt to generate better heuristics for each specific domain or to demonstrate that the produced heuristics are "statistically better" than previous ones, but to gauge the performance of a system that is forced to operate over multiple different domains. This study looks at how the system performs if we assume that only one representation is available and we have to solve problems from more than one domain. The chapter addresses the following questions:

1. Can a single grammar be designed in such a way that it is capable of producing constructive heuristics for multiple domains? (assuming a common graph-based problem representation)

2. How effectively can the grammar be used to evolve a) reusable and b) disposable heuristics in multiple domains?

3. What is the cost, in terms of quality of the results, of using the same representation for problems that are radically different from each other, in this case the Multidimensional Knapsack Problem and the Travelling Salesman Problem, but both NP-hard?

Furthermore we investigate key differences in the components of the heuristics for each domain and highlight an affinity between packing problems and heuristics that make use of cosine distance metric to prioritise the order of packing.

4.2 Contribution

We describe a novel approach for the automatic generation of low-level constructive heuristics called constructive Hyper-Heuristic with Grammatical Evolution (cHHGE). Our approach is based on grammatical evolution which evolve programs that are applicable to geometric graphs in order to choose a sequence that will become a solution to a given combinatorial optimisation problem. The heuristics are tested on a set of well known benchmarks for the Travelling Salesman Problem and Multidimensional Knapsack Problem. The evolved heuristic outperform simple human written heuristics in both domains. The heuristics generated are further analysed and a relationship between the cosine distance between pair of objects in the Multidimensional Knapsack Problem and their preference order is found (Section 4.8.1).

4.3 Benchmarks

Many real world problems can be formulated as ordering problems, however we specifically choose two problems of which one is a classical ordering problem (routing) and the other is not (packing). This is due to the fact that we want to investigate further the idea that ordering representations and heuristics can be used outside their usual domain. Routing problems are a class of mathematical optimisation problems that formalise the task of visiting a number of locations while minimising some variable or set of variables such as tour length, fuel consumed, number of vehicles or agents used, risk, pollution generated. The most well known routing problem is the Travelling Salesman Problem (TSP) where some agent has to visit n cities and wants to minimise the length of the tour as in eq.4.1. Solutions of the problem come in the form of a list of weighted edges E = e_1,e_2,...,e_n that connect all the cities where each city is visited exactly once. A precursor of this problem was formalised by William Rowan Hamilton which also gave the name to the property of a graph in which every vertex is visited once, each vertex has degree (number of connected edges) equal to 2 and there is only one cycle forming an “Hamiltonian Cycle”.

Minimise:

$$\sum_{i=1}^{n} e_i \tag{4.1}$$

Packing problems are a class of mathematical optimisation problems that formalise the task of packing items into containers. The two most well known families of these problems are the Knapsack Problem, where there is one container that has to be packed with maximum density, and the Bin Packing Problem, where the goal is to use as few containers as possible but all the items have to be packed.

Knapsack problems have been studied since the work of Dantzig [41], and are researched due to their immediate applications to operational research, industry and financial management. A knapsack problem is normally presented as a linear programming problem subject to constraints:

Given a set of n items with profits p_i and weight w_i which are packed in one or more knapsack of capacity C as defined in eq. 4.2 and 4.3.

Maximise:

$$\sum_{i=1}^{n} p_i x_i \tag{4.2}$$

Subject to the constraints:

$$\begin{aligned} &\sum_{i=1}^{n} w_i x_i \leq C \\ &x_i \in \{0,1\} \end{aligned} \tag{4.3}$$

where x_i is a binary variable that describe if the i-th object should be placed in the knapsack(1) or not(0).

Knapsack problems have numerous applications, for example: one investor can choose from a pool of n projects and the profit obtainable from the i-th project is p_i = 1,2,3,...,n. It costs w_i to invest in the project i, and the investor has only C amount of money. An optimal investment plan could be found by solving the underlying Knapsack Problem. Another application can appear in the restaurant where a person has to choose k courses, without surpassing the amount of C calories as defined by his diet. Assuming that there are n dishes to choose for each course i = 1, 2, ...,k and w is the nutritional value while p is a rating that defines how good each dish tastes. An optimal meal can be found by solving a knapsack problem.

Knapsack problems appear also in cargo loading, the cutting of material and portfolio management. Bin Packing problems are usually classified depending on the dimensionality of the problem: **1-Dimensional:** normally deal with problems such as data packets, sets of jobs to be processed one at a time, organise containers of equal size.

2-Dimensional: mostly used to describe the optimisation of surface usage, such as a sheet of some material that has to be cut where the amount of unused material has to be minimised

3-Dimensional: describe problems regarding the volume of the objects and containers they are placed in, this is especially common in problems related to the transportation of goods.

Multi-dimensional: it is used in problems where there are many constraints, such as budgeting, where cost, time and number of employees available need to be taken in consideration. This tends to be the reality of many transportation problems as one must take into consideration various other factors on top of the volume of an object (i.e. the weight, decay time, temperature etc). This is the type of knapsack problem that is used in the all empirical evaluations. Specifically we use a set of benchmarks in which there is one knapsack with a number of arbitrary constraints and a collection of objects with a cost for each type of constraint and a profit value. The goal is to maximise the profit of all the objects in the knapsack while respecting the constraints.

4.3.1 Representation

In the proposed system we encode all the properties of the problems into a graph embedded in some arbitrary space. Whenever possible, we convert properties into some spatial concept to which we can associate some arbitrary metric. In the case of the Travelling Salesman Problem, the cities to visit can be trivially encoded as vertices in 2-D Euclidean space. For the knapsack problem we use one vertex for each object, and one vertex for the knapsack. The properties of the vertex can be interpreted as coordinates that determine the location of the vertices in some constraint-profit space. A geometric interpretation of the problem can be intuitively described as follows: when an object is chosen (connected to the knapsack vertex) the properties of the object are added to the knapsack and it is moved in the constraint-profit space. The amount of motion is equal to the values of the object's vector in constraint space and in the direction of the profit space. The configuration of objects connected to the knapsack that move the knapsack the furthest in profit space without the knapsack crossing the line corresponding to its maximum capacity in any of its constraint dimensions is the best configuration. An example with just one constraint (weight) is drawn in Fig. 4.1.

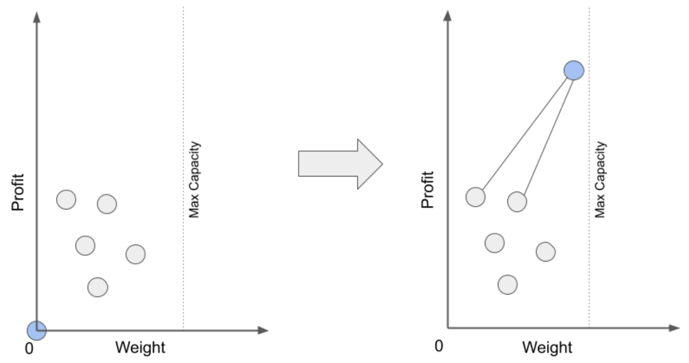


Figure 4.1: Geometric interpretation of the knapsack problem simplified to two dimensions (weight constraint and profit). On the left the knapsack at initialisation has 0 weight and 0 profit. As objects are connected to the knapsack it is moved by an amount equivalent to the vectors defined by the sum of the objects.

4.4 Grammatical Evolution

Grammatical Evolution is a population based approach used to construct sequences of symbols by exploring the space of codon values. The codons are used to chose the production rules, and the possible expansions within one production rule, in a given grammar. The production rules are then used to produce the symbols in the defined language. In our case the language described by the grammar is a subset of the programming language Python. Our implementation builds on top of the Python implementation of Fenton *et al* [54]. Their implementation proved to be accessible, straightforward to reuse and is the most recent version of GE. A description of the algorithm can be found in Chapter 3.1, and a complete description plus additional information can be found in the original paper [54]. The code is also open-source and available on *github*[1]. Implementation specific details are as follows:

Genome. Fenton's implementation uses a linear genome representation that is encoded as a list of integers (codons). Codons are responsible for selecting the symbols in production rules with more than one possible product (branches). The mapping between the genotype and the phenotype is actuated by the use of the modulus operator on the value of the codon, i.e. *Selected node* $= c$ mod

[1]https://github.com/PonyGE/PonyGE2

n, where c is the integer value of the codon to be mapped and n is the number of options available in the specific production rule.

Mutation. An integer flip at the level of the codons is used. One of the codons that has been used for the phenotype is changed each iteration and substituted with a completely new codon.

Crossover. Variable one-point crossover, where the crossing point between 2 individuals is chosen randomly.

Replacement. Generational replacement strategy with elitism 1, i.e one genome is guaranteed to stay in the pool on the next generation.

Selection. A Tournament based selection of size 2 where the best of each pair is selected.

4.5 Geometric Graph Grammar

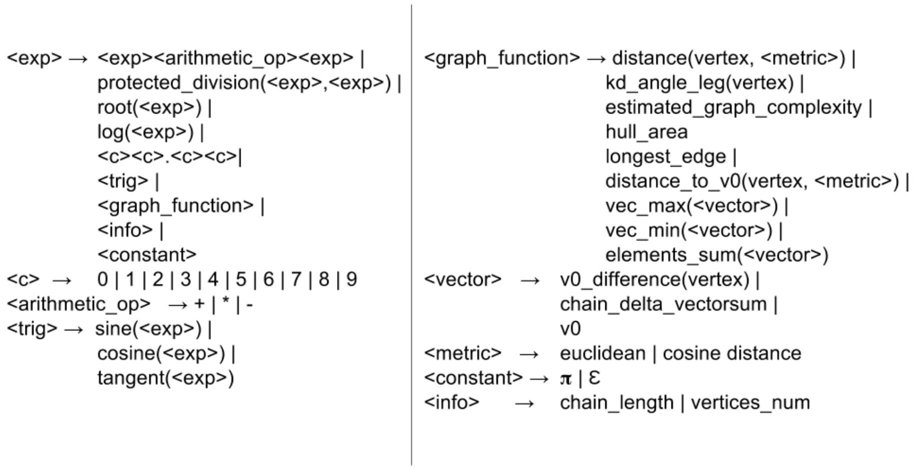
The specified grammar is the heart of a Grammatical Evolution algorithm [111]. Symbols in the grammar can be whole heuristics, mathematical operations, raw numbers and even elements of other grammars. The grammar is typically defined using Backus-Naur Form (BNF) [113], a notation used to express grammars in the form of production rules. Similar to the mechanics used in Genetic Programming, BNF consists of terminal nodes and non-terminal nodes. Terminal nodes are symbols that can appear in the final language and do not have any successive production rules associated with them. Terminal nodes are also known as leaf nodes, external nodes or outer nodes in other applications that rely on tree data structures. Non-terminal nodes are nodes that can be expanded into one or more nodes (either terminal or non-terminal). Non-terminal nodes are also known as branch nodes, internal nodes or parent nodes. In BNF notation non-terminal nodes are surrounded by <>brackets. Production rules consist of instructions on how to substitute (or expand) non-terminal nodes, starting from a specifically defined non-terminal node, until all the nodes are terminal nodes. This is often denoted as *start*. Production rules offer the possibility to substitute one non-terminal node with more than one possible choice. The different substitution choices are delimited by the symbol " | ". This is a key element of the notation: in Grammatical Evolution the codons of the genome specify exactly which choice should be made on each branch. In the grammar we have specified that the terminal nodes are pieces of python programs that can be composed to create an executable python program which is used as one of our heuristic ranking functions.

Here, we restrict ourselves to using only basic arithmetic operations and geometric properties of the given graph (either given or derived). One of the goals is to produce a system that is independent from the specific problems we are going to solve, yet reducible to concrete or abstract spacial concepts. This includes, measuring distances over a metric space, counting (vertices), measuring areas, measuring angles, arithmetic operation and basic trigonometry operations. Within the grammar there are a number of terminal nodes that are custom functions and properties that the generator can access:

- *distance(vertex, metric)* returns the distance between the last chosen vertex and the currently evaluated vertex over a specified metric.
- *euclidean* selects the euclidean distance as a metric
- *cosine distance* selects the cosine similarity as a metric
- *kd-leg-angle(vertex)* returns the angle that would be formed by connecting the currently evaluated vertex to the previously chosen vertex
- *estimated graph complexity* returns the difference between the total number of vertices and the number of vertices in the convex hull constructed around the graph. We do not give the program the actual vertices of the hull as this would count as a heuristic and facilitate excessively the algorithm.
- *hull area* returns the area of the convex hull constructed around the graph
- *longest edge* returns the highest value in the distance matrix
- *v0* is an optional vertex that can be used as a reference. In the TSP solvers it is used to store the starting point of the tour. In the MKP solvers it is used for the knapsack.
- *chain delta vector sum* returns the vectorial sum of the vectorial differences of each pair of vertices in the order in which they are in the chain.
- *distance to v0* returns the distances between the currently evaluated vertex and the reference vertex using a specified metric

- *vec_max* returns the highest value of the given vector
- *vec_min* returns the smallest value of the given vector
- *elements_sum* returns the sum over all elements of a vector

The grammar used to generate heuristics for both TSP and MKP can be seen in fig. 5.3 (the rules are the same for both problems and all rules are used).



```
<exp> →  <exp><arithmetic_op><exp> |
         protected_division(<exp>,<exp>) |
         root(<exp>) |
         log(<exp>) |
         <c><c>.<c><c>|
         <trig> |
         <graph_function> |
         <info> |
         <constant>
<c>  →   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<arithmetic_op>  → + | * | -
<trig> → sine(<exp>) |
         cosine(<exp>) |
         tangent(<exp>)

<graph_function> → distance(vertex, <metric>) |
                   kd_angle_leg(vertex) |
                   estimated_graph_complexity |
                   hull_area
                   longest_edge |
                   distance_to_v0(vertex, <metric>) |
                   vec_max(<vector>) |
                   vec_min(<vector>) |
                   elements_sum(<vector>)
<vector>   →  v0_difference(vertex) |
              chain_delta_vectorsum |
              v0
<metric>   →  euclidean | cosine distance
<constant> →  π | Ɛ
<info>     →  chain_length | vertices_num
```

Figure 4.2: Complete grammar used to build the heuristics expressed in BNF notation

4.6 Experimental Setup

It is well known that having better training instances leads to better outcomes [23]. However, as the ultimate goal of this work is to produce a system that can produce acceptable heuristics in an unknown domain in which good training examples might not be available (or in an existing domain in which we cannot predict characteristics of future problems) in the following experiments we synthesise a random set of training instances in each case.

4.6.1 Reusable Heuristics

The first experiment generates *reusable* heuristics. A training set is synthesised for each domain. For TSP, a set of cities are generated using a uniform-random distribution over 2 dimensions. Each

Algorithm 1 Generate Constructive Heuristic

```
$T \leftarrow$ collection of training instances ($t$)
$h \leftarrow$ heuristic
$p \leftarrow$ population of heuristics
Init $p$
while stop != false do
  Select $p'$ from $p$
  Crossover $p'$
  Mutate $p'$
  for each $h$ in $p'$ do {Evaluate heuristics}
    $Tf \leftarrow$ list of fitnesses
    for each t in T do
      $u \leftarrow$ unchosen vertices
      $c \leftarrow$ solution chain
      while $u > 0$ do
        for each $v$ in $u$ do {Rank unchosen vertices}
          Rank $v$ according to $h$
        end for
        $v^* \leftarrow$ Lowest value in u
        append $v^*$ to $c$
        remove $v^*$ from $u$
      end while
      $f \leftarrow$ fitness of $c$ [The formula is domain specific]
      append f to Tf
    end for
    $h's$ fitness$\leftarrow$ Tf's median
  end for
end while
best $\leftarrow$ Highest value in p
return best
```

instance has 50 cities. For MKP, instances have 20 objects with 10 constraints each. 10 instances are generated for each domain. A uniform random distribution is used to generate the object constraints and normal distributions for the profits of the objects and the constraints of the knapsack. The exact parameters of the synthesisers can be found in Table 4.3. The chosen parameters are not the result of a fine tuning process, but are for the most part default values and in part chosen in such a way to produce outputs within a reasonable time-frame. For MKP, the mean profit of a single object is proportional to the total constraints of the objects and the size of each knapsack is large enough to accommodate, on average, half of the objects in the problem instance. This is to ensure that there are no cases in which it is possible to fit either all the objects in the knapsack or no objects at all, which would make the instance useless for training purposes. As in all learning algorithms, the quality of the training set is important. However, here, we aim to simply give the generator minimal examples of the type of problem it might expect to solve, assuming we have no *a priori* information regarding the specific distributions of points that might be found in future problems. For both the TSP and MKP instances, the fitness of a heuristic on the training set is calculated as the *median* fitness returned from the set of training instances.

For each domain we generate heuristics using the method described in Algorithm 1 and repeat the process in order to produce 5 TSP heuristics and 5 MDK heuristics. We test the TSP heuristics over 9 instances from TSPlib benchmark [129]; and we test for MKP heuristics over 7 instances of the mdknap1 benchmark from the OR-library[2]. In the case of the TSP instances, due to the fact that there is a difference of more than 2 orders of magnitude between the synthesised instances and the test problem, we normalise the coordinate of the data to the range 0-100. For TSP, we compare against two human-designed heuristics, *nearest-neighbour(NN)* and *insertion* based on the minimum spanning tree *(MST)* [124]. These heuristics are always applied starting from the same initial city and hence provide the same results. For MKP, we compare our generated heuristics against a greedy *depth first search* algorithm [87].

The experiments are run on a desktop computer with a Intel i5 3.3Ghz CPU and 8 GB of memory.

[2]http://people.brunel.ac.uk/ mastjjb/jeb/info.html

Parameter	Value
Number of Generations	100
Population	100
Mutation	int flip
Crossover Prob.	0.80
Crossover type	one point
Max initial tree	10
Max tree depth	17
Replacement	generational
Tournament size	2

Table 4.1: Resuable Heuristics

Parameter	Value
Generations	200
Population	100
Mutation	int flip
Crossover Prob.	0.80
Crossover type	one point
Max initial tree	10
Max tree depth	17
Replacement	generational
Tournament size	2

Table 4.2: Disposable Heuristics

4.6.2 Disposable Heuristics

The second experiment generates *disposable* heuristics. This means that for each problem instance we generate one specialised heuristic that uses the result of only one specific problem as the fitness function. Normally disposable heuristics work well on the instance they were created for but very poorly on other instances even of the same domain. For each instance we run GE for 200 iterations with a population of 100 to produce a single heuristic for the specific instance. Both experiments are repeated 20 times and we account for best, median and worse cases for each scenario. Creating disposable heuristics is considerably more computationally expensive therefore the experiments are run on ”n1-standard-1” instances of Google Compute Engine. The instances have 1 virtual CPU using 2.3 GHz Intel Xeon E5 v3 (Haswell Architecture) and 3.75GB of Ram. Scenarios that are run multiple times for statistical validation use multiple independent virtual machines.

4.7 Results and Analysis

Reusable Heuristics: The results in Table 4.4 and Figure 4.3 show a comparison of the best, worse and median fitness of the evolved heuristics on each problem of the TSP benchmark. These are compared to the deterministic values obtained from a single run of the human heuristic. It can be seen that in 5 out of 9 instances the median performance of the evolved heuristics is better than both the simple human heuristics. In all 9 instances at least one evolved heuristic is better than both the human heuristics.

Parameter	Value
Number of cities	50
Cities distribution type	Uniform
Cities distribution range	0-100
Number of objects	20
Number of constraints	10
Object constraints distribution	Uniform
Object constraints range	0-100
Object profit distribution	Normal
Object profit mean	Sum of constraints
Object profit deviation	15
Knapsack constraints distribution	Normal
Knapsack constraints mean	500
Knapsack constraints deviation	100

Table 4.3: Parameters of the problem synthesiser used to produce the traning instances

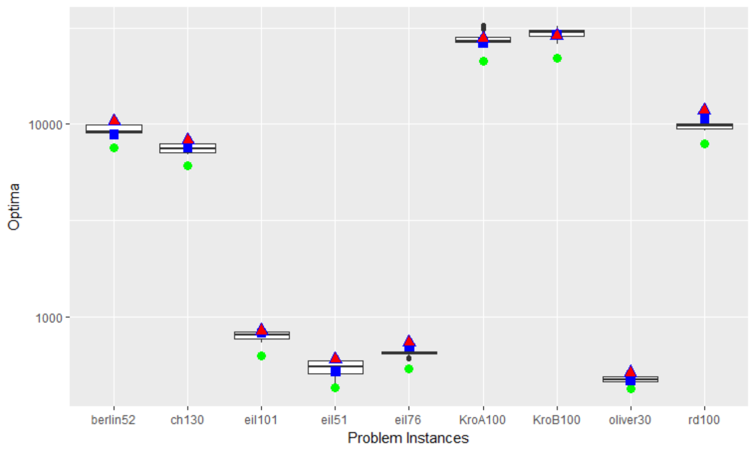


Figure 4.3: Reusable Heuristics (TSP): optima (green circles), nearest neighbour (blue squares) and minimum spanning trees (red triangles)

Instance Name	Optima	NN	MST	Rand	cHHGE Median	cHHGE Best	cHHGE Worse
berlin52	7542	8868	10404	30290	9196	**8452**	*10515*
ch130	6110	7575	8277	46099	**7501**	**6942**	*8469*
eil101	629	826	846	3434	**803**	**736**	897
eil51	426	521	605	1635	547	**451**	*620*
eil76	538	700	739	2494	**652**	**603**	678
KroA100	21282	26506	27921	171940	26914	**25603**	*32683*
KroB100	22141	29264	28804	170564	30182	**26064**	*32213*
oliver30	423	464	513	1379	474	**449**	499
rd100	7910	10733	11855	55064	**9863**	**9260**	11338

Table 4.4: Reusable Heuristics: TSP. Comparison between nearest neighbour, minimum spanning tree heuristic and constructive Hyper Heuristics with Grammatical Evolution(cHHGE)

From Table 4.5, we can see that the best evolved heuristics for the MKP outperform the greedy (deterministic) heuristic in 4 out of 7 instances, where in 2 cases the global optima is reached by the best heuristics. The worst performing evolved heuristics are still better than the greedy heuristic in 3 out of 7 instances. The median profit reached by the evolved heuristics is better than the greedy heuristic in 2 out of 7 instances instances but an overall statistical analysis using the Wilcoxon rank test highlighted that they are not significantly better and should be considered on par with the greedy depth first heuristic.

Instance	optima	worst rand	median rand	best rand	worst greedy	median greedy	best greedy	worst cHHGE	median cHHGE	best cHHGE
mknap1-1	3800	100	1700	3100	1200	2700	3800	**1800**	3300	**3800**
mknap1-2	8706.1	1482.1	5059	8687.5	4340.7	6504.8	8650.1	4212.4	7059.8	**8706.1**
mknap1-3	4015	985	2235	2860	1895	3325	3765	**2390**	2480	3725
mknap1-4	6120	1320	3240	5820	2460	3525	5390	**2480**	3020	**5640**
mknap1-5	12400	3770	7770	10340	7590	8990	11550	6855	8150	10510
mknap1-6	10618	4286	6566	9770	7400	8032	10345	7238	7641	9352
mknap1-7	16537	5661	9509	12769	8770	12363	15330	8335	10887	**15668**

Table 4.5: Reusable Heuristics: Multidimensional Knapsack Problem. Comparison between random search, a greedy depth first heuristic and constructive Hyper Heuristics with Grammatical Evolution(cHHGE)

Disposable Heuristics: The results in Table 4.6 show the best and median fitness obtained by evolving a single heuristic per instance. The generator constructs a heuristic that finds the global optima on the simplest instance (oliver30). It reaches fitness within 15% of the global optima in 8 out of 9 instances.

The results in Table 4.7 show the best and median fitness reached by the generator directly used on each problem of the MKP benchmark. Here, the global optima is found on 3 out of 7 instances, even in the worst run. In 4 out of 7 instances, the global optima is reached by the median performing generator. In one instance the global optima is missed by only 0.24%. Finally the very worst result of all 140 runs on the MKP benchmark is only 3.57% away from the global optima which happens in the hardest instance.

Instance name	Optima	Best	Median
berlin52	7542	8212.09	8972.69
ch130	6110	6781.6	7171.57
eil101	629	724.62	775.47
eil51	426	474.19	509.86
eil76	538	592.08	638.08
KroA100	21282	23866.88	26858.24
KroB100	22141	25019.18	27959.35
oliver30	423	**423**	442.56
rd100	7910	8796.43	9369.94

Table 4.6: Disposable Heuristics (TSP)

Instance name	Optima	Best	Median
mknap1-1	3800	**3800**	**3800**
mknap1-2	8706.1	**8706.1**	**8706.1**
mknap1-3	4015	**4015**	**4015**
mknap1-4	6120	**6120**	**6120**
mknap1-5	12400	12370	12355
mknap1-6	10618	10532	10462
mknap1-7	16537	16260	16022.5

Table 4.7: Disposable Heuristics (MKP)

4.8 Phenotype Analysis

A table with the complete list of heuristics evolved can be found in Appendix A. Below we report a few examples taken from the results that are particularly small and intelligible.

Listing 4.1: MKD Heuristics

```
a)

np.exp(np.exp(self.distance\_to\_v0(vertex,'cosine')))


b)

psqrt(self.distance\_to\_v0(vertex,'cosine'))-

np.sin(plog(np.exp(self.distance\_osms(vertex,'cosine'))))
```

The python code can be interpreted in natural language as follows:

The priority of the current vertex is equal to:

a) The Euler number to the power of the Euler number to the power of the cosine distance between the current vertex and the container vertex.

b) The root of the cosine distance between the current vertex and the container vertex minus the sine of the distance between the last vertex of the chain and the current vertex

Listing 4.2: TSP Heuristics

```
c)
self.distance\_osms(vertex,'euclidean')-
np.exp(np.cos(self.distance\_to\_v0(vertex,'euclidean')))

d)
self.distance\_osms(vertex,'euclidean')*pdiv(psqrt(len(self.vertices\_vecs)),
psqrt(self.distance\_to\_v0(vertex,'euclidean')))
```

c) The distance between the last vertex of the chain and the current vertex minus the Euler number at the power of the cosine of the Euclidean distance between the current vertex and the starting vertex.

d) The Euclidean distance between the last vertex of the chain and the current vertex times the ratio between the square of the length of the chain and the square of the Euclidean distance between the current vertex and the starting vertex.

These heuristics albeit convoluted and hard to interpret can be understood with enough familiarity with geometry. This is an highly desirable feature that many modern successful algorithm (i.e. neural network) do not possess.

4.8.1 Distance metrics usage in the evolved heuristics

The result of the experiments where we produced a collection of reusable heuristics in Table 4.5 highlighted that the phenotype of the evolved heuristics for the knapsack problem appears to use the gene that takes a measure of the cosine distance when ranking items to be packed in almost all cases. While the TSP heuristics use, unsurprisingly, Euclidean distance. Therefore we evolved a supplementary set of heuristics with the goal of measuring the frequency of these genes. In total we create 100 new heuristics for each problem domain then we account for the usage of cosine distance metric and Euclidean distance metric within the heuristics. The results confirmed this phenomenon as shown in Table 4.8. While it is easy to see why the Euclidean distance is useful when solving TSP problems, it is not clear why the cosine distance is so effective in the context of Knapsack problems.

4.9 Analysis of the Search Parameters

In this section we expand the range of benchmark instances on which we test our heuristics and perform a parameter sweep of the Grammatical Evolution around a range of values to verify the consistency of our results. A total of 20 instances taken from the TSPlib benchmark are used. For each instance 10 heuristics are trialled and compared with 5 well-known human heuristics. We account for the best median and worst case scenario in each instance.

While for the MKP we test the heuristic on 6 different benchmarks, with a smaller group of 5 benchmarks that contain 54 instances and a larger benchmark with 270 instances for a total of 324 instances. The results are compared with a greedy best fit heuristic. Finally we verify the robustness of the heuristics generator by performing a parameter sweep using a Monte Carlo method that samples the parameter space of the generator over a range of values. Each sample consists in a specific configuration of the generator that will output one heuristic that is then used to optimise a random set of instances taken from the TSPlib and Chu and Baes [34]. In the case of the TSP we have removed instances that were too easy and increase the overall number slightly. In the case of the MKP we have substantially increased the number of instances as this is the secondary domain which uses an usual representation for which more rigorous testing is required. The results are examined using a Kolmogorov-Smirnov test in order to assess if changing a given parameter effectively changes the performance of the generator. We provide a confidence measure for each parameter taken into consideration.

4.10 Results

	Euclidean	Cosine
TSP	100	12
MKD	17	97

Table 4.8: Frequency of occurrence of the distance metrics (Euclidean/Cosine) in each of 100 heuristics evolved for the each of the two domains

Dataset	%-gap to optima	
	Evolved Heuristic	Greedy
hp	7.386	6.699
pb	5.138	4.562
pet	4.370	7.174
sento	3.231	2.280
weing	3.852	5.224
weish	4.870	7.277

Table 4.9: Average % gap of the best result on MKD instances compounded by dataset

	Human Heuristics					Reusable Evolved Heuristic		
instance	nearest	cheapest	farthest	arbitrary	nn	Best	Median	Worst
berlin52	9004.9	8983.0	8278.4	8476.2	9313.3	8361.7	8666.7	8971.7
bier127	145508.8	139858.8	127820.7	131273.5	147631.0	127995.2	130854.1	133713.0
ch130	7362.2	7158.3	6644.5	6677.1	7950.3	6734.7	7051.7	7368.7
d198	17831.5	17555.9	16484.1	16718.7	18636.3	16582.1	17365.5	18149.0
eil101	740.8	731.2	681.2	697.4	864.6	691.0	714.7	738.3
eil51	494.8	490.6	450.8	465.6	559.1	458.1	467.3	476.5
eil76	626.0	618.2	586.8	594.5	659.8	590.7	600.0	609.3
kroA150	31153.7	30558.6	28770.4	28848.6	33737.9	28979.4	30016.1	31052.8
kroA200	36362.3	35088.7	31745.0	32791.7	37297.1	32285.5	32596.0	32906.5
kroB150	31973.4	31010.4	27798.9	28281.9	34371.2	27991.2	29228.9	30466.6
kroB200	36326.0	35780.0	32121.2	32012.5	36707.8	32483.0	32650.1	32817.2
kroC100	25807.6	25334.1	21801.8	22827.8	26173.9	22194.8	22527.1	22859.4
kroD100	25199.2	25252.2	22555.9	22923.3	27539.2	22799.2	23500.1	24201.0
kroE100	27168.1	25744.3	23223.6	23838.4	27647.5	23331.1	23716.2	24101.4
lin105	18323.9	17320.3	15559.3	15768.3	18977.7	15647.9	15897.0	16146.0
pr107	52443.7	51185.0	45332.4	45584.1	50678.8	45835.0	48080.3	50325.7
pr136	107128.5	110219.2	105483.2	105029.0	123294.3	106275.5	107913.8	109552.1
pr144	73619.0	71672.1	61710.1	63266.3	65606.2	62731.2	63873.4	65015.7
pr152	87315.8	89237.4	76349.8	78163.8	86472.6	76904.3	80675.9	84447.4
pr226	99460.6	92345.1	82129.1	84146.7	99190.9	83535.0	85518.6	87502.3
u159	51567.0	49852.0	46507.3	46747.8	54587.4	46770.9	47227.2	47683.5

Table 4.10: Comparison between 5 different human heuristics taken from the 'R' package for TSP and the evolved heuristics for the TSP

4.11 Discussion

In order to give a more precise account of the frequency with which specific distance measures are used within the heuristics, we created an extra 200 reusable heuristics. For each heuristic we simply count if the gene that is associated with each metric, Euclidean or cosine, is present and we show the totals in the Table 4.8. We count one if the gene is present at least once, which means that

Parameters	Gens.	Pop.	Samples	Crossover	Tree size (start)	Tree size (max)	Tournam. size
Range	20-100	20-100	400-10000	50-90	5-10	10-20	2-4
Effect(p)	0.09	0.7	0.03	21.7	32.5	12.4	43.6

Table 4.11: Relationship between change in parameter within the range and effect on the generator (the lower the number the greater the chances of affecting the generator)

if the distance is used multiple times in different parts of the heuristic we still count one. It can be seen that all the TSP heuristics make use of the Euclidean distance. This is not surprising as the Euclidean distance is part of the fitness function. Again 97 out of 100 heuristics for the multidimensional knapsack problem make use of the cosine distance confirming that there is some correlation between this type of metric and the type of problem. This experiment is not sufficient to explain what is the reason behind these results, however we can venture into some conjectures: as the cosine distance is a metric of symmetry or proportions between the variables of two objects, those objects that are similar or have ratios similar to the container or the amount of constraints still available in the container are preferable when having to pack them.

The reusable TSP heuristics have been trialled on a larger set of problems from the TSPlib and compared with 5 very well known heuristics used to initialise TSP problems. These heuristics are taken from the 'R' package for the TSP. Specifically nearest insertion, cheapest insertion, farthest insertion, arbitrary insertion and nearest neighbour. The results are collected in Table 4.10 where the human heuristics are juxtaposed to the best, median and worst performance of the evolved heuristics. The median evolved heuristic consistently performs better than 4 out of 5 human heuristics with the exclusion of the farthest insertion heuristic. This human heuristic differs from the other human and evolved heuristics as it is a multi steps heuristic which may be the key factor that gives it a performance advantage.

The reusable MKP heuristics have been trialled on a much larger set of instances from 7 different datasets. We increase our experiments on the knapsack problem as the representation we have chosen is commonly used for the TSP and we want to verify that the system is able to tackle new domains too, in this case the MKP. We compare the results of the evolved heuristic and one common human greedy heuristic on the first 6 smaller benchmarks in table A.1 (see appendix), a summary of the results presented as % gap between the known optima and the average best compounded by dataset can be seen in 4.9. The experiments show that the heuristics obtain results that are comparable or marginally better than the baseline greedy heuristic.

The experiments performed on the largest benchmark (from Chu and Beasly [34]) contains 270 instances and in this case we compounded the results into gap ratios (percentage) for each bundle of instances as this is the most common approach in the literature. This allows us to compare our results with those from the literature. In Table 4.12 it can be seen that the average performance of our heuristics is better than two other human heuristics, specifically designed for the MKP, in 7 out of 9 times. However it should be noted that the improvements are only marginal. Finally a sweep of the generator's parameter has been performed in order to verify which parameters most greatly influence the performance of the generator. It can be seen that in Table 4.11 the strongest factor is given by the total number of samples used during evolution (given by population size and number of generations), while parameters such as crossover, initial tree size and tournament size, within the selected range, are very weakly correlated to the final performance of the heuristic generated.

4.12 Summary

This section presented a study on a generator of heuristics and its application to two different problem domains in combinatorial optimisation. We proposed a representation based on geometric graphs and partial permutations with a method to automatically generate low-level constructive heuristics that are applicable to instances of both the Travelling Salesman Problem and the Multidimensional Knapsack Problem. In both problems the heuristics work by constructing a solution one vertex at a time using a *rank and choose* approach (where the heuristics that assigns priorities to vertices adapt to use different criteria depending on the problem domain). We have shown that the proposed grammar and representation, operated by Grammatical Evolution, can be used to develop both reusable and disposable heuristics using very few simple examples. This empirical study has shown that, given a “sufficiently general” grammar, it is possible to create heuristics that are comparable with human made heuristics by searching in the space of heuristics. The analysis of the heuristics generated by the grammar highlighted that the use of a cosine distance metric is useful when a priority has to be assigned to items in a Multidimensional Knapsack Problem. Similarly to how the Euclidean distance provides useful information in the TSP (intuitively cities that are nearer to the current city should be preferred) objects that are similar, in the sense of ratios between the constraints, to the container or the room available in the container should be preferred.

In this chapter we focused on constructive heuristics which is one of the two main approaches to provide solutions to optimisation problems in hyper-heuristic approaches, the other being iterative methods that repeatedly modify a solution by applying small changes to it. This will be the focus of the next chapter where similar ideas will be applied to the generation of perturbative heuristics.

problem			%-gap		
m	n	a	M&O	V&Z	Ours
5	100	0.25	11.57	10.64	9.88
		0.5	6.54	7.63	6.64
		0.75	4.79	5.29	5.12
		average	7.63	7.85	**7.21**
5	250	0.25	7.69	8.46	7.52
		0.5	4.44	4.12	4.2
		0.75	3.65	4.34	3.58
		average	5.26	5.64	**5.10**
5	500	0.25	5.08	4.06	4.25
		0.5	3.37	3.13	3.02
		0.75	2.87	1.77	1.56
		average	3.77	2.99	**2.94**
10	100	0.25	16.06	13.94	12.85
		0.5	10.62	9.68	9.6
		0.75	5.66	5.68	5.46
		average	10.78	9.77	**9.30**
10	250	0.25	10.62	9.86	8.23
		0.5	6.74	6.41	6.35
		0.75	4.36	3.74	4.12
		average	7.24	6.67	**6.23**
10	500	0.25	9.22	8.42	8.37
		0.5	4.63	4.54	4.35
		0.75	3.33	2.89	3.59
		average	5.73	5.28	5.44
30	100	0.25	17.9	16.41	15.94
		0.5	11.09	9.93	10.45
		0.75	6.6	6.81	6.52
		average	11.86	11.05	**10.97**
30	250	0.25	12.75	12.11	12.34
		0.5	8.34	7.46	6.12
		0.75	4.34	3.9	4.51
		average	8.48	7.82	**7.66**
30	500	0.25	10.34	9.37	9.65
		0.5	6.78	5.47	5.33
		0.75	3.96	3.21	3.15
		average	7.03	6.02	6.04

Table 4.12: Disposable heuristics applied to 270 instances of the MKP grouped by problem size. Where m is the number of dimensions, n the number of objects and a the ratio between the number of objects and how many can be fitted in the knapsack

Chapter 5

Synthesising Perturbative Heuristics

Following the work on the synthesis of Constructive Heuristics in this chapter we look at the second major class of hyper-heuristics, where some generative technique is used to produce new perturbative heuristics. Perturbative heuristics can be used after having initialised a solution using some special techniques or simply by using a randomly initialised solution. This approach proceeds by applying "small changes" to the solution, sometimes called 'local moves' or neighbourhood operators, and then decide if the new solution is accepted by some criteria (i.e. if the new solution is better keep it, otherwise keep the previous solution). The solvers synthesised in the study are part of a class called iterated local search. ILS algorithms solve problems by using an already available solution and altering it "slightly" according to some *move* operator. Move operators are also known as perturbations and the move operators based on heuristic decisions are called *perturbative heuristics*.

Perturbative Heuristics are heuristics that start from an already complete solution and attempt to improve it by perturbing some property of the solution while maintaining its validity. This could affect the order of the items in the solution or the values of the variables in the solution depending on the specific problem class. Perturbative heuristics can be applied an arbitrary number of times to the solution, usually determined by the number of iterations, amount of time passed or until the fitness attained by the solution stops improving.

One of the key advantages of perturbative heuristics is their ability to provide an answer if interrupted half-way through the run. This is thanks to the fact that these heuristics work by producing a stream of new solutions and if interrupted can provide an answer simply by returning the last or best-so-far solution.

5.1 Contribution

We describe a novel approach for the automatic generation of low-level perturbative heuristics. Our approach is based on grammatical evolution which evolve programs that are applicable to sequences via the modification of a given solution of a given combinatorial optimisation problem. The heuristics are tested on a set of well known benchmark for the Travelling Salesman Problem, Multidimensional Knapsack Problem and an additional set of Load Balancing Problems developed as part of this chapter. The heuristic generated are analysed and compared with the results obtained by other domain specific techniques taken from the literature.

5.2 Method

Our generator makes use of Grammatical Evolution [109] for the production of new heuristics and we will refer to it as *pHHGE* (perturbative Hyper-Heuristic Grammatical Evolution). In particular we specify *one* grammar and this single grammar is used to produce heuristics in three different domains. Our method can be described by four fundamental steps:

1. Represent the problem-domain of interest as an ordering problem
2. Use Grammatical Evolution to breed heuristics that perturb the order of a solution, using a small training set of examples.
3. Evaluate the heuristics according to their effectiveness as a mutation operator in an iterated local-search algorithm.
4. Re-use the evolved heuristics on unseen instances from the same domain

5.3 Grammar and mechanics of the operator

The operator constructed by our grammar can be thought of as a form of k-opt that is configurable and includes extra functions to determine where to break a sequence. The formulation and implementation is vertex centric instead of edge centric. The mechanics of the algorithm are as follows:

Number of cuts: This determines in how many places a sequence will be cut creating $(k-1)$ subsequences where k is the number of cuts. The number of possible loci of the cuts is equal to $n+1$, where n is the number of vertices (the sequence can be cut both before the first element and after the last element).

Location of cuts: The grammar associates a strategy to each cut that will determine the location of the specific cut. A strategy may contain a reference location such as the ends of the sequence or subsequence, a specific place in the sequences or a random location. The reference can be used together with a probability distribution that determines the chances of any given location to be the place of the next cut. These probability distributions *de facto* regulate the length of each subsequence. Two probability distributions can be selected by the grammar: a discretised triangular distribution and a negative binomial distribution. An example can be seen in Fig.5.1-A and 5.1-B. The former has been chosen as it is the simplest linear non monotonic distributions and the latter because it is a function that is parametrised by just 3 values but can describe a wide range of natural phenomenons [158].

After the cutting phase the subsequences are given symbols with S always being the leftmost subsequence and E being the rightmost subsequence such as in fig. 5.1-C. The start and end sequences (S, E) are never altered by the evolved operator which only acts on the sequences labelled α-β in Fig. 5.1-C. Note that subsequences may be empty. This can happen if the leftmost cut is on the left of the first element (leaving S empty), if the rightmost cut is after the last element (leaving E empty) or if two different cuts are applied in the same place.

Permutation of the subsequence: After cutting the sequence the subsequences become the units of a new sequence. The grammar can specify if the subsequence will be reordered to a specific permutation (including the identity, i.e no change) or to a random permutation. An example can be seen in 5.2-a.

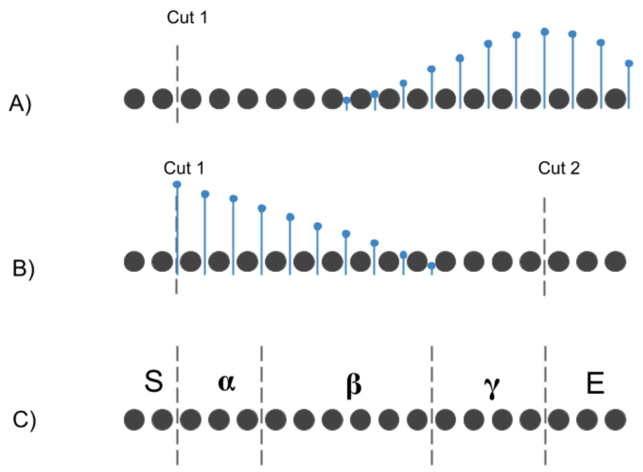
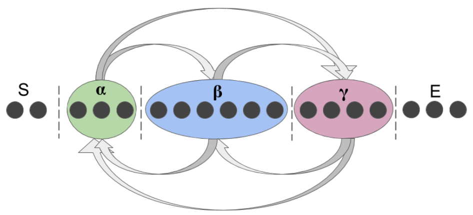
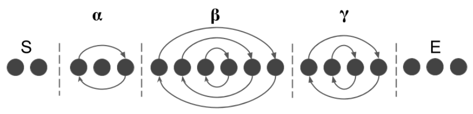


Figure 5.1: A) Example of a sequence with one cut and a probability mass function (blue lines) that will decide the loci of the second cut. B) Both cuts now shown with the probabilities for the loci of the third cut C) Final set of subsequences after k-cuts



(a) Subsequence permutation



(b) Subsequence inversion

Figure 5.2: Example perturbations of the subsequences produced by the grammar

Inversion of the subsequences: The grammar specifies whether the order of each specific subsequence should be reversed or if the reversing should be decided randomly for each subsequence and each iteration.

Iteration effect: Another component of the grammar is the iteration effect which may associate a specific function that regulates the change or shift in the initial cutting location at each iteration. We have specified four types of effects: *random*, which means that the starting location of the first cut will be random; *oscillate* that makes the starting position move in a wave like manner and returns to the initial loci after a number of iterations; *step* that simply moves one step to the right of the previous starting position and finally *none* which has no effect.

```
<op>                  →   addCut(<loci_ref>,<distance>)
                          <c>
                          <ExtraCuts>
                          Iteration_effect(<motion>,<loci_computation>)
                          permutation(<perm_behaviour>)
                          inversions(<inv_behaviour>)
<ExtraCuts>           →   ∅ | <c> | <c><c> | <c><c><c>
<c>                   →   addCut(<loci_ref>,<distance>)
                          isInverted(<invert>)
                          permutationFactor(<r>)
<motion>              →   'random' | 'oscillate' | 'steps' | 'none'
<loci_ref>            →   'none' | 'left' | 'right' | 'limit'
<distance>            →   'linear' | 'negative_binomial',(<r>,<p>)
<r>                   →   1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
<p>                   →   0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7
<loci_computation>    →   'once' | 'always'
<perm_behaviour>      →   'fixed' | 'random'
<inv_behaviour>       →   'fixed' | 'random'
<invert>              →   0 | 1
```

Figure 5.3: Grammar used to produce the local search operator

5.4 Benchmarks

In this section we reuse the benchmarks utilised in the previous chapter for both the TSP and Knapsack problem.

As part of the empirical evaluation of the evolved programs, many instances and several repetitions of each heuristic had to be computed. In order to maximise the utilisation of a 64-core instance of a Google Compute Engine a load balancing problem had to be solved. This is due to the fact that instances are divided into groups of different sizes that require substantially different

Algorithm 2 Generate Perturbative Heuristic

```
$T \leftarrow$ collection of training instances (t)
$h \leftarrow$ heuristic
$p \leftarrow$ population of heuristics
Init(p)
while stop != false do
  Select $p'$ from $p$
  Crossover $p'$
  Mutate $p'$
  for each $h$ in $p'$ do {Evaluate heuristics}
    $Tf \leftarrow$ list of fitnesses
    for *each* t in T do
      $c \leftarrow$ random solution chain
      while *iterations* $<$ MAXperturbations do
        $cMutant \leftarrow$ perturb $c$ with $h$
        if $cMutant.fitness > c.fitness$ then
          $c \leftarrow cMutant$
        end if
        $iterations++$
      end while
      $f \leftarrow$ fitness of $c$ [The formula is domain specific]
      *append f to Tf*
    end for
    $h's$ fitness $\leftarrow$ *Tf's average*
  end for
end while
*best* $\leftarrow$ *Highest value in p*
return *best*
```

times to be evaluated and CPUs are paid in blocks, in the sense that if one CPU is computing and the others are waiting we will still pay for the 64 CPUs. Also we want to have all the results as soon as possible because until the last instance is computed we cannot proceed with the successive analysis. Organising the schedule of processes in order to balance the load is a well known combinatorial optimisation problem, and therefore an ideal candidate to test the ability of the generator of heuristics to provide heuristics for a new domain. For this reason we add this problem as a specific use case to our collection of benchmarks.

To do this we calculate the span of the ideal configuration of the relaxed problem. We relax the constraints of the problem by assuming that the specific jobs can be arbitrarily divided and assigned to different CPUs and compute the lower bound as the sum of the jobs' times t divided by the number of CPUsC.

Lower bound :

$$\sum_{i=1}^{n} t_i/C \tag{5.1}$$

We use this bound to estimate the quality of the balancing found by the heuristics.

5.5 Experiments

5.5.1 Travelling Salesman and Knapsack Problems

Training Phase: In our experiments single-point local-search heuristics are generated using an off-line learning approach. The system is applied *separately* to each domain, but uses an *identical* grammar in both. At each iteration of the GE, each heuristic in the population is applied within a hill-climbing algorithm to each of the 5 training instances starting from a randomly initialised solution. The hill-climber runs for x iterations with an improvement only acceptance criteria. For TSP, $x = 1000$ and for MKP, $x = 2500$ (based on initial experimentation). The fitness at the end-point is averaged over the 5 instances and assigned to the heuristic (i.e. distance for TSP and profit for MKP). Experiments are repeated in each domain 10 times, with a new set of 5 problems generated for each run. The best performing heuristic from each run is retained, creating an ensemble of 10 heuristics as a result. All the parameters of the synthesisers are give in Table 5.1 while the GE parameters are in Table 5.1b.

Testing Phase: The generated ensemble is tested on benchmark instances from the literature. For TSP, we use 19 problems taken from the TSPlib. MKP heuristics are tested on a total of 54 problems from 6 benchmark datasets from the OR-library. Each of the 10 heuristics is applied 5 times to each problem for 10^5 iterations, starting from a randomly initialised solution, using an improvement only acceptance criteria (hill-climber). We record the average performance of each heuristic over 5 runs, as well as the best, and the worst.

For TSP, we compare the results with 50 runs per instance of a classic two opt algorithm, using the R package TSPLIB that implements the method described by Croes [40], chosen as a commonly used example of high-performing local-search heuristic. For MKP, the vast majority of published results use meta-heuristic approaches. We compare with two approaches from [33], the Chaotic Binary Particle Swarm Optimisation with Time Varying Acceleration Coefficient (CBPSO), and an improved version of this algorithm that includes a self-adaptive check and repair operator (SACRO CBPSO), the most recent and highest-performing methods in MKP optimisation. Both algorithms use problem specific knowledge: a penalty function in the former, and a utility ratio estimation function in the latter, with a binary representation for their solution. Both are allocated a considerably larger evaluation budget than our experiments. The heuristics evolved using our approach would not be expected to outperform these approaches. However, we wish to investigate whether the approach can produce solutions within reasonable range of known optima that would be acceptable to a practitioner requiring a quick solution.

5.5.2 Load Balancing

For the load balancing problem we use a different approach as the goal is to find an extremely good solution to one specific problem instance. In this case we use an on-line-learning approach in which we try to learn on the fly how to solve the problem and in the mean time produce a set of heuristics that can be recycled for future problems belonging to the same domain. Using pHHGE we produced 10 heuristics that are the best heuristics picked from 10 independent runs. We plug our problem directly into pHHGE, differently from previous experiments we do not produce synthetic instances beforehand, but we use the results of the trials against the problem directly as fitness function. The results are compared with a maximal hypothetical balancing calculated analytically in section 5.4.

Parameter	Value
Number of cities	100
Cities distribution type	Uniform
Cities distribution range	0-100
Number of objects	100
Number of constraints	10
Object constraints distribution	Uniform
Object constraints range	0-100
Object profit distribution	Normal
Object profit mean	Sum of constraints
Object profit deviation	50
Knapsack constraints dist.	Normal
Knapsack constraints mean	2500
Knapsack constraints deviation	300
Number of workers	4-256
Number of jobs	16-1024
Jobs length	10-100

(a) Problem synthesisers

Parameter	Value
Generations	80
Population	100
Mutation	int flip
Crossover Prob.	0.80
Crossover type	one point
Max initial tree	10
Max tree depth	17
Replacement	generational
Tournament size	2

(b) Grammatical Evolution

Table 5.1: Experimental Parameters. On the left the parameters used to synthesise the training instances and on the right the parameters used by grammatical evolution to generate heuristics

The 10 heuristics are then used for a new experiment in which the heuristics so generated are trialled against a set of 100 synthesised problems of the same domain. The instances are in the form of j jobs that take $t(j)$ times and k identical workers (which is the formulation of load balancing problems for processors with homogeneous cores). We create instances with k equal to 2 to the power of n with n from 2 to 11. Furthermore we ensure that in each instance we have at least 4 times as many jobs as there are workers and the biggest job is 10 times the smallest job.

5.6 Results and Analysis

Table 5.3 shows the best, worst and median performance of the evolved heuristics and the two-opt based algorithm for TSP. With the exception of a single case, the evolved heuristics perform better in term of best, worst and median results. For each instance, we apply a Wilcoxon Rank-sum test on the 50 pairs of samples, and provide a p-value in the rightmost column. Improvements are statistically significant at the 5% level in all cases. Results for MKP the results are reported in Table A.8, averaged over 10 heuristics in each case. Note that despite the simplistic nature of our approach, a hill-climber with an evolved mutation operator, our approach out-performs CBSPO in

22 out of 54 instances when considering average performance[1]. SACRO-BPSO (currently the best available meta-heuristic) performs better across the board, as expected.

In Table 5.2 we compare the Average Success Rate (ASR) across all instances group by dataset against the results presented by [33] on 2 versions of SACRO algorithms and an additional swarm based method. In [33], ASR is calculated as the number of times the global optima was found for each instance divided by the number of trials. For pHHGE, we define a trial as successful if at least one of the 10 heuristics found the optima in the trial, and repeat this 5 times. It can be seen that the results are comparable to those of specialised algorithms, and in fact outperform these methods on Weing and HP sets.

In Table A.5 we show the compounded results by average %-gap grouped by instance size and number of constraints. The %-gap is calculated as the difference between the estimated profit given by Linear Programming (LP) relaxation and actual profit found. This value is considered as it is the one used in the results of the approaches we compare ours to. A detailed description of the derivation method and analysis of its application to the multidimentional knapsack problem can be found here [125]. In Table 5.4 we show how our approaches compare against 12 other approaches. It is important to note that the table include both constructive and perturbative approaches. Here is a brief explanation of the type of algorithms that appear in the table. Algorithms 11-13 and 14 are single trial constructive heuristics, algorithm 10 uses a portfolio of mixed algorithms and algorithm 1 is its latest version, algorithms 2 and 4 use a population of adaptive perturbative heuristics, algorithm 3 selects from a collection of specialised heuristics, algorithm 5 and 6 both use a single human written perturbative heuristic, algorithm 9 uses a generator of constructive algorithms with specialised components (it is the technique most similar to our generator of constructive algorithms at number 12), our generator of perturbative heuristics performs somewhere in between a meta-heuristics (8) and a human written perturbative heuristic (6). This table ranks algorithms based on average performance across all the instances of the dataset. It can be seen that algorithms that are more nuanced and are built to tackle large sets of diverse instances fare better than single strategy algorithms and that specialised algorithms tend to fare better than more general approaches as expected. However until now the ranking only contained specialised representations and operators for those representations,

[1]We do not provide statistical significance information as the PSO results, which are reported directly from [33], use a population based approach and vastly different number of evaluations

with these experiments we introduced the idea of non specialised representations with generator of operators that can synthesise an ad-hoc heuristic for the new problem domain(at the cost of some computation and some optimality).

In Table A.7 we report the results of the experiments dedicated to our load balancing problem in which 270 jobs of different length have to be divided among 64 workers. It can be seen that in the use case scenario the majority of the heuristics can find the optima and when reusing the heuristics on new synthesised problems at least half the heuristics can find the optima in each instance.

Problem Set	Instances	ASR			
		IbAFSA	BPSO–TVAC	CBPSO–TVAC	**pHHGE**
Sento	2	1.000	0.9100	0.9100	0.90
Weing	8	0.7875	0.7825	0.7838	0.80
Weish	30	0.9844	0.9450	0.9520	0.907
Hp	2	0.9833	0.8000	0.8600	1.00
Pb	6	1.000	0.9617	0.9517	0.967
Pet	6	na	na	na	1.00

Table 5.2: Comparison with latest specialised meta-heuristics (PSO) from the literature: a fish-swarm algorithm IbAFSA and the two most recent SACRO algorithms, results taken directly from [33]

5.7 Conclusions

We have presented a method based on grammatical evolution for generating perturbative low-level heuristics for multiple problem domains that is cross-domain: the same grammar generates heuristics for a domain that can be represented as an ordering problem. The method was demonstrated on three specific domains, TSP (a natural ordering problem), MKP and load balancing. We have compared the synthesised heuristics with a specialised human-designed heuristic in the TSP domain where the synthesised heuristic outperformed the well-known 2-opt heuristic. In the MKP domain, we compared the generated heuristics against two of the latest specialised meta-heuristics. The heuristics outperform one of these methods, and are at least comparable to the best method. We also note that the ensemble of 10 generated heuristics demonstrates high success rates in finding known optima when each heuristic is applied several times. Furthermore we extended our experiments and evaluated the evolved heuristic using larger MKP problem instances and compared the results

	pHHGE			2-opt			
	Best	**Worst**	**Median**	**Best**	**Worst**	**Median**	**Ranksum p-value**
berlin52	7793	8825	8170	7741	9388	8310	0.0033
ch130	6418	7108	6722	6488	7444	6984	0.0030
d198	16256	17033	16651	16400	18213	17291	$\ll 0.001$
eil101	674	739	702	680	749	709	0.0073
eil51	435	484	456	442	494	473	$\ll 0.001$
eil76	563	616	593	583	628	611	$\ll 0.001$
kroA150	28109	31473	29344	29223	31994	30509	$\ll 0.001$
kroA200	31470	34528	32634	31828	35170	32893	0.0005
kroB150	27028	30283	28767	28114	30941	29134	$\ll 0.001$
kroB200	31315	35319	33029	31509	35077	33422	0.0455
kroC100	21418	24353	22885	22953	25503	23977	$\ll 0.001$
kroD100	21817	24405	23233	22772	26428	23430	$\ll 0.001$
kroE100	22660	25509	24178	23012	26695	24216	0.0021
lin105	14675	16965	15642	14966	17057	16191	$\ll 0.001$
pr107	45547	50313	47560	47597	51932	50002	0.0001
pr144	58847	68722	61534	59058	67272	64660	0.0002
pr152	75615	81458	78073	77307	81850	79964	$\ll 0.001$
pr226	81811	96484	86244	83566	101582	91512	0.0021
u159	44826	51353	47461	45297	51505	48124	0.1276

Table 5.3: Comparison between evolved heuristics and classic two-opt. For each instance we compute the Wilcoxon Rank-sum test using 50 pairs of samples

against 12 well established methodologies. In the load balancing domain we used a real problem encountered during the evaluation of the many instances of the MKP domain and used it as a use case. We compared the result obtained by our system with analytically derived lower bounds and showed that the heuristics could find the optima in the majority of cases. The 10 heuristics synthesised while optimising the load balancing problem were further analysed by testing them on synthesised instances of the load balancing problem. The results showed that the heuristics generated can be re-utilised on different problem instances of the same domain at the cost of some drop in performance.

The approach represents another step towards increasing the cross-domain nature of hyper-heuristics: current approaches tend to focus on the high-level hyper-heuristics as cross-domain, while relying on specialised low-level heuristics. Our approach extends existing work by also making methods for the automated generation of low-level heuristics cross-domain, without requiring specialist human-expertise. The proposed approach is applicable to a subset of domains that can be represented as ordering problems. While we believe this subset is large, it clearly does not include all domains. However, the same approach could be generalised to develop a portfolio of modifiable

Type	**Reference**	**%-gap**
MIP	Drake et al.(CPLEX 12.2)	0.52
MA	Chu and Beasley	0.54
Selection HH	Drake et al	0.7
MA	Ozcan and Basaran	0.92
Heuristic	Pirkul	1.37
Heuristic	Freville and Plateau	1.91
Generation HH	**perturbative HHGE**	**1.96**
Metaheuristic	Qian and Ding	2.28
Generation HH	Drake et al	3.04
MIP	Chu and Beasley (CPLEX 4.0)	3.14
Heuristic	Akc¸ay et al.	3.46
Generation HH	**constructive HHGE**	**6.77**
Heuristic	Volgenant and Zoon	6.98
Heuristic	Magazine and Oguz	7.69

Table 5.4: Ranking of the average performance over the whole Chu and Bae knapsack benchmark of algorithms taken from the literature and both our methods

grammars, each addressing a broad class of problems. Recall that in each case, HHGE was trained using a very small, uniformly generated set of instances, and in the case of MKP, applied to a non-typical representation, yet still provides acceptable results. We believe this fits with the original intention of hyper-heuristics, i.e. to provide quick and acceptable solutions to a range of problems with minimal effort. Although specialised representations and large sets of specialised training instances undoubtedly have their place in producing very high-quality results when required, these results demonstrate that a specialised representation is not *strictly* necessary and can be off-set by an appropriate move-operator at the cost of some computation and some optimality.

In this chapter we have confirmed again that we can use a representation designed for a specific domain and re-use it for a different domain as long as we search for a move operator (perturbator) appropriate to the new domain. This lead us to the question : Can we design a general, flexible representation that can be used to represent combinatorial optimisation problems from multiple domains AND the solvers that are used to optimise them?

In the following chapter we will propose a formalisation of a graph based representation that has the goal to describe and synthesise combinatorial optimisation problems and solvers structures without focusing on any one in particular. We equip this representation with a formal language that can be used to transform the proposed representation.

Chapter 6

A Conceptual Framework for the Synthesis of Graphs

The previous chapters considered domains in which the solutions were described as graphs. In particular, in Chapter 4 we defined a generator of algorithms that enabled the construction of a graph, starting from scratch, using a classical *constructive* approach that adds and connects one vertex at each iteration of the algorithm. In chapter 5 we defined another generator of algorithms that perturbed already fully constructed graphs by modifying the ordering of the vertices of the graph. The experimental results of the previous chapters showed that, given a scenario in which a single low-level encoding of the solution is imposed (i.e. a partial permutation), one can spend computational effort searching for new specific operators that specialise and improve their efficiency in order to be applicable to new domains. In the previous chapter, it was necessary to separately define and implement the constraints of the algorithms that evolved these operations in each case. From an engineering perspective, it would be more convenient if there was a way to describe these constraints at a generic level that would enable both types of operation (and new ones) to be composed from a set of basic building blocks, in order to make them applicable to as many domains as possible.

The goal of this chapter is to introduce a *higher level representation* as an attempt to provide a way to specify graphs that have sufficient flexibility to cover very different parts of our system, which include low level operations such as selection of heuristics, heuristic components and learning mechanisms, yet built on a specific formalism. These types of frameworks are often called *Meta-Models* [8].

The key idea is that a representation must be able to describe both heuristic solvers (i.e. a program generated with grammatical evolution), their management (i.e. a specific selection of heuristics out of a collection of heuristics) and specific problem domain solutions (i.e. a tour of a TSP problem or collection of objects that are the solution of a packing problem). Note that this is a different interpretation of the term "representation" used in evolutionary computation and hyper-heuristic literature where it typically refers to low level encodings of problem's solutions and no other part of the hyper-heuristic system (i.e. the generator and selector of heuristics).

The necessity of introducing this new representation arises as a consequence of the fact that the low-level solution representations currently available in the literature have different purposes from the one described above. In the majority of cases the representation specifically targets some underlying properties of the problem's domain and exploits these properties to make the problem easier to solve using some specific operator [133]. This approach becomes inconvenient (i.e. not manageable from a software engineering perspective) if one has to solve problems from a stream of problems where the domain of the problems changes over time. Unfortunately this is often the case in a real world application where business requirements change over time due to expansion or change in the law. Further problems rarely come one at time in perfect silos, instead they come in bulk of problems that are interconnected to some degree [2]. In this chapter we also try to address a current limitation that generative hyper heuristic systems have which is that they can normally only generate heuristics for one domain at a time. Commonly a different generator must be used for each problem domain that has a different representation and current systems a use specific representation for each domain [138] [150].

It is conjectured that in real scenarios any system left operating long enough will likely have to deal with a problem from a new domain other than that for which it was designed in order to continue operating. This makes generators of heuristics and representations applicable to multiple domains beneficial in autonomous systems as having a flexible system can save the effort of having to manually re-write or re-invent software or can give at least one option when no others are available.

Further, due to the methods developed so far in the literature being highly specialised in order to give best results within a domain (or a dataset), these methods tend to be very different from each other. This is partly due to the importance given to representations tailored around problem domains. This makes it difficult to build systems that grow in the number of domains that a single system can tackle as each approach is different/incompatible with the previous ones.

Finally, the goal is to have a tool kit able to describe objects that go from higher dimensional geometries (i.e. static connected structures like the one utilised in the MKP) to grammars that generate networks with various structural properties based on graph production rules.

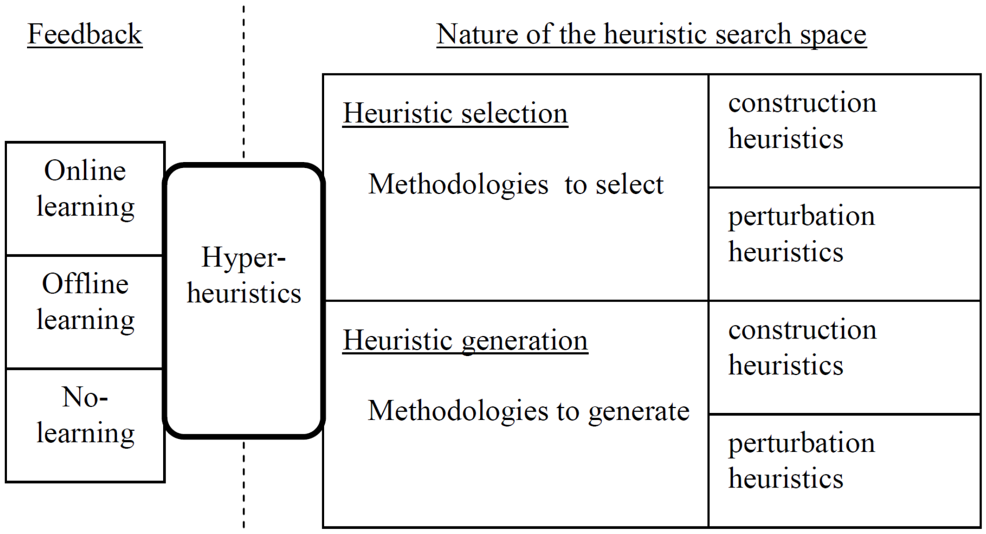


Figure 6.1: Classification of Hyper-Heuristics approaches

We briefly review again the essential methodologies in hyper-heuristic systems that can be seen in fig. 6.1. Hyper-heuristics may be be driven by creation or selection of heuristics. Heuristics may construct solutions from scratch or perturb already existing ones. Normally it is sufficient to implement one of these methodologies to be considered hyper-heuristics. Many hyper-heuristics focus only on generating new constructive heuristics [61] [29][121] or apply perturbative heuristics following the use a constructive heuristic [110] , although there are a few exceptions [146].

We define Autonomic Hyper-Heuristics (AHH) as a hyper-heuristic system that: specifies both low-level components and high-level components. It has a heuristics' management system that creates and selects heuristics. It is able to modify elements of the systems at different levels specifically heuristics, solutions and generators. It is able to modify its internal parameters. Feedback loops and memory can be attached to all its elements which can be used for both online and offline learning.

In order to create such a system we need to be able to answer at least these questions:

1. What is the set of components that could describe an Autonomic Hyper-Heuristic system?
2. To what extent can we develop a theoretical framework that can contribute to the HH field by facilitating the specification of AHH?
3. What are the essential requirements of a language that describes the above in a manner that is compact/efficient ?

We could tackle the above in a number of ways, e.g. from an algebraic perspective, set theoretic, logic or agent based. However, extending the knowledge gained in previous chapters, we have selected to use a graph-theoretic approach. Graph theory ideas have also been previously used in the HH literature. In particular, graph colouring and hyper-heuristics have been combined to solve timetabling problems [28]. Hyper-heuristics have also been used in combination with ant algorithms which require the construction of some graph that the ant must be able to navigate [32].

6.1 Definitions

The remainder of this chapter is going to use a graph based representation to tackle the above questions but before going into details it is useful to explain some terms:

Graph: a collection of objects and arrows or more formally a labelled multigraph.

Vertex: a generic object. In this manuscript we consider vertices, objects and nodes as equivalent.

Arrow: an ordered pair of vertices. In this manuscript we consider arrows, edges and arcs as equivalent.

Subgraph: is a graph formed by using a subset of the vertices and arrows of another given graph.

Production rule: is a formal graph rewriting rule composed of two graphs that play the parts of pre-condition and post-condition respectively. If a match of the pre-condition graph is found it is substituted by the post-condition graph. A rule is normally written in the form $\mathrm{G} \rightarrow \mathrm{H}$, where G is replaced by H.

Transformation: the execution of a production rule, this is equivalent to a graph rewriting operation.

Grammar: a collection of production rules which is equivalent to a collection of specific graph rewriting operations.

Colour: colouring is the graphical representation of types or labels. Vertex or edges of the same colour correspond to objects of the same type or having the same label.

Heuristic: a choice or decision that come in the form of one or more transformations with at least partially unknown outcomes and efficiency .

Family: a collection of objects or arrows sharing the same property .

Progenitor: an object with specific properties from which a family can be built, the symbol of a progenitor will be used to identify the family and their specific shared properties within the manuscript.

6.2 A Higher Level Graph Representation Language

The key idea behind the proposed representation is that we have one graph "entity" defined as a collection of objects and arrows with a set of properties that enable the language to describe the topological structures of the objects required to define a HH system, including problems and solvers. This allows any object in our system to be morphed into a different encoding which can be reverted back to what it was or any of the other mappings described. Nevertheless whenever possible we will encode objects into their most commonly used form, in the sense that each component introduced has a role that is often mentioned in the literature or is a "common element" within hyper heuristic systems. For example, genotypes used in grammatical evolution will be sequences, programs will be trees, collections of solutions will be multisets etc.

We approach this from a first principles approach and propose a set of initial symbols (progenitors) that describe container types (in term of fundamental topological features) but also are the descriptors of types of processes (types of actions). These types are then used to define the types of objects at the input and output of transformations.

We propose that four base types (Multiset, sequences, tree and cycles) are required in order to balance between the ease of use, management of the growth in complexity and in part due to pragmatic reasons. These types are in fact influenced in large part by structure ubiquitous in computer science.

All the transformations between these fundamental types form a family of 16 transformations (each category has 3 functors into other categories and one endomorphism that turns it back into the same categories for a total of 16 transformations. A diagram of the four types and all the transformations between these types can be seen in fig. 6.2. These transformations are a small set of a greater family of transformations. The 4 types of progenitor can be combined and give rise to 16 hybrid objects (Hasse diagram or powerset) which go from the null object to the object that has all four topological properties as shown in Fig. A.1. These 16 objects give rise to a total of 256 transformations that describe all the transformations that take one object from one category to another or themselves. In this chapter we will only describe and show concrete examples of the 16 hybrid objects.

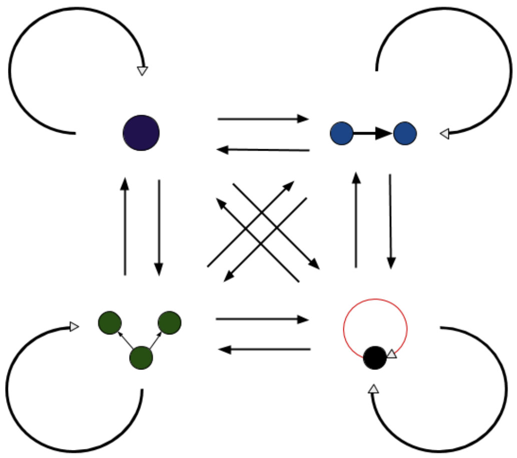


Figure 6.2: The four base categories. Each category has 3 functors into other categories and one endomorphism that turns it back into the same categories for a total of sixteen transformations (arrows)

6.2.1 Four Base Graph Families

Multiset or Disjoint objects(•)

The multiset, fig. 6.3, can be considered as an unordered collection of disjoint vertices. The smallest element of the family of the non-empty multiset is the single vertex. Multisets appear in a large variety of combinatorial applications [149].

The property of disjointness describes the possibility of having more than one vertex in the sub-graph with no relations between them. In our case it has both the role of declaring the existence of a collection and that the elements of a given collection are disjoint (at least at some level). This doesn't mean that the collection cannot have connections among its elements but that the specific sub-graph should be such that the objects' connection are not included in the sub-graph. This property is used to distinguish graphs that are connected from those graphs that are collections of disconnected graphs.

A graph with only this property is sometimes controversially called an 'empty graph' in the literature. It should be clarified that in this manuscript a graph is considered 'empty' only if it has no vertices, but it may still have labels or properties.

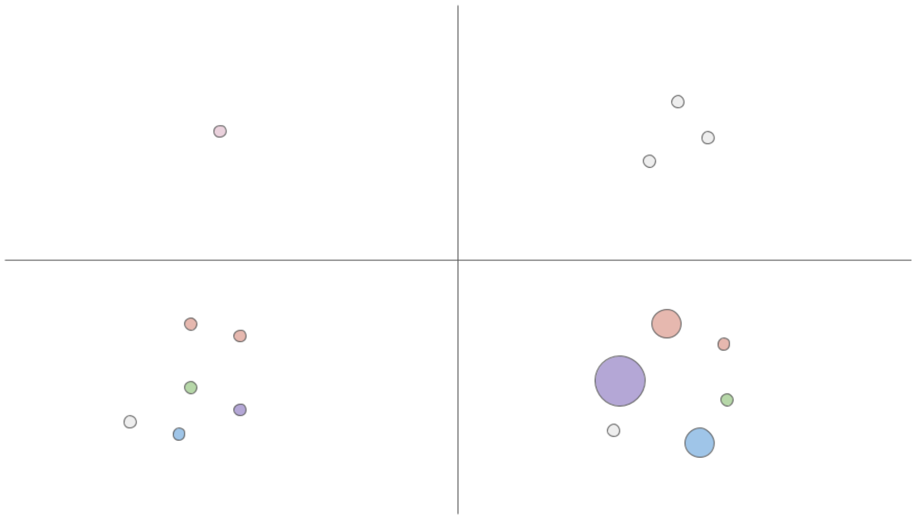


Figure 6.3: Multiset (•): Some arbitrary examples of graphs where the top-left graph is the smallest non-empty element (progenitor). The property is invariant regardless of quantity, colour and size of the vertices

Sequences (→) A sequence (fig. 6.4) is a collection of vertices in a given order. The smallest non-empty sequence contains two vertices and one arrow (that specifies the order). Every sequence has a defined beginning and end, which must be different vertices. The main interpretations of the minimal sequences is **A** *becomes* **B**, **A** *is transformed into* **B** and **A** *is followed by* **B**. An important characteristic of the sequence is that each element is followed by one and only one object, apart from the end vertex which has no following object. The sequence can be considered equivalent to a one dimensional ordered set. This is a common object used in computer science and mathematics as it is used in series, time-series, strings, lists, 1D vectors, serialised objects, numeric notation and linear forms. The DNA and chromosomes used in meta-heuristic algorithms are often implemented as sequences.

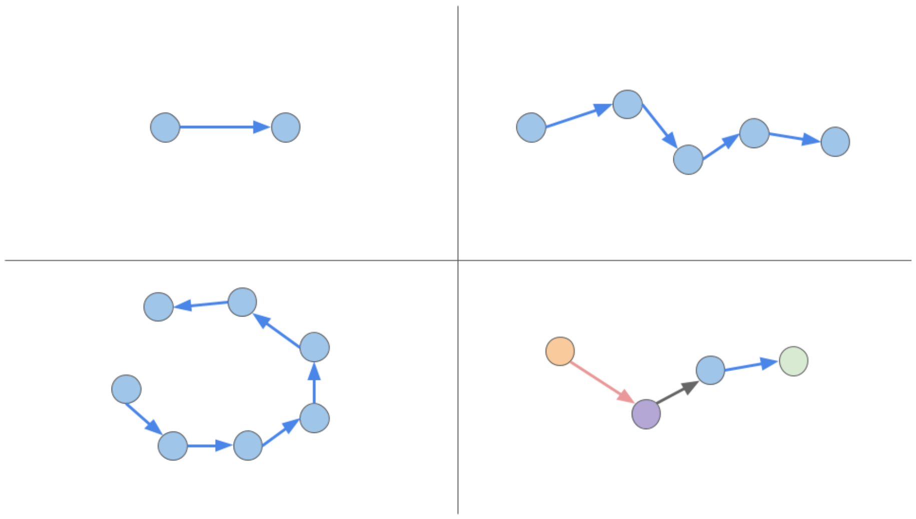


Figure 6.4: Sequences (→): Some arbitrary examples of sequence graphs where the top-left graph is the smallest non-empty element (progenitor). The topological property of the sequences is independent and invariant regardless of the number of vertices in the chain, their colour or position.

Cycles (⥁)

A cycle (fig. 6.5) is a collection of vertices in a given order where the beginning node and the end node are the same node. The minimal cycle is the self-loop composed by one vertex and one arrow from the vertex into itself. This structure is common in computer science and engineering but it is also used to describe combinatorial problems such as The Travelling Salesman Problem.

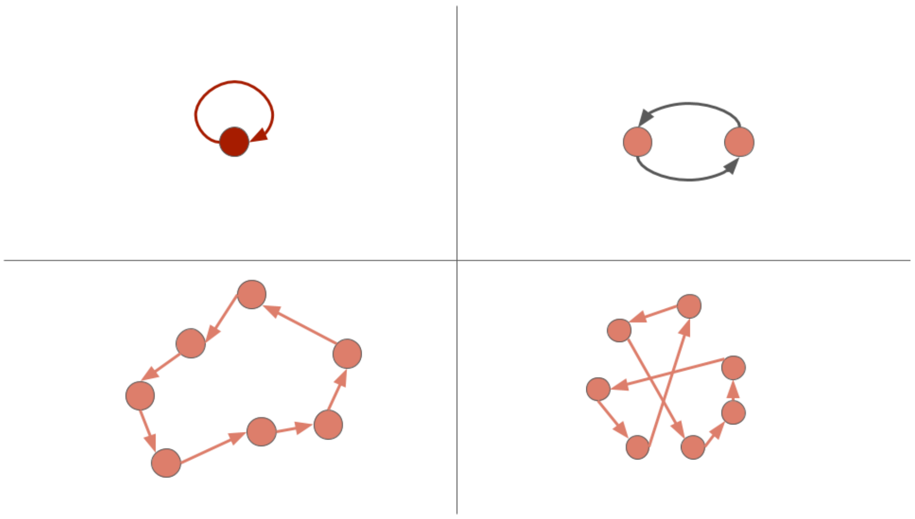


Figure 6.5: Cycles (⥁): four examples of cyclic graphs where the top-left graph, the self-loop, is the smallest non-empty element (progenitor)

Trees (⋎)

A tree (fig. 6.6) is a collection of connected vertices without cycles. The minimal non-empty distinguishable tree is the ramification composed of 3 vertices and 2 arrows. Trees are one of the most important data structure in computer science and are used in file systems, data storage, web-pages, genetic programming, grammatical evolution, syntax tree, hierarchical models and several other applications [86].

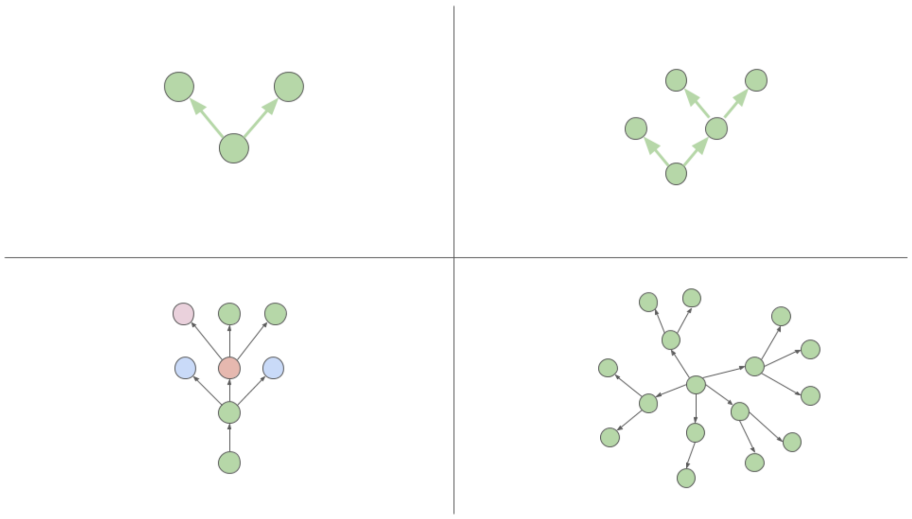


Figure 6.6: Trees ($\curlyvee$): four examples of trees where the top-left graph, the single ramification, is the smallest non-empty element (progenitor)

6.2.2 Power Set of the Base Families

Starting from the initial four properties it is possible to combine them and generate their **power set** and create more complex graphs which are listed below.

Disjoint Sequences ($\bullet, \rightarrow$)

Disjoint sequences (fig. 6.7) are obtained by combining the property of disjoint objects and the property of ordered sequences and the outcome is a collection of sequences with no relationship or adjacent vertices between the specific sequences. A population of chromosomes in a genetic algorithm, a collection of independent time-series, a dictionary of lists or strings are examples of this category.

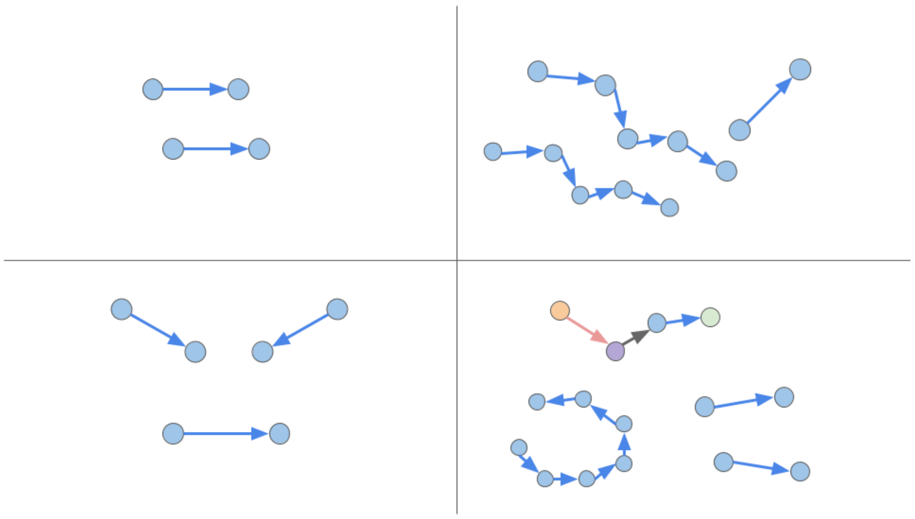


Figure 6.7: Disjoint sequences: four examples of disjoint sequences

Disjoint trees (•, ⋎)

A disjoint collection of trees (fig.6.8) is the result of the combination of disjoint objects and trees resulting in a collection of trees with no relationship or adjacent vertices across the specific trees. The definition is equivalent to the one of *forest*. A population of programs in genetic programming, file systems in a multi-disk computer, random forests, collections of independent websites and collection of decision trees are some examples of this category.

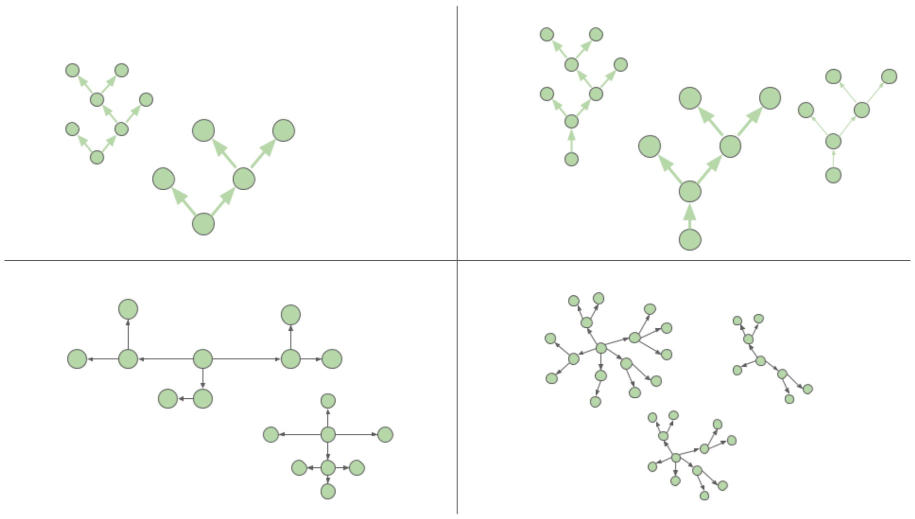


Figure 6.8: Disjoint trees: four examples of disjoint trees (forests)

Collection of Cycles (•, ↻)

A disjoint collection of cycles (fig. 6.9) is the result of the combination of disjoint objects and cycles interpreted as a collection of cycles with no relationship or adjacent vertices across the specific loops. The special case of multiple travelling salesmen that start from different depots and each return to their own depot is an example of this category.

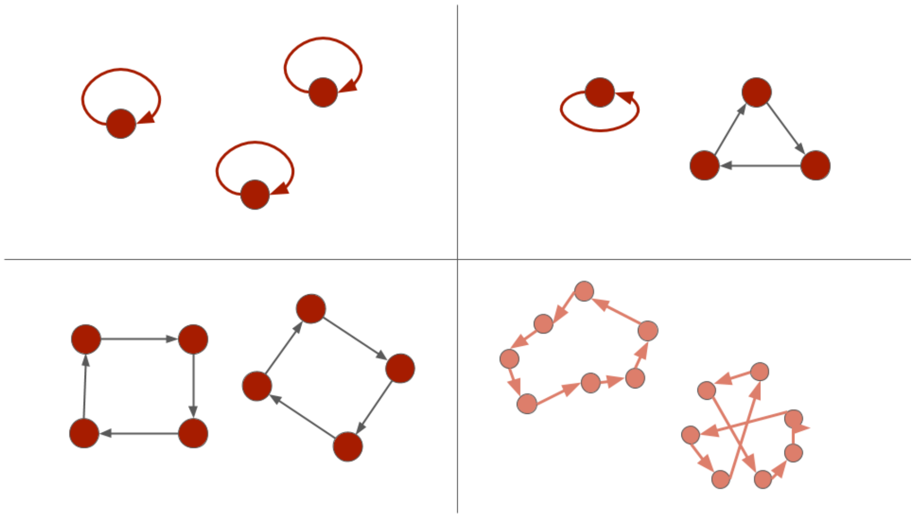


Figure 6.9: Disjoint trees: four examples of disjoint cycles

Multi tree (→, ⋎)

A multi-tree graph (fig. 6.10) in which the set of nodes reachable from any node form a tree. In this interpretation of the combination of sequences and trees we assume many roots and many intersecting leaves. Diamonds are allowed (branches that diverge at one level and then and re-converge at a lower level). The flow from the roots to the leaves is mono directional and has no cycles. Examples of this topology appear in P2P systems and databases [95].

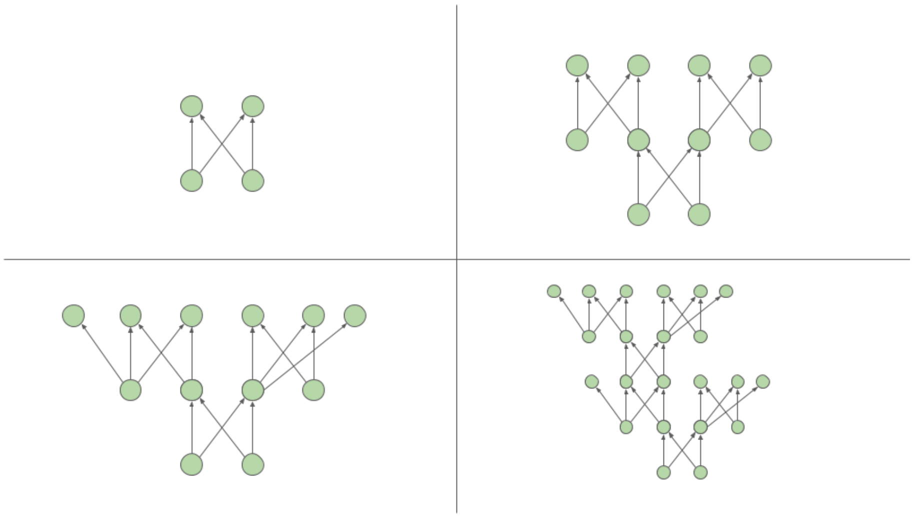


Figure 6.10: Four examples of multi-tree

Branching Cycles (⥁, ϒ)

A branching cycle or branching loop (fig. 6.11), which is the combination of trees and cycles, can be constructed using a tree where every leaf is connected back to the root. The circulatory system and data flow (with return statements after branching statements) in a software program are a few examples of this category.

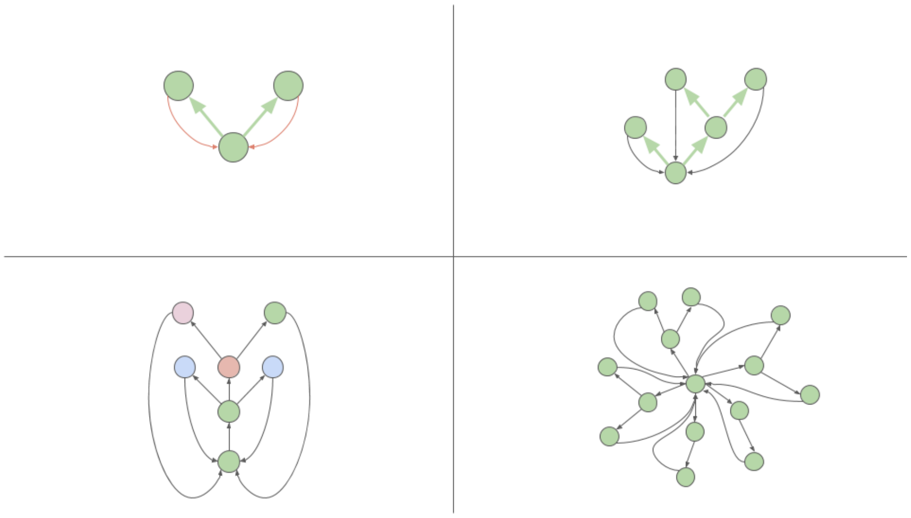


Figure 6.11: Four examples of branching cycles

Sequence of cycles $(\rightarrow, \circlearrowright)$

A sequence of cycles (fig. 6.12) is a joined collection of ordered loops all having the same beginning and the same end. It is the result of combining the property of sequences and the property of cycles. A vehicle routing problem where the vehicle has to return multiple times to the depot or multi-vehicle routing problems where the vehicles start and return to the same depot are examples of this structure.

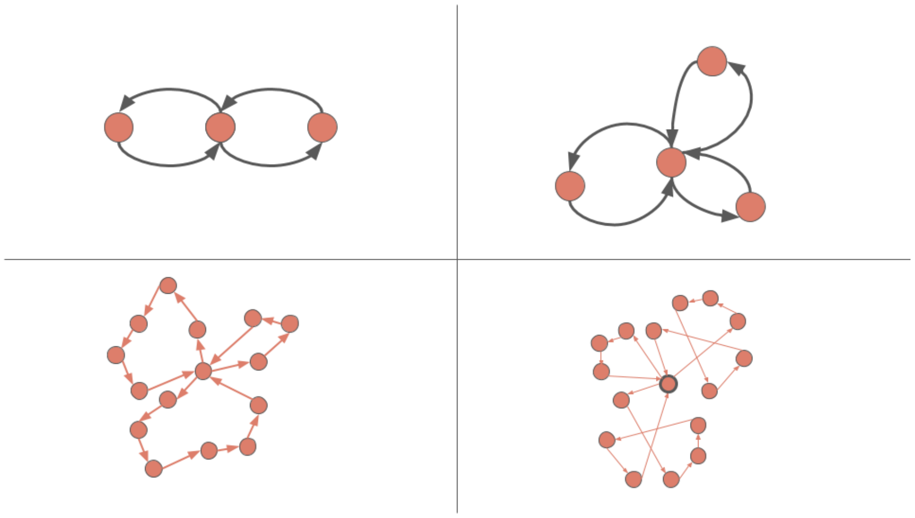


Figure 6.12: Four examples of sequence of cycles

Collection of multitree $(\bullet, \rightarrow, \curlyvee)$

A disjoint collection of multitree (forest of multitrees) or more formally a disjoint collection of directed acyclic graphs in which the set of nodes reachable from any node form a tree. It is the result of the combination of the properties of disjoint objects, sequences and trees.

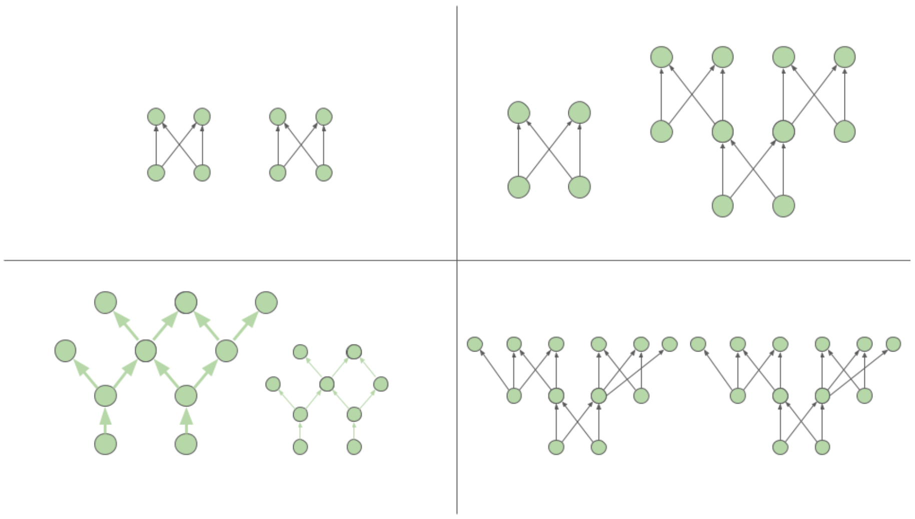


Figure 6.13: Four examples of collections of multitree

Cyclic Multitree $(\rightarrow, \curlyvee, \circlearrowright)$

A cyclic multitree or sequence of branching cycles is constructed using a multitree and connecting every leaf back to its own roots (fig. 6.14). It is the combination of the properties of sequences, trees and cycles.

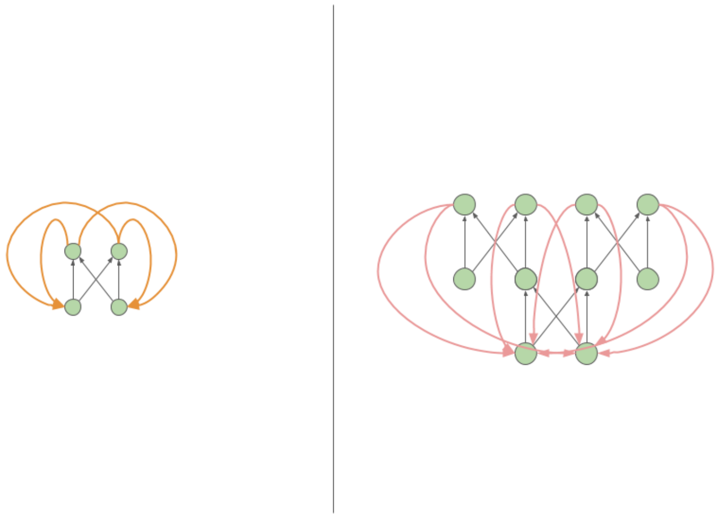


Figure 6.14: Two examples of cyclic multitree

Collection of Branching Cycles $(\bullet, \curlyvee, \circlearrowright)$

It is a collection of disjoint branching loops (fig. 6.15). It is the combination of the properties of disjoint objects, trees and cycles.

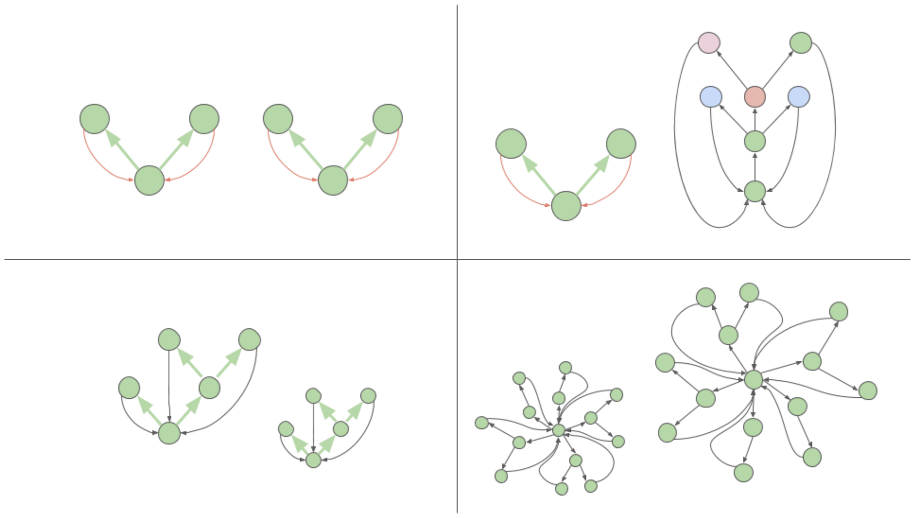


Figure 6.15: Four examples of collection of disjoint branching loops

Collection of Chained Loops $(\bullet, \rightarrow, \circlearrowleft)$

It is a collection of disjoint chained loops (fig. 6.16) . It can be used to describe special cases of multi-city, multi-depot vehicle routing problems.

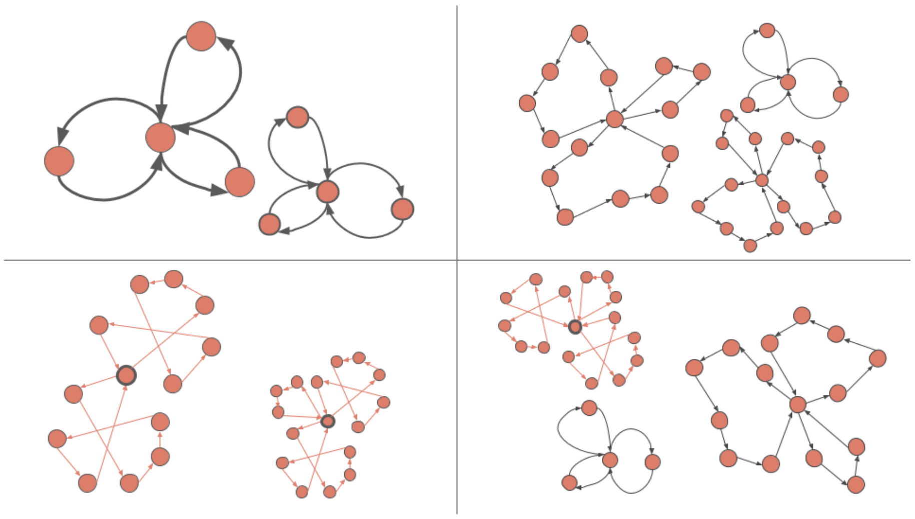


Figure 6.16: Four examples of collections of disjoint chained loops

Tetra Graph $(\bullet, \rightarrow, \circlearrowleft, \curlyvee)$

The tetra graph (fig. 6.17) or collection of cyclic multitrees is a collection of disjoint ordered branching loops. It is the combination of all four base properties.

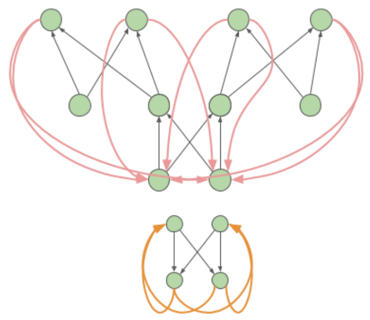


Figure 6.17: A single example of a collection of cyclic multitrees

The Null Graph

It is a graph with no labels, no properties, no vertices and no arrows. The null-graph is used as an empty container (not to be confused with an empty graph). It is important to note that using and transforming null graphs is not cost-less; however the cost is dependent on implementation details.

The four initial properties and their composition gave us a total of 16 families of graphs. These graphs cover a very large variety of possible configuration and special cases. However these graphs should be seen as building blocks that can be "glued" together in order to create vastly more complex graphs. In the following section we define some approaches where this could be achieved.

6.3 Operating on Graphs

In this section we present a meta-model and an associated transformation system for the representation of combinatorial optimisation problems and hyper-heuristic solvers. We limit ourselves to using unary (single transformation or production rule) and a few binary operations (leaving a door open for future work that generalises to n-ary operations)

The goal is to provide a language with which problems can be described by abstract models in the form of graphs, where the language provides tools to alter models and turn them into different problems. At the same time the language must provide a way to design solvers and give them a structure in which collection of solvers can be organised. Finally and most importantly, it must be able to alter them in order to produce new solvers. This is in contrast with the bulk of the effort in theoretical Evolutionary Computation that focuses on run-time analysis or tries to give guarantees relative to the performance of specific algorithms [55] [9]. It should be clarified that this is not the goal of this contribution, which instead attempts to provide a formalisation that is capable of describing and composing complex structures and is also computable.

Graph transformations provide a theoretical tool-set that allows us to create and edit graphs with the use of production rules. As discussed in 2.3 a language based on graph transformations equipped with composition and iteration allows it to be Turing complete. This means that potentially any program can be produced in this way, however extreme generality is not always ideal as it can cause specific applications to become longer or more verbose than desired. Therefore on top of the special families of graphs described in this chapter we define 3 binary graph operations in order to narrow the scope of the language and equip it with short cuts useful in the areas of hyper-heuristics and combinatorial optimisation problems.

All the operations take two graphs as an input and return a graph as an output.

Multi-set Disjoint Union is the classic union operator $\cup$ that combines two graphs without adding edges or vertices but only modifying subset-hood. This can be expressed as $H = G_1 \cup G_2$ in which $V_H = V_1 \cup V_2$ and $E_H = E_1 \cup E_2$ where H is the resulting graph, V_H its vertices and E_H its edges.

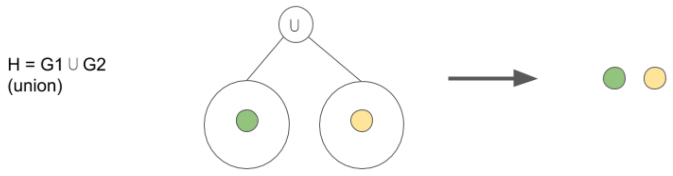


Figure 6.18: Example of the union operator, two graph with one single vertex each turn into a graph with two disjoint vertices

Sub-graph Multi Assignment: connects all the vertices from the first sub-graph to all the vertices of the second sub-graphs. This can be expressed as $H = G_1 + G_2$ with $V_H = V_1 \cup V_2$, $E_H = E_1 \cup E_2 \cup E_s$, and $E_s = \{e(v_i, v_j) \in V_H \times V_H | v_i \in V_1 \wedge v_j \in V_2\}$ where E_1 are the edges of the first graph, E_2 are the edges of the second graph and E_s is the set of edges built from the vertices of the first graph to those of the second graph.

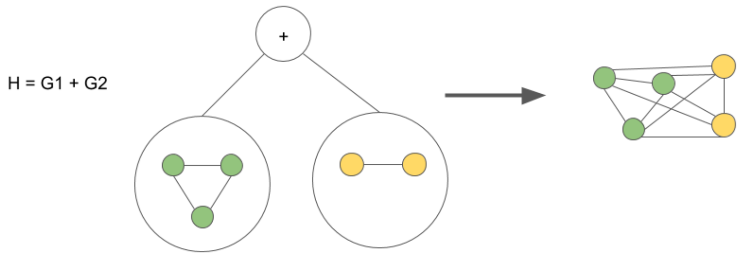


Figure 6.19: Example of the Multi assignment operator, all the vertices of the first graph are now connected to the elements of the second graph

Nesting: each vertex of the first graph will contain an instance of the second graph. This can be expressed as $H = G_1 * G_2$ with $V_H = V_1'$ and $E_H = E_2$ where $V_1' = \{v_i \in V_1 | v_i \cup G2\}$. This will create as many copies of G_2 as there are vertices in G_1.

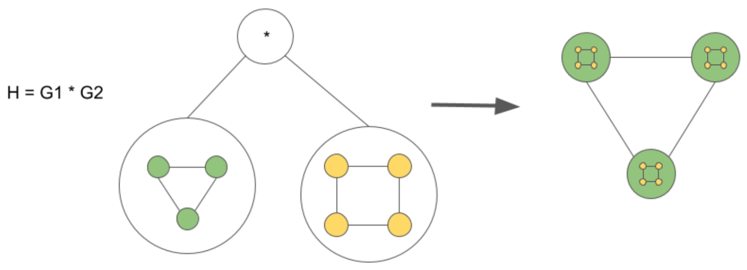


Figure 6.20: Example of the nesting operator, all the vertices of the first graph contain one instance of the second graph

6.4 Components

In this section some common components used in hyper-heuristic systems are modelled using the notions introduced in this chapter.

Genotype:

This is a common entity used in evolutionary algorithms. With graphs both Biological DNA and artificial DNA can be represented. In the case of biological DNA the four letter chains (i.e. $A \rightarrow G \rightarrow T \rightarrow H$) can be thought as a concatenation of objects but also these objects can be described at a more refined level as complex structure of atoms using a graph [36]. In artificial genotypes used in computer science the sequence often refers to a sequence of symbols where the most common underlying type are numbers (binary, integers or real). In our experiments the genotype was always encoded as a concatenation of Integers.

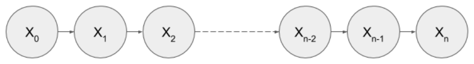


Figure 6.21: An example of DNA sequence where each label X_i can be substituted by a number or a symbol

Genotype to phenotype mapping:

These are the procedures with which the genotype is transformed into phenotype. In O'Neill's Grammatical Evolution [111] each element of the genotype sequence triggers a *production rule*. Using our proposed language this can be described by:

$$(\rightarrow) \rightarrow \curlyvee$$

This would impose that the input is a sequence and the output a tree.

Classic Heuristic Program:

A decision making criteria, encoded into a software program, with no assurances about the optimality in space, time or quality of the solution. In the case of hyper-heuristics the heuristic programs are generally involved in making decisions to solve combinatorial optimisation problems. Heuristics often trade optimality in exchange of time, this allows a solver to output a solution, at least good enough, in a feasible time. In many cases (such as generative HH that make use of GP and GE used in this manuscript), the heuristic program is encoded in a tree which represent what is often called "phenotype". This can then take many forms and names depending on the context. i.e. decision forests[126], regression trees and synthesised programs [91].

Measurements:

These play a central role in all control systems, optimisation and automated learning. In hyper-heuristic systems several key measures are normally implemented such as: evaluation of the fitness function, magnitude of the optimisation cost either in time and or space (e.g. computational memory), or diversity among solutions or solvers. Graphically we can represent them as the association of each instance (object or event) to the length of a single arc (belonging to the category of sequences): $(\bullet) \to (\to)$

It has been shown that in some cases it is useful to change the type of metric used [82], with known or surrogate ones, however the concept of associating an object with a "measured" value still holds and the diagram can represent the whole class of possible associations.

Feedback loops:

In conjunction with the measure there is the *feedback loop.* It is used to adjust parameters in control systems, for example, they implement the effect of selection in evolutionary optimisation or adjust mutation parameters, and affect the learning rate and rewards in machine-learning algorithms.

It is simple to define the diagram of a system with feedback loops using the proposed Graph properties as cycles are one of the base family (i.e. fig 6.22).

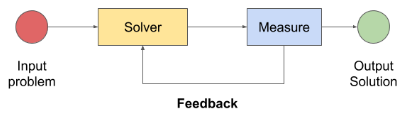


Figure 6.22: A simple example of feedback loop diagram typical of learning and evolutionary algorithms

Selection of heuristic:

In the beginning the notion of hyper-heuristic was intended as 'heuristics to choose heuristics'. It focused on the problem of selecting heuristics from an already available collection of heuristics. The term hyper-heuristics now encompass a much larger variety of tasks, such as generation and combination of heuristics [24], but *heuristics selection* is still a fundamental problem that needs to be managed in some way [96] [160] [47]. The greater the number of heuristics, the number of problem domains the system has to deal with, and the more diverse the heuristic available, the greater the complexity of the problem of selecting a heuristic.

In fig. 6.23 we propose the simplest graph interpretation that models this choice. This can be seen as an assignment problem in which one arrow associates the problem to the chosen heuristic.

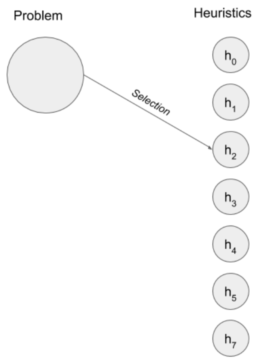


Figure 6.23: Selector: heuristic selection modelled as the assignment of an arrow between the problem and one of the heuristic available

The finite discrete selection:

It is common among many hyper-heuristic systems to choose an ensemble of heuristics out of a larger set of heuristics, and combine them to obtain more consistent results [76].

In this case we can model the selection as a tree (or more specifically **a star**) as in fig. 6.24

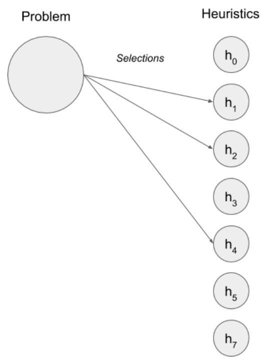


Figure 6.24: Multi-Heuristic Selector: heuristics selection modelled as a tree with the problem as root and heuristics as leaves

The multi-problem multi-heuristic selection:

The previous heuristic selection models can be generalised further in order to model the scenario of primary focus within the manuscript: having to deal with a variety of problem domains and the possibility of creating a variety of heuristics.

In diagram C we show the case where many problems (possibly of radically different domains) have many heuristics assigned to them. This model can be described by a simple multi-tree of depth one as in fig. 6.25

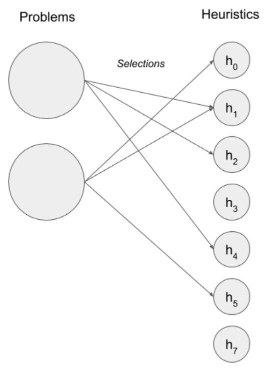


Figure 6.25: Multi-Problem/Heuristic Selector: heuristics selection modelled as a Multi-tree with the problems as roots and heuristics as leaves

The constructive heuristic:

In chapter 4 we have created a generator of constructive heuristics. Here we provide a diagrammatic model of the behaviour of a constructive heuristic (fig. 6.26). From the diagram it is easy to see the relationship between our approach and the use of sequences, trees and cycles. At each step of the sequence a tree of options is created, when one of these options is chosen a new edge is added. The edges will eventually form a cycle at the end of the sequence. As the algorithm only moves forward it highlights how functions such as backtracking are missing from our proposed approach which would likely improve the quality of the solver.

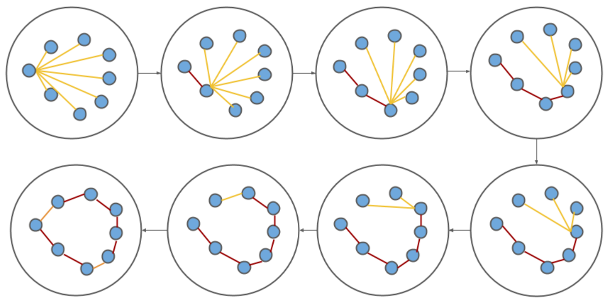


Figure 6.26: Example of constructive heuristic where a sequential program assign a tree of candidate edges and then select one edge to be added to the solution until completion

The perturbative heuristic:

In chapter 5 we have created a generator of perturbative heuristics. Here (fig. 6.27) we provide a diagrammatic model of the behaviour of perturbative heuristic. At each step of the sequence a perturbation is applied which changes the connectivity of the graph while leaving invariant the cyclic property of the graph.

Perturbative heuristics tend to be very different depending on the underlying representation (i.e. sequences, trees etc). However they all revolve around the idea of Iterated Local Search [74] [127], where the new solution is in the "neighbourhood" of the previous solution. What usually remains unvaried is the topology of the solution i.e. sequences will be sequences after perturbation, trees will stay trees and so on.

The meta-model can specify these requirements with very few symbols. i.e. a perturbation is a transformation from cycle to cycle $(\circlearrowright) \rightarrow (\circlearrowright)$

Finally it shall be noted from a category theory point of view that all perturbations are endomorphisms. This means that perturbing a sequence will generate a sequence, perturbing trees generate trees and perturbing cycles generate cycles. This vastly restricts the space of possible outputs and ensures constraint satisfaction.

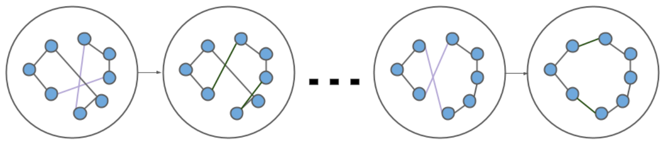


Figure 6.27: Example of perturbative heuristic where a sequential program re-orders the vertices of a cycle

Adaptive grammar modelling capability:

The rule of substitution can be used to substitute the initial properties and morphism with new ones. This means that a given grammar can transform some of its rules into new different starting and ending graphs and properties.

It is important to highlight that all the ideas introduced so far dedicated to the modification of graphs are also applicable in the description and transformation of grammars if the given grammar is represented as a graph.

6.5 Conclusions

In this chapter we have developed an abstract framework built using ideas and tools borrowed from graph theory and graph transformations.

The goal was to assemble a small compendium of concepts that can be useful when solving problems that requires the specification and transformation of some structure. These concepts form a language that can be used to describe a wide variety of combinatorial optimisation problems, their representations, and a method for solving them. It is small as it uses very few initial symbols making them easy to remember and implement however its expressive power allows it to combine properties that can quickly lead to highly complex objects. The intention is to use these concepts in a variety of situations and recycle them when new problem domains appear.

The hope is that the tools can provide at least some common/generic way to describe a structure and the conditions that defines the ways in which it can be transformed. In this way when a new problem domain has to be described one can express it with the same building blocks that were used in previous problems. This is convenient when facing a stream of different problem domains as new encodings and representations do not have to be developed for each new domain.

These tools do not focus on specifications that assure optimality, completeness or facilitate the search by a given operator, instead they try to describe the structural characteristic of a system with some rigour but very few symbols. This formalism become useful when focusing on the specification of problems' structural requirements as opposed to implementation choices at the lower level such as the low level encodings (i.e. using boolean and boolean operators vs using floats and floating point operations).

In particular we asked the questions: Q1: What are the set of components that could describe an Autonomic Hyper-Heuristic system?

We have shown that the notions we have proposed can describe the components commonly used in hyper-heuristic systems and defined the problems that the components are meant to solve. The language is extremely compact as it uses very few symbols and it is able to describe the components high level requirements but also describe them diagrammatically at a more fine grained level.

Then we have asked Q2: to what extent can we develop a theoretical framework that can contribute to the HH field by facilitating the specification of AHH?

We have shown that the notions we have borrowed from graph transformations can be utilised to model both elements of the solver's internal mechanics but also the problems that hyper-heuristics system usually tackles. It is useful as the same language and tools can be used to modify and create very different part of a given system instead of having to use different area of mathematics that require a multitude of different symbols, semantics and incompatible operators. This is also fundamental for the development of autonomic hyper-heuristic systems that can adjust their own parameters and high level heuristics.

Finally: Q3: What are the essential requirements of a language that describes the above in a manner that is compact/efficient ?

We have proposed 4 fundamental properties that can be used to describe a very large variety of graph structures usable to define both solvers and problems. We have defined 3 binary operators on top of classical graph transformation operators. We have shown that these properties and operators are sufficient in the description of Selective hyper-heuristics and Generative hyper-heuristics that uses Grammatical Evolution (an approach we have used to build in chapter 4 and 5) and simple constructive and perturbative heuristics.

While we have limited ourselves to describe established mechanics in hyper-heuristics and the tools we used in the previous chapters, in the following chapter we will use the notions proposed here to implement a number of problem scenario generators with applications in combinatorial optimisation.

Chapter 7

Graph Based Problem Scenarios Synthesis

Engineering consideration related to building multi-domain systems.

Due to the focus in the literature on domain specific problem solvers and problem specific representations, building heuristic solvers that can tackle a variety of different domains poses a number of challenges. Firstly, each new domain tends to have its own sets of benchmarks upon which various different representations, intended as low level encoding, have been developed that are not compatible with each other. This makes creating systems that grow in the *number of domains* burdensome especially if each new algorithm requires a comparison over an existing benchmark. Domain specific benchmarks are not built to fit within complex multi-domain systems, but simply as a specialised set of problem instances that are used only within a given mono-domain context. At a practical level this means that each new domain tackled by the system encounters an overhead cost caused by the necessity of implementing ad-hoc parsers plus the translation required to go from the benchmark specific encoding to the solvers specific representation, again intended as low level encoding. While there are some exceptions to this [21, 60], it is not the norm in evolutionary computation and hyper-heuristic literature.

A practical example of these issues appears in the TSPlib collection which contains instances of the travelling salesman problem [130] where some instances are given as list of pairs of coordinates while other instances are given as square adjacency matrices. So even within the same domain and same library there isn't a standardised approach.

In the case of the knapsack problems some benchmarks use the second line of the file for the profits of the objects [34] while others use the penultimate [47]. Again some benchmarks use the row of a matrix for the constraints of the objects [120] while others the columns of the same matrix [47].

This makes testing algorithms over multiple domains cumbersome as each domain, and sometimes instance, requires a unique ad-hoc parser. Finally it is the belief of the author that advanced optimisation systems should be able to spawn their own *synthetic scenarios* where they can practice and guess if a given problem is solvable and to what degree. This approach is already used and has proved extremely successful in Generative Adversarial Networks in which one network synthesises instances and another classifies them leading to both networks improving over time [62]. These are the reasons why in this section we developed a multi-problem benchmark generator that uses a standardised encoding that allows the creation and parsing of problems from a variety of domains using the specified language. This fulfils a necessity to have access to multi problem domain benchmarks that can facilitate assessing the performance of multi-domain-solvers. Further this chapter completes the previous one (Chapter 6) dedicated to the description and management of solvers with the description and implementation of some problem domains.

The representation developed in the previous chapter is more complex than other more direct and minimalistic encodings (such as a list of booleans for the knapsacks problem). On the other hand now we have a system that can describe the problem, solver and solution graph using the same building blocks. It can also describe a larger variety of domains. In this chapter we use this representation to develop a generator of instances for multiple domains.

The contributions of the this chapter are:

1. 3 Graph based models of optimisation problems of public utility
2. An implementation of a system for the embedding of synthetic problems into real world networks

3. An implementation of a system for the extraction of real world road networks and their visualisation
4. A novel generator of semi-synthetic problem instances
5. Baseline results on a sample of instances optimised using well established methods

7.1 Domains considered

We use three use cases to describe a single generic method for creating generators of problem instances for multiple domains. We demonstrate how to implement generators of problem scenarios/instances. These specific cases are chosen as they have different structures and different goals. Each problem can be considered as a sub-problem of some larger collection of problems. Allocating the position of ambulances has a number of related problems to solve (i.e. paramedics shifts, time of the day, vehicle maintenance schedule and more). However the goal of the present chapter is not to solve or model any specific problem domain but to create a system of scenarios where algorithms can be tested on multi-domain problems. The long term goal is to facilitate the implementation of 'streams of domains' as opposed to streams of problems of one specific domain. Clearly three is hardly a "stream" yet it can be viewed as a step in that direction. The current goal is to develop a generic method that can be used to generate instances for multiple domains using exactly the same language.

7.1.1 Ambulance placement problem

Ambulances are vehicles utilised across the world to carry people that need medical attention from some location to an healthcare facility. Ambulances may be parked at hospitals waiting for an emergency and intervene. In many cases it can be convenient to place the ambulance somewhere close to where it is likely to be required such as a large gathering of people (i.e. festivals or football matches). When the density distribution of the population is not trivial as in festivals it is not obvious where the most logistically convenient place to wait for an emergency is, so that the intervention time is minimised. Here we propose a model that contains a collection of vehicles A that can move across the road network, a collection of hospitals H that are fixed points in the network and a collection of emergencies Em that are vertices of the network where the vehicle A must travel and then go to the nearest hospital.

More formally: Given a network, with nodes associated to a probability P of Em, and a subset of nodes as H that are fixed, the problem is to place a collection of vehicles A on the network such that the average intervention time t_x calculated as the average travelling time of the shortest paths between $A \longrightarrow E \longrightarrow H$ is minimised.

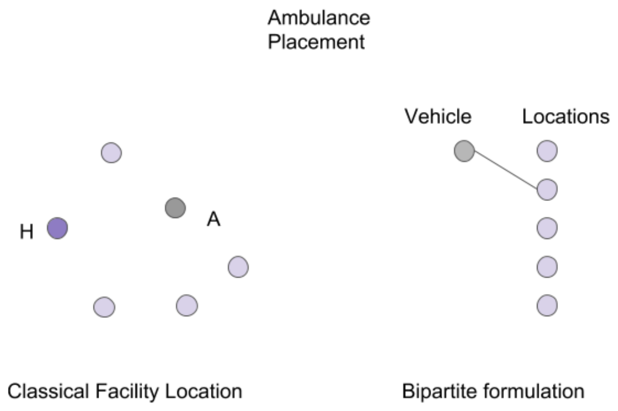


Figure 7.1: Example of an instance of an Ambulance Placement Problem, on the in its more common form and on the right formulated as a bipartite graph. On the Left location are represented by vertices, an ambulance placed on a vertex occupies the vertex with vertices placed in space according to their real position. On the right a bipartite graph interpretation where the vehicle vertices are assigned to locations via an arrow

7.1.2 Fire Brigade intervention problem

Among the various problems related to firefighting there is a resources allocation problem associated with each emergency. When one or more fire related emergencies arises the resources available must be deployed in such a way that all the fires are extinguished with minimal damage but also minimal costs. In this model we assume the existence of a number of fire stations deployed across a road network. Each fire station can deploy fire brigades that will move using the shortest path toward an assigned fire. Depending on the amount of forces deployed and the intensity of the fire, the chances of extinguishing the fire are computed.

More formally: Given a network, with nodes embedded on a square lattice where each cell of the grid is associated with a boolean state F, a collection of nodes FD that contains a set of resources FDR, where a FDR can be assigned to a node in order to turn the state F to false with some probability PFE, the problem is to assign the FDR resources to the nodes with state F such that all the state of the grid are equal to false while minimising the use of the resources (over all FD)

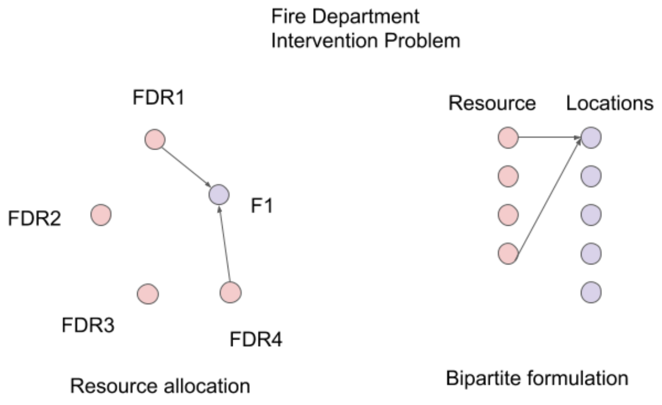


Figure 7.2: Example of an instance of an Fire Brigade intervention problem, on the left in its natural form and on the right formulated as a bipartite graph

7.1.3 School bus problem

The school bus problem is a combinatorial problem similar to the travelling salesman problem. Instead of going from city to city to sell items this problem is about picking up students that are all delivered at once at the last stop (the school). Further, the pick up stops are not given a priory but must be decided given a collection of houses where the students live. In this model we assume that the school bus is parked, starts and stops at the school.

Given a network, find the Hamiltonian cycle subset of the network G that includes all the points from a set of “stop vertices” Where stop vertices are single points each taken from a subset of vertices of G such that the traversing time of the cycle is minimised and the distance between a stop vertex and its associated houses vertices is minimised.

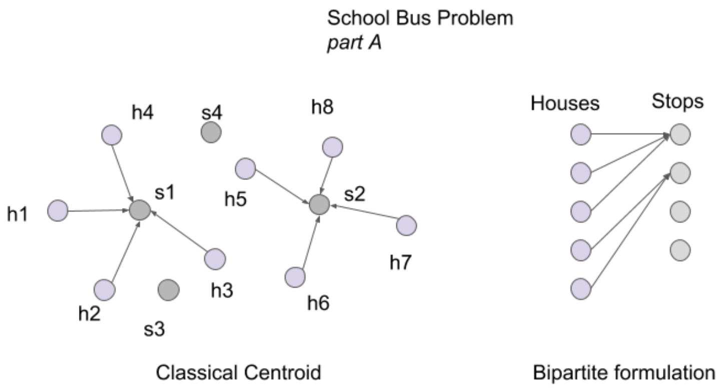


Figure 7.3: Example of a component of the School bus problem where stop vertices have to be assigned in order to minimise the travel time from each house, on the left in its natural form and on the right formulated as a bipartite graph

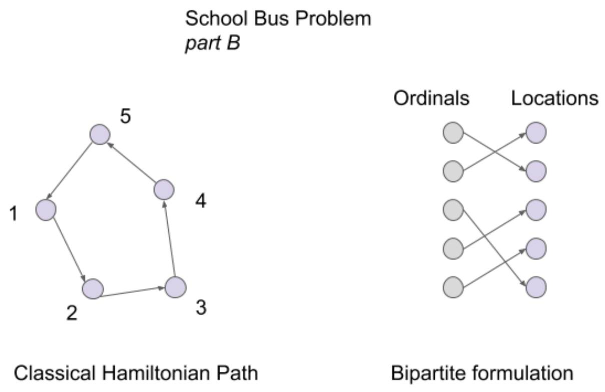


Figure 7.4: Example of a component of the School bus problem where a tour connects all the stop vertices, on the left in its natural form and on the right formulated as a bipartite graph

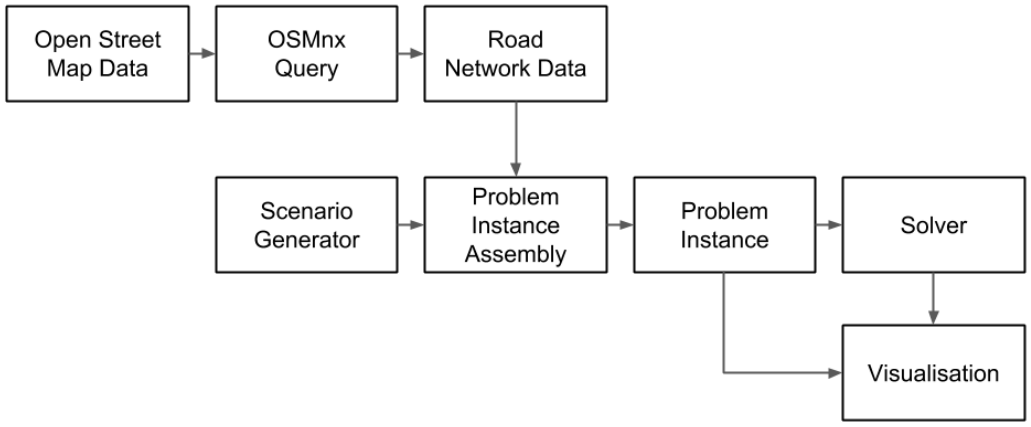


Figure 7.5: Pipeline of the generator that goes from the raw map data (top left) to the final visualisation of the solution (bottom right)

7.2 Implementation of the generator and data pipeline

In this section we describe the implementation of the models discussed in the previous sections that are part of the pipeline that extract data from real world road networks such as the location of buildings, streets and junctions. A diagram of the pipeline can be seen in figure 7.5. The real world networks extracted from OpenStreetMap via OSMNnx queries are embedded with “problem scenarios” and the combinations of the two are used to synthesise whole benchmarks. The problem scenarios can be parameterised and it is possible to input the number of hospitals, ambulances or other variables that are present in the specific scenario generator. The system can produce extremely large instances (i.e. the whole road network of the City of London or Scotland can be extracted in one query) much larger than what can be tested within the scope of this manuscript.

7.2.1 OSMnx

OSMnx [1] is a library created by G. Boeing [17] that uses the Open Street Map API [73] and the NetworkX library [72] that allows the extraction of real world information and makes them available

[1]https://github.com/gboeing/osmnx

as python data structures. The OSMnx library allows one to query OSM about streets, buildings, terrains and other geographical information. Subset of the map can be selected by specifying the name of a city or the coordinates and area of the selection. The generator developed in this instance makes use of road network information, but much more information is available such as walking trails, shapes of buildings, height maps, rivers, parks, forests, railway networks that could be added to problem scenarios in the future. We use these OSMnx queries to extract the road network information and then convert them into NetworkX graph structures.

7.2.2 Scenario Generator

Once the roads information are extracted and turned into a NetworkX graph structure a subset of the vertices are selected (according to a rule specific to the domain in question) and their IDs added to new lists.

- In the case of the ambulance problem n vertices are chosen to become hospitals and k vertices are chosen to become ambulances.
- In the case of the school bus problem n vertices become houses requiring pick ups.
- In the case of the fire brigade intervention problem n vertices become fire department resources and k vertices become fires.

From a data structure point of view the scenarios are described by two lists of vertices and one tensor. The dimensions of the tensor can be set to any an arbitrary combination of values, but in all three use cases it is just a 2D square distance matrix.

The initial graph G extracted from OSM is then embedded with the labels indicated in the two lists and the single tensor becomes G'. A simple example can be seen in 7.6. The graph G' is the instance of the problem. To become a fully fledged problem scenario the instances must be associated to a "Goal Function". The goal function or objective may be used directly as fitness function but doesn't have to be. One example can be seen in 7.7. The type of graph and goal function pair determine the domain of the problem.

- In the case of the ambulance placement problem the goal is to minimise the average intervention time.

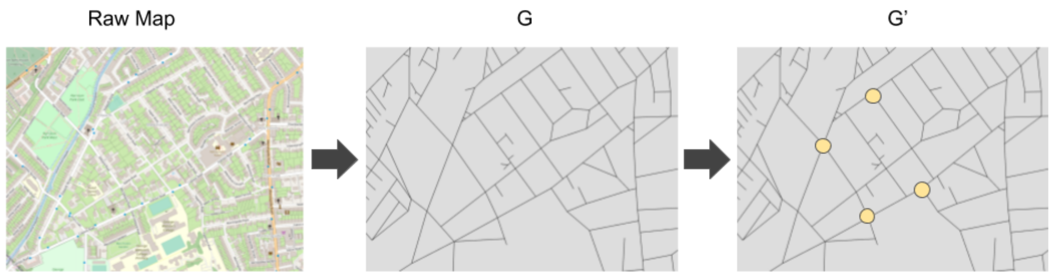


Figure 7.6: The Raw Map contained in Open Street Map is queried and a graph G is extracted. Some vertices of G are injected with some properties transforming it into G'

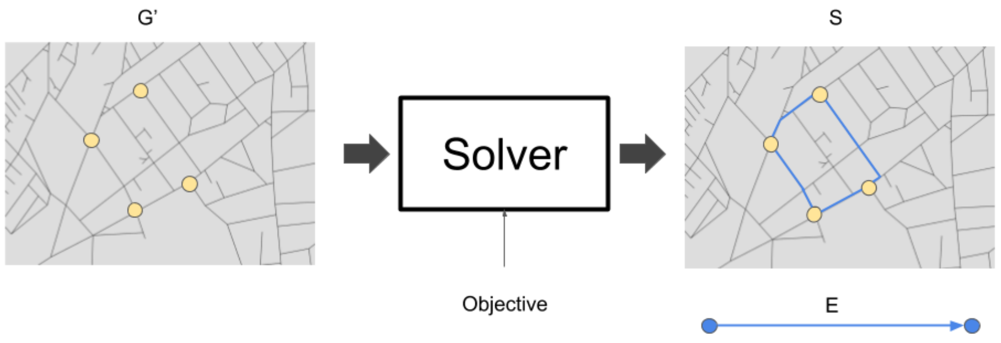


Figure 7.7: The instance G' and the objective function are used as inputs to the solver. The solver than outputs a solution graph S and an evaluation value E. In this case equal to the sum of the length edges.

- In the case of the school bus problem the goal is to minimise the length of the cycle that connected j points that minimise the maximum distance from houses vertices to stop vertices.
- In the case of the fire brigade problem the goal is to minimise the time required to turn all the vertices labelled as "fire" to false.

Each graph G' has a function that associates a morphism M of the graph G' to a solution graph of type S and each domain has a specific metric M. In the simplest cases the evaluation graph E is commonly an arc of some length or a single vertex with an integer or real numbers (i.e. fitness landscape of single valued outputs) in it. This is also the case in the scenarios generated in this chapter, however the evaluation graph E can have more complex structures, i.e. stress of the network or star tree of objectives.

At this point the problem scenario can be turned into a JSON format and exported (in order to be solved by some other program or visualised) or it can be given to a solver. For practical purposes we did our experiments directly after generating the instances and packaged the solutions into JSON format. This may depend on the language that the solver used, in our case the solvers are python programs and it was convenient to do both generation and solution of the problem instances under the same environment.

7.2.3 Visualisation

The generators are accompanied with a visualisation tool that allows to plot both problems and solutions on the browser. This allows to visually inspect the results, a common approach in quality control [142], and makes them more accessible. The visualisation engine is built in JavaScript and relies on the Three.JS library [2] for the graphs rendering. The system has a core logic for plotting the road network and 3 different subsections for each specific domain. In all scenarios the engine expects a JSON which contains a collection of vertices and edges belonging to the road network. The vertices are expected to be labelled with OSM ids. Then in the case of ambulance placement domain the engine expects a list of hospitals locations and ambulances locations, these will be treated as disjoint points (*multiset*), optionally the emergency calls and routes used to evaluate the quality of the choice can be provided and plotted.

[2]https://threejs.org/

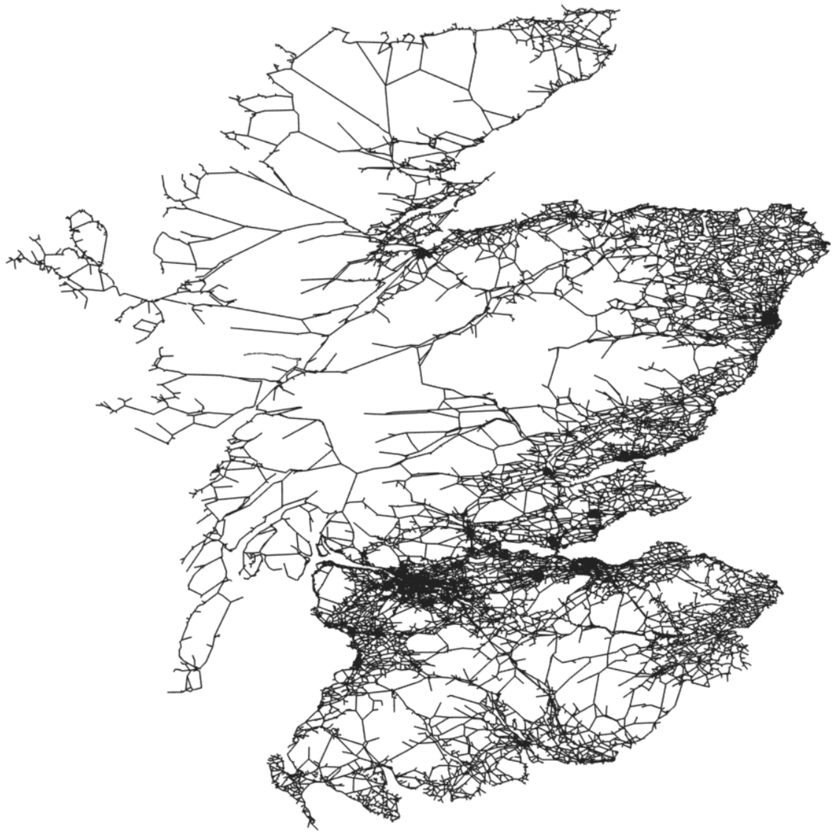


Figure 7.8: The entire road network of Scotland (islands excluded) containing more than 230k nodes and 1 million edges

In the case of school bus routing domain the engine expects a list of stop vertices that will be treated as *cycles*.

In the case of the fire brigade domain the engine expects a list of fire brigade vertices (multiset), a list of emergencies (multiset) and a collection of shortest path routes from the chosen resources and to the emergency locations (as a *collection of sequences* that will form a *forest of multitree* but in the simplest cases they will appear as a *tree* or a *sequence*)

The visualisation can handle very large numbers of nodes and edges even on low spec computers. A laptop without a dedicated graphic card or mid range mobile phone (in the year 2018-19) can handle graphs with more than 40k nodes and 200k edges. Devices equipped with a modern graphic card can visualise much larger graphs such as those in the Fig. 7.9 and 7.8.

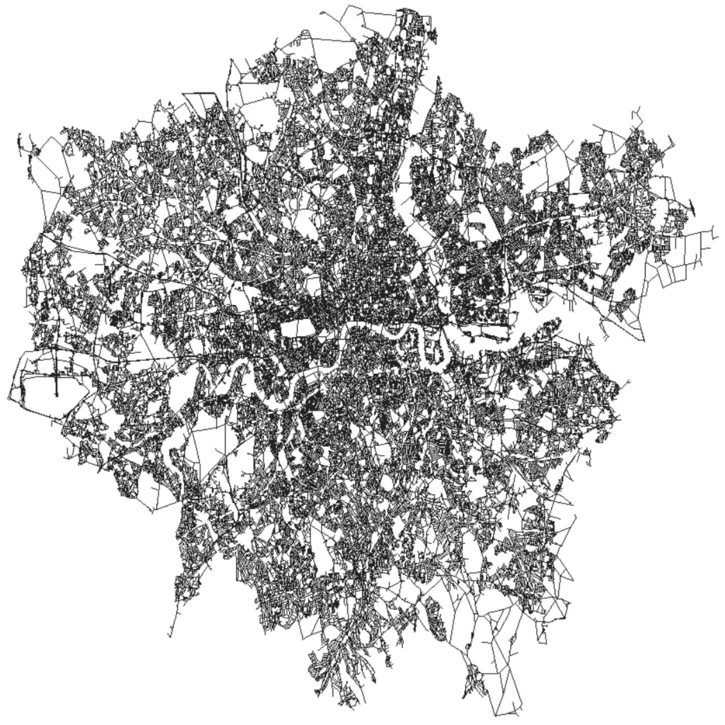


Figure 7.9: The entire road network of London containing more than 120k nodes and 590k edges

7.3 Data set Generated

For each domain we generated 10 sample instances for a total of 30 instances. An example of an instance of the school bus problem can be seen in the figure 7.10, one for the ambulance placement problem can be seen in the figure 7.12 and one for the fire brigade intervention problem 7.11. For each instance generated we verified that the type of graph is correct both programmatically and by visual inspection confirming that the instances have no violations. Specifically we verify that the solutions to the instances of the school bus problem form *cycles*, the solutions of the ambulance placement problem form a *collection of disjoint vertices* (ambulances) and a *tree of depth 2* for each vertex and finally that the solutions of the fire brigade intervention problem form a *tree* for each fire.

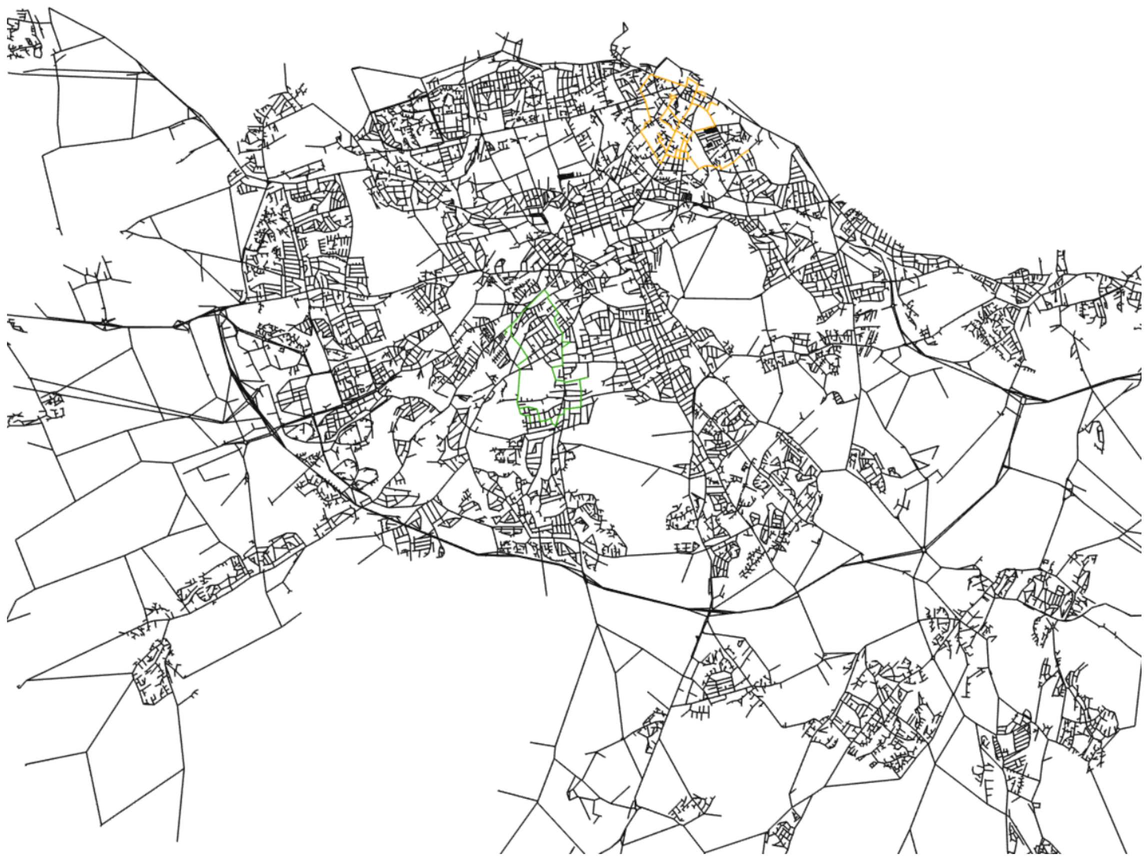


Figure 7.10: Two simple examples of the School bus problem embedded on Edinburgh's road network. In green an optimised tour and in orange a non-optimised tour. Both instances have 10 stops

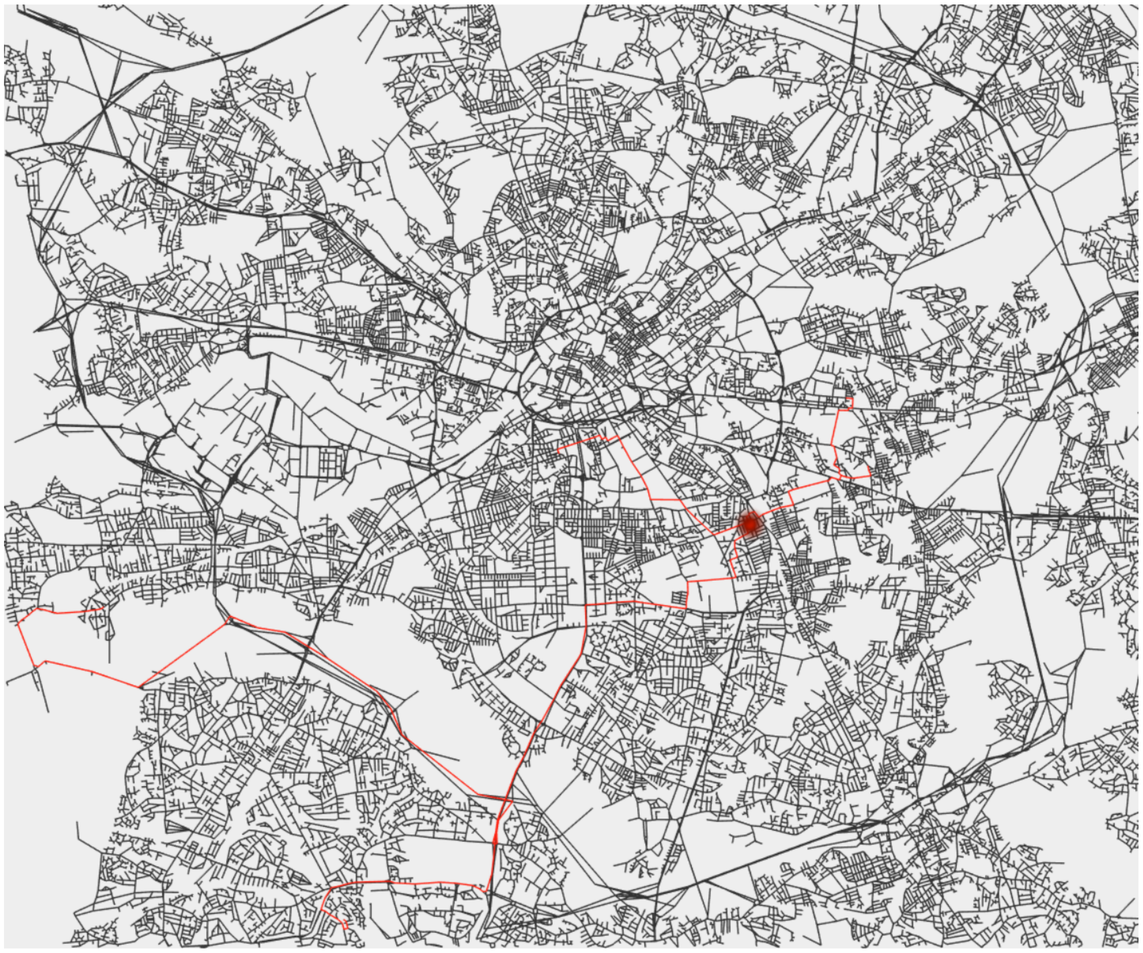


Figure 7.11: Example of the Fire department intervention problem embedded on Manchester's road network. The problem contains 1 fire and the optimised solution uses forces from 5 different sources

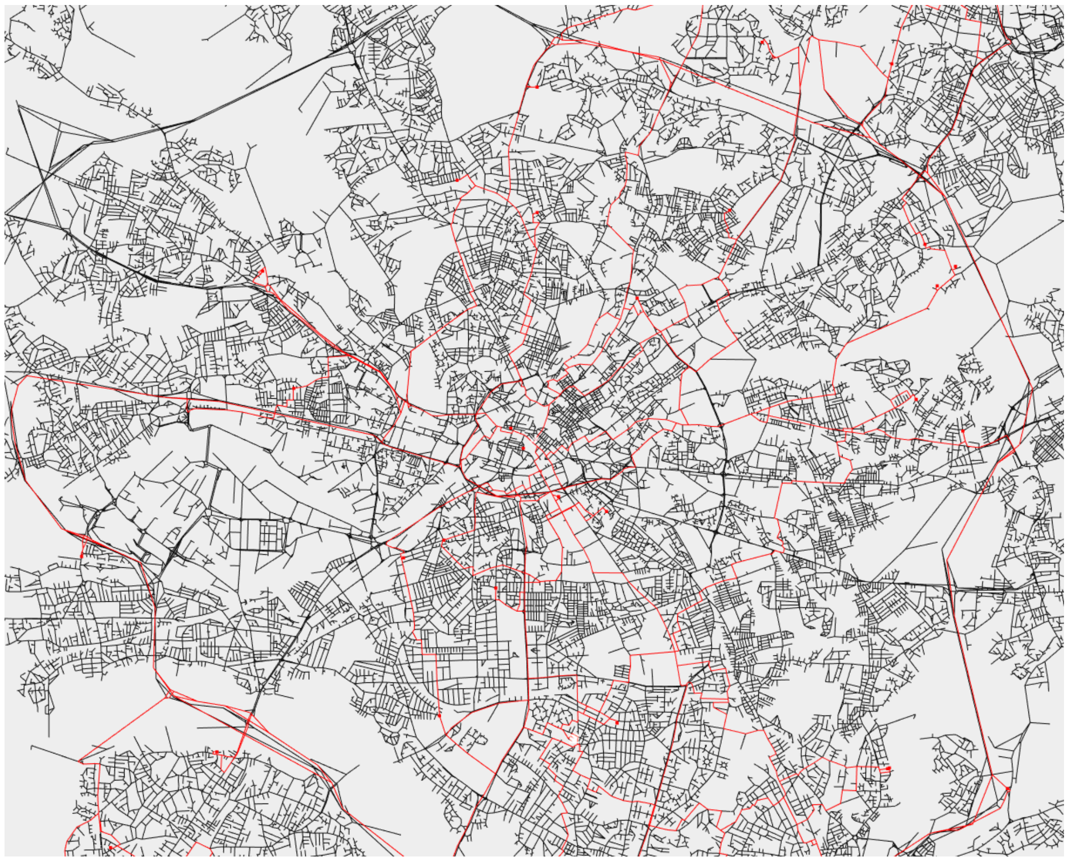


Figure 7.12: Example of the ambulance placement problem embedded on Manchester's road network. Pink lines are paths from the chosen location to the emergency location (red points) and blue lines are paths from the emergency location to the hospital.

7.4 Testing

In order to empirically verify the validity (in the sense that it contains the required information in the correct form to be optimised via a solver) of the generated instances we utilise two well known algorithms, simulated annealing [85] and self-organising maps [88]. These algorithms are well established and have been used on a large variety of problem domains. Both algorithms are used in their 'vanilla' version and we refer the reader to the original paper for the details of the algorithms. Each algorithm has a budget of 1000 iterations. In the case of simulated annealing we use a starting temperature of 10 and a cooling factor of 0.9. The temperature determines the chances of accepting a worst solution at each step and the cooling factor determines how quickly this value decreases. At each step the algorithm reorders the vertices of the bipartite graph via a simple swap operator.

In the case of the self-organising maps we randomly initialise N_C and we use $\alpha(t) = 0.9 * (1 - t/1000)$. N_C determines the initial state of the mesh and $\alpha(t)$ the amount of displacement of the mesh vertices overtime. Each vertex of the mesh is associated to the closest vertex belonging to the problem graph and only one vertex is associated in this way creating a 1-to-1 relationship.

We compare these algorithms with a simple random search where 1000 samples are taken and the best one selected for comparison.

The algorithms are applied to all the domains. We do not expect optimal results but the objective is not to show the quality of the results. The purpose is to show that an off the shelf solver can be plugged to any of the domains, that it can at least trial all the domains available and that the system functions correctly.

7.4.1 Results

The table 7.1 shows the results of the runs of the 2 algorithms, simulated annealing(SA) and self-organising maps(SOM), next to runs of a simple random search (RS). Each result is the best value obtained after 10 restarts for a total of 10000 samples each. All the objective functions lead the optimisation algorithms in the correct direction: shorter tours in the case of the school bus problems, faster intervention time in the ambulance problems and faster fire extinguishing time in the fire brigade problem. We note that in all cases using an optimisation algorithm is better than just using random search. This means that the instances ares not trivial and some combinatorial structure is present in the instances.

7.5 Conclusions

In this chapter we presented the implementation of a single system for the definition and generation of combinatorial optimisation scenarios using graphs. This method allows the generation of multiple instances in multiple domains without changing the representation or format.

The problem scenarios use ideas introduced in the previous chapter where we have shown examples of solvers and hyper-heuristic mechanics. Differently from the previous chapter, that focused on theory and abstractions, in this chapter we focused on the engineering and practical aspect of problem scenario generation. We developed a library that extracts data from the real world and uses it to generate new instances of selected problem domains. These semi-synthetic instances and their solutions can be visualised in any web browser and can be viewed from both computer desktops and mobile phones that support WebGL technology. The hope is that these type of tools could make the work done in operational research more accessible to both academics and non-academics.

It should be highlighted that the goal of the library is not to provide instances from the 3 specified examples but to demonstrate a methodology that can be used to specify new domains in the context of multi-domains generators.

Finally we have shown the flexibility of bipartite-graph's assignments and how graph types can be turned into different types while maintaining the properties that we desire to optimise.

Instance Name	RS	SA	SOM
SchoolBus-10	7300	2465	2407
SchoolBus-20	13435	3956	4043
SchoolBus-30	20531	6666	5621
SchoolBus-40	26986	9294	8074
SchoolBus-50	33601	10739	10602
SchoolBus-60	40217	11949	10942
SchoolBus-70	46832	12235	14515
SchoolBus-80	53448	18186	17901
SchoolBus-90	60063	19283	20779
SchoolBus-100	68679	18680	18460
Ambulance-1-1	80	62	53
Ambulance-2-1	63	45	44
Ambulance-2-2	50	34	27
Ambulance-2-3	48	37	39
Ambulance-2-4	45	37	23
Ambulance-3-2	30	22	17
Ambulance-3-3	28	14	19
Ambulance-3-4	25	14	14
Ambulance-3-5	23	13	12
Ambulance-3-6	21	17	11
Firebrigade-7-2	41	23	17
Firebrigade-10-2	35	15	20
Firebrigade-11-3	50	18	18
Firebrigade-11-4	55	24	23
Firebrigade-15-4	42	18	19
Firebrigade-16-4	40	19	26
Firebrigade-20-5	34	17	21
Firebrigade-20-8	60	22	35
Firebrigade-20-10	72	47	39
Firebrigade-22-10	65	33	26

Table 7.1: Best result obtained from 10 runs of the random search (RS), simulated annealing(SA) and self-organising maps(SOM) applied to the semi-synthetic instances. The number next to school bus is the number of pick ups with results in meters. The first value next to ambulance is the number of vehicle and the second value is the number of hospitals with results in minutes. The first value next to firebrigade is the number of teams available and the second value the number of fires with results in minutes

Chapter 8

Conclusions

This chapter sums up the previous chapters of this thesis, highlights their contributions to the objectives set and discusses the finding that emerged during their development. The original aim of the thesis was increasing the generality and applicability of hyper-heuristic methodologies and investigating how this could be achieved. To accomplish this we asked the following questions:

1. To what extent can a *single* hyper-heuristic generator be used to synthesise heuristics for multiple combinatorial domains, assuming a common representation?

2. To what extent is a graph-based formalism able to express combinatorial problems and the constituent parts of complex hyper-heuristic algorithms so that more general hyper-heuristic optimisers could be developed?

To answer these questions we proposed two different methods that utilised a fixed representation for multiple domains and searched in the space of operators in order to find heuristics that could suit the specific domains. We showed that given a sufficiently general grammar it is possible to evolve heuristics adapted to specific representation-problem pairs. Then we proposed a framework for the production of graphs with different characteristics that can be used to describe both hyper-heuristic algorithms and combinatorial optimisation problems. The intuition is that if a system is able to describe and modify both then it will be able to optimise both.

8.1 Generative Hyper-Heuristics

Chapter 4 and 5 focused on heuristic methods that do not make use of existing heuristics but build heuristics from domain agnostic building blocks. This study helped us determine and measure our ability to solve problems from new domains when heuristics for that domain are not available neither are domain specific representation.

In both chapters the generator of heuristics made use of Grammatical Evolution where a grammar contained the building blocks and structure available to produce the heuristics in the form of Python scripts. A classical evolutionary process refined the heuristics depending on the type of domain they were evaluated on. In both cases we have shown that even if the representation of the knapsack problem went from 2^n to n! configurations it was possible to find heuristics that operated intelligently on the representation that led to solutions of quality comparable to the ones obtained with ideal representation and problem specific operators. These experiments provided a fundamental contribution to answering Question 1 as it has shown that even if the representation at hand is not specifically tailored to the domain it is possible to search in the space of heuristics for operators that, in part, offsets this shortcoming.

8.1.1 Generating Constructive Heuristics

Chapter 4 introduced a grammar that made use of the geometric graph's properties in order to build constructive heuristics. This system helped us answering question 1 as it has shown that it is possible to develop general generative methods applicable to more than one domain and produce results that are of acceptable quality, even if not optimal and close to optimal in some cases. The proposed method showed a way to trade more computational resources for increasing the quality of solution when no handcrafted representations and operators are available.

Of particular interest is the role played by the metric chosen by the heuristics depending on the problem domain. The grammar available to the generator could optionally use different metrics when measuring the distance between two vertices. An analysis of the best heuristics in each domain highlighted that the cosine distance was chosen by the evolutionary process in almost all the heuristics that were generated for the multidimensional knapsack problem.

8.1.2 Generating Perturbative Heuristics

Chapter 5 introduced a grammar that made use of cuts, inversions and permutation of sub-sequences of a partial permutation in order to assemble perturbative heuristics. This chapter complemented Chapter 4 and strengthened the idea that it is possible to develop general methods applicable to more than one domain with a relatively small loss in terms of optimality of the solutions generated. The generator of heuristics was tested on instances of the TSP and MKP and compared to other state of the art methods.

A further case study was investigated using the Load Balancing Problem. This problem had to be solved to organise the schedule of a large amount of experiments. The same system used to generate the heuristics for TSP and MKP was used successfully to assign groups of experiments to different cores of a multicore machine so that all the experiments finished at the same time. This minimised the cost of the experiments as cloud based multicore machines are paid as a whole regardless of how many cores are actually used, it is therefore ideal to spread the load among the cores as evenly as possible.

8.2 Graph Representations for Hyper-Heuristics

Chapter 6 and 7 were dedicated to the creation of representations for graphs with particular topological structures and ways to describe the composition of these properties to create new types of graphs. Representation plays a fundamental part in optimisation however we suggest that it is not feasible to create ad-hoc representations for each single new domain and few representations will have to suffice for a variety of problem domains. Furthermore a unifying representation for both solver and problem instances will enable the creation of a system that can synthesise both using the same building blocks. This would allow hyper-heuristic systems to operate on its own internal mechanics that often involve the optimisation of some combinatorial problem. These chapters helped to answer question 2 as they provided a formalism that describes the structure of graphs, followed by an explanation on how the graph types developed the mechanics found in hyper-heuristic solvers and finally an implementation of a set of problem instance generators built using the same concepts.

8.2.1 Solver Mechanics

In Chapter 6 we presented a graph representation, constructed from first principles, that can be used to create data structures that are ubiquitous in computer science. Starting with disjoint collections, sequences, trees and cycles we have built a larger collection of structures using property composition. We have shown how these graph structures can be used to describe the type of processes and decision problems within hyper-heuristic systems. This has contributed to answering question 2 as it has shown that this computable language can be use to for different parts of the system. This provides the groundwork required to build hyper-heuristic systems that can optimise classic combinatorial problems but are also applicable to hyper-heuristic internal components that also require optimisation.

8.2.2 Problems instances representation and generation

In Chapter 7 we used the high level graph representations developed in Chapter 6 to describe 3 different models of combinatorial optimisation problems inspired by real world problems. Then we implemented scenario generators that extract real world network graphs and embed them with the problems we have modelled. This enabled the automatic creation of semi-synthetic instances that could be further extended by adding extra information in the network or adding extra models to it. This chapter focused on the practical and engineering aspects and contributed to answering question 2 by showing that the same graph structures can be used for both combinatorial problems and solvers. The part of the work was also an attempt to build a multi-problem benchmark as a useful tool in the development of future multi-problem solver systems.

8.3 Cost of crossing the domain

This section provides some final insights into the cost of crossing the domain, that is, the price one has to pay for using a single representation to represent multiple domains and creating a heuristic generator that manipulates the representation in different ways. This cost spans several measures. The first is in terms of loss of quality of the output, our experiments confirmed a well known fact that general systems tend to perform worse than specialised and ad-hoc algorithms given the same amount of computation. Then there is the time required to evolve the new heuristics which is orders of magnitude greater than the time required by the heuristic to solve the problem. This contains the computational cost required to evaluate all the heuristics that end up being discarded but are created as part of the evolutionary process. Finally the heuristic generated in this way must be validated over a collection of problem instances in order to be confident of their performance. These costs are balanced by the fact that the procedure is automatic and its feasibility depends on the computational power available.

8.4 Future work

We present some natural extensions to this thesis and ideas for future research.

Automatic synthesis of tuned instances

The instance generators developed in Chapter 7 can produce randomised instances of unspecified complexity where only the size of the problem can be defined. The natural extension to these generators would include the possibility to specify how "hard" the instances generated are based on some criteria of difficulty. Further, each generator is programmed by hand for each specific domain. Ideally one should be able to define the high level properties, such as the topological characteristic of the graphs and types of vertices, and a generator of instances should be automatically produced. These ideas are currently pursued by the author and some results are already available in [3].

Merging automated modelling and hyper heuristics in order to navigate both representation space and heuristic space

In this research we have presented a system that forces one representation over multiple problem domains. It is well understood that a good representation can provide huge efficiency gains when paired with the right operator. It would be desirable to move in the space of representations in automated manner in the same way we navigated the space of operators in Chapters 4 and 5. The field of automated modelling could provide the right tools to achieve this as several applications in problem reformulation already exist [105]. Research in this direction would require tools to help facilitating the choice of representation and heuristic pairings as these spaces are vastly larger and more complex. Research on this area would also require substantially more computational efforts. These type of systems would view problems as representable in many different ways and for each representation a plethora of different strategies and operators could be generated.

Streams of novel domains and novel heuristics in one macro system

In the work presented we have generated heuristics dealing with a small number of domains. The successive step, following the work on online problem optimisation when a stream of problems of a specific domain is fed into the system (and possibly generated by it), is to create a system for online optimisation where each successive problem may belong to a new domain. Similarly on already existing methodology for online optimisation the system should be able to pick a heuristic or generate novel ones as required.

What next?

It is fair to assume that autonomous systems will have to deal with problems from a stream (or multiple streams) of problems from different domains, it would be natural to equip such a system with some predictive subsystem. Similarly with systems that attempt to predict which instance (in terms of features, difficulties, size, etc...) will appear next, such a system would be required to identify what domain the next problems in the stream will belong to. However predicting which problem domain is most likely to appear next is only part of the problem as this prediction only commences the search for the strategy that will tackle the problem.

Finally, even if hyper-heuristic systems that go beyond the “domain barrier” are desirable, solving problems belonging to many different domains and choosing algorithms that are applicable across several domains is vastly more complex than solving single instances of NP-hard problems. Therefore it can be expected that hyper-heuristics will benefits and become ever more interconnected to the topic of algorithm selection and configuration [89].

Appendix A

Supplementary Material

			Evolved reusable heuristics					Greedy heuristic	
	Dataset	Instance	Best	Worst	Average	Median	Optima	Best	Average
1	hp	hp1	3318	2696	3089.790	2901	3418	3303	2790.9
2	hp	hp2	3131	3113	3260.121	3228	3186	2963	2803.8
3	pb	pb1	3055	2655	2979.098	2820	3090	2762	2214.1
4	pb	pb2	3087	2470	2914.383	2697	3186	2901	2680.5
5	pb	pb4	94034	84893	89602.704	88978	95168	84632	76627.5
6	pb	pb5	1963	1943	1980.401	1998	2139	1976	1700.5
7	pb	pb6	749	663	715.630	692	776	716	674.3
8	pb	pb7	965	822	910.350	901	1035	921	752.8
9	pet	pet2	79524	66603	73553.746	71482	87061	78619	63628.9
10	pet	pet3	3796	3674	3737.356	3863	4015	3852	3113.3
11	pet	pet4	5923	5092	5703.799	5662	6120	6029	5947.6
12	pet	pet5	11375	9157	10290.201	10183	12400	11989	11530.1
13	pet	pet6	9931	8437	9592.361	9033	10618	10296	8453.9
14	pet	pet7	16509	13825	15813.989	14971	16537	15001	12699.9
15	sento	sento1	7601	6452	7187.535	6970	7772	7424	7137.4
16	sento	sento2	7893	7137	7643.002	7520	8722	8428	6900.2
17	weing	weing1	135431	127387	135560.0	137406	141278	134060	113855.5
18	weing	weing2	119272	108445	114216.0	113873	130883	125028	110224.4
19	weing	weing3	90780	86960	90838.0	89999	95677	94598	89136.1
20	weing	weing4	112325	100183	107035.6	105563	119337	111737	94311.3
21	weing	weing5	95874	78054	91079.5	86996	98796	96943	96830.3
22	weing	weing6	120344	103124	111851.1	107599	130623	119183	97484.5
23	weing	weing7	1005070	820977	925634.2	903362	1095445	968895	872501.1
24	weing	weing8	578621	521090	553942.1	559165	624319	590726	510130.3
25	weish	weish01	4461	3915	4382.388	4202	4554	4063	3722.2
26	weish	weish02	4224	3884	4106.429	4185	4536	4127	3945.9
27	weish	weish03	3962	3484	3883.436	3767	4115	3696	3355.1

Table A.1: Comparison between the evolved reusable heuristics and a greedy heuristics (part1)

			Evolved reusable heuristics					Greedy heuristic	
	Dataset	Instance	Best	Worst	Average	Median	Optima	Best	Average
28	weish	weish04	4299	4038	4329.434	4282	4561	4476	3802.2
29	weish	weish05	4429	3597	4065.256	3849	4514	4457	3583.6
30	weish	weish06	5474	4642	5168.923	4912	5557	5232	4931.1
31	weish	weish07	5362	4502	4948.143	4847	5567	4896	4602.0
32	weish	weish08	5392	4712	5137.568	5161	5605	5588	4694.6
33	weish	weish09	5226	4228	4870.974	4715	5246	4992	4481.9
34	weish	weish10	5741	5126	5649.510	5443	6339	6263	5138.8
35	weish	weish11	5107	4415	4988.925	4821	5643	5236	4360.1
36	weish	weish12	5715	4573	5370.955	5197	6339	6154	5584.6
37	weish	weish13	5759	5623	5926.084	5943	6159	5579	5012.6
38	weish	weish14	6565	6154	6514.844	6618	6954	6463	6262.2
39	weish	weish15	7405	7017	7253.824	7139	7486	7219	7150.7
40	weish	weish16	6977	6340	6723.443	6660	7289	6634	6202.8
41	weish	weish17	8079	6556	7497.267	7152	8633	8221	6650.8
42	weish	weish18	9498	8281	9074.954	8936	9580	9027	8486.6
43	weish	weish19	7612	6681	7327.741	7139	7698	7166	6095.4
44	weish	weish20	9244	8736	9228.757	9409	9450	9193	8450.1
45	weish	weish21	8415	7098	7848.938	7568	9074	9004	7332.8
46	weish	weish22	8079	7360	8094.395	7871	8947	8644	7226.0
47	weish	weish23	7553	7004	7426.393	7405	8344	7510	6314.9
48	weish	weish24	9496	9118	9546.735	9631	10220	9261	8843.0
49	weish	weish25	9614	9048	9529.524	9483	9939	9909	8734.6
50	weish	weish26	9429	9005	9318.846	9486	9584	9509	8316.3
51	weish	weish27	9569	9167	9799.804	9540	9819	9169	7864.6
52	weish	weish28	9371	8022	9023.719	8846	9492	8488	6799.0
53	weish	weish29	8504	8432	8724.097	8628	9410	9247	7743.1
54	weish	weish30	11044	9867	10483.937	10252	11191	11155	9087.0

Table A.2: Comparison between the evolved reusable heuristics and a greedy heuristic (part2)

Knapsack Reusable Heuristics
pdiv(pdiv(pdiv(distance_to_v0(vertex,'cosine')*plog(psqrt(pdiv(distance_to_v0(vertex,'cosine'), e))),len(vertices_vecs)),elements_sum(ref_difference(chain_vsum())))+cos(pdiv(len(vertices_vecs), psqrt(longest_potential_edge)))-len(vertices_vecs),pdiv(distance_to_v0(vertex,'cosine'), psqrt(distance_osms(vertex,'cosine'))))
distance_to_v0(vertex,'cosine')+pdiv(exp(pdiv(exp(19.84),exp(vec_min(s_vertex0))))* plog(pdiv(hull_area(),sin(plog(vec_min(chain_delta_vsum()))))))), exp(distance_osms(vertex,'cosine')))
distance_to_v0(vertex,'cosine')-pdiv(tanh(e),plog(psqrt(exp(psqrt(plog(plog(plog(hull_area()))- len(vertices_vecs)))))))- hull_area()
plog(pdiv(distance_to_v0(vertex,'cosine'),psqrt(pdiv(49.40, pdiv(95.05,distance_osms(vertex,'cosine')))))+pdiv(sin(47.81),distance_osms(vertex,'euclidean'))))
distance_to_v0(vertex,'cosine')*plog(pdiv(pdiv(18.71,exp(e)),pdiv(exp(e),plog(cos(pi)))-pi+ exp(plog(elements_sum(ref_difference(ref_difference(vertex))))))*distance_to_v0(vertex,'cosine'))- plog(kd_leg_angle(vertex))
pdiv(distance_to_v0(vertex,'cosine'),pdiv(elements_sum(chain_vsum())* vec_max(ref_difference(chain_vsum())),psqrt(pdiv(distance_osms(vertex,'euclidean')* exp(pdiv(longest_potential_edge,pi)),e))))-exp(psqrt(psqrt(pi)))
distance_to_v0(vertex,'cosine')+cos(pdiv(distance_osms(vertex,'cosine'), pdiv(len(solution_chain),psqrt(18.81))))
plog(pdiv(distance_to_v0(vertex,'cosine'), exp(pdiv(pdiv(longest_potential_edge, len(vertices_vecs)), pdiv(e,distance_osms(vertex,'cosine'))))))
distance_to_v0(vertex,'cosine')-pdiv(tanh(plog(e))*distance_osms(vertex,'cosine'), distance_to_v0(vertex,'cosine'))
psqrt(distance_to_v0(vertex,'cosine'))-sin(plog(exp(distance_osms(vertex,'cosine'))))
distance_to_v0(vertex,'cosine')-psqrt(psqrt(plog(pdiv(distance_osms(vertex,'cosine'), pdiv(76.22,plog(kd_leg_angle(vertex)))))))
exp(exp(distance_to_v0(vertex,'cosine')))
plog(psqrt(pdiv(psqrt(longest_potential_edge),pdiv(plog(tanh(elements_sum(chain_delta_vsum())- len(solution_chain)))+pi,pdiv(psqrt(distance_to_v0(vertex,'euclidean')), cos(plog(len(solution_chain)))-pdiv(distance_osms(vertex,'cosine'), psqrt(pdiv(psqrt(len(vertices_vecs)),psqrt(vec_max(s_vertex0))))))))))
distance_to_v0(vertex,'cosine')-exp(pdiv(psqrt(93.73),len(vertices_vecs)))-len(solution_chain)* tanh(37.02)-pdiv(exp(e*estimated_graph_complexity()), pdiv(38.57,psqrt(distance_osms(vertex,'euclidean'))))
exp(plog(exp(pdiv(distance_to_v0(vertex,'cosine')-92.38,35.09))* pdiv(distance_osms(vertex,'cosine')- 62.18,pdiv(exp(pi),e))))*len(solution_chain)
psqrt(distance_to_v0(vertex,'cosine')+distance_to_v0(vertex,'cosine')* psqrt(exp(exp(pi*tanh(65.33))))-psqrt(vec_max(vertex)*psqrt(exp(exp(pi*tanh(63.52))))- pdiv(pi*tanh(63.59),len(solution_chain))))*exp(exp(len(solution_chain)))
plog(distance_to_v0(vertex,'cosine')*distance_to_v0(vertex,'euclidean'))*pdiv(tanh(plog(exp(e))), plog(exp(distance_osms(vertex,'cosine'))))
plog(distance_to_v0(vertex,'cosine'))-pdiv(psqrt(cos(exp(plog(len(solution_chain)))* psqrt(plog(plog(len(solution_chain))))*plog(distance_to_v0(vertex,'euclidean')))), psqrt(psqrt(cos(hull_area()))))*plog(tanh(pdiv(04.79,len(vertices_vecs))))
exp(distance_to_v0(vertex,'cosine'))+hull_area()
psqrt(distance_to_v0(vertex,'euclidean'))-sin(exp(elements_sum(vertex)))+sin(exp(63.02))

Table A.3: Phenotypes of the evolved MKP heuristics

TSP Reusable Heuristics
psqrt(pdiv(distance_osms(vertex,’euclidean’),pdiv(hull_area()*distance_to_v0(vertex,’euclidean’), longest_potential_edge)+psqrt(pi+len(vertices_vecs))))
distance_osms(vertex,’euclidean’)+sin(pdiv(41.83*e,plog(tanh(plog(sin(longest_potential_edge)+e)) - distance_osms(vertex,’euclidean’))))
pdiv(longest_potential_edge*psqrt(distance_osms(vertex,’euclidean’))-psqrt(10.79),psqrt(distance_to_v0(vertex,’euclidean’)))
psqrt(distance_to_v0(vertex,’euclidean’)-pdiv(cos(plog(62.24-25.73)),plog(pi)*vec_min(vertex)))* pdiv(psqrt(len(vertices_vecs))*pdiv(plog(pi),pdiv(psqrt(05.77),exp(kd_leg_angle(vertex)))), tanh(pdiv(cos(e),pi))*distance_osms(vertex,’euclidean’))
pdiv(distance_osms(vertex,’euclidean’),kd_leg_angle(vertex)+ pdiv(psqrt(psqrt(exp(psqrt(kd_leg_angle(vertex))))-e),plog(pdiv(cos(e),plog(kd_leg_angle(vertex)-plog(cos(e)))*plog(distance_to_v0(vertex,’cosine’))-psqrt(distance_to_v0(vertex,’cosine’))))))
distance_osms(vertex,’euclidean’)-plog(cos(psqrt(pdiv(40.97,vec_max(ref_difference(vertex))))+ plog(exp(psqrt(plog(20.17))))))
pdiv(distance_osms(vertex,’euclidean’),distance_to_v0(vertex,’euclidean’)* plog(pdiv(cos(psqrt(82.29)),psqrt(20.27)))+ psqrt(e)+psqrt(plog(vec_max(vertex))-psqrt(distance_osms(vertex,’euclidean’))))
distance_osms(vertex,’euclidean’)*e-psqrt(psqrt(len(vertices_vecs)+len(solution_chain))+ tanh(vec_max(vertex)+ exp(plog(distance_osms(vertex,’euclidean’)))+ sin(len(solution_chain)* distance_osms(vertex,’euclidean’)))* pdiv(sin(exp(len(vertices_vecs))),len(vertices_vecs)))
distance_osms(vertex,’euclidean’)-plog(psqrt(pdiv(distance_to_v0(vertex,’euclidean’), psqrt(plog(len(solution_chain))))))*len(solution_chain)
distance_osms(vertex,’euclidean’)*pdiv(psqrt(len(vertices_vecs)), psqrt(distance_to_v0(vertex,’euclidean’)))
psqrt(exp(pdiv(pdiv(distance_osms(vertex,’euclidean’)-pdiv(cos(len(solution_chain)), tanh(len(solution_chain))),psqrt(distance_to_v0(vertex,’euclidean’))), psqrt(psqrt(distance_osms(vertex,’euclidean’)))*psqrt(sin(psqrt(len(solution_chain))))*exp(pi))))
32.30+psqrt(distance_to_v0(vertex,’cosine’))-cos(len(vertices_vecs)*distance_osms(vertex,’cosine’))-pi+distance_osms(vertex,’euclidean’)+exp(32.39)
pdiv(distance_osms(vertex,’euclidean’),exp(distance_to_v0(vertex,’cosine’))*e-pdiv(psqrt(len(solution_chain)), psqrt(60.44)+pi)+plog(sin(39.84*cos(37.06)*37.20-cos(psqrt(len(solution_chain))))))+ distance_osms(vertex,’cosine’)* distance_osms(vertex,’euclidean’)
plog(plog(psqrt(psqrt(exp(distance_osms(vertex,’euclidean’)+pi)))+exp(cos(len(vertices_vecs)))+ exp(pdiv(cos(sin(plog(plog(59.90))))+plog(tanh(sin(plog(e)))),tanh(plog(kd_leg_angle(vertex)-vec_max(ref_difference(chain_vsum())))))))+sin(sin(vec_max(vertex))-len(vertices_vecs))))
distance_osms(vertex,’euclidean’)+e*plog(distance_osms(vertex,’euclidean’))* pdiv(distance_osms(vertex,’euclidean’), distance_to_v0(vertex,’euclidean’))+ 81.74-estimated_graph_complexity()
distance_osms(vertex,’euclidean’)-sin(distance_to_v0(vertex,’cosine’))
distance_osms(vertex,’euclidean’)-exp(cos(distance_to_v0(vertex,’euclidean’)))
distance_osms(vertex,’euclidean’)+ exp(sin(plog(psqrt(psqrt(11.39*plog(cos(elements_sum(vertex))))))+len(vertices_vecs)))
psqrt(pdiv(distance_osms(vertex,’euclidean’),exp(distance_to_v0(vertex,’cosine’)))-psqrt(pdiv(plog(tanh(exp(sin(psqrt(len(vertices_vecs)))))),68.30)))

Table A.4: Phenotypes of the evolved TSP heuristics in their python program form

MKP group			
m	n	a	HHGE
5	100	0.25	2.54
		0.5	1.32
		0.75	0.58
		average	1.48
5	250	0.25	1.07
		0.5	0.79
		0.75	0.48
		average	0.78
5	500	0.25	1.18
		0.5	1.32
		0.75	0.6
		average	1.03
10	100	0.25	3.86
		0.5	1.95
		0.75	1.26
		average	2.36
10	250	0.25	2.26
		0.5	0.82
		0.75	0.78
		average	1.29
10	500	0.25	2.21
		0.5	2.35
		0.75	1.19
		average	1.92
30	100	0.25	5.71
		0.5	3.1
		0.75	2.01
		average	3.61
30	250	0.25	4.11
		0.5	2.15
		0.75	0.93
		average	2.40
30	500	0.25	3.77
		0.5	3.21
		0.75	1.42
		average	2.80

Table A.5: Results grouped by number of constraints (m), size (n), ratio of the expected number of object that could fit the knapsack (a) and %-gap

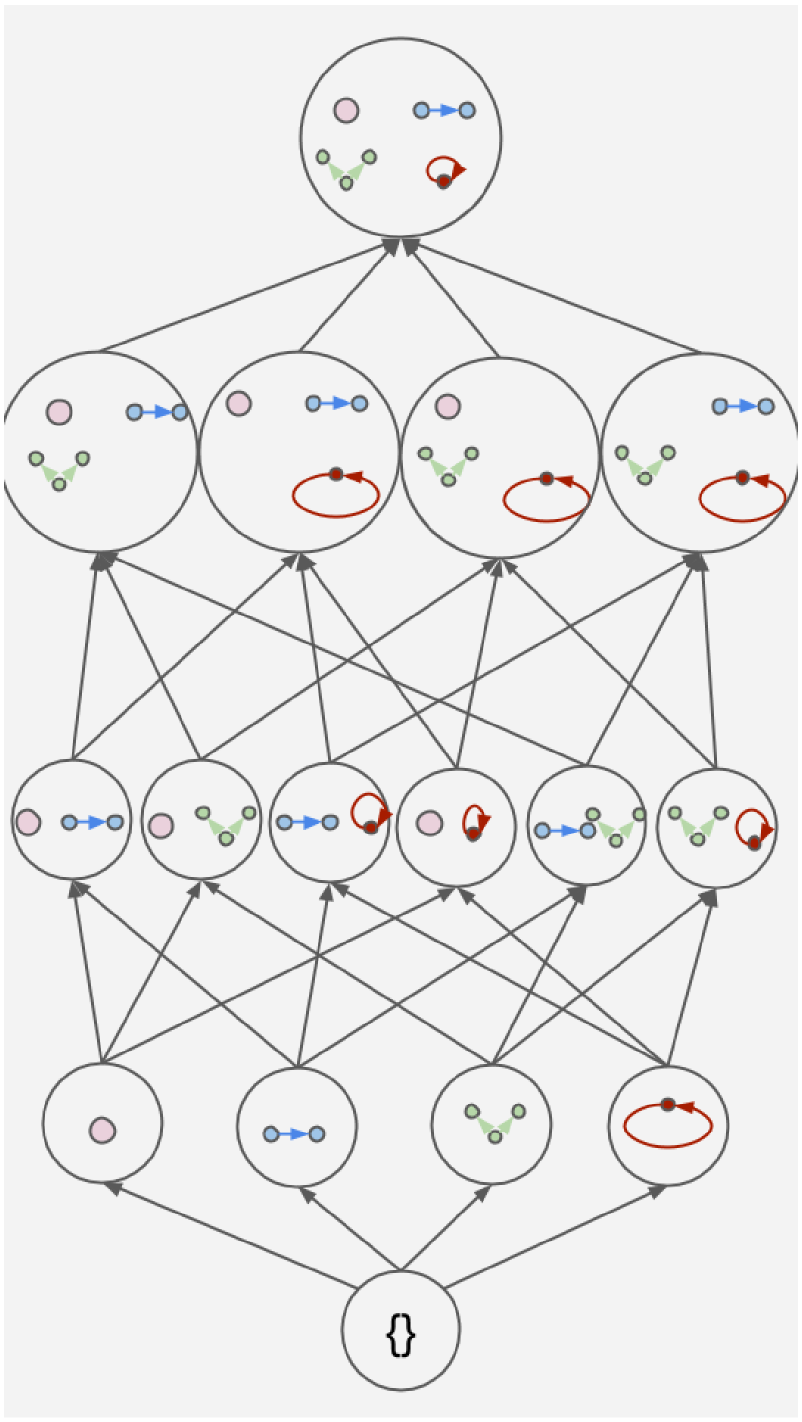


Figure A.1: The diagram of the power-set that contain all the combination of the proposed four properties

Jobs	Time(s)
5x100	25
5x250	50
5x500	80
10x100	50
10x250	100
10x500	210
30x100	125
30x250	150
30x500	500

Table A.6: Jobs and their execution times used as a use case for the load balancing problem

Heuristic	Optimality hits - use case instance(%)	Optimality hits - synthesised instances(%)
h1	85	62
h2	86	70
h3	79	68
h4	75	79
h5	89	57
h6	82	79
h7	80	74
h8	85	67
h9	84	80
h10	87	78

Table A.7: Hit percentage of the estimated optima against the synthesised instances of the load balancing problem

	HHGE					CBPSO		SACRO-BPSO	
Instance	Best	Worst	Average	Median	Optima	Best	Average	Best	*Average*
hp1	3418	3385	**3410.56**	3418	3418	3418	3403.9	3418	*3413.38*
hp2	3186	2997	3171.54	3186	3186	3186	3173.61	3186	*3184.74*
pb1	3090	3057	**3083.32**	3090	3090	3090	3079.74	3090	*3086.78*
pb2	3186	3114	**3179.88**	3186	3186	3186	3171.55	3186	*3186*
pb4	95168	90961	93515.54	93897	95168	95168	94863.67	95168	*95168*
pb5	2139	2085	**2120.06**	2130.5	2139	2139	2135.6	2139	*2139*
pb6	776	641	733.12	735.5	776	776	758.26	776	*776*
pb7	1035	983	1018.9	1025	1035	1035	1021.95	1035	*1035*
pet2	87061	78574	85409.32	87061	87061	-	-	-	-
pet3	4015	3165	3955.8	4015	4015	-	-	-	-
pet4	6120	5440	6040.2	6110	6120	-	-	-	-
pet5	12400	12090	12363.1	12400	12400	-	-	-	-
pet6	10618	10107	10592.1	10604	10618	-	-	-	-
pet7	16537	15683	16504.48	16537	16537	-	-	-	-
sento1	7772	7491	**7706.92**	7749.5	7772	7772	7635.72	7772	*7769.48*
sento2	8722	8614	**8691.02**	8704	8722	8722	8668.47	8722	*8722*
weing1	141278	135673	140619.36	141278	141278	141278	141226.8	141278	*141278*
weing2	130883	118035	128542.94	130712	130883	130883	130759.8	130883	*130883*
weing3	95677	77897	93099.5	94908	95677	95677	95503.93	95677	*95676.39*
weing4	119337	100734	117811.56	119337	119337	119337	119294.2	119337	*119337*
weing5	98796	78155	95912	98475.5	98796	98796	98710.4	98796	*98796*
weing6	130623	117715	129452.56	130233	130623	130623	130531.3	130623	*130623*
weing7	1095382	1088277	**1093583.14**	1093595	1095445	1095382	1084172	1095382	*1094349*
weing8	624319	525663	**606175.12**	613070	624319	624319	597190.6	624319	*622079.9*
weish01	4554	4298	4494.34	4530	4554	4554	4548.55	4554	*4554*
weish02	4536	4164	4485.12	4536	4536	4536	4531.88	4536	*4536*
weish03	4115	3707	3963.08	3985	4115	4115	4105.79	4115	*4115*
weish04	4561	3921	4385.5	4455	4561	4561	4552.41	4561	*4561*
weish05	4514	3754	4265.56	4479.5	4514	4514	4505.89	4514	*4514*
weish06	5557	5238	5503.16	5538	5557	5557	5533.79	5557	*5553.75*
weish07	5567	5230	5496.56	5542	5567	5567	5547.83	5567	*5567*
weish08	5605	5276	5534.82	5597.5	5605	5605	5596.16	5605	*5605*
weish09	5246	4626	5062.24	5128	5246	5246	5232.99	5246	*5246*
weish10	6339	5986	6244.82	6314	6339	6339	6271.84	6339	*6339*
weish11	5643	5192	5522.18	5631.5	5643	5643	5532.15	5643	*5643*
weish12	6339	5951	6217.14	6322.5	6339	6339	6231.5	6339	*6339*
weish13	6159	5780	6032.28	6056	6159	6159	6120.38	6159	*6159*
weish14	6954	6581	6827.9	6852	6954	6954	6837.77	6954	*6954*
weish15	7486	7113	**7391**	7445.5	7486	7486	7324.55	7486	*7486*
weish16	7289	6902	7154.82	7159.5	7289	7289	7288.7	7289	*7288.7*
weish17	8633	8506	**8609**	8633	8633	8633	8547.71	8633	*8633*
weish18	9580	9310	**9527**	9560.5	9580	9580	9480.86	9580	*9578.46*
weish19	7698	7272	7505.3	7527	7698	7698	7528.55	7698	*7698*
weish20	9450	9117	**9381.32**	9430	9450	9450	9332.11	9450	*9450*
weish21	9074	8655	**8972.9**	9025	9074	9074	8948.22	9074	*9074*
weish22	8947	8466	**8814.7**	8871	8947	8947	8774.2	8947	*8936.92*
weish23	8344	7809	**8202.06**	8217.5	8344	8344	8165	8344	*8344*
weish24	10220	9923	**10154.54**	10185.5	10220	10220	10106.28	10220	*10219.7*
weish25	9939	9667	**9872.48**	9909.5	9939	9939	9826.57	9939	*9939*
weish26	9584	9175	**9434.92**	9473	9584	9584	9313.87	9584	*9584*
weish27	9819	9244	**9652.3**	9671	9819	9819	9607.54	9819	*9819*
weish28	9492	8970	**9328.52**	9347.5	9492	9492	9123.26	9492	*9492*
weish29	9410	8794	**9217.28**	9279	9410	9410	9025.5	9410	*9410*
weish30	11191	10960	**11135.64**	11161	11191	11191	10987.21	11191	*11190.12*

Table A.8: Generated heuristics vs specialised meta-heuristics from [33]. Highlighted values for HHGE indicate where it outperforms CBPSO. SACRO-BPSO performs best in all instances

Bibliography

[1] Salwani Abdullah, Nasser R Sabar, Mohd Zakree Ahmad Nazri, Hamza Turabieh, and Barry McCollum. A constructive hyper-heuristics for rough set attribute reduction. In *2010 10th International Conference on Intelligent Systems Design and Applications*, pages 1032–1035. IEEE, 2010.

[2] Russell L Ackoff. The future of operational research is past. *Journal of the operational research society*, 30(2):93–104, 1979.

[3] Özgür Akgün, Nguyen Dang, Ian Miguel, András Z Salamon, and Christopher Stone. Instance generation via generator instances. In *International Conference on Principles and Practice of Constraint Programming*, pages 3–19. Springer, 2019.

[4] Shun-ichi Amari and Hiroshi Nagaoka. *Methods of information geometry*, volume 191. American Mathematical Soc., 2007.

[5] José María Amigó, Jorge Gálvez, and Vincent M Villar. A review on molecular topology: applying graph theory to drug discovery and design. *Naturwissenschaften*, 96(7):749–761, 2009.

[6] Rafael Andrade, Tibérius Bonates, Manoel Campêlo, and Mardson Ferreira. Minimum linear arrangements. *Electronic Notes in Discrete Mathematics*, 62:63–68, 2017.

[7] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[8] Uwe Aßmann, Steffen Zschaler, and Gerd Wagner. *Ontologies, Meta-models, and the Model-Driven Paradigm*, pages 249–273. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[9] Anne Auger and Benjamin Doerr. *Theory of randomized search heuristics: Foundations and recent developments*, volume 1. World Scientific, 2011.

[10] Artin Avanes and Johann-Christoph Freytag. Adaptive workflow scheduling under resource allocation constraints and network dynamics. *Proceedings of the VLDB Endowment*, 1(2):1631–1637, 2008.

[11] Mohamed Bader-El-Den and Riccardo Poli. Generating SAT local-search heuristics using a GP hyper-heuristic framework. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 37–49. Springer, 2007.

[12] Márcio P Basgalupp, Rodrigo C Barros, and Vili Podgorelec. Evolving decision-tree induction algorithms with a multi-objective hyper-heuristic. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 110–117. ACM, 2015.

[13] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global Constraint Catalog, 2005.

[14] Norman Biggs, E. Keith Lloyd, and Robin J. Wilson. *Graph Theory, 1736-1936.* Clarendon Press, 1976.

[15] Burak Bilgin, Ender Özcan, and Emin Erkan Korkmaz. An experimental study on hyper-heuristics and exam timetabling. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 394–412. Springer, 2006.

[16] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3):268–308, 2003.

[17] Geoff Boeing. Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, 65:126–139, 2017.

[18] T Bogers. Movie recommendation using random walks over the contextual graph. *Proc. of the 2nd Intl. Workshop on Context-Aware*, 2010.

[19] Mohammad Reza Bonyadi, Zbigniew Michalewicz, and Luigi Barone. The travelling thief problem: The first step in the transition from theoretical problems to realistic problems. In *2013 IEEE Congress on Evolutionary Computation*, pages 1037–1044. IEEE, 2013.

[20] Anthony Brabazon and Michael O'Neill. Bond-issuer credit rating with grammatical evolution. In *Workshops on Applications of Evolutionary Computation*, pages 270–279. Springer, 2004.

[21] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[22] Csilla Bujtás, György Dósa, Csanád Imreh, Judit Nagy-György, and Zsolt Tuza. The graph-bin packing problem. *International Journal of Foundations of Computer Science*, 22(08):1971–1993, 2011.

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

[24] Edmund K Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R Woodward. A classification of hyper-heuristic approaches. In *Handbook of metaheuristics*, pages 449–468. Springer, 2010.

[25] Edmund K Burke, Matthew R Hyde, and Graham Kendall. Evolving bin packing heuristics with genetic programming. In *Parallel Problem Solving from Nature-PPSN IX*, pages 860–869. Springer, 2006.

[26] Edmund K Burke, Matthew R Hyde, and Graham Kendall. Grammatical evolution of local search heuristics. *IEEE Transactions on Evolutionary Computation*, 16(3):406–417, 2011.

[27] Edmund K Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In *Handbook of metaheuristics*, pages 457–474. Springer, 2003.

[28] Edmund K Burke, Barry McCollum, Amnon Meisels, Sanja Petrovic, and Rong Qu. A graph-based hyper-heuristic for educational timetabling problems. *European Journal of Operational Research*, 176(1):177–192, 2007.

[29] EK Burke, B McCollum, A Meisels, and S Petrovic. A graph-based hyper-heuristic for educational timetabling problems. *European Journal of*, 2007.

[30] Giulio Erberto Cantarella and Antonino Vitetta. The multi-criteria road network design problem in an urban area. *Transportation*, 33(6):567–588, 2006.

[31] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[32] Pai-Chun Chen, Graham Kendall, and Greet Vanden Berghe. An ant based hyper-heuristic for the travelling tournament problem. In *2007 IEEE Symposium on Computational Intelligence in Scheduling*, pages 19–26. IEEE, 2007.

[33] Mingchang Chih. Self-adaptive check and repair operator-based particle swarm optimization for the multidimensional knapsack problem. *Applied Soft Computing*, 26:378–389, 2015.

[34] Paul C Chu and John E Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of heuristics*, 4(1):63–86, 1998.

[35] Alberto Colorni, Marco Dorigo, Francesco Maffioli, Vittorio Maniezzo, GIOVANNI Righini, and Marco Trubian. Heuristics from nature for hard combinatorial optimization problems. *International Transactions in Operational Research*, 3(1):1–21, 1996.

[36] Bruno Contreras-Moreira. 3d-footprint: a database for the structural analysis of protein–dna complexes. *Nucleic acids research*, 38(suppl_1):D91–D97, 2010.

[37] Leon Cooper. Heuristic methods for location-allocation problems. *SIAM review*, 6(1):37–53, 1964.

[38] Gérard Cornuéjols. Combinatorial optimization: packing and covering. 2000.

[39] Peter Cowling, Graham Kendall, and Eric Soubeiga. A hyperheuristic approach to scheduling a sales summit. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 176–190. Springer, 2000.

[40] Georges A Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812, 1958.

[41] GB Dantzig. Discrete-variable extremum problems. *Operations research*, 1957.

[42] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[43] Jörg Denzinger and Marc Fuchs. High performance atp systems by combining several ai methods. IJCAI, 1996.

[44] Marco Dorigo and Mauro Birattari. *Ant colony optimization*. Springer, 2010.

[45] John H Drake, Nikolaos Kililis, and Ender Özcan. Generation of vns components with grammatical evolution for vehicle routing. In *European Conference on Genetic Programming*, pages 25–36. Springer, 2013.

[46] John H Drake, Ender Özcan, and Edmund K Burke. An improved choice function heuristic selection for cross domain heuristic search. In *International Conference on Parallel Problem Solving from Nature*, pages 307–316. Springer, 2012.

[47] John H Drake, Ender Özcan, and Edmund K Burke. A case study of controlling crossover in a selection hyper-heuristic framework using the multidimensional knapsack problem. *Evolutionary computation*, 24(1):113–141, 2016.

[48] Ben Dushnik and Edwin W Miller. Partially ordered sets. *American journal of mathematics*, 63(3):600–610, 1941.

[49] Stefan Edelkamp and Arend Rensink. Graph transformation and ai planning. *Knowledge Engineering Competition (ICKEPS), Rhode Island, USA*, 2007.

[50] H Ehrig, R Heckel, M Korff, and M Löwe. Algebraic approaches to graph transformation: Part II: Single pushout approach and comparison with double pushout approach. *Handbook of Graph*, 1997.

[51] H Ehrig, R Heckel, G Rozenberg, and G Taentzer. *Graph Transformations*. Springer, 2008.

[52] Hartmut Ehrig, Grzegorz Rozenberg, and Hans-J rg Kreowski. *Handbook of graph grammars and computing by graph transformation*, volume 3. world Scientific, 1999.

[53] OI Elgerd and HH Happ. Electric Energy Systems Theory: An Introduction. *IEEE Transactions on Systems, Man, and*, 1972.

[54] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. PonyGE2. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '17*, pages 1194–1201, New York, New York, USA, 3 2017. ACM Press.

[55] Tobias Friedrich and Frank Neumann. What's hot in evolutionary computation. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[56] A Gamst. Application of graph theoretical methods to GSM radio network planning. *Circuits and Systems, 1991., IEEE International*, 1991.

[57] P Garrido and C Castro. Stable solving of cvrps using hyperheuristics. *Proceedings of the 11th Annual conference on*, 2009.

[58] Gregor Gassner. *Discontinuous Galerkin methods for the unsteady compressible Navier-Stokes equations*. 2009.

[59] Herbert Gelernter. Realization of a geometry-theorem proving machine. *Computer and thought*, 1963.

[60] Ian P Gent and Toby Walsh. Csplib: a benchmark library for constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 480–481. Springer, 1999.

[61] Ranulph Glanville, David Griffiths, Philip Baron, John H Drake, Matthew Hyde, Khaled Ibrahim, and Ender Ozcan. A genetic programming hyper-heuristic for the multidimensional knapsack problem. *Kybernetes*, 2014.

[62] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[63] Melissa D. Goodman, Kathryn A. Dowsland, and Jonathan M. Thompson. A grasp-knapsack hybrid for a nurse-scheduling problem. *Journal of Heuristics*, 15(4):351–379, 11 2007.

[64] Leo John Grady et al. *Space-variant computer vision: A graph-theoretic approach*. PhD thesis, Boston University, 2004.

[65] Ronald L Graham. *Handbook of combinatorics*. Elsevier, 1995.

[66] Jonathan L. Gross, Jay Yellen, and Ping Zhang. *Handbook of Graph Theory, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2013.

[67] Martin Grötschel and Lászlo Lovász. Combinatorial optimization: A survey. 1993.

[68] Jun Gu, Paul W Purdom, John Franco, and Benjamin W Wah. Algorithms for the satisfiability (sat) problem: A survey. Technical report, Cincinnati Univ oh Dept of Electrical and Computer Engineering, 1996.

[69] Xianfeng Gu, Yalin Wang, Hsiao-Bing Cheng, Li-Tien Cheng, and Shing-Tung Yau. Geometric methods in engineering applications. In *Mathematics and Computation, a Contemporary View*, pages 1–19. Springer, 2008.

[70] J Haantjes and G Laman. On the definition of geometric objects. i. *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen: Series A: Mathematical Sciences*, 56(3):208–215, 1953.

[71] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *International Conference on Foundations of Software Science and Computation Structures*, pages 230–245. Springer, 2001.

[72] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[73] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.

[74] Pierre Hansen, Nenad Mladenović, and José A Moreno Pérez. Variable neighbourhood search: methods and applications. *Annals of Operations Research*, 175(1):367–407, 2010.

[75] E Hart and P Ross. A heuristic combination method for solving job-shop scheduling problems. *International Conference on Parallel Problem Solving*, 1998.

[76] Emma Hart and Kevin Sim. A hyper-heuristic ensemble method for static job-shop scheduling. *Evolutionary computation*, 24(4):609–635, 2016.

[77] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *International Conference on Graph Transformation*, pages 161–176. Springer, 2002.

[78] Erik Hemberg, Anthony Erb Lugo, Dennis Garcia, and Una-May O'Reilly. Grammatical evolution with coevolutionary algorithms in cyber security. In *Handbook of Grammatical Evolution*, pages 407–431. Springer, 2018.

[79] Juraj Hromkovič. *Algorithmics for hard problems: introduction to combinatorial optimization, randomization, approximation, and heuristics.* Springer Science & Business Media, 2013.

[80] J Hughes. Electrical Networks—A Graph Theoretical Approach. 2009.

[81] Shyh-In Hwang and Sheng-Tzong Cheng. Combinatorial optimization in real-time scheduling: theory and algorithms. *Journal of combinatorial optimization*, 5(3):345–375, 2001.

[82] Yaochu Jin. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61–70, 2011.

[83] David S Johnson. Combinatorial optimization: Algorithms and complexity. by christos h. papadimitriou and kenneth steiglitz. *The American Mathematical Monthly*, 91(3):209–211, 1984.

[84] Leonid V Kantorovich. The mathematical method of production planning and organization. *Management Science*, 6(4):363–422, 1939.

[85] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[86] Donald Ervin Knuth. *The Art of Computer Programming: Fundamental Algorithms. Fundamental Algorithms.* Addison-Wesley, 1997.

[87] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[88] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[89] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data Mining and Constraint Programming*, pages 149–190. Springer, 2016.

[90] Ahmed Kouider, Hacene Ait Haddadene, Samia Ourari, and Ammar Oulamara. Mixed integer linear programs and tabu search approach to solve mixed graph coloring for unit-time job shop scheduling. In *2015 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1177–1181. IEEE, 8 2015.

[91] John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[92] Ladislav Kvasz. History of geometry and the development of the form of its language. *Synthese*, 116(2):141–186, 1998.

[93] Eugene L Lawler. *Combinatorial optimization: networks and matroids.* Courier Corporation, 2001.

[94] Rhyd Lewis. *A guide to graph colouring: Algorithms and Applications(1st. ed.)*, volume 7. Springer, 2015.

[95] Chao Liang, Yong Liu, and Keith W Ross. Topology optimization in multi-tree based p2p streaming system. In *2009 21st IEEE International Conference on Tools with Artificial Intelligence*, pages 806–813. IEEE, 2009.

[96] Falk Lieder and Thomas L Griffiths. When to use which heuristic: A rational solution to the strategy selection problem. In *CogSci*, 2015.

[97] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[98] Thainá Mariani, Giovani Guizzo, Silvia R. Vergilio, and Aurora T.R. Pozo. Grammatical Evolution for the Multi-Objective Integration and Test Order Problem. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO '16*, pages 1069–1076, New York, New York, USA, 2016. ACM Press.

[99] Jiří Matoušek. *Lectures on discrete geometry*, volume 108. Springer, 2002.

[100] Julian F Miller, Dominic Job, and Vesselin K Vassilev. Principles in the evolutionary design of digital circuits—part i. *Genetic programming and evolvable machines*, 1(1-2):7–35, 2000.

[101] Melanie Mitchell. *An introduction to genetic algorithms.* MIT press, 1998.

[102] Jason H Moore and Lance W Hahn. An improved grammatical evolution strategy for hierarchical petri net modeling of complex genetic systems. In *Workshops on Applications of Evolutionary Computation*, pages 63–72. Springer, 2004.

[103] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[104] Nikolaos E. Myridis. Applications of combinatorial optimization, edited by vangelis th. paschos. *Contemporary Physics*, 53(2):178–179, 2012.

[105] Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in savile row. *Artificial Intelligence*, 251:35–61, 2017.

[106] Gabriela Ochoa, Matthew Hyde, Tim Curtois, Jose A. Vazquez-Rodriguez, James Walker, Michel Gendreau, Graham Kendall, Barry McCollum, Andrew J. Parkes, Sanja Petrovic, and Edmund K Burke. HyFlex: A Benchmark Framework for Cross-Domain Heuristic Search. In *European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 136–147. Springer Berlin Heidelberg, 2012.

[107] Gabriela Ochoa, James Walker, Matthew Hyde, and Tim Curtois. Adaptive evolutionary algorithms and extensions to the hyflex hyper-heuristic framework. In *International Conference on Parallel Problem Solving from Nature*, pages 418–427. Springer, 2012.

[108] Michael O'Neill, James McDermott, John Mark Swafford, Jonathan Byrne, Erik Hemberg, Anthony Brabazon, Elizabeth Shotton, Ciaran McNally, and Martin Hemberg. Evolutionary design using grammatical evolution and shape grammars: Designing a shelter. *International Journal of Design Engineering*, 3(1):4–24, 2010.

[109] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[110] Ender Ozcan, Yuri Bykov, Murat Birben, and Edmund K Burke. Examination timetabling using late acceptance hyper-heuristics. In *2009 IEEE Congress on Evolutionary Computation*, pages 997–1004. IEEE, 2009.

[111] Michael O'Neil and Conor Ryan. Grammatical Evolution. In *Grammatical Evolution*, pages 33–47. Springer US, Boston, MA, 2003.

[112] Michael O'Neill, Anthony Brabazon, Conor Ryan, and JJ Collins. Evolving market index trading rules using grammatical evolution. In *Workshops on Applications of Evolutionary Computation*, pages 343–352. Springer, 2001.

[113] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):339–363, 2010.

[114] János Pach. *Towards a theory of geometric graphs*. Number 342. American Mathematical Soc., 2004.

[115] János Pach. The beginnings of geometric graph theory. In *Erdős Centennial*, pages 465–484. Springer, 2013.

[116] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[117] Madhumangal Pal and Anita Pal. Scheduling algorithm to select optimal programme slots in television channels: A graph theoretic approach. *International Journal of Applied and Computational Mathematics*, 3(3):1931–1950, 2017.

[118] Christos H Papadimitriou. *Computational complexity.* John Wiley and Sons Ltd., 2003.

[119] Diego Perez-Liebana and Miguel Nicolau. Evolving behaviour tree structures using grammatical evolution. In *Handbook of Grammatical Evolution*, pages 433–460. Springer, 2018.

[120] Carsten Peterson. Artificial neural networks. *Modern heuristic techniques for combinatorial problems*, 1993.

[121] Nelishia Pillay. Evolving hyper-heuristics for the uncapacitated examination timetabling problem. *Journal of the Operational Research Society*, 63(1):47–58, 2012.

[122] Nelishia Pillay and Rong Qu. *Hyper-Heuristics: Theory and Applications.* Springer, 2018.

[123] Hilary A Priestley. Ordered sets and complete lattices. In *Algebraic and coalgebraic methods in the mathematics of program construction*, pages 21–78. Springer, 2002.

[124] Robert Clay Prim. Shortest connection networks and some generalizations. *Bell Labs Technical Journal*, 36(6):1389–1401, 1957.

[125] Jakob Puchinger, Günther R Raidl, and Ulrich Pferschy. The multidimensional knapsack problem: Structure and algorithms. *INFORMS Journal on Computing*, 22(2):250–265, 2010.

[126] Yanjun Qi. Random forest for bioinformatics. In *Ensemble machine learning*, pages 307–323. Springer, 2012.

[127] R Raghavjee and N Pillay. A genetic algorithm selection perturbative hyper-heuristic for solving the school timetabling problem. *ORiON*, 31(1):39–60, 2015.

[128] Samuel Reid. On contact numbers of finite lattice sphere packings and the maximal coordination of monatomic crystals. *arXiv preprint arXiv:1602.04246*, 2016.

[129] Gerhard Reinelt. TSPLIB—A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384, 11 1991.

[130] Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

[131] Shervin Roshanisefat, Harshith K Thirumala, Kris Gaj, Houman Homayoun, and Avesta Sasan. Benchmarking the capabilities and limitations of sat solvers in defeating obfuscation schemes. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pages 275–280. IEEE, 2018.

[132] P Ross, S Schulenburg, JG Marín-Blázquez, and E Hart. Hyper-heuristics: learning to combine simple heuristics in bin-packing problems. *GECCO*, 2002.

[133] Franz Rothlauf. *Representations for genetic and evolutionary algorithms*. Springer, 2006.

[134] Grzegorz Rozenberg. Handbook of graph grammars and computing by graph transformation: volume I. foundations. 2 1997.

[135] Conor Ryan and JJ Collins. *Handbook of Grammatical Evolution*. Springer, 2018.

[136] Conor Ryan, John James Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *European Conference on Genetic Programming*, pages 83–96. Springer, 1998.

[137] Nasser R. Sabar, Masri Ayob, Graham Kendall, and Rong Qu. Grammatical Evolution Hyper-Heuristic for Combinatorial Optimization Problems. *IEEE Transactions on Evolutionary Computation*, 17(6):840–861, 12 2013.

[138] Nasser R Sabar, Masri Ayob, Graham Kendall, and Rong Qu. A dynamic multiarmed bandit-gene expression programming hyper-heuristic for combinatorial optimization problems. *IEEE Transactions on Cybernetics*, 45(2):217–228, 2015.

[139] Nasser R Sabar, Masri Ayob, Rong Qu, and Graham Kendall. A graph coloring constructive hyper-heuristic for examination timetabling problems. *Applied Intelligence*, 37(1):1–11, 2012.

[140] Sarah E Salvioli et al. On the theory of geometric objects. *Journal of Differential Geometry*, 7(1-2):257–278, 1972.

[141] Hanamantagouda P Sankappanavar and Stanley Burris. A course in universal algebra. 1981.

[142] JW Schoonahd, John D Gould, and LA Miller. Studies of visual inspection. *Ergonomics*, 16(4):365–379, 1973.

[143] Jouko Seppänen and James M. Moore. Facilities Planning with Graph Theory. *Management Science*, 17(4):B–242–B–253, 12 1970.

[144] Yossi Shiloach. A minimum linear arrangement algorithm for undirected trees. *SIAM Journal on Computing*, 8(1):15–32, 1979.

[145] Kevin Sim. *Novel Hyper-heuristics Applied to the Domain of Bin Packing.* PhD thesis, Edinburgh Napier University, 2014. School: iidi Department: Institute for Informatics and Digital Innovation.

[146] Kevin Sim and Emma Hart. A combined generative and selective hyper-heuristic for the vehicle routing problem. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 1093–1100. ACM, 2016.

[147] Kevin Sim, Emma Hart, and Ben Paechter. Learning to solve bin packing problems with an immune inspired hyper-heuristic. In *Artificial Life Conference Proceedings 13*, pages 856–863. MIT Press, 2013.

[148] Kevin Sim, Emma Hart, and Ben Paechter. A Lifelong Learning Hyper-heuristic Method for Bin Packing. *Evolutionary Computation*, 23(1):37–67, 3 2015.

[149] D Singh, A Ibrahim, T Yohanna, and J Singh. An overview of the applications of multisets. *Novi Sad Journal of Mathematics*, 37(3):73–92, 2007.

[150] Jerry Swan, Ender Özcan, and Graham Kendall. Hyperion–a recursive hyper-heuristic framework. In *International Conference on Learning and Intelligent Optimization*, pages 616–630. Springer, 2011.

[151] John Tabak. *Geometry: the language of space and form.* Infobase Publishing, 2014.

[152] Ioannis Tsoulos, Dimitris Gavrilis, and Euripidis Glavas. Neural network construction and training using grammatical evolution. *Neurocomputing*, 72(1-3):269–277, 2008.

[153] Fang Tu, Krishna R. Pattipati, Somnath Deb, and Venkata Narayana Malepati. Computationally efficient algorithms for multiple fault diagnosis in large graph-based systems. *IEEE*

Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, 33(1):73–85, 2003.

[154] Enrique Urra, Claudio Cubillos, and Daniel Cabrera-Paniagua. A hyperheuristic for the dial-a-ride problem with time windows. *Mathematical Problems in Engineering*, 2015, 2015.

[155] D Van Dantzig. On the principles of intuitionistic and affirmative mathematics. *Indagationes Mathematicae*, 9:429–440, 1947.

[156] Peter Vanbekbergen, Gert Goossens, Francky Catthoor, and Hugo J De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. *IEEE transactions on computer-aided design of integrated circuits and systems*, 11(11):1426–1438, 1992.

[157] Stephan M. Wagner and Nikrouz Neshat. Assessing the vulnerability of supply chains using graph theory. *International Journal of Production Economics*, 126(1):121–129, 7 2010.

[158] Gary C White and Robert E Bennetts. Analysis of frequency count data using the negative binomial distribution. *Ecology*, 77(8):2549–2557, 1996.

[159] Laurence A Wolsey and George L Nemhauser. *Integer and combinatorial optimization.* John Wiley & Sons, 2014.

[160] Kamal Z Zamli, Fakhrud Din, Graham Kendall, and Bestoun S Ahmed. An experimental study of hyper-heuristic selection and acceptance mechanism for combinatorial t-way test suite generation. *Information Sciences*, 399:121–153, 2017.

[161] Min Zhang, Nabil Anwer, and Luc Mathieu. Discrete geometry for product specification and verification. *IDMME-Virtual Concept*, pages 2–4, 2010.