Noname manuscript No.
(will be inserted by the editor)

A Hyperheuristic Approach based on Low-Level Heuristics for the Travelling Thief Problem

Mohamed El Yafrani · Marcella Martins · Markus Wagner · Belaïd Ahiod · Myriam Delgado · Ricardo Lüders ·

Received: date / Accepted: date

Abstract In this paper, we investigate the use of hyper-heuristics for the Travelling Thief Problem (TTP). TTP is a multi-component problem, which means it has a composite structure. The problem is a combination between the Travelling Salesman Problem (TSP) and the Knapsack Problem (KP). Many heuristics were proposed to deal with the two components of the problem separately. In this work, we investigate the use of automatic online heuristic selection in order to find the best combination of the different known heuristics. In order to achieve this, we propose a genetic programming based hyper-heuristic called GPHS*, and compare it to state-of-the-art algorithms. The experimental results show that the approach is competitive with those algorithms on small and mid-sized TTP instances.

Keywords Heuristic selection · Genetic Programming · Travelling Thief Problem · Multi-component problems

Mohamed El Yafrani (✉) · Belaïd Ahiod
LRIT, associated unit to CNRST (URAC 29)
Mohammed V University in Rabat
B.P. 1014 Rabat, Morocco
E-mail: m.elyafrani@gmail.com, ahiod@fsr.ac.ma

Markus Wagner
Optimisation and Logistics
School of Computer Science
University of Adelaide, Australia
E-mail: markus.wagner@adelaide.edu.au

Marcella S. R. Martins · Myriam R. B. S. Delgado · Ricardo Lüders
Federal University of Technology - Paraná (UTFPR)
Av. Sete de Setembro, 3165. Curitiba PR, Brazil
E-mail: {marcella,myriamdelg,luders}@utfpr.com.br

1 Introduction

Over the years, a large variety of heuristic methods has been proposed to deal with combinatorial optimization problems. Non-deterministic search methods such as evolutionary algorithms, local search, and others metaheuristics offer an alternative approach to exhaustive search to achieve good solutions for difficult computational problems in a reasonable amount of time. These methods can generate good quality solutions, but there is no guarantee that an optimal solution will be produced.

It is important to highlight that these approaches still find difficulties in terms of adaptation to newly encountered problems, or even new instances of a similar problem. Additionally, these search strategies can be resource-intensive to implement and develop.

Hyper-heuristics aim to address some of these issues. In fact, hyper-heuristics are a trending class of high-level search techniques designed to automate the heuristic design process and raise the level of generality at which search methods operate [37]. Hyper-heuristics are search methodologies for selecting Low Level Heuristics (LLHs) or generating new heuristics combining and adapting components of heuristics, in order to solve a range of optimization problems. They operate on a search space of heuristics unlike traditional computational search methods which operate directly on a search space of solutions [16].

Genetic Programming (GP) [25] is one of the most commonly used approaches in the field of hyper-heuristics [6], and usually used as a heuristic generation approach. However, in this work we explore the use of GP from a heuristic selection perspective. Heuristic selection using GP has been previously investigated by Nguyen et al. [34] in his adaptive model called GPAM. Hunt et al. [22] proposed another heuristic selection approach for the feature selection problem.

Our proposed hyper-heuristic based on GP is evaluated on the Travelling Thief Problem (TTP), a relatively new NP-hard problem [3]. TTP is an optimization problem that provides interdependent sub-problems, which is a problem aspect often encountered in real-world applications.

Despite the fact that many heuristics were proposed to deal with the TTP components separately, most existing algorithms are based on local search heuristics and meta-heuristic adaptation. None of the proposed approaches investigate the use of the heuristic selection in order to find the best combination of the different heuristics, which is the main motivation of this paper.

Indeed, we believe that the composite structure of the problem makes it a good benchmark for heuristic selection, or more precisely low-level-heuristic selection. Therefore, in this paper we propose a heuristic selection framework based on genetic programming. The approach uses well known low-level-heuristics in order to evolve combinations of these heuristics aiming to find a good model for the instance at hand.

Through this work, we aim especially to answer the following question: how efficient a GP heuristic selection approach can be to find good quality solutions for the TTP considering a low computational budget? In order to answer it, we conduct an in-depth experimental study on a subset of the TTP library.

While GP is usually used in an off-line fashion, the proposed approach is based on online training where the model continuously learns from feedback. Our proposal relies on two aspects: The tree representation and the standard GP crossover. In order to show the effectiveness of using our GP representation, we conduct experiments comparing the proposed GP approach with an off-line GP and to what we consider as a standard GA. The obtained results and the statistical tests tend to confirm the efficiency of our online hyperheuristic.

To have more insight on the efficiency of the proposed approach, further experiments are performed against three state-of-the-art algorithms. The results show that the proposed approach performs well on many small and mid-size instances.

This paper is organized as follows. Section 2 presents some related work on using hyper-heuristics for combinatorial problems. The TTP and a brief history of TTP algorithms are introduced in Section 3. Section 4 describes our proposed approach. The experiments, results, and discussion are presented in Section 5. Finally, Section 6 concludes the paper and outlines some future directions.

2 On the use of hyper-heuristics in combinatorial optimization

This section provides a background and some related work in combinatorial optimization problems using hyper-heuristics. Two classes of hyper-heuristics are presented, and the use of genetic programming as a hyper-heuristic is briefly revisited.

2.1 Heuristic selection vs heuristic generation

A hyper-heuristic can be defined as a high-level heuristic (HLH) that controls low-level heuristics (LLH) [13]. Two main categories of hyper-heuristics can be distinguished according to Burke et al. [7]: heuristic selection methodologies and heuristic generation methodologies. Heuristic selection frameworks select a LLH to apply at a given point during the search process. The framework is provided with a set of pre-existing, generally widely known heuristics for solving the target problem. On the other hand, the objective of heuristic generation methodologies is to automatically design new heuristics using components of previously known heuristics [41]. Heuristic generation approaches mostly use a training phase, in which the model is evolved using a subset of problem instances, called the training set. These hyper-heuristics are classified as off-line learning hyper-heuristics.

Burke et al. [7] also define other categories related to the nature of the LLH heuristics used in the hyper-heuristic framework. In either case, the set of LLHs being selected or generated can be further split to distinguish between those which construct solutions from scratch (constructive) and those which modify an existing solution (perturbative) [16]. Furthermore, most of the hyper-heuristics approaches incorporate a learning mechanism to assist the selection of LLH during the solution process. Hyper-heuristics which apply online learning continuously adapt throughout the search process based on the feedback they receive. Hyper-heuristics using offline learning train the model on a subset of instances before being applied to another, frequently larger, set of unseen instances.

Some related work present good results for automating the design and adaption of heuristic methods with hyper-heuristics. Many meta-heuristics and machine learning techniques have been used in the context of heuristic selection.

Previous work describe hyper-heuristic frameworks that rank adaptively the LLHs based on a choice function [11, 12]. The choice function is the weighted sum of heuristic performance, joint performance of pairs of heuristics, and CPU time from the previous time the LLH was called. So the ranks are used to decide which heuristic to select in the next call. This method is simple and can be easily applied to new problem domains.

Another interesting method was developed by Krasnogor and Smith [26]. The approach shows how a simple inheritance mechanism is capable of learning what is the best local search heuristic to use at different stages of the search process. An individual is composed of its genetic material and its memetic material. The memetic material specifies the strategy the individual will use to apply local search in the vicinity of the solution encoded in its genetic part. This method was applied to solve different combinatorial optimization problems and showed very promising results.

Local search methods have also been used for heuristic selection such as the Tabu Search based hyper-heuristic [8]. The authors proposed a hyper-heuristic framework for timetabling and rostering in which heuristics compete using rules based on the principles of reinforcement learning. A tabu list is maintained in order to prevent certain heuristics from being chosen at certain times during the search. This framework was applied in two different problem domains and obtained good results on both.

In [38], the authors used a genetic algorithm as a hyper-heuristic that learns heuristic combinations for solving one-dimensional bin-packing problems. Another approach described in [13] proposed a very basic ACO based hyper-heuristic to evolve sequences of (at most) five heuristics for 2D packing. Both approaches provide competitive results.

In [15], the authors proposed a Simulated Annealing hyper-heuristic for the shipper rationalization problem, and the results showed that the approach was able to incorporate changes without the need for extensive experimentation to determine their impact on solution speed or quality.

2.2 Genetic Programming as a Hyper-heuristic

In recent years, Genetic Programming (GP) has been widely applied as a hyper-heuristic framework due its flexible structure, that can be used to build either construction or perturbation heuristics, with successful implementations in real world problems [23, 24].

One of the most common applications of GP as a hyper-heuristic is the automatic generation of heuristics. For example, Bölte and Thonemann [2] successfully adapted GP to learn new heuristics. The proposed method used standard GP to evolve annealing schedule functions in simulated annealing to solve the Quadratic Assignment Problem (QAP). Another example is the use of GP as a hyper-heuristic methodology to generate constructive heuristics in order to solve the Multidimensional 0-1 Knapsack Problem [16]. The results over a set of standard benchmarks showed that this approach leads to human competitive results. Furthermore, the work in [5] presented a GP system which automated the heuristic generation process producing human-competitive heuristics for the online bin packing problem. The framework evolved a control program that rates the suitability of each bin for the next piece, using the attributes of the pieces to be packed and the bins.

The authors in [32] also proposed a heuristic generation evolution with GP for the TTP. The approach consists of employing GP to evolve a gain and a picking function, respectively, in order to replace the original manually designed item selection heuristic in TSMA (Two-Stage Memetic Algorithm). The aim of this paper was to conduct a more systematic investigation on the item picking heuristic in TSMA.

The author in [21] proposed the Composite heuristic Learning Algorithm for SAT Search (CLASS). CLASS is a GP framework that discovers satisfiability testing (SAT) local search heuristics. The evolved heuristics were shown to be competitive compared to well-known SAT local search algorithms.

According to Nguyen et al. [34], GP can be also applied for heuristic selection methods for NP-hard combinatorial optimization problems. Nguyen et al. [34] investigated a GP based hyper-heuristic approach that evolves adaptive mechanisms called GPAM. The hyper-heuristic chooses from a set of LLH and constructs an adaptive mechanism, which is evolved simultaneously with the problem solution, providing an online learning system. Results showed that GPAM presented good quality solutions that performed competitively alongside existing hyper-heuristics for the MAX-SAT, bin packing and flow-shop scheduling problem domains.

Another GP-based heuristic selection framework was proposed in [22] for the feature selection problem. The proposed method evolves new heuristics using some basic components (building blocks). The evolved heuristics act as new search algorithms that can search the space of subsets of features.

A recent work presented in [9] introduced a hyper-heuristic with a GP search process that includes a self-tuning technique to dynamically update the crossover and mutation probabilities during a run of GP. The approach assigns different crossover and mutation probabilities to each candidate solution. In order to evaluate the performance of the proposed approach, the authors considered seven different symbolic regression test problems.

Although GP learning techniques are more often used for off-line approaches. In our proposal, which is based on LLH selection methodology as presented in [22], the framework can learn from feedback concerning heuristic performance throughout the search process. This approach is tested on TTP problem instances. TTP instances were also investigated by Mei et al. [32] from a hyper-heuristic perspective. However, the main difference between the work proposed in [32] and our approach is that we aim to investigate the LLHs combination instead of evolving the item selection function.

3 The Travelling Thief Problem (TTP)

The Travelling Thief Problem (TTP) is an artificially designed problem that combines the Travelling Salesman Problem and the Knapsack Problem. The motivation of designing such a problem is to provide a benchmark model closer to real optimization problems, which in many situations, are composed of multiples interacting sub-problems.

In this section, the TTP is presented and mathematically defined. Then, some of best performing algorithms are briefly introduced.

3.1 TTP and hyper-heuristics

The Travelling Thief Problem was first introduced by Bonyadi et al. [3] with the purpose of providing a benchmark for problems with multiple interdependent components. The problem was then simplified and reformulated in [35] and many benchmark instances were proposed in the same paper.

Since the problem has two components, many approaches focus on designing a heuristic for each part of the problem. We believe this presents a good testbed for heuristic selection in particular and hyper-heuristics in general. Indeed, many local search routines and disruptive operators have been proposed for the two components of the problem. Therefore, a heuristic selection approach would provide a way to automatically combine heuristics and mutation operators, in order to determine the sequence that they must be applied to assure the best achievable objective value.

A first attempt on using hyper-heuristics to solve the problem was proposed by Mei et al. [32] in an above mentioned paper. The approach consists of using genetic programming to evolve packing routines for the KP component of the problem.

A recent paper by Wagner et al. [44] presents a detailed study on algorithm selection for the TTP. The paper uses 21 algorithms for the TTP and all 9720 instances in order to create a portfolio of algorithms. This portfolio is used to determine what algorithm performs best for what instance.

3.2 Problem Definition

In the TTP, we are given a set of n cities, the associated matrix of distances d_{ij} between cities i and j, and a set of m items distributed among the n cities. Each item k is characterized by its profit p_k and weight w_k. A thief must visit all cities exactly once, stealing some items on the road, and return to the first city.

The total weight of the collected items must not exceed a specified capacity W. In addition, we consider a renting rate per time unit R that the thief must pay at the end of the travel, and the maximum and minimum velocities denoted v_{max} and v_{min} respectively. Each item is available in only one city, and we note $A_k \in 1, \ldots, n$ the availability vector. A_k contains the reference to the city that contains the item k.

Naturally, a TTP solution is coded in two parts. The first is the tour $x = (x_1, \ldots, x_n)$, a vector containing the ordered list of cities. The second is the picking plan $z = (z_1, \ldots, z_m)$, a binary vector representing the states of items (0 for packed, and 1 for unpacked).

To make the sub-problems mutually dependent, the speed of the thief changes according to the knapsack weight. Therefore, the thief's velocity at city x_i is defined in Equation 1.

$$v_{x_i} = v_{max} - C \times w_{x_i} \tag{1}$$

where $C = (v_{max} - v_{min})/W$ is a constant value, and w_{x_i} the weight of the knapsack at city x_i.

We note $g(z)$ the total value of all collected items and $f(x, z)$ the total travel time which are defined in Equations 2 and 3 respectively.

$$g(z) = \sum_m p_m \times z_m \tag{2}$$

$$\text{subject to} \sum_m w_m \times z_m \leq W$$

$$f(x, z) = \sum_{i=1}^{n-1} t_{x_i, x_{i+1}} + t_{x_n, x_1} \tag{3}$$

where $t_{x_i, x_{i+1}} = \frac{d_{x_i, x_{i+1}}}{v_{x_i}}$ is the travel time from x_i to x_{i+1}.

The objective is to maximize the travel gain, as defined in Equation 4, by finding the best tour and picking plan.

$$G(x, z) = g(z) - R \times f(x, z) \tag{4}$$

Note that it has been shown in [3, 33] that optimizing the sub-problems in isolation, even to optimality, does not guarantee finding good solutions to the overall problem.

3.3 A brief history of TTP algorithms

Since the appearance of the TTP, several heuristic algorithms were proposed to solve it. The first version of the problem was proposed by Bonyadi et al. [3], in which an item can appear in multiple cities. Mei et al. [33] investigated the interdependence in this first formulation of the TTP, and proposed two algorithms to solve it. The algorithms are a Cooperative Coevolution heuristic and Memetic Algorithm.

Afterwards, [35] proposed a simplified version of the problem where an item can occur only once in a city. The paper also presented a very large library for the TTP containing 9720 instances, and three simple heuristics to solve these instances. The heuristics are a constructive heuristic named Simple Heuristic (SH), a Random Local Search (RLS), and a (1+1) Evolutionary Algorithm (EA). EA and RLS were able to obtain a positive gain, but were lately surpassed by more sophisticated heuristics.

Mei et al. [31] introduced a memetic algorithm named MATLS able to solve very large instances. The algorithm's success is mainly due to the use of domain knowledge to speed up local search, and the efficient greedy packing routines. The algorithm is shown to be efficient for instances having more than 10000 cities. The same framework was later used to design a GP-based heuristic generation approach in order to improve the packing process [32].

The paper by Bonyadi et al. [4] also presented a framework called CoSolver. The idea is to handle the components of the TTP separately, but maintain a communication tunnel between the two sub-problems in order to take interdependence into consideration.

Faulkner et al. [20] implemented multiple local search and greedy algorithms. The algorithms are combined in different fashions in order to design more sophisticated approaches. The authors proposed two families of heuristics as a result of the combinations, simple heuristics (S1-S5) and complex ones (C1-C6). Surprisingly, the best performing heuristic is S5 which belongs to simple heuristics group.

Most proposed heuristics use a Lin-Kernighan tour to initialize the solution. In a tentative to explore this bias in heuristic design for the TTP, Wagner [43] investigated longer tours using the Max-Min Ant System. This approach focuses on improving the tour accordingly to the overall TTP problem instead of using a Lin-Kernighan tour, which is designed for the TSP component independently. The approach is shown to be very efficient for small TTP instances.

Recently, El Yafrani and Ahiod [19] presented two heuristics. The first is a memetic algorithm using 2-opt and bit-flip local search heuristics, named MA2B. The second is a combination of a 2-opt local search and a simulated annealing based heuristic for efficient packing, named CS2SA. The two proposed heuristics were shown to be very competitive to other heuristics such as MATLS and S5. MA2B particularly performs well on small instances, while CS2SA was more efficient on large instances.

Wu et al. [45] investigated the impact of the renting rate on a special case of the TTP in which the tour is supposed fixed called the Nonlinear Knapsack

Problem. The paper presents an in-depth theoretical study of the renting rate and its effect on the hardness of a given instance. The authors also proposed an approach to generate hard instances based on theoretical and experimental study.

Finally, the authors in [18] focus on designing a TTP specific neighborhood instead of using a sequential structure as in most heuristics. The paper presented the use of speedups in the context of the proposed neighborhoods. The results show that this approach was competitive to EA and RLS on different small instances. However, the algorithms lacked of exploration capabilities as they are mainly designed for exploitation purposes.

4 The proposed approach

In our proposed approach, the goal is to apply GP as a heuristic selection technique, aiming to evolve combinations of heuristics in order to find good problem solutions. In this section we explain how this strategy can be implemented detailing the proposed algorithm.

The motivation behind choosing GP in the context of heuristic selection is mainly due to the fact that GP preserves the correlation between the terminals in sub-trees. The correlations are transferred to the offspring to produce new trees using mainly crossover.

In this section, the details of our GP adaptation are presented.

4.1 Representation

In the GP population, every individual is encoded as a tree that represents the program to be executed. The tree's internal nodes are functions, while leaf nodes are terminals, or LLHs.

We use two types of functions:

- **Connectors:** which are used to sequentially execute the child sub-tree from left to right. In our implementation, we use four types of connectors, which we refer to as con_N, such as $N \in \{1, \dots, 4\}$. The first has two child sub-trees, while the second has three.
- **If nodes:** which represent acceptance functions. We use the following three if nodes, each having two child sub-trees:
  - **if_improvement:** runs the left sub-tree if the current solution improves the former one, runs the right sub-tree otherwise.
  - **if_local_optimum:** runs the left sub-tree if the current solution does not improve the former one; runs the right sub-tree otherwise.
  - **if_no_improvement:** runs the left sub-tree if there is no improvement for 20 consecutive iterations; runs the right sub-tree otherwise.

This is rather a simple adaptation compared to other possibilities like using more conditional statements and loops. Nevertheless, these connectors allow representing different combinations of LLHs heuristics.

On the other hand, we use a set of several terminals. The terminals are either a component heuristic or a disruptive operation. In the following, we present a list of the used terminals:

- **term_kpbf:** A neighborhood search heuristic targeting the KP part. This heuristic uses a simple bit-flip hill climbing algorithm empowered with speedup techniques. It is part of the memetic algorithm MATLS proposed in [31].
- **term_kpsa:** A simulated annealing adapted for the KP sub-problem. It is used in CS2SA presented in [19]. The heuristic uses a random neighbor generator using the bit-flip operator. For each fixed picking plan, multiple random neighbors are generated and evaluated.
- **term_tsp2opt:** A 2-opt based local search heuristic used for the TSP component. It is used in many TTP algorithms [17, 19, 32, 33]. This search heuristic generates an entire set of tours based on the 2-opt operator. Additionally, the Delaunay triangulation is used within the 2-opt local search to generate candidate solutions while reducing the time complexity [14]. Indeed, in a Delaunay graph, each city has only 6 surrounding cities on average. Thus, the time complexity of the 2-opt local search heuristic becomes linear.
- **term_otspswap:** A disruptive move for the TSP sub-problem that randomly swaps two cities.
- **term_otsp4opt:** A double bridge move for the TSP sub-problem that randomly selects the cities. The double bridge is a 4-opt move used as a disruptive operator.
- **term_okpbf$N$:** A disruptive routine that toggles the state of $N\%$ of the picking plan items. The item is chosen randomly, and the level of disruption depends on N.

The last three disruptive terminals were provided in order to enlarge the search space, while the other terminals are components of state-of-the-art TTP algorithms.

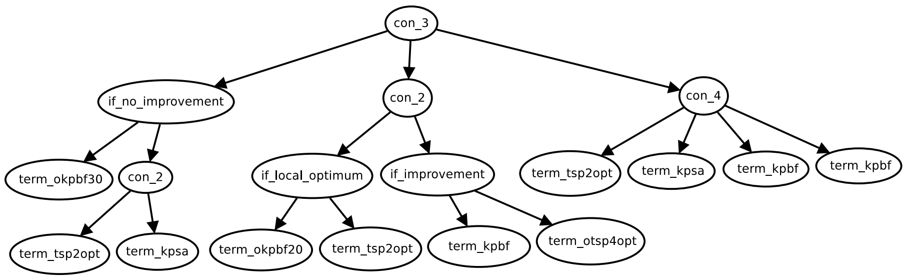

Fig. 1: Example of a GP individual. The terminals are TTP heuristics, while the internal nodes represent connectors.

An example of a tree individual can be seen in Figure 1. Once the leaves of the tree are executed based on an pre-order parse of the entire tree, this example produces the sequence: [*term_okpbf30 term_okpbf30 term_okpbf30 term_kpbf term_tsp2opt term_kpsa term_tsp2opt*], a GP individual, which will have its fitness assessed.

4.2 The fitness function

The fitness of a GP individual M, denoted $fitness_{GP}(M)$, depends on its performance on the given instance. First, the tree is parsed in-order to get the list of leaf nodes. These are then applied sequentially on the problem instance, starting from an initial TTP solution. The fitness of the individual is set to the achieved TTP objective, according to the Equation 4, when the processing of the list of leaf nodes is completed. Therefore, the fitness of a GP model M is given in Equation 5.

$$fitness_{GP}(M) = G(x*, z*) \tag{5}$$

such that $(x*, z*)$ is the TTP solution obtained by the model. Note that since most of the used LLHs have a stochastic behavior, the GP fitness is not deterministic as it depends on the objective value of the TTP solution.

The initial TTP solution is created using the Lin-Kernighan heuristic (for the tour) [28] and the PackIterative routine (for the picking plan) [20]. This TTP solution is used as the input of the first LLH in the GP sequence, the LLH is then executed, and a new TTP solution is produced. This process is repeated for each LLH in the sequence provided by the tree traversal. The output of each LLH serves as the input of the next LLH. Finally, the GP fitness is obtained from the quality of the low level solution provided by LLHs.

4.3 The framework

The main steps performed by our approach are described in Algorithm 1. The proposed algorithm makes use of specific strategies presented in the next subsections.

4.3.1 Initialization

The Initialization process loads the problem instance, sets the GP parameters GP_{param} and generates an initial population of random compositions of functions and terminals of the problem, Steps 1, 2 and 3, respectively.

There are different approaches to generate the random initial population [27]. In this work, we use the *Full Initialization* method [25] in order to provide full trees and to increase the initial search space.

In this method, the initial individuals are generated according to a given maximum depth. Nodes are taken at random from the function set until the

Algorithm 1 GP framework

```
INPUT: $Instance$: problem instance
       $GP_{param}$: set of GP parameters
       $N$: population size
       $N_{\text{offspring}}$: number of offspring individuals
       $Max_{runtime}$: maximum runtime
       $Max_{ger}$: maximum number of generations
OUTPUT: $T$: the final individual tree
    {Initialization}
 1: $\mathbf{I} \leftarrow$ LoadInstance($Instance$)
 2: $\mathbf{P} \leftarrow$ SetParams($GP_{param}$)
 3: $Pop^1 \leftarrow$ RandomGenerate($N, I, P$) {initial population}
 4: $g \leftarrow 1$;
    {GP: main loop}
 5: while $g \leq Max_{ger}$ and $\neg Max_{runtime}$ do
 6:     $Pop^g$.fitness $\leftarrow$EvaluateFitness($Pop^g, I, P$)
 7:     $Pop_{\text{offspring}}$ $\leftarrow$GeneticOperator($Pop^g, P, N_{\text{offspring}}$); {offspring population}
 8:     $Pop_{\text{offspring}}$.fitness $\leftarrow$EvaluateFitness($Pop_{\text{offspring}}, I, P$)
        {GP: Survival}
 9:     $Pop^{g+1} \leftarrow$ Survival($\{Pop^g \cup Pop_{\text{offspring}}\}, N$);{new population}
10:     $g \leftarrow g + 1$
11: end while
    {GP: best individual}
12: $T_{best} \leftarrow$ SelectBest($Pop^{g-1}$)
```

maximum tree depth is reached, and the last depth level is limited to terminal nodes. As a result, trees initialized with this method will be balanced with the same length for all individuals of the initial population. This method prevents small shapes to be generated, and enlarges the genetic operations possibilities.

4.3.2 Main framework

The proposed GP algorithm searches the space of possible heuristic combinations. In the context of our implementation, each GP program is a tree of heuristics. The program is executed in an infix manner to search the space of possible problem solutions in order to find good ones.

The *EvaluateFitness*, in Step 6, calculates $fitness_{GP}$ based on the TTP objective function. In Step 7 *GeneticOperator* applies genetically inspired operations of mutation and crossover in selected parent individuals, in order to produce a new population of programs $Pop_{\text{offspring}}$. The selection of these parent individuals is probabilistically based on fitness. That is, better individuals are more likely to have more child programs than inferior individuals.

The most commonly employed method for selecting these parents in GP is tournament selection [25], which is used in our framework. This method selects a random number of individuals from the population and the best of them is chosen[1].

[1] We apply the Lexicographic Parsimony Pressure technique [30]. If two individuals are equally fit, the tree with less nodes is chosen as the best. This technique has shown to effectively control bloat in different types of problems [40].

The genetic operators adopted for the selected parents are standard tree crossover and standard mutation. Given two parents, standard crossover randomly selects a crossover point in each parent tree. Then, it creates the offspring by replacing the sub-tree rooted at the crossover point in a copy of the first parent with a copy of the sub-tree rooted at the crossover point in the second parent [27]. In standard tree mutation, a randomly created new tree replaces a randomly chosen branch (excluding the root node) of the parent tree [40].

This offspring population $Pop_{\text{offspring}}$ of $N_{\text{offspring}}$ programs is evaluated using the fitness function and merged with the current population Pop^g, where g is the generation number. However, N individuals are selected in *Survival* process to continue in the evolutionary process as a new population Pop^{g+1}, as can be seen in Step 9. The best individual from both Pop^g and $Pop_{\text{offspring}}$ is kept in the new population Pop^{g+1}. The fittest individuals from $Pop_{\text{offspring}}$ followed by the fittest ones from Pop^g compose the remaining $N-1$ individuals from Pop^{g+1}.

This process is iteratively performed until the termination criterion (Max_{ger} or $Max_{runtime}$) has been satisfied, whichever comes first. At the end the best individual T_{best} is selected from the population, according *SelectBest* function in Step 12.

4.4 Variants

Two variants of our approach were implemented. The first is an off-line hyperheuristic that uses one instance for the training phase in order to generate a model. The obtained model can be used on other unseen instance. This approach is mainly based on the adaptive system introduced by Nguyen et al. [34], and will be referred to as GPHS.

The second approach is an online implementation that tackles exclusively one instance at a time. Since there is no point in using the conditional node in an online framework, this implementation uses only the *con_2* connector. Note that a different sequence is evolved for each problem instance as the initialization process generates an initial population of random compositions of functions and terminals of the problem according to the instance being tackled. In the rest of this manuscript, we will refer to this approach as GPHS*.

5 Experiments and discussion

5.1 Benchmark Instances

The experiments conducted in this paper are performed on a comprehensive subset of the TTP benchmark instances [2] from [35]. The characteristics of these

[2] All TTP instances can be found in the website: `http://cs.adelaide.edu.au/~optlog/research/ttp.php`.

instances vary widely, and in this work we consider the following diversification parameters:

- The number of cities is based on TSP instances from the TSPlib, described in [36]
- For each TSP instance, there are three different types (we will refer to this parameter as T) of knapsack problems: uncorrelated (unc), uncorrelated with similar weights (usw), and bounded strongly correlated (bsc) types
- For each TSP and KP combination, the number of items per city (item factor, denoted F) is $F \in \{1, 5, 10\}$
- For each TTP configuration, we use 3 different knapsack capacities $C \in \{1, 5, 10\}$. C represents a capacity class.

To evaluate the proposed hyper-heuristic, we use seven representative small and mid-sized TTP instance groups: *eil51*, *berlin52*, *eil76*, *kroA100*, *a280*, *pr439* and *rat783*. Therefore, using this setting, a total of 189 instances are considered in this paper. While though significantly larger TTP instances exist, we are covering 59% of the 81 TTP instance sizes (when measured in the number of cities) with this subset. The global optima for instances with less than 1000 cities are not known yet. Therefore, we believe the focus on this subset is an acceptable limitation once large performance gains are still possible there. In addition, we still do not know what the optimal solutions are even for the tiniest instances. Moreover, we use the following instance naming convention: $instance\_group(F, T, C)$. For instance, $eil51(01, bsc, 10)$ represents the TTP instance having $eil51$ as the TSP base instance, with a KP bounded strongly correlated, a knapsack capacity of 10, and 1 item per city.

5.2 GP tuning

The hyper-heuristic framework is developed based on GPLAB [39], a GP toolbox for MATLAB. The GP parameters we used for the experiments are shown in Table 1.

Table 1: Parameters of GPHS* algorithm.

Description	GPHS*	GPHS
Population size N	10	20
Maximum runtime $Max_{runtime}$	$600s$	$-$
Maximum number of generations Max_{ger}	500	1000
Crossover Rate	0.9	0.9
Mutation Rate	0.1	0.1
Reproduction Rate	0.1	0.1
Max Depth	7	7
Min Depth	2	2
Tournament size	3	3
Number of offspring individuals $Pop_{\text{offspring}}$	N	N

Some of these parameters were empirically tuned, as Crossover and Mutation Rate, and the other are off-line tuned using I/F-Race [1], which is a state-of-the-art automatic configuration method. We use the implementation of I/F-Race provided by the *irace* package [29], which is implemented in R and is based on the iterated racing procedure. Since we are training GPHS for a given instance, we used relatively larger values for N and Max_{ger} and no maximum runtime.

Additionally, we use the *Full Initialization* technique alongside a minimum depth threshold of 2 in order to encourage the use of deeper initial trees. The reproduction (replication) rate is 0.1, meaning that each selected parent has a 10% chance of being copied to the next generation without suffering the action of the genetic operators. Standard tree mutation and standard crossover (with uniform selection of crossover and mutation points among different tree levels) were used with probabilities of 0.1 and 0.9, respectively. Also, a maximum tree depth *Max Depth=7* is imposed to any new branch created in order to avoid bloat.

A different tree traversal can also provide different orders, and this can directly affect its fitness, once different sequences can be created. In our experiments we applied a pre-established order (parsing the tree from left to right), but any other mode can be done or even included as a parameter in the search. Regarding this, an offspring tree can inherit its parsing mode from its parents, once the same individual could be evaluated with different tree traversals.

All algorithms are performed on a core i3-2370M CPU 2.40GHz machine with 4 GB of RAM, running Linux, for a maximum of 10 minutes per instance. In addition, 10 independent executions are conducted for each algorithm. The obtained results are then used to perform statistical tests in order to compare the performance of GPHS* to the other algorithms.

5.3 GP-based heuristic selection vs GA

In order to show the difference in performance between the proposed approach and a standard GA, we developed a basic GA framework based on one-point crossover. Aiming to make a fair comparison, all the parameters used in the GP-based framework are reconsidered in the GA. Such parameters include the population size, selection operator, stopping criteria, crossover rate, and minimum and maximum solution size. The minimum and maximum depth considered in the GP tree are 2 and 7 respectively. Therefore, the equivalent minimum chromosome size is $2^1 = 2$ and the maximum size correspond to $2^6 = 64$.

Afterwards, the two approaches are compared using the average (AVG) and relative standard deviation (RSD) as shown in Table 2. Additionally, the A-test is also used in order to gain further insight on the performance of the two approaches, as presented in Table 3. The entries representing when GP-based approach is showing a better performance than GA are highlighted in blue.

Table 2: Average (AVG) and relative standard deviation (RSD) for GP-base variants and GA

	GPHS*		GPHS		GA	
	AVG	RSD	AVG	RSD	AVG	RSD
eil51(01,bsc,50)	4079	2.13	3732	6.33	3811	12.34
berlin52(01,bsc,51)	4062	1.31	3654	7.68	4056	2.3
eil76(01,bsc,75)	3718	1.9	3201	7.16	3329	7.57
kroA100(01,bsc,99)	4454	1	4174	4.73	4142	11.66
a280(01,bsc,279)	18359	0.38	16567	4.21	17301	3.95
pr439(01,bsc,438)	36548	0.1	30518	6.17	30414	10.86
rat783(01,bsc,782)	41073	3.1	36999	2.59	31359	17.37

Table 3: A-test results between GPHS* ,GPHSand GA

	GPHS* x GPHS	GPHS* x GA	GPHS x GA
eil51(01,bsc,50)	0.94	0.61	0.3
berlin52(01,bsc,51)	0.98	0.65	0.04
eil76(01,bsc,75)	0.99	1	0.34
kroA100(01,bsc,99)	1	0.75	0.375
a280(01,bsc,279)	1	1	0.22
pr439(01,bsc,438)	1	0.985	0.51
rat783(01,bsc,782)	0.97	0.99	0.85

The reported results, from Table 3, show clearly that the GPHS* outperforms the off-line version GPHS and the standard GA on all the considered instances. However GPHS presents better results than standard GA for mid-sized instances, i.e. $pr439(438, bsc, 01)$ and $rat783(782, bsc, 01)$. Due to these results we define GPHS* as our standard version and it will be compared with other traditional approaches.

5.4 Comparison with state-of-the-art algorithms

Comparing different optimization techniques experimentally involves the notion of performance. Herein, we present a comparison between our GPHS* approach and three other algorithms for the TTP: MA2B [19], MATLS [31], and S5 [20]. To the best of our knowledge, few algorithms are efficient for the TTP, and the three algorithms above-mentioned are among the best performing ones for the considered instances. For a comprehensive comparison of 21 algorithms across all 9720 instances, we refer the interested reader to [44].

To measure the quality of the approaches, we consider for each algorithm the average objective score of 10 independent runs, and the best objective score found by any algorithm. The ratio between the average and the best objective values found gives us the approximation ratio. According to Wagner

[43], this ratio allows us to compare the performance across the chosen set of instances, since the objective values vary across several orders of magnitude.

In Figure 2, we show a summary of over 756 (189 instances and 4 algorithms) average approximation ratios as trend lines. The curves are polynomials of degree six, as considered in [43] showing a general trend qualitatively, but not necessarily with high accuracy. We can observe in Figure 2 that our GPHS* approach outperforms all the three algorithms for all TTP instances with more than 100 and less than 783 cities. For the instances having less than 100 cities, GPHS* is competitive with MATLS and S5. However, for these instances, MA2B still outperforms all the other approaches. On the other hand, For the instances with more than 783 cities, S5 provides the best average approximation ratio for almost all cases.

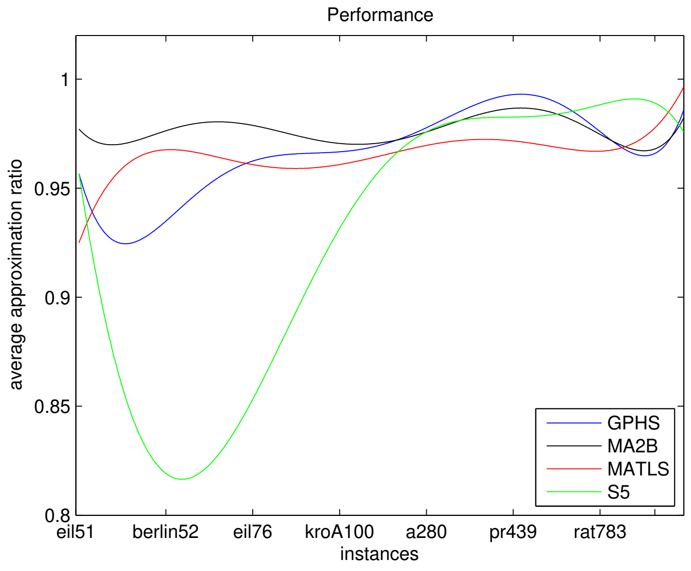

Fig. 2: Performance comparison - Summary of results shown as trend lines. The $x-$axis represents the 189 instances: 27 of *eil51*; 27 of *berlin52*; 27 of *eil76*; 27 of *kroA100*; 27 of *a280*; 27 of *pr439*; and 27 of *rat783*.

A deeper analysis can show that individual approaches do not strictly follow the trends. Figures 3 to 9 in Appendix A represent the average approximation ratios achieved in 10 independent runs for each instance group.

In order to provide a statistical analysis of the results, the A-test (Vargha-Delaney A measure [42]) was used. This test tells us how often, on average, one technique outperforms the other. Its a non-parametric test called the measure of stochastic superiority.

The A-test test returns a value between 0 and 1, representing the probability that a randomly selected observation from one sample is bigger than a randomly selected observation from other sample. Therefore it provides how much the two samples overlap. The two samples are composed by objective values from each algorithm run. Then each sample has 10 runs.

Tables 4 – 10 show the pairwise comparison between these algorithms for each instance, respectively.

When the A-measure is exactly 0.5, there is no statistical difference between the two techniques. When the A-measure is less than 0.5, the first technique has the worse performance. Lastly, when the A-measure is greater than 0.5, the second technique is the worst performing one.

The entries representing when GPHS* is showing a better performance than another approach are highlighted.

Table 4: A-test over *eil51* instances.

Algorithm	GPHS* x MA2B	GPHS* x MATLS	GPHS* x S5
$eil51(01, bsc, 01)$	0.4900	1.0000	1.0000
$eil51(05, bsc, 01)$	0.3000	0.7600	1.0000
$eil51(10, bsc, 01)$	0.0000	1.0000	0.9000
$eil51(01, bsc, 05)$	0.5900	0.5100	1.0000
$eil51(05, bsc, 05)$	0.0650	0.9100	1.0000
$eil51(10, bsc, 05)$	0.0000	1.0000	0.0000
$eil51(01, bsc, 10)$	0.7000	0.7500	1.0000
$eil51(05, bsc, 10)$	0.1200	0.2800	1.0000
$eil51(10, bsc, 10)$	0.0000	1.0000	1.0000
$eil51(01, usw, 01)$	0.0000	0.0000	1.0000
$eil51(05, usw, 01)$	0.4000	0.1600	1.0000
$eil51(10, usw, 01)$	0.9000	0.5000	1.0000
$eil51(01, usw, 05)$	0.0500	0.0000	0.6000
$eil51(05, usw, 05)$	0.0000	0.0800	1.0000
$eil51(10, usw, 05)$	0.0000	0.0200	1.0000
$eil51(01, usw, 10)$	0.0000	0.1400	0.2000
$eil51(05, usw, 10)$	0.1000	0.2000	1.0000
$eil51(10, usw, 10)$	0.0000	0.0000	1.0000
$eil51(01, unc, 01)$	0.0000	0.0000	1.0000
$eil51(05, unc, 01)$	0.0000	0.0200	1.0000
$eil51(10, unc, 01)$	0.0000	0.0000	1.0000
$eil51(01, unc, 05)$	0.0000	0.2400	1.0000
$eil51(05, unc, 05)$	0.0000	0.0600	1.0000
$eil51(10, unc, 05)$	0.0000	0.0600	1.0000
$eil51(01, unc, 10)$	0.3200	1.0000	1.0000
$eil51(05, unc, 10)$	0.2700	0.1600	1.0000
$eil51(10, unc, 10)$	0.0000	0.0000	1.0000

We can observe in Table 4 that MA2B and MATLS show the best results for *eil51* instances. However, GPHS* is better than S5 for almost all instances. The same observations can be made in Table 5.

We can note in Table 6 that GPHS* is better than S5 for most *eil76* instances, and more competitive with MATLS. Additionally, GPHS* was able to outperform MA2B for some instances.

The A-measures reported in Table 7 show that GPHS* is better than S5 for most *kroA100* instances and better or at least similar to MATLS. We can also observe that GPHS* presents good results when compared with MA2B.

Table 5: A-test over *berlin52* instances.

Algorithm	GPHS* x MA2B	GPHS* x MATLS	GPHS* x S5
$berlin52(01, bsc, 01)$	0.3200	0.0000	1.0000
$berlin52(05, bsc, 01)$	0.0000	0.2000	0.1000
$berlin52(10, bsc, 01)$	0.0000	0.0000	1.0000
$berlin52(01, bsc, 05)$	0.0000	0.0000	1.0000
$berlin52(05, bsc, 05)$	0.4800	0.0000	1.0000
$berlin52(10, bsc, 05)$	0.3500	0.1200	1.0000
$berlin52(01, bsc, 10)$	0.0000	1.0000	1.0000
$berlin52(05, bsc, 10)$	0.2600	0.9500	1.0000
$berlin52(10, bsc, 10)$	0.2900	1.0000	1.0000
$berlin52(01, usw, 01)$	0.4000	0.0600	1.0000
$berlin52(05, usw, 01)$	0.0000	0.0000	1.0000
$berlin52(10, usw, 01)$	0.0000	0.0000	1.0000
$berlin52(01, usw, 05)$	0.6800	1.0000	1.0000
$berlin52(05, usw, 05)$	0.6700	0.4900	1.0000
$berlin52(10, usw, 05)$	0.6900	0.0400	1.0000
$berlin52(01, usw, 10)$	0.0750	0.0200	1.0000
$berlin52(05, usw, 10)$	0.2300	0.1000	1.0000
$berlin52(10, usw, 10)$	0.5300	0.1400	1.0000
$berlin52(01, unc, 01)$	0.3900	0.5000	1.0000
$berlin52(05, unc, 01)$	0.1200	0.5000	1.0000
$berlin52(10, unc, 01)$	0.4950	0.1000	1.0000
$berlin52(01, unc, 05)$	0.5000	1.0000	1.0000
$berlin52(05, unc, 05)$	0.0900	0.0000	1.0000
$berlin52(10, unc, 05)$	0.6600	0.1400	1.0000
$berlin52(01, unc, 10)$	0.1200	1.0000	1.0000
$berlin52(05, unc, 10)$	0.6000	0.2000	1.0000
$berlin52(10, unc, 10)$	0.7000	0.0000	1.0000

Table 6: A-test over *eil76* instances.

Algorithm	GPHS* x MA2B	GPHS* x MATLS	GPHS* x S5
$eil76(01, bsc, 01)$	0.0900	0.4500	0.9000
$eil76(05, bsc, 01)$	0.9000	1.0000	0.9000
$eil76(10, bsc, 01)$	0.6000	1.0000	1.0000
$eil76(01, bsc, 05)$	0.0000	0.0000	0.0000
$eil76(05, bsc, 05)$	0.1600	0.6600	0.6500
$eil76(10, bsc, 05)$	0.0400	1.0000	0.9000
$eil76(01, bsc, 10)$	0.0000	0.3200	0.9200
$eil76(05, bsc, 10)$	0.8750	0.9500	0.9700
$eil76(10, bsc, 10)$	0.1600	0.8600	0.9800
$eil76(01, usw, 01)$	0.0900	0.0400	0.9800
$eil76(05, usw, 01)$	0.0000	0.0000	1.0000
$eil76(10, usw, 01)$	0.0000	0.0000	1.0000
$eil76(01, usw, 05)$	0.0000	0.0000	0.6000
$eil76(05, usw, 05)$	0.4500	0.0000	0.9700
$eil76(10, usw, 05)$	0.0550	0.0000	0.9600
$eil76(01, usw, 10)$	0.1150	0.4900	1.0000
$eil76(05, usw, 10)$	0.3700	0.2800	0.4000
$eil76(10, usw, 10)$	0.4500	0.9000	1.0000
$eil76(01, unc, 01)$	0.0900	0.9800	1.0000
$eil76(05, unc, 01)$	0.2000	0.8000	1.0000
$eil76(10, unc, 01)$	0.5200	1.0000	1.0000
$eil76(01, unc, 05)$	0.1000	0.9000	1.0000
$eil76(05, unc, 05)$	0.6000	0.6000	1.0000
$eil76(10, unc, 05)$	0.0000	0.0000	1.0000
$eil76(01, unc, 10)$	0.4300	0.9000	1.0000
$eil76(05, unc, 10)$	0.5050	0.5000	1.0000
$eil76(10, unc, 10)$	0.2700	0.1000	1.0000

Table 7: A-test over *kroA100* instances.

Algorithm	GPHS* x MA2B	GPHS* x MATLS	GPHS* x S5
$kroA100(01, bsc, 01)$	0.1700	0.0100	1.0000
$kroA100(05, bsc, 01)$	0.7000	1.0000	1.0000
$kroA100(10, bsc, 01)$	0.3700	1.0000	0.4000
$kroA100(01, bsc, 05)$	0.4500	0.9000	1.0000
$kroA100(05, bsc, 05)$	0.2650	1.0000	1.0000
$kroA100(10, bsc, 05)$	0.0750	1.0000	1.0000
$kroA100(01, bsc, 10)$	0.8000	0.7700	1.0000
$kroA100(05, bsc, 10)$	0.5500	1.0000	1.0000
$kroA100(10, bsc, 10)$	0.5000	1.0000	1.0000
$kroA100(01, usw, 01)$	0.4050	0.0800	1.0000
$kroA100(05, usw, 01)$	0.1650	0.0000	1.0000
$kroA100(10, usw, 01)$	0.1400	0.0300	1.0000
$kroA100(01, usw, 05)$	0.3600	0.3900	1.0000
$kroA100(05, usw, 05)$	0.7900	0.2800	1.0000
$kroA100(10, usw, 05)$	1.0000	0.4700	1.0000
$kroA100(01, usw, 10)$	0.5700	0.3900	1.0000
$kroA100(05, usw, 10)$	0.5000	0.1000	1.0000
$kroA100(10, usw, 10)$	0.7300	0.6600	1.0000
$kroA100(01, unc, 01)$	0.0000	1.0000	0.4000
$kroA100(05, unc, 01)$	0.1000	0.7300	0.1000
$kroA100(10, unc, 01)$	0.1500	1.0000	1.0000
$kroA100(01, unc, 05)$	0.5900	0.5500	0.9000
$kroA100(05, unc, 05)$	0.0000	1.0000	1.0000
$kroA100(10, unc, 05)$	0.9000	1.0000	1.0000
$kroA100(01, unc, 10)$	0.6200	0.6900	1.0000
$kroA100(05, unc, 10)$	0.0000	1.0000	1.0000
$kroA100(10, unc, 10)$	0.1000	1.0000	1.0000

Table 8: A-test over *a280* instances.

Algorithm	GPHS* x MA2B	GPHS* x MATLS	GPHS* x S5
$a280(01, bsc, 01)$	1.0000	1.0000	0.6500
$a280(05, bsc, 01)$	1.0000	1.0000	0.5000
$a280(10, bsc, 01)$	0.8100	1.0000	0.1100
$a280(01, bsc, 05)$	0.8400	0.8000	0.0000
$a280(05, bsc, 05)$	0.9400	1.0000	1.0000
$a280(10, bsc, 05)$	1.0000	1.0000	0.0000
$a280(01, bsc, 10)$	0.9800	0.8800	0.4000
$a280(05, bsc, 10)$	0.8200	1.0000	0.5000
$a280(10, bsc, 10)$	0.9500	0.9700	0.0100
$a280(01, usw, 01)$	0.1400	0.5500	0.1000
$a280(05, usw, 01)$	0.6350	1.0000	0.9000
$a280(10, usw, 01)$	0.8000	1.0000	1.0000
$a280(01, usw, 05)$	0.5800	0.5200	0.5000
$a280(05, usw, 05)$	0.9650	0.6700	0.6000
$a280(10, usw, 05)$	0.8650	0.4500	0.6700
$a280(01, usw, 10)$	0.7800	0.9300	0.8000
$a280(05, usw, 10)$	1.0000	0.9200	0.9200
$a280(10, usw, 10)$	0.8200	0.8100	0.4600
$a280(01, unc, 01)$	0.4800	0.2500	1.0000
$a280(05, unc, 01)$	0.3800	0.7700	0.8000
$a280(10, unc, 01)$	0.2000	0.9800	0.6000
$a280(01, unc, 05)$	0.2300	0.9500	0.4800
$a280(05, unc, 05)$	1.0000	0.9500	0.9500
$a280(10, unc, 05)$	0.8100	0.9300	0.2300
$a280(01, unc, 10)$	0.8250	1.0000	1.0000
$a280(05, unc, 10)$	0.9600	0.9600	0.7400
$a280(10, unc, 10)$	0.7900	0.6800	0.0300

Table 9: A-test over *pr439* instances.

Algorithm	GPHS* x MA2B	GPHS* x MATLS	GPHS* x S5
$pr439(01, bsc, 01)$	0.0000	1.0000	0.9000
$pr439(05, bsc, 01)$	1.0000	1.0000	0.9400
$pr439(10, bsc, 01)$	0.7100	0.8200	0.1900
$pr439(01, bsc, 05)$	0.1200	0.6300	0.0900
$pr439(05, bsc, 05)$	0.3100	0.7100	0.0900
$pr439(10, bsc, 05)$	0.3900	0.8200	0.0000
$pr439(01, bsc, 10)$	0.7200	0.9800	0.8200
$pr439(05, bsc, 10)$	0.7700	1.0000	0.3300
$pr439(10, bsc, 10)$	0.4000	0.9900	0.4000
$pr439(01, usw, 01)$	0.6600	0.4900	0.6500
$pr439(05, usw, 01)$	0.5700	0.9000	0.7600
$pr439(10, usw, 01)$	0.6400	1.0000	0.3600
$pr439(01, usw, 05)$	0.9200	1.0000	1.0000
$pr439(05, usw, 05)$	0.6850	0.9200	0.9100
$pr439(10, usw, 05)$	0.5400	0.9400	0.5700
$pr439(01, usw, 10)$	0.7400	1.0000	0.7600
$pr439(05, usw, 10)$	0.8500	0.9700	0.8900
$pr439(10, usw, 10)$	0.1100	0.5700	0.0200
$pr439(01, unc, 01)$	0.8900	0.9000	0.8600
$pr439(05, unc, 01)$	0.9300	1.0000	0.0100
$pr439(10, unc, 01)$	0.8000	0.8300	0.0400
$pr439(01, unc, 05)$	0.3100	0.8900	0.6900
$pr439(05, unc, 05)$	0.8150	0.8700	0.8000
$pr439(10, unc, 05)$	0.4500	0.7900	0.4300
$pr439(01, unc, 10)$	0.4500	0.9300	0.3600
$pr439(05, unc, 10)$	0.7600	1.0000	0.8000
$pr439(10, unc, 10)$	0.2650	0.7900	0.0200

Table 10: A-test over *rat783* instances.

Algorithm	GPHS* x MA2B	GPHS* x MATLS	GPHS* x S5
$rat783(01, bsc, 01)$	0.6900	0.8600	0.5300
$rat783(05, bsc, 01)$	0.5900	0.6000	0.1500
$rat783(10, bsc, 01)$	0.6000	0.6000	0.1200
$rat783(01, bsc, 05)$	0.9900	1.0000	1.0000
$rat783(05, bsc, 05)$	0.9000	0.9300	0.8800
$rat783(10, bsc, 05)$	0.6500	1.0000	0.3000
$rat783(01, bsc, 10)$	0.9500	1.0000	1.0000
$rat783(05, bsc, 10)$	1.0000	1.0000	0.2000
$rat783(10, bsc, 10)$	0.7000	0.5500	0.0000
$rat783(01, usw, 01)$	0.2700	0.1400	0.0000
$rat783(05, usw, 01)$	0.2800	0.0700	0.0000
$rat783(10, usw, 01)$	0.0000	0.0100	0.0000
$rat783(01, usw, 05)$	0.9000	0.2600	0.0000
$rat783(05, usw, 05)$	1.0000	0.1800	0.0600
$rat783(10, usw, 05)$	0.5000	0.2800	0.1300
$rat783(01, usw, 10)$	1.0000	0.4000	0.2600
$rat783(05, usw, 10)$	0.8900	0.2100	0.0400
$rat783(10, usw, 10)$	0.8800	0.4600	0.1300
$rat783(01, unc, 01)$	0.2000	0.0200	0.4200
$rat783(05, unc, 01)$	0.6800	0.3800	0.0000
$rat783(10, unc, 01)$	0.0000	0.4300	0.0000
$rat783(01, unc, 05)$	0.8300	0.3200	0.4500
$rat783(05, unc, 05)$	0.8700	0.3000	0.1600
$rat783(10, unc, 05)$	0.3500	0.1700	0.0600
$rat783(01, unc, 10)$	0.9600	0.7400	0.9800
$rat783(05, unc, 10)$	1.0000	0.0000	0.0000
$rat783(10, unc, 10)$	0.5000	0.3600	0.0000

Tables 8 and 9 show a significant increase in the performance of GPHS* compared with the three other algorithms.

Finally, in Table 10 we observe that GPHS* presents better results for several instances in comparison with MA2B, however its performance decreases when it is compared with MATLS and S5.

The results in Tables 4–10 show that the proposed hyperheuristic performs comparatively better (apart from MATLS) when the sizes of the addressed problems increase.

In order to summarize the statistical analysis of the results concerning each group of instances, the Friedman test and a post-hoc was applied to the obtained objectives values, considering the non-normality observed by Shapiro-Wilk test [10]. Friedman test reveals all the algorithms have statistically significant difference.

Table 11 shows the statistical analysis of pairwise comparisons between GPHS* and the state-of-art algorithms using Dunn-Sidak's post-hoc test with a significance level of $\alpha = 0.05$. When the test result is greater than α, there is no statistical difference between the two approaches. All tests have been executed with a confidence level of 95% ($\alpha = 0.05$) considering the 10 independent runs of each algorithm.

The entries representing a statistically significant difference between GPHS* and the other technique are emphasized (bold). The background is highlighted when GPHS* shows better performance (average of its objective values).

Table 11: Results for pairwise comparisons among GPHS* and state-of-art algorithms using Friedman and Dunn-Sidak's post-hoc tests with $\alpha = 0.05$ for each group of instances.

Algorithm	GPHS* x MA2B	GPHS* x MATLS	GPHS* x S5
*eil*51	0.0032	0.9012	**0.0008**
*berlin*52	0.3096	0.3561	**0.0000**
*eil*76	0.1420	1.0000	**0.0006**
*kroA*100	0.9980	0.1112	**0.0000**
*a*280	**0.0324**	**0.0000**	0.9799
*pr*439	0.9811	0.0000	0.9999
*rat*783	0.3007	1.0000	0.0053

We can observe from Table 11 that there are statistical differences between GPHS* and S5 for all small sized instances (*eil51*, *berlin52*, *eil76* and *kroA100*), where GPHS* is better. In comparison with MA2B, GPHS* is not statistically different for the same instances set. This can also be observed for *a280* and *pr439*, where there are no statistical differences between GPHS* and S5, however, GPHS* is better than MA2B and MATLS for *a280* instances. Finally, for *rat783* set we observe that GPHS* presents no statistical differences between MA2B and MATLS; and S5 provides better results than all the other approaches.

5.5 Discussion

We believe that the GP-based heuristic selection outperforms the standard GA due to two main factors:

- **The tree representation:** a solution can be represented in many different ways using a GP tree, which allows exploring more possibilities.
- **The standard GP crossover:** which is able to explore more combinations than the 1-point crossovers usually used in a standard GA, while preserving the LLHs order.

We believe that the superior performance that MA2B shows is mainly due to two properties. The first is the use of a population of solutions instead of a single solution as is the case with the building blocks of our hyper-heuristic and S5. The second advantage is the excellent balance between diversification (disruptive crossover and strong mutations) and intensification (fast local search).

Our proposed framework is still a preliminary attempt to investigate the use of heuristic selection for the TTP. We used disruptive operators to increase the search space exploration, which worked to some extent, but needs further improvement.

Nevertheless, even with less sophisticated LLHs, GPHS*was able to beat MATLS which uses a population of solutions and sophisticated components. Although MATLS is a faster algorithm, the improvement brought by GPHS* was significant regarding the solution quality.

Perhaps a fairer comparison would be against S5, due to the fact that it is also very time consuming. Indeed, the reported results show clearly that GPHS* outperformed S5 algorithm on the majority of TTP instances.

It is worth noting that GPHS*'s performance decrease for *rat783* is mainly due to the runtime limit and the size of these instances. In fact, due to the 10 minutes stopping criteria which is used as a standard in all the algorithms for the TTP, there was significantly less computational effort for this group of instances.

In addition, it is not obvious to extract a general pattern for the obtained models, or even a pattern given a single TTP instance. However, according to 21 results we can observe that the generated models likely start with a disruptive operator (52%) or a search heuristic (48%). In the middle stages of the models, search heuristics are more likely to be applied (65%) compared to disruptive operators (35%). Local search heuristics are always applied at the end of all the studied models.

6 Conclusions and Future Work

In this paper, we studied the Travelling Thief Problem, an NP-hard multi-component problem, from a hyper-heuristic perspective. Firstly, we briefly revisited various hyper-heuristic techniques used in the field of combinatorial optimization. Then, the problem was formally defined and state-of-the-art algorithms were revisited.

The main focus of this work was to investigate the use of heuristic selection for the TTP with a low computational budget. Therefore, we proposed a heuristic selection technique based on a GP framework in order to search for the best combination between low-level-heuristics and disruptive operators.

We have analyzed the performance of the proposed approach on small sized TTP instances, considering an average of approximation ratio. We also provided a statistical analysis of a pairwise comparison between our approach and three other state-of-the-art algorithms.

Based on the experiments we can conclude that for the small and mid-size instances addressed in this work, the proposed heuristic selection is competitive when compared with the other investigated algorithms. The explanation is based on the fact that GP framework can obtain good combinations of LLH as well as provide a high diversity of the search space. Additionally, GP trees structures have preserved the correlation between the terminals in sub-trees, and this has been transferred to the offspring population.

In the future, we can use a self-tuning technique to dynamically update the LLHs parameters during a run. Furthermore another interesting research direction is expanding the approach for larger instances in order to test its scalability.

Acknowledgements M.Martins acknowledges CAPES/Brazil. M.Delgado acknowledges CNPq grant Nos.: 309197/2014-7 (Brazil Government).

A Appendix

In this appendix we provide a closer look of the average approximation ratio achieved in 10 independent runs (stated as trend lines in Section 5).

According to Figures 3–9, for some instances, the average approximation ratios are close to 100%, while the same achievement seems to be very difficult on others. For example, GPHS* regularly achieves better results than S5 on almost all instances of small size (*eil51*, *berlin52*, *eil76* and *kroA100*), as can be seen in Figures 3, 4, 5 and 6. Another example can be seen in Figures 7 and 8, where GPHS* presents similar results as S5 for almost all instances of *a280* and *pr439* set. Finally, in Figure 9 we observe that GPHS* presents similar results as MA2B and MATLS but worse than S5 for almost all instances of *rat783* set.

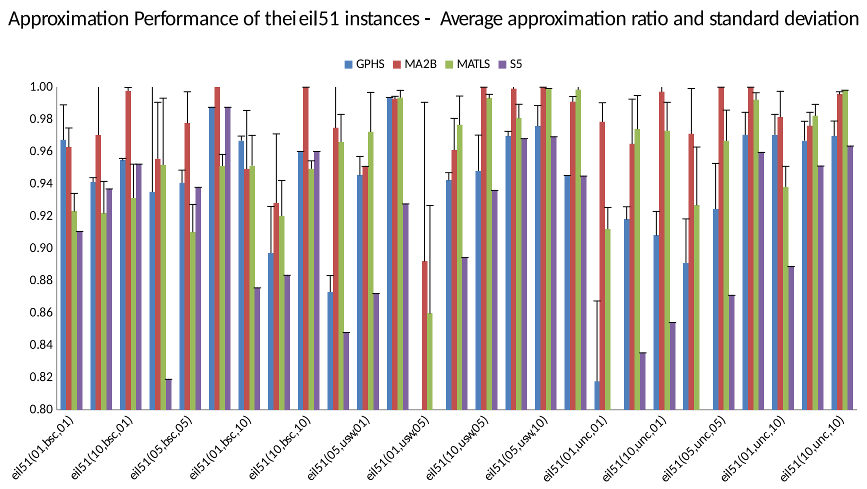

Fig. 3: Average approximation ratios over 10 independent runs of the eil51 instances.

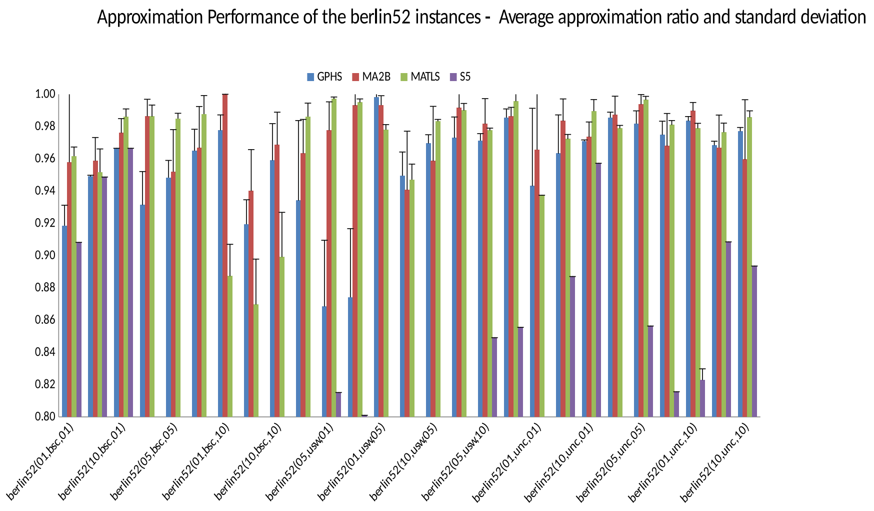

Fig. 4: Average approximation ratios over 10 independent runs of the berlin52 instances.

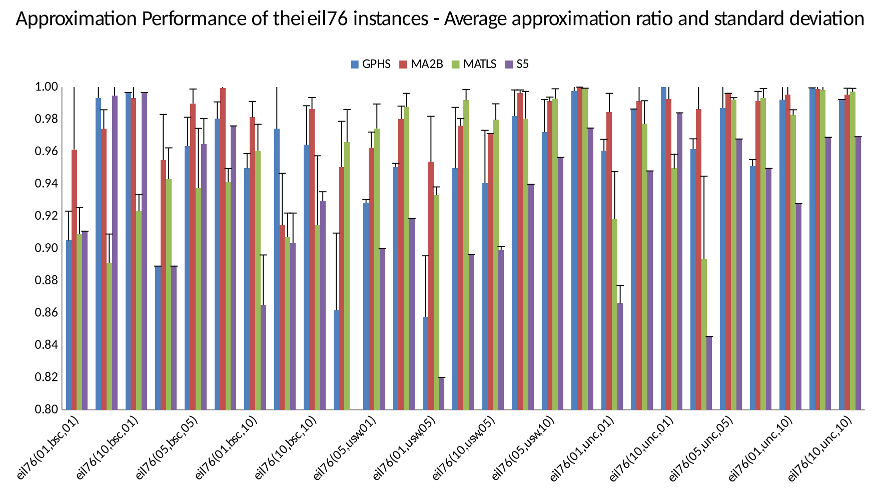

Fig. 5: Average approximation ratios over 10 independent runs of the eil76 instances.

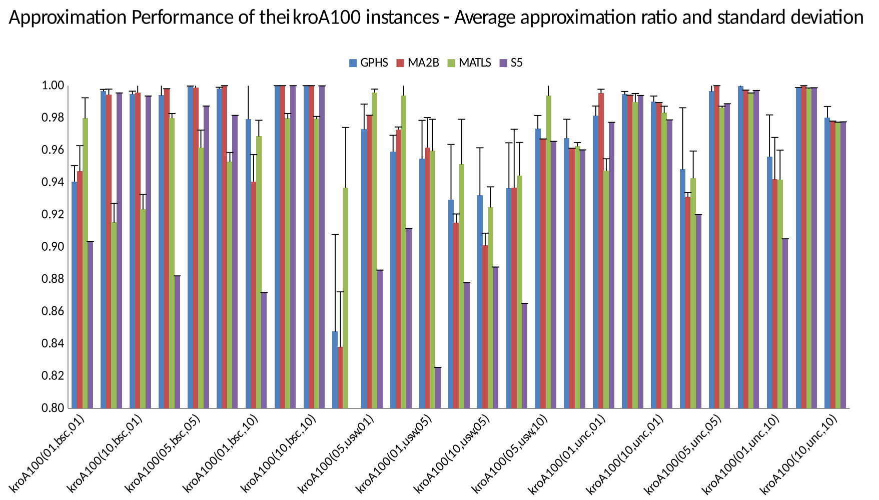

Fig. 6: Average approximation ratios over 10 independent runs of the kroA100 instances.

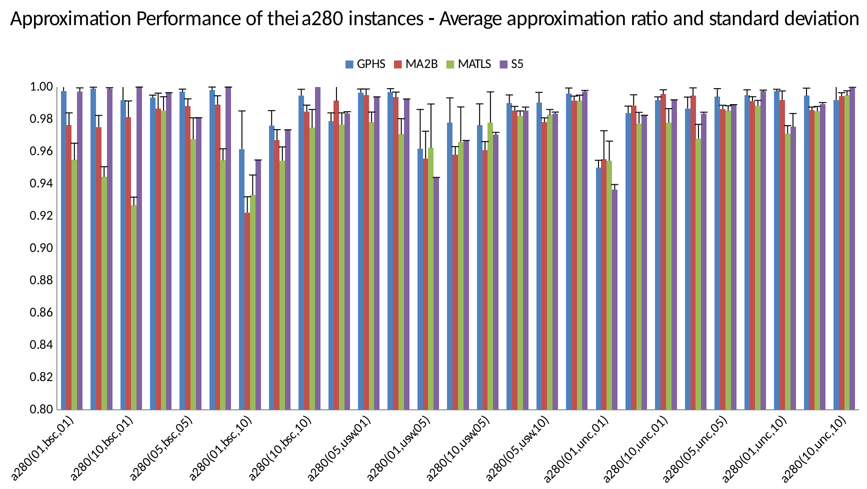

Fig. 7: Average approximation ratios over 10 independent runs of the a280 instances.

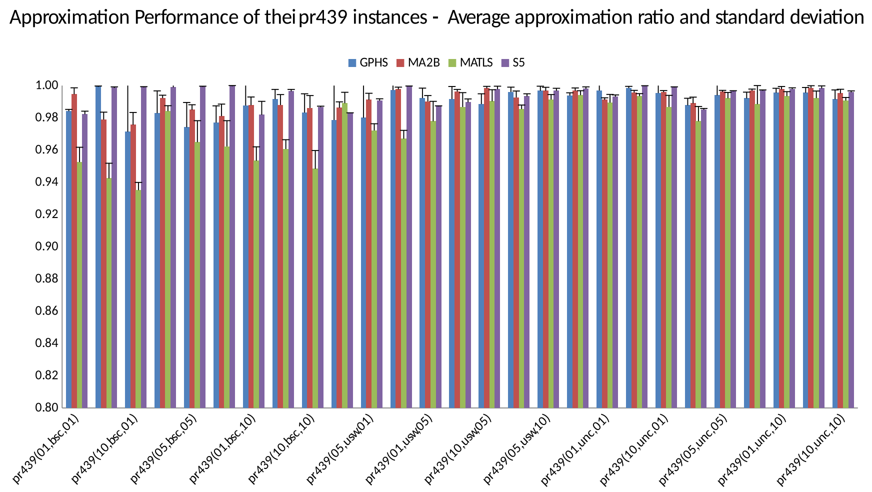

Fig. 8: Average approximation ratios over 10 independent runs of the pr439 instances.

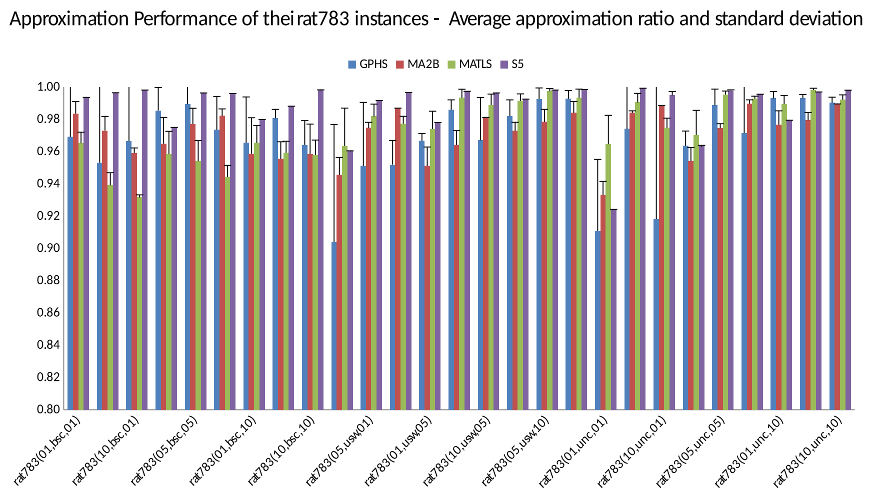

Fig. 9: Average approximation ratios over 10 independent runs of the rat783 instances.

References

1. Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. *F-Race and Iterated F-Race: An Overview*, pages 311–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
2. Andreas Bölte and Ulrich Wilhelm Thonemann. Optimizing simulated annealing schedules with genetic programming. *European Journal of Operational Research*, 92(2):402–416, 1996.
3. M. R. Bonyadi, Z. Michalewicz, and L. Barone. The travelling thief problem: The first step in the transition from theoretical problems to realistic problems. In *Proceedings of the 2013 IEEE Congress on Evolutionary Computation*, pages 1037–1044, June 2013.
4. Mohammad Reza Bonyadi, Zbigniew Michalewicz, Michal Roman Przybylek, and Adam Wierzbicki. Socially inspired algorithms for the travelling thief problem. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, GECCO'14*, pages 421–428. ACM, 2014.
5. Edmund K. Burke, Matthew R. Hyde, Graham Kendall, and John Woodward. Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, GECCO'07*, volume 2, pages 1559–1565, London, 2007. ACM.
6. Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward. *Exploring Hyper-heuristic Methodologies with Genetic Programming*, pages 177–201. Springer Berlin Heidelberg, 2009.
7. Edmund K. Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R. Woodward. *A Classification of Hyper-heuristic Approaches*, pages 449–468. Springer US, Boston, MA, 2010.
8. E.K. Burke, G. Kendall, and E. Soubeiga. A Tabu-Search Hyperheuristic for Timetabling and Rostering. *Journal of Heuristics*, 9(6):451–470, 2003.
9. Mauro Castelli, Luca Manzoni, Leonardo Vanneschi, Sara Silva, and Aleš Popovič. Self-tuning geometric semantic genetic programming. *Genetic Programming and Evolvable Machines*, 17(1):55–74, 2016.
10. W.J. Conover. *Practical Nonparametric Statistics*. Wiley, third edition, 1999.

11. Peter Cowling, Graham Kendall, and Eric Soubeiga. A Hyperheuristic Approach to Scheduling a Sales Summit. In *Proceedings of the Third International Conference on Practice and Theory of Automated Timetabling, PATAT 2000*, pages 176–190, Konstanz, Germany, 2000. Springer Berlin Heidelberg.
12. Peter Cowling, Graham Kendall, and Eric Soubeiga. A Parameter-Free Hyperheuristic for Scheduling a Sales Summit. In *Proceedings of the 4th Metaheuristic International Conference, MIC 2001*, pages 127–131, 2001.
13. Alberto Cuesta-Cañada, Leonardo Garrido, and Hugo Terashima-Marín. Building Hyper-heuristics Through Ant Colony Optimization for the 2D Bin Packing Problem. In *Proceedings of the 9th International Conference, KES 2005*, pages 654–660, Melbourne, Australia, 2005. Springer Berlin Heidelberg.
14. Boris Delaunay. Sur la sphere vide. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7(793-800):1–2, 1934.
15. Kathryn A. Dowsland, Eric Soubeiga, and Edmund Burke. A simulated annealing based hyperheuristic for determining shipper sizes for storage and transportation. *European Journal of Operational Research*, 179(3):759 – 774, 2007.
16. John H Drake, Matthew Hyde, Khaled Ibrahim, and Ender Ozcan. A genetic programming hyper-heuristic for the multidimensional knapsack problem. *Kybernetes*, 43(9/10): 1500–1511, 2014.
17. Mohamed El Yafrani and Belaïd Ahiod. Cosolver2B: An efficient Local Search Heuristic for the Travelling Thief Problem. In *Proceedings of the 2015 IEEE/ACS 12th International Conference of Computer Systems and Applications, AICCSA*, pages 1–5, Nov 2015. doi: 10.1109/AICCSA.2015.7507099.
18. Mohamed El Yafrani and Belaïd Ahiod. A local Search based Approach for Solving the Travelling Thief Problem. *Applied Soft Computing*, 2016. ISSN 1568-4946. doi: http://dx.doi.org/10.1016/j.asoc.2016.09.047.
19. Mohamed El Yafrani and Belaïd Ahiod. Population-based vs. Single-solution Heuristics for the Travelling Thief Problem. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO'16, pages 317–324, Denver, Colorado, USA, 2016. ACM. ISBN 978-1-4503-4206-3.
20. Hayden Faulkner, Sergey Polyakovskiy, Tom Schultz, and Markus Wagner. Approximate Approaches to the Traveling Thief Problem. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO'15, pages 385–392, Madrid, Spain, 2015. ISBN 978-1-4503-3472-3.
21. Alex S. Fukunaga. Automated Discovery of Local Search Heuristics for Satisfiability Testing. *Evolutionary Computation*, 16(1):31–61, March 2008. ISSN 1063-6560.
22. Rachel Hunt, Kourosh Neshatian, and Mengjie Zhang. A Genetic Programming Approach to Hyper-Heuristic Feature Selection. In *Proceedings of the 9th International Conference on Simulated Evolution and Learning, SEAL 2012*, pages 320–330, Hanoi, Vietnam, 2012. Springer Berlin Heidelberg.
23. Miha Kovačič. Modeling of total decarburization of spring steel with genetic programming. *Materials and Manufacturing Processes*, 30(4):434–443, 2015.
24. Miha Kovačič and Franjo Dolenc. Prediction of the natural gas consumption in chemical processing facilities with genetic programming. *Genetic Programming and Evolvable Machines*, 17(3):231–249, 2016.
25. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.
26. Natalio Krasnogor and Jim Smith. Emergence of Profitable Search Strategies Based on a Simple Inheritance Mechanism. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, GECCO'01, pages 432–439, San Francisco, California, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-774-9.
27. William B. Langdon, Riccardo Poli, Nicholas F. McPhee, and John R. Koza. *Genetic Programming: An Introduction and Tutorial, with a Survey of Techniques and Applications*, pages 927–1028. Springer Berlin Heidelberg, 2008.
28. Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.
29. M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The irace Package: Iterated Racing for Automatic Algorithm Configuration. IRIDIA Technical Report

Series 2011-004, Universit? Libre de Bruxelles, Bruxelles,Belgium, 2011.
30. Sean Luke and Liviu Panait. Lexicographic Parsimony Pressure. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO'02, pages 829–836, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. ISBN 1-55860-878-8.
31. Yi Mei, Xiaodong Li, and Xin Yao. Improving efficiency of heuristics for the large scale traveling thief problem. In *Proceedings of the Asia-Pacific Conference on Simulated Evolution and Learning*, pages 631–643. Springer, 2014.
32. Yi Mei, Xiaodong Li, Flora Salim, and Xin Yao. Heuristic evolution with genetic programming for traveling thief problem. In *Proceedings of the 2015 IEEE Congress on Evolutionary Computation, CEC*, pages 2753–2760. IEEE, 2015.
33. Yi Mei, Xiaodong Li, and Xin Yao. On investigation of interdependence between subproblems of the travelling thief problem. *Soft Computing*, 20(1):157–172, 2016.
34. Su Nguyen, Mengjie Zhang, and Mark Johnston. A Genetic Programming Based Hyper-heuristic Approach for Combinatorial Optimisation. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO'11, pages 1299–1306, Dublin, Ireland, 2011. ACM. ISBN 978-1-4503-0557-0.
35. Sergey Polyakovskiy, Mohammad Reza Bonyadi, Markus Wagner, Zbigniew Michalewicz, and Frank Neumann. A Comprehensive Benchmark Set and Heuristics for the Traveling Thief Problem. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO'14, pages 477–484, Vancouver, BC, Canada, 2014. ACM. ISBN 978-1-4503-2662-9.
36. Gerhard Reinelt. Tsplib a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.
37. Peter Ross. *Hyper-Heuristics*, pages 529–556. Springer US, Boston, MA, 2005.
38. Peter Ross, Javier G. Marín-Blázquez, Sonia Schulenburg, and Emma Hart. Learning a Procedure That Can Solve Hard Bin-Packing Problems: A New GA-Based Approach to Hyper-heuristics. In *Proceedings of the Genetic and Evolutionary Computation 2003, GECCO'03*, pages 1295–1306, Chicago, IL, USA, 2003. Springer Berlin Heidelberg.
39. Sara Silva. Gplab a genetic programming toolbox for matlab, version 4.0 (2015). University of Coimbra, 2009. URL `http://gplab.sourceforge.net/download.html`.
40. Sara Silva and Jonas Almeida. Gplab-a genetic programming toolbox for matlab. In *Proceedings of the Nordic MATLAB Conference (NMC-2003)*, pages 273–278, 2005.
41. Alejandro Sosa-Ascencio, Gabriela Ochoa, Hugo Terashima-Marin, and Santiago Enrique Conant-Pablos. Grammar-based generation of variable-selection heuristics for constraint satisfaction problems. *Genetic Programming and Evolvable Machines*, 17(2): 119–144, 2016.
42. András Vargha and Harold D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
43. Markus Wagner. Stealing Items More Efficiently with Ants: A Swarm Intelligence Approach to the Travelling Thief Problem. In *Proceedings of the 10th International Conference on Swarm Intelligence, ANTS 2016*, pages 273–281, Brussels, Belgium, 2016. Springer International Publishing.
44. Markus Wagner, Marius Lindauer, Mustafa Mısır, Samadhi Nallaperuma, and Frank Hutter. A case study of algorithm selection for the traveling thief problem. *Journal of Heuristics*, pages 1–26, 2017. ISSN 1572-9397. doi: 10.1007/s10732-017-9328-y. URL `http://dx.doi.org/10.1007/s10732-017-9328-y`.
45. Junhua Wu, Sergey Polyakovskiy, and Frank Neumann. On the impact of the renting rate for the unconstrained nonlinear knapsack problem. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, GECCO'16*, pages 413–419. ACM, 2016.