A regression analysis of the impact of routing and packing dependencies on the expected runtime

Mohamed El Yafrani[1,4], Marcella Scoczynski[2*], Markus Wagner[3] and Peter Nielsen[1]

[1]Materials and Production, Aalborg University, Aalborg, Denmark.
[2]Department, Federal University of Technology - Parana, Ponta Grossa, Brazil.
[3]School of Computer Science, The University of Adelaide, Adelaide, Australia.
[4]Quantitative Assets, Norlys Energy Trading A/S, Aalborg, Denmark.

*Corresponding author(s). E-mail(s): marcella@utfpr.edu.br;
Contributing authors: mey@mp.aau.dk,mey@net.com; markus.wagner@adelaide.edu.au; peter@mp.aau.dk;

Abstract

Problems with multiple interdependent components offer a better representation of the real-world situations where globally optimal solutions are preferred over optimal solutions for the individual components. One such model is the Travelling Thief Problem (TTP); while it may offer a better benchmarking alternative to the standard models, only one form of inter-component dependency is investigated. The goal of this paper is to study the impact of different models of dependency on the fitness landscape using performance prediction models (regression analysis). To conduct the analysis, we consider a generalised model of the TTP, where the dependencies between the two components of the problem are tunable through problem features. We use regression trees to predict the instance difficulty using an efficient Memetic Algorithm that is agnostic to the domain knowledge to avoid any bias. We report all the decision trees resulting from the regression model, which is the core in understanding the relationship between the dependencies (represented by the features) and problem difficulty (represented by the runtime). The regression model was able to predict the expected runtime of the algorithm based on the problem features. Furthermore, the results show that the contribution of the item value drop dependency is significantly higher than the velocity change dependency.

Keywords: Runtime regression analysis, Evolutionary Algorithms, Travelling Thief Problem

1 Introduction

Tackling real-world problems can be very challenging compared to standard benchmarking integer programming models [17, 33, 34, 37]. Multiple aspects contribute to creating this gap, such as bad modelling practices (e.g., over-simplification, linearisation), intractability (high computational complexity of solution methods), hard and soft constraints (heavy constraints...), external factors (stochastic environment, uncertainty about data), and composition of interdependent (mutually dependent) sub-problems [4, 28].

In this paper, we are interested in the last aspect, namely, dependencies between the problem's components (sub-problems). This type of problems are referred to as problems with multiple interdependent components [2] or multi-hard problems [32], as the components are NP-hard when tackled separately. To the best of our knowledge, the first benchmark model covering this aspect is the Travelling Thief Problem (TTP), introduced by Bonyadi et al [2] as a combination of the Travelling Salesman Problem and the Knapsack Problem. The model was then simplified by Polyakovskiy et al [31], and a large dataset of instances was published. Furthermore, an extension has been proposed in Chand and Wagner [5] where multiple thieves are considered.

The standard TTP formulation in [31] considers a combination of the Travelling Salesman Problem (TSP) and the Knapsack Problem (KP). The problem considers a set of items scattered in different cities where a thief should visit each city once, picking some items on the way and returning to the first city, while trying to maximise the thief's gain. The dependency in this formulation is modelled in a number of manners, such as:

- Penalising the travelling time by tying the thief's velocity with the knapsack load (standard model [31]),
- Decreasing the value of items as the thief progresses in his journey,
- A combination of the above (bi-objective TTP model [2]).

Note that because the final aim is to cover some aspects from real-world problems, both dependency types were designed to reflect realistic situations. First, the load-velocity dependency can be adapted to reflect the relationship between the load and fuel consumption in the transportation of goods [16].

Second, the value drop dependency can have even more realistic and important applications within the shipping sector, such as the transportation of perishable goods [7, 14].

Since it was introduced, the TTP received the attention of researchers from the fields of evolutionary computation, metaheuristics and operations research, mainly due to the fact that the problem is easy to understand, yet challenging to solve. Several papers proposed heuristic solution methods [9, 10, 26, 38], and fewer ones tried to analyse the problem empirically and theoretically [11, 42, 43]. These analyses focus on the impact of problem features, with little to no attention to the impact of the dependencies between the components. Furthermore, all of the above-mentioned works only consider the velocity change constraint, which is probably the result of the standard TTP library only supporting this dependency.

The gap we are trying to close with this work is the analysis of the impact of the above-mentioned dependency formulations and investigation of their impact on the difficulty of tackling the problem. To conduct the analysis, we consider cost models as a tool to empirically analyse the difficulty of the generalised TTP model, that embeds all these dependency models. The novelty here considers imposing that the value of a picked item drops by time, besides the evaluation of the difficulty of problem instances is done using a Memetic Algorithm based on MA2B [9]. The findings show that the item value drop dependency significantly impacts the difficulty of solving the TTP instances. More importantly, according to our analysis, its impact is stronger than the velocity change constraint.

In Section 2, we provide a background information with a brief literature review on the algorithms and analyses for the TTP and cost model-based fitness landscape analysis. Section 3 introduces the methodology adopted to analyse the problem, including the feature-based analysis and the adopted algorithm. In Section 4 describes the experiments and discusses the results. Finally, Section 5 summarises the findings and concludes the paper.

2 Background and Related Works

2.1 The Travelling Thief Problem

The standard TTP can be informally stated as follows: *Given are a set of cities and a set of items distributed among these cities. Each item is defined by its individual profit and weight. A thief must visit all the cities exactly once, pick some items while travelling, and return to the starting city. The knapsack has a limited capacity, which should not be exceeded. We also consider a knapsack renting rate (per time unit) which determines the amount that the thief must pay at the end of the journey.*

What makes the two components of the TTP interdependent is the velocity of the thief, because it changes according to the weight of the knapsack. Specifically, the heavier the knapsack gets, the slower the thief becomes. The objective is to maximise the total gain function defined as the total profit, minus the cost of the journey.

The interdependence, such as the one present in the TTP, can reflect the characteristics and the complexity of real-world problems [2]. For this reason, several authors have addressed the TTP by applying different methods. Polyakovskiy et al [31] presented the first heuristics for solving the TTP, generating a TSP tour using the classical Chained Lin–Kernighan heuristic [1] and with the fixed tour they applied some packing heuristics for improving the solution, such as Random Local Search (RLS) and (1+1)-EA.

As the problem is NP-hard and the objective function is non-linear, many researchers focused on (meta-)heuristic algorithms. The work proposed by Bonyadi et al [3] introduces a method named CoSolver, which is inspired by cooperative coevolution. The framework consists in splitting the problem into sub-problems and tackling them in a parallel and synchronous fashion. Mei et al [26] proposed a fast Memetic Algorithm called MATLS, which embeds multiple complexity reduction methods to solve large scale TTP instances. Faulkner et al [13] explored multiple operators for optimising the packing plan combining them in a number of simple and complex heuristics. The work in [9] proposed and compared two heuristic algorithms, a Memetic Algorithm (MA2B) and Simulated Annealing-based algorithm (CS2SA) which resulted in competitive performances for various problem sizes. Wagner [38] investigated the use of swarm intelligence approaches with two different TSP-specific local search operators and of "boosting" TTP solutions using TTP-specific local search. Two algorithms were proposed in [10] based on combining the 2-OPT steepest ascent hill climbing algorithm for the TSP component and the simulated annealing metaheuristic for the KP component, named CS2SA* and CS2SA-R. The obtained results showed that the proposed algorithms are competitive in many TTP instances.

In [39] the authors created a dataset with performance data of 21 TTP algorithms on the full original set of 9720 TTP instances. They also defined 55 characteristics for TTP instances that can be used to select the best algorithm on a per-instance basis, and they used these algorithms and features to construct algorithm portfolios for TTP in order to outperform the single best algorithm.

Recently, the authors in [29] investigated the inter-dependency of the TSP and the KP by means of quality diversity (QD) approaches, conducting experimental studies that show the usefulness of using the QD approach applied to the TTP. Besides, the authors introduced a MAP-Elite based evolutionary algorithm called BMBEA, using well-known TSP and KP search operators. In the MAP-Elite solutions compete with each other to survive. Their results showed that QD approach can improve the best-known solution for a wide range of TTP instances.

2.2 Empirical algorithm analysis

Fitness landscapes represents the association between the search process and the fitness space [41]. A heuristic algorithm can be seen as a strategy for navigating the solution landscape structure in the search for an optimal solution. Thus, fitness landscape analysis is a set of tools and methods used to investigate the dynamics of heuristic search algorithms applied for specific optimisation problems [22].

Cost models are fitness landscapes methods that can help predicting the performance of algorithms by identifying features that make a problem more or less difficult to solve. These models can be expressed as linear, multiple regression models [25], decision trees [30], or other models of features and search cost; also, some models are more amendable to human interpretation than others. To aid interpretability, the features are extracted from the problem structure and the model can at times explain their influence in the difficulty level during the search [11].

Some authors have presented fitness landscape analysis for several problems, as described in [44]. The work developed by [27] studied the relation between features of fitness landscapes and recombination and/or mutation operators for the Quadratic assignment problems. The authors in [35, 36] analysed the fitness landscape of a dynamic optimisation problem, investigating the influence on the performance of the algorithm.

The work proposed by [18] designed a prediction model for the algorithmic performance of CMA-ES variants by using a random forest regression model based on exploratory features of fitness landscapes. Also, the authors in [24] presented a spatial-domain fitness landscape analysis framework to visualise the fitness landscapes regarding a specific combinatorial optimisation problem and evaluate its properties. By extracting characteristics of combinatorial optimisation problems allowed study the behaviour of algorithms effectively.

Recently, the paper in [21] focused on the adaptability landscape features of optimisation problems by applying differential evolution algorithm. The authors presented a quantitative analysis of the fitness distance correlation information, evaluating the difficulty of solving the problem.

Some papers explored TTP using fitness landscape analysis. In [43] the authors considered local search operators and investigated the fitness landscape characteristics of some smaller instances of the TTP. The local search operators included 2-opt, Insertion, Bitflip and Exchange and metaheuristics included multi-start local search, iterated local search and genetic local search.

Another recent work in [12] investigated 3 dependency models of the TTP, in addition to a dependency-free model for comparison purposes. The authors used local optima networks as a fitness landscape analysis tool to study the difficulty of the search landscape for these dependency models. The results show that the dependency-free landscape is the most difficult to navigate based on the basin sizes and their correlation to the fitnesses. Such a result is difficult to interpret as it is expected that the dependencies create more difficult instances [4]. The authors speculated that this result is due to the join neighbourhood search algorithm, an algorithm that joins two neighbourhood sets into one, which creates an additional complexity even for the dependency-free model.

3 Proposed approach

3.1 Generalised TTP model

Herein, we provide the mathematical formulation of the generalised TTP, based on the standard TTP as introduced in Section 2, and the TTP2 formulation in [2]. The model embeds two types of dependency, and is conceived such that the strength of these dependencies is tunable through problem features.

Input data

We define the following problem input parameters:

- $N = \{1, \ldots, n\}$ is the set of labels, representing the cities to be visited,
- $\{d_{ij}\}$ is the matrix of distances between these cities,
- $M = \{1, \ldots, m\}$ is a set of labels corresponding to the items scattered among cities,
- p_k and w_k represent the profit and weight of an item $k \in M$, respectively,
- W is the knapsack capacity,
- R is the knapsack renting rate, which is used to determine the cost of the journey,
- v_{max} and v_{min} represent the maximum and minimum velocities, respectively.
- $A = \{A_1, \ldots, A_m\}$ denotes the availability vector, such that $A_k \in \{1, \ldots, n\}$ contains the reference to the city that contains the item k.

Decision variables

A TTP solution is represented using two components: The first is the tour $X = (x_1, \ldots, x_n)$, a vector containing the ordered list of cities, which is encoded as a permutation. The second is the picking plan $Z = (z_1, \ldots, z_m)$, a binary vector representing the states of items, where 1 is associated to the packed items, and 0 to the unpacked ones.

Interdependencies

What makes the two components of the TTP interdependent is the velocity of the thief which changes according to the weight of the knapsack. Therefore, the velocity at city x_i is defined in Equation 1.

$$v_{x_i} = v_{max} - \left(\frac{v_{max} - v_{min}}{W}\right) \times w_{x_i} \tag{1}$$

where w_{x_i} the weight of the knapsack at city x_i.

In addition to the velocity change dependency in Equation 1, we add a second dependency on the value of items. This is achieved by imposing that the value of a picked item k drops by time from its initial value p_k to p_k^{final} following Equation 2.

$$p_k^{final} = p_k \times \mathcal{D}_r^{\lceil \frac{T_k}{Q} \rceil} \tag{2}$$

where $\mathcal{D}_r$ is the item value dropping rate, p_k is the initial value of item k, T_k is the carrying time of item k, and Q is a constant calculated as shown in Equation 3.

$$Q = T_{min} \frac{ln(\mathcal{D}_r)}{ln\left(\frac{P_{min}}{2 \times P_{max}}\right)} \tag{3}$$

Equation 3 seeks to satisfy $P_{max} \times \mathcal{D}_r^{\frac{T_k}{Q}} = \frac{1}{2} \times P_{min}$, where P_{max} and P_{min} represent the maximum and minimum item values, respectively, while T_{min} is the minimum travelling time.

Furthermore, the dependencies in the Generalised TTP model are tunable through the parameters $v_{min} \in [0, v_{max}]$ and $\mathcal{D}_r \in [0, 1]$, where:

- The velocity drop dependency is controlled through v_{min}. When $v_{min} = v_{max}$, this dependency is cancelled as the thief will be always travelling at the maximum velocity v_{max} independently from the knapsack load.
- The item value drop dependency is controlled through $\mathcal{D}_r$. Similarly, this dependency is deactivated by setting $\mathcal{D}_r = 1$.

Based on the above, the proposed model covers all dependency models in [12], including the standard TTP.

There are two special cases to consider: (1) Setting $v_{min} = 0$ can lead to a velocity of 0. Based on Equation 1, we have $v_x = v_{max}(1 - \frac{w_x}{W})$, which leads to the thief remaining stationary if the knapsack load reaches the maximum capacity W. (2) Setting $\mathcal{D}_r = 0$ makes all the picked items valueless in the end as $p_k^{final} = 0$. Based on these remarks, the considered values for v_{min} and $\mathcal{D}_r$ are always chosen strictly positive in our empirical study.

Note that it is difficult to predict how a change in dependency feature values, e.g., lower values of $\mathcal{D}_r$ and v_{min}, will impact the difficulty of the problem instances. This partially depends on the algorithm used to tackle the problem.

To illustrate how these interdependencies work, we consider a TTP instance with 6 cities and 20 items, and a potential solution as shown in Figure 1. Note that this is a simplified illustration as the distances, item values and weights among other problem parameters are not shown for simplicity. Following the velocity change dependency, assuming that $v_{min} < v_{max}$, the velocity will start decreasing from city B where items 1 and 4 are picked, passing through city C does not influence the velocity as no items are picked, and so on. Now considering the item value drop dependency, the items 1 and 4 lose most of their value as they are carried for almost the entire journey, while item 17 loses the least of its initial value.

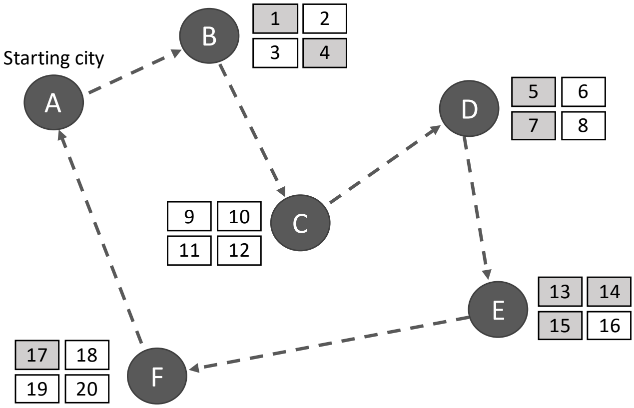


Figure 1: A simplified illustration of a TTP instance and solution. The circles represent the cities, labelled by letters from A to F. The rectangles represent the items associated to each city (except the first), labelled by numbers from 1 to 20. The dashed arrows form the route, and the highlighted rectangles represent the picked items.

Objective function

To focus on the dependency analysis, we consider a linear combination of the total profit and cost of the journey as shown in Equation 4.

$$\text{Maximise} \quad G(X,Z) = \sum_m p_m^{final} \times z_m - R \times (\sum_{i=1}^{n-1} \frac{d_{x_i,x_{i+1}}}{v_{x_i}} + \frac{d_{x_n,x_1}}{v_{x_n}}) \tag{4}$$

Subject to:

$$\sum_m w_m \times z_m \quad \leq W \tag{5}$$

$$x_i \in \mathbb{Z}^+ \quad \forall i \in \{1, \dots, n\} \tag{6}$$

$$z_k \in \{0, 1\} \quad \forall k \in \{1, \dots, m\} \tag{7}$$

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}.

As mentioned earlier, the introduction of the velocity change and item value drop dependencies leads to a non-linear objective function. Note that the non-linearity exists in the first term of the equation as well, since p_k^{final} has a non-linear formulation (Eq. 2) and depends on z. This increases the difficulty of the problem as it cannot be solved using standard MILP methods, and rather requires an ad-hod solution.

3.2 Memetic Algorithm

As a baseline for our analysis, we consider the memetic algorithm presented in Algorithm 1[1]. The algorithm uses genetic operators (tournament selection and crossover) combined with a hill climbing local search algorithm to evolve a population of candidate solutions iteratively. The implementation is based on the Memetic Algorithm MA2B initially designed for the standard TTP [9], but differs from the original implementation by removing all domain knowledge from the logic of the algorithm.

Algorithm 1 Memetic Algorithm for the Generalised TTP

$P \leftarrow \varnothing$
$T \leftarrow 0$
for $i \in \{0, \ldots, N_{pop}\}$ **do**
  $S_i \leftarrow \{rand\_perm(n), rand\_bin(m)\}$ ▷ Random initialisation
  $G(S_i)$ ▷ Evaluate
  $T \leftarrow T + 1$
  $\{S_i, T\} \leftarrow hill\_climbing(S_i, T)$ ▷ Local search improvement
  $P \leftarrow P \cup S_i$
end for
$P \leftarrow sort(P)$
$S_{bsf} \leftarrow \{rand\_perm(n), rand\_bin(m)\}$
repeat
  if $G(P_1) > G(S_{bsf})$ **then**
    $S_{bsf} \leftarrow P_1$
  end if
  $Q \leftarrow \varnothing$
  for $i \in \{0, \ldots, N_{offspring}\}$ **do**
    $\{p_1, p_2\} \leftarrow tournament(P, N_{tournament})$ ▷ Select parent solutions
    $S_{new} \leftarrow crossover(p_1, p_2)$ ▷ Crossover
    $G(S_{new})$
    $T \leftarrow T + 1$
    if $random(0, 1) < R_{LS}$ **then**
      $\{S_{new}, T\} \leftarrow hill\_climbing(S_{new}, T)$ ▷ Local search improvement
    end if
    if $S_{new} \notin Q$ **then** ▷ add to offspring population
      $Q \leftarrow Q \cup \{S_{new}\}$
    else ▷ Generate new solution to reduce premature convergence
      $S_{rand} \leftarrow \{rand\_perm(n), rand\_bin(m)\}$
      $\{S_{rand}, T\} \leftarrow hill\_climbing(S_{rand}, T)$
      $Q \leftarrow Q \cup \{S_{rand}\}$ ▷ add to offspring population
    end if
  end for
  $P' \leftarrow sort(P \cup Q)$
  $P \leftarrow \{P'_1, \ldots, P'_{N_{pop}}\}$
until $T \geq T_{max} \lor \frac{g^* - G(P_1)}{g^*} \leq \epsilon$

[1] The implementation of the memetic algorithm is done in Java based on the codes available at https://github.com/yafrani/ttplab

The motivation behind choosing a Memetic Algorithm is due to the fact that it borrows aspects from classical local search heuristics and evolutionary algorithms (exploratory operators) [40]. Most of the other algorithms are either based on (stochastic or deterministic) local search, or include domain knowledge from TTP standard model, which is practically unusable for the generalised formulation we proposed (due to the second dependency - item value drop). These aspects combined result in an algorithm that can both explore and exploit the solution space. Furthermore, the other TTP heuristics use problem knowledge as the main component of their logic [13, 26], making them unsuitable for this analysis.

Table 1: Notations used in Algorithm 1. The remaining notations are aligned with the problem formulation in Sub-section 3.1

Notation	Description
P	Main population of solutions
T	Number of evaluations
S_i	An initial solution
S_{bsf}	Best-so-far solution
R_{LS}	Local search probability
Q	Offspring population
p_1 and p_2	Solutions selected to generate the offspring
S_{new}	Solutions generate using genetic operators and local search
S_{rand}	A randomly generated solution to avoid premature convergence
g^*	The optimal objective value

Note that, to simplify the pseudocode notations, we use the same notation to evaluate a solution and to access its objective value. Therefore, only the first call of evaluation function $G(.)$ is considered to increase the counter of the number of evaluations T, which will be used to calculate the expected runtime (ert). Furthermore, the algorithm does not take into account the dependencies as part of the domain knowledge used for the optimisation. This is done on purpose to avoid giving the algorithm an unfair advantage for some specific instance categories, and to obtain insights which can be used to improve the efficiency the the algorithm.

Each solution in the population is initialised with a random permutation for the tour and random binary vector for the picking plan (line 4. As we are interested in the features making the problem difficult, random initialisation is important to for the same reason stated above, i.e., ensuring a fair algorithm and not favouring any specific instance categories.

A best improvement hill climbing procedure is then used to refine the solutions by finding a local optima. The hill climbing algorithm uses two neighbourhood searches for the tour and the picking plan sequentially. A simplified pseudocode of the hill climbing procedure is shown in Algorithm 2. The $\mathcal{N}_{2\text{-OPT}}(S)$ neighbourhood function returns a set of solutions where a new tour is obtained by applying the 2-OPT operator [6] on the tour of S, while the picking plan is copied from S as is. The same logic is followed for the $\mathcal{N}_{\text{bit-flip}}(S)$ function [8] where only the picking plan is updated. More details about these operators can be found in [8].

Algorithm 2 Best improvement hill climbing algorithm

function $hill\_climbing(S, T)$
    repeat ▷ First neighbourhood search on the tour
        for $S^* \in \mathcal{N}_{\text{2-OPT}}(S)$ **do**
            if $G(S^*) > G(S)$ **then**
                $S \leftarrow S^*$
            end if
            $T \leftarrow T + 1$
        end for
    until S is not improved
    repeat ▷ Second neighbourhood search on the picking plan
        for $S^{**} \in \mathcal{N}_{\text{bit-flip}}(S)$ **do**
            if $G(S^{**}) > G(S)$ **then**
                $S \leftarrow S^{**}$
            end if
            $T \leftarrow T + 1$
        end for
    until S is not improved
    return $\{S, T\}$
end function

The population is then sorted and the best solution is identified (lines 10-15). The tournament selection is applied to select 2 parent solutions, with a tournament size of $N_{tournament}$ to produce $N_{offspring}$ (lines 18). This is followed by the Maximal Preservative Crossover (MPX) operator, which returns a new solution by combining the parents (line 19). The hill climbing function (Algorithm 2) is then applied to the new solution with a probability R_{LS} (lines 22-24). If the new solution does not already exists in the offspring population Q, it is included. Otherwise, a new solution is generated randomly (lines 25-31).

Once the offspring is generated, it is combined with the population and the best solutions are kept for further improvement (lines 33-34). The algorithm stops when the maximum number of evaluations T_{max} is reached, or a near-optimal solution (solution with a gap to optimal smaller than ϵ) is found (line 35).

The parameters of Algorithm 1 are summarised in Table 2. The fine tuning of the parameters is done taking into account the size of the instances used for the analysis, with values chosen empirically based on a random sample of instances (diversified based on the features in Section 4, and on previous studies [9, 12]. This is ensure that the algorithm find a near-optimal solution in most cases, which is important to conduct the analysis.

Table 2: Memetic Algorithm parameters

Parameter	Description	Value
T_{max}	Maximum number of evaluations	100000
N_{pop}	Population size	6
$N_{offspring}$	Offspring size	4
$N_{tournament}$	Tournament size	3
R_{LS}	Local search probability	0.1
ϵ	ϵ-approximation	0.01

3.3 Estimation of the Expected Runtime (*ert*)

One way to measure the performance of an algorithm $\mathcal{A}$ (search cost), is consider the expected number of function evaluations necessary to achieve an ϵ-approximation. To achieve this, we save the number of function evaluations until an ϵ-approximation is found which characterises a "success". Otherwise, the search cost is set to a predetermined maximum T_{max}. This approach is similar to the one presented by Hansen et al [15], Liefooghe et al [23].

In this approach, we denote $p_s \in [0, 1]$ as the probability of success of algorithm $\mathcal{A}$, and T_f as the random variable measuring the number of function evaluations for unsuccessful runs (failures). After $(t - 1)$ failures, each one requiring T_f evaluations, and the final successful run of T_s evaluations, the total runtime is $T = \sum_{i=1}^{t-1} T_f + T_s$, where t is the random variable representing the number of runs. t follows a geometric distribution with parameter p_s. Equation 8 takes the expectation by considering independent runs for each instance, stopping at the first success:

$$\mathbb{E}[T] = (\mathbb{E}[t] - 1)\mathbb{E}[T_f] + \mathbb{E}[T_s] \tag{8}$$

Here, the estimated success rate ($\hat{p}_s$) is computed by the ratio of successful runs over the total number of executions. The expectation of a geometric distribution for t with parameter p_s is equal to $1/p_s$. The expected runtime for unsuccessful runs $\mathbb{E}[T_f]$ is set as a constant limit (T_{max}) on the number of function evaluation calls, and the expected runtime for successful runs $\mathbb{E}[T_s]$ is estimated as the average number of function evaluations performed by successful runs. Given these assumptions, *ert* can be expressed as an estimation of the expected runtime $\mathbb{E}[T]$ as presented in Equation 9.

$$ert = \frac{1 - \hat{p}_s}{\hat{p}_s} T_{max} + \frac{1}{t_s} \sum_{i=1}^{t_s} T_i \tag{9}$$

where t_s is the number of successful runs, T_i is the number of evaluations for successful run i.

4 Experiments, Results and Discussion

4.1 Experimental setting

The experimental framework consists of generating enumerable instances based on the problem features of interest, then running the algorithm and modelling the expected runtime (ert) [2].

The instance generator considers the following features:

- **Number of cities (n):** Represents the number of cities to visit. This feature is not considered as an input to the regression model. Instead, the experiments are replicated for multiples values of $n \in \{5, 6, 7, 8\}$. Small values of n are chosen in order to be able to enumerate the solutions, which is used to derive the optimal values, as there is no exact methods able to solve the problem. Note that we will exclude some illustrations for $n = 5, 6, 7$ for the sake of simplicity, but result summaries will be given for all the values of n.
- **Dropping rate ($\mathcal{D}_r$)**: Represents the dropping rate at which the value of an item decreases through time as shown in Equation 2. $\mathcal{D}_r$ takes values from the set $\{0.7, 0.75, 0.8, \ldots, 1\}$.
- **Minimum velocity (v_{min})** is the minimum speed of the thief following Equation 1. v_{min} takes values from $\{0.1, 0.2, \ldots, 1\}$.
- **Profit-weight correlation ($\mathcal{T}$):** Defines the correlation between the weight (w_i) and profit (p_i) of each item. Three correlations have been defined in the TTP standard library [31]. We generate random weights and profits for each correlation type as follow:
  - *Uncorrelated* (0): $p_i \sim \mathcal{U}(10, 1000)$ and $w_i \sim \mathcal{U}(10, 1000)$,
  - *Uncorrelated with similar weight* (1): $p_i \sim \mathcal{U}(10, 1000)$ and $w_i \sim \mathcal{U}(1000, 1010)$, and
  - *Bounded strongly correlated* (2): $p_i \sim \mathcal{U}(10, 1000)$ and $w_i = p_i + 100$.

  As this feature can be considered an ordinal variable, numerical values (between parentheses) are assigned to represent the correlation strength, which will be useful for the regression analysis.
- **Knapsack capacity class ($\mathcal{C}$):** Takes values from $\{3, \ldots, 10\}$. $\mathcal{C}$ is a factor occurring in the maximum weight of the knapsack which is given in Equation 10.

$$W = \frac{\mathcal{C}}{11} \sum_{x=2}^{n} \sum_{y=1}^{\mathcal{F}} w_{xy} \tag{10}$$

  Note that values lower than 3 have been excluded as they can result in capacities smaller than the smallest item weight.

The features $\mathcal{D}_r$ and v_{min} are considered as they control the strength of the dependencies. While the motivation for choosing $\mathcal{T}$ and $\mathcal{C}$ is to compare their impact on the expected runtime with that of the dependency features.

[2] We use Python3.10 with the statistical learning packages scikit-learn for the statistical analysis and regression models

Furthermore, the choice of $\mathcal{T}$ and $\mathcal{C}$, instead of other features, is based on previous analyses [11, 12].

10 instances are generated for each combination of feature values, resulting in a total of $|n| \times |\mathcal{D}_r| \times |v_{min}| \times |\mathcal{T}| \times |\mathcal{C}| \times 10 = 67,200$ instances. Then, 30 independent runs of Algorithm 1 are performed for each generated instance. Afterwards, the *ert* is calculated for each instance based on Equation 9.

It is worth noting that the experiments result in 23 instances where the algorithm fails to identify an optimal or near-optimal solution, i.e., $\hat{p}_s = 0$, which results in a division-by-zero. To solve the issue, we set $ert = \infty$, and consider these results as outliers for the analysis. Note that, based on a deeper look into the individual results, we concluded that this happens for different categories (combinations of features), i.e., all the categories are covered in the analysis.

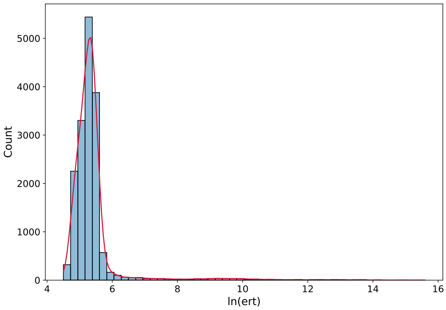
4.2 Results and analysis

4.2.1 Preliminary analysis of the runtime

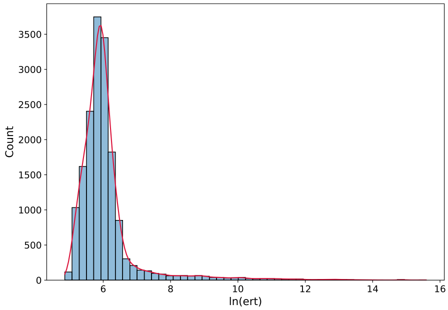
The histograms in Figure 2 shows the distributions of $ln(ert)$, where $ln(.)$ denotes the natural logarithm function for the different values of n. The natural logarithm is only used to better visualise the *ert* data. Indeed, *ert* follows a distribution that is heavy-tailed (with a very high kurtosis $kurt[ert] = 1390.38$ for $n = 8$) and asymmetric (with a skewness $\mu_3 = 32.51$ for $n = 8$).

The nature of *ert* makes it virtually impossible to use standard linear regression models. The most significant factor is the existence of outliers (large *ert* values) – representing hard instances – which are difficult to properly include in the regression model. Note that the existence of heavy tails in the dependent variable is not a problem in itself, except when it leads to heavy-tailed residuals, which is the case here (based on experiments with linear regression and regression tree models).

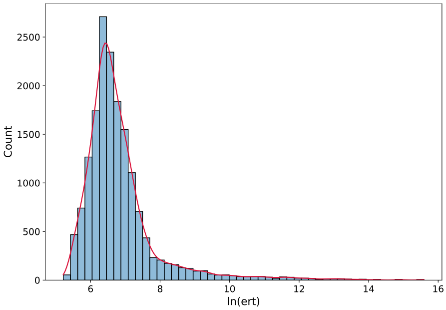
It is usually favourable for the observed residuals to be approximately normally distributed. In order to obtain (approximately) normal residuals, a transformation, such as the logarithm, square root or cubic root, should be applied to the *ert*. While the mentioned transformations can be efficient in taming outliers in many cases, the resulting residuals, even when using regression trees and other regression models, remain heavy-tailed due to the high variation is the *ert* values. A better alternative in this case is to use the reciprocal, $\frac{1}{ert}$, which results in a non-normal, but smaller-tailed distribution of the residuals for a regression tree as shown in Figure 3. Furthermore, in order to preserve the *ert* orders and improve the illustrations, we apply the transformation $1 - \frac{min(ert)}{ert}$ instead, which does not impact the regression analysis results. We will refer to the resulting values as the *runtime scores*.



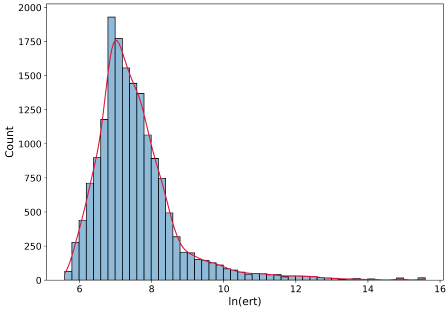
(a) $n = 5$



(b) $n = 6$



(c) $n = 7$



(d) $n = 8$

Figure 2: Histogram of expected runtimes in logarithmic scale

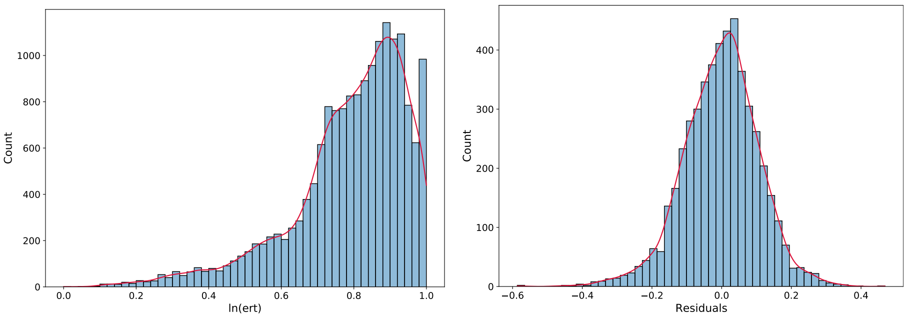


Figure 3: Histogram of the runtime scores (left) and residuals using a regression tree (right) for $n = 8$

It is worth noting that $\frac{1}{ert}$ should not be interpreted as the expected rate. Indeed, as the reciprocal is a convex function, we have:

$$E[rate] = E\left[\frac{1}{runtime}\right] \leq \frac{1}{E[runtime]} = \frac{1}{ert}$$

based on Jensen's inequality [19]. Furthermore, as the distribution of *ert* is unknown and difficult to fit, it is hard to deduce $E[rate]$ using the law of the unconscious statistician. Therefore, $\frac{1}{ert}$ can be simply interpreted as the *ert* reciprocal.

As linear models were not efficient in capturing the variability of the data, we utilise regression trees for two main reasons: (1) they are efficient in handling larger data with outliers, (2) they can produce explainable outcomes, which is important to understand the features' impact on the expected runtime and identify what makes some TTP instances harder to solve. In other words, regression trees offer a better trade-off between efficiency and explainability compared to other alternatives.

4.2.2 Regression analysis

In order to analyse the impact of the problem features on the difficulty of tackling the instances, one must map the problem features to the expected runtime to identify near-optimal solutions. This can be achieved using non-linear regression analysis. Specifically, we consider regression trees using the Mean Squared Error (MSE) to measure the quality of split, and two parameters to control the model's complexity. The first is the maximum depth, which represents the maximum number of node splits the regression tree makes before returning a prediction. The second is the minimum sample size, which represents the minimum number of samples at each leaf. We denote by $RT^n_{d_{max},s_{min}}$ the regression tree [3] obtained for problem size n, by using the parameter values d_{max} as the maximum depth and s_{min} as the minimum sample size, during the training phase.

Model tuning is achieved using a 5-fold cross-validation, and based on a grid search with different values of d_{max} and s_{min}. Figure 4 shows a heat-map of the trade-off between the regression tree complexity and the model fitness for $n = 8$. A maximum R^2 of 0.53 can be reached using the regression model $RT^8_{9,40}$, i.e., the corresponding models can explain 53% of the variation in expected runtime that is predictable from the problem features. The remaining 47% of the variation can be attributed to many other aspects, the most likely is the stochasticity of the considered Memetic Algorithm, which leads to high variation in the *ert* within some instance categories (combination of features), and high variation in the *number of evaluations* for the same instances. Another, less likely scenario is that there are combinations of features that the regression tree was not able to identify. This is believed to have minimum impact as other machine learning algorithms (random forest and neural networks) were investigated and could not result in a better coefficient of determination.

Naturally, more complex trees can result in higher R^2 values. Nevertheless, this comes with the cost of losing the ability to interpret the model. For the purpose of this analysis, the goal is to understand the impact of the feature combinations on the expected runtime. Therefore, we favour small explainable

[3] The implementation is done in Python 3.8.10 using scikit-learn 1.1.0

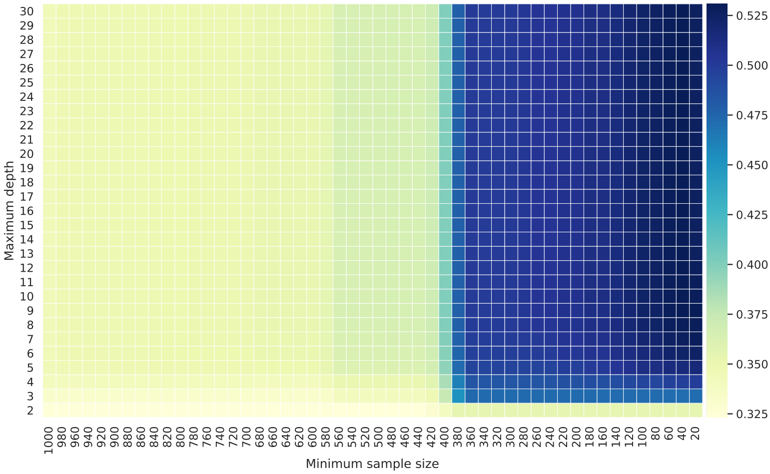


Figure 4: Regression tree complexity vs. coefficient of determination (R^2) for $n = 8$. The complexity is represented by the maximum depth, shown in the y-axis, and the minimum sample size per leaf, shown in the x-axis; while R^2 values are illustrated in the colour bar

trees, even if they explain a lower percentage of variability between the features and the expected runtime.

Based on the above, we consider two regression models for each n. $RT^5_{6,280}$, $RT^6_{6,20}$, $RT^7_{7,20}$, and $RT^8_{9,40}$ are the models with the highest accuracy; while $RT^5_{3,380}$, $RT^6_{3,380}$, $RT^7_{3,380}$, and $RT^8_{3,380}$ represent a good trade-off between the tree complexity and quality-of-fit for all n values. The evaluation metrics of the two resulting models are shown in Table 3 for each value of n, where the first model corresponds to the explainable one, and the second corresponds to the one with the highest coefficient of determination.

Table 3: Evaluation metrics of the obtained regression trees. MAE is the mean absolute error, MSE is the mean squared error, RMSE is the root mean squared error, and R^2 is the coefficient of determination

n	**5**		**6**		**7**		**8**	
Model	$RT^5_{3,380}$	$RT^5_{6,280}$	$RT^6_{3,380}$	$RT^6_{6,20}$	$RT^7_{3,380}$	$RT^7_{7,20}$	$RT^8_{3,380}$	$RT^8_{9,40}$
MAE	0.08	0.08	0.09	0.09	0.1	0.1	0.09	0.08
MSE	0.12	0.12	0.12	0.12	0.13	0.12	0.11	0.11
RMSE	0.02	0.01	0.02	0.01	0.02	0.02	0.01	0.01
R^2	0.37	0.4	0.45	0.48	0.4	0.45	0.5	0.55

In general, Table 3 shows that the loss in quality between the simple and complex models, in terms of error metrics and R^2, is minimal and can be neglected due to the gain in explainability. Nevertheless, a difference in the performances can be noticed between the different models for each n value. Looking back at the different *ert* distributions, this can be explained by the fact that as n grows, the variance in *ert* decreases.

On the one hand, the complex regression trees ($RT^5_{6,280}$, $RT^6_{6,20}$, $RT^7_{7,20}$ and $RT^8_{9,40}$) can provide useful global insights on the importance of features and to what extent we can explain the variability in the runtime variable, but it could be less practical to extract which combination of feature values lead to a specific expected runtime. On the other hand, the explainable regression trees ($RT^5_{3,380}$, $RT^6_{3,380}$, $RT^7_{3,380}$ and $RT^8_{3,380}$) result in a competitive quality while having the additional benefit of being explainable, allowing us to draw useful conclusions on what makes instances easier or harder to solve.

Table 4: Feature importance based on all regression tree models

n	**5**		**6**		**7**		**8**	
Model	$RT^5_{3,380}$	$RT^5_{6,280}$	$RT^6_{3,380}$	$RT^6_{6,20}$	$RT^7_{3,380}$	$RT^7_{7,20}$	$RT^8_{3,380}$	$RT^8_{9,40}$
$\mathcal{T}$	0.13904	0.13409	0.13814	0.13800	0.17611	0.18711	0.27582	0.27084
$\mathcal{C}$	0.79609	0.79498	0.72660	0.67902	0.60393	0.54678	0.47955	0.44658
$\mathcal{D}_r$	0.05560	0.05555	0.12460	0.13655	0.20094	0.19975	0.22137	0.22135
v_{min}	0.00927	0.01537	0.01066	0.04643	0.01902	0.06636	0.02327	0.06123

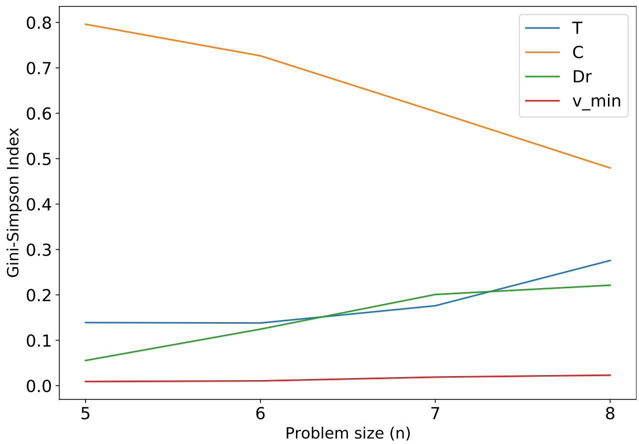


Figure 5: Feature importance versus the problem size (n)

Table 4 provides a macroscopic idea on the impact of the problem features using the Gini-Simpson Index [20]. The feature importance values vary between the two models, but are aligned in sense that they rank features similarly. For this holistic investigation, we focus on the models having the better quality-of-fit.

The results show that the capacity class ($\mathcal{C}$) is the most influential feature contributing to the difficulty of instances. This is followed by the dropping rate ($\mathcal{D}_r$) and profit-weight correlation ($\mathcal{T}$), while the minimum velocity (V_{min}) has a much lower impact index. Comparing the two dependency features shows that items losing value with time is highly important to predict the efficiency of the algorithm in tackling the problem. However, the change of velocity of the thief based on the knapsack load seems to have minimum impact on predicting the performance of the algorithm when tackling the problem.

Figure 5 shows the evolution of the feature importance indices in terms of n. The dependency features tend to have a higher importance index as the problem size increases, this phenomenon is stronger for D_r compared to v_{min}. The same can be said about the feature T. On the other hand, the importance index for C is decreasing significantly.

While the above analysis can help in identifying important features (and potentially eliminating less important ones), a more in-depth analysis should be done at a microscopic scale, which can be achieved using $RT^{n}_{3,380}$.

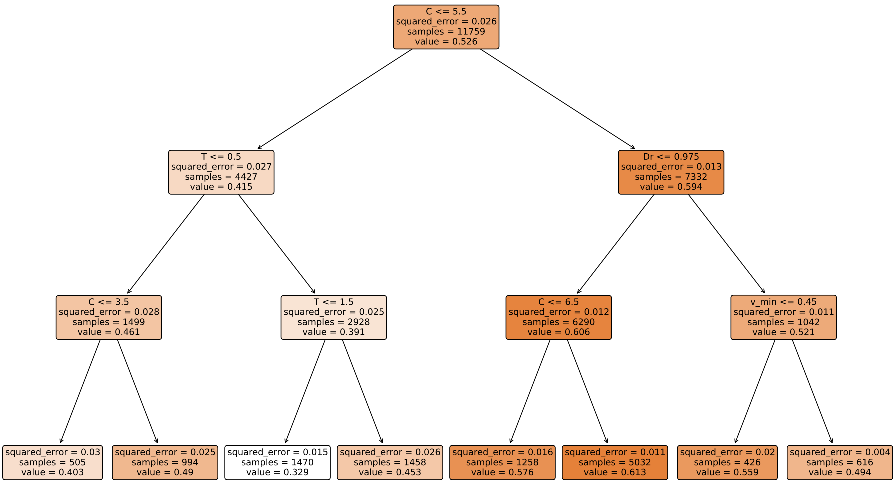


Figure 6: Regression tree $RT^{5}_{3,380}$. Colour brightness represents the difficulty of instance sets, where darker colour represents harder instances.

Figures 6, 7, 8, and 9 show the resulting regression tree models $\{RT^{n}_{3,380}\}, n \in \{5, \ldots, 8\}$, which will help us compare the regression logic across different instance sizes. Additionally, in Appendix A we reproduce the regression models for $n = 6, 7$ to show that the results can be generalised, i.e., they are not dependent on the generated instances, but on the problem features.

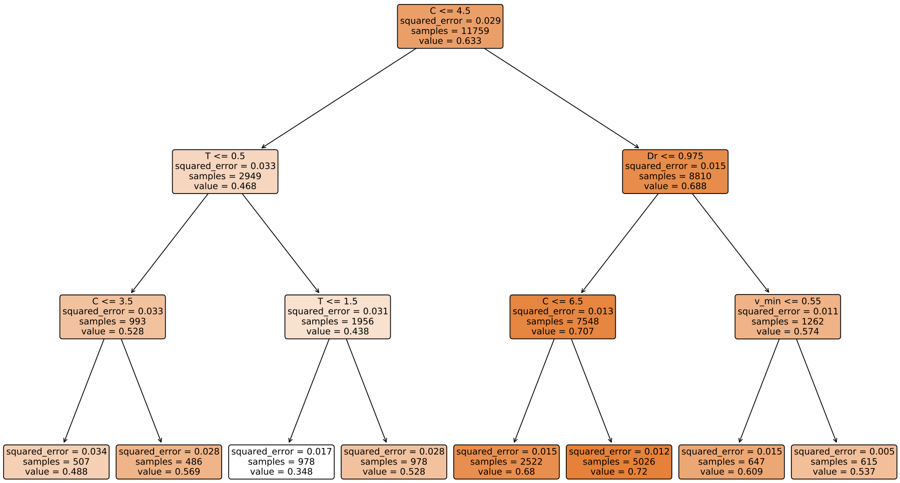


Figure 7: Regression tree $RT^{6}_{3,380}$. Colour brightness represents the difficulty of instance sets, where darker colour represents harder instances.

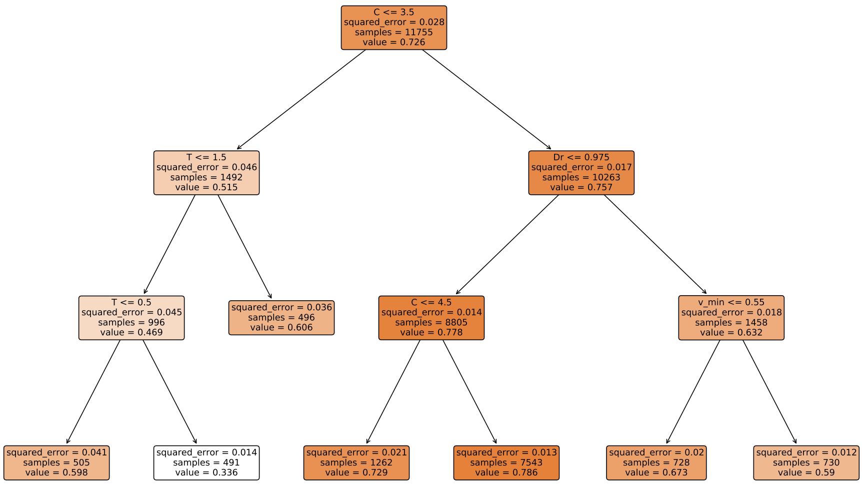


Figure 8: Regression tree $RT^{7}_{3,380}$. Colour brightness represents the difficulty of instance sets, where darker colour represents harder instances.

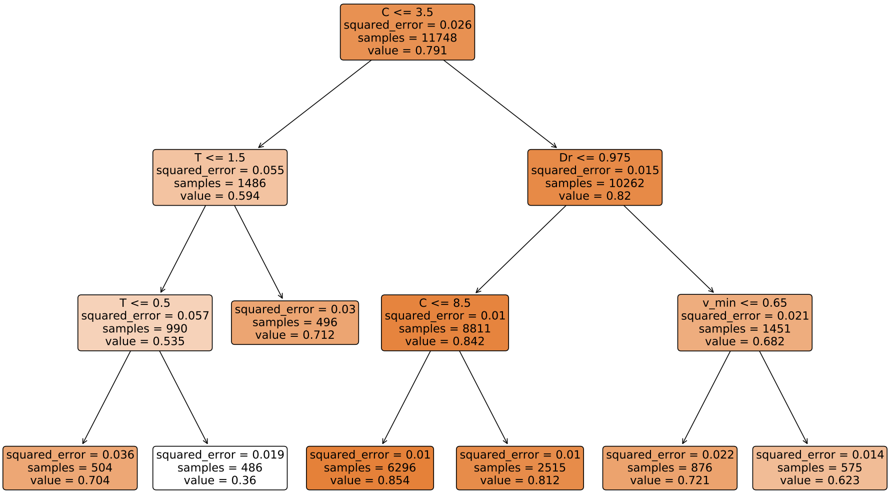


Figure 9: Regression tree $RT^8_{3,380}$. Colour brightness represents the difficulty of instance sets, where darker colour represents harder instances.

Based on this, we observe that the regression models follow a similar logic which can be clearly seen in the branching nodes. Small deviations exist in the branching nodes and predicted values. Looking at all the four models, we can report the following general observations and explanations:

- Setting $\mathcal{D}_r = 1$ results in the standard TTP model as the dropping rate dependency is deactivated, making the minimum velocity feature the main separator as it represents the velocity drop dependency. Furthermore, the lower minimum velocities (v_{min}) result in slightly harder instances.
- $\mathcal{D}_r$ has a clear impact on the difficulty of instances. Lower dropping rate values result in more difficult instances.
- Larger knapsack capacities ($\mathcal{C}$) result in harder instances, as clearly seen in branches B_2, B_4, B_5 and B_6. This is in contradiction with the findings in [11, 12] where the difficulty of instances is associated with the size of basins of attraction, in the context of a local search algorithm applied to the standard TTP. Further experiments considering only standard TTP instances ($\mathcal{D}_r = 1$) show a similar behaviour, confirming that larger knapsack capacities increase the difficulty of the instances. This is suggesting that the contradiction is due to the nature of the search algorithm, not to the complexity added by considering the dropping rate dependency. Indeed, the local optima network analysis is only suitable for embedded neighbourhood search algorithms; and the conclusions cannot be extrapolated to more sophisticated algorithms such as the considered evolutionary algorithms.

- For smaller capacities, the profit-value correlation ($\mathcal{T}$) has a significant impact on the problem difficulty. The hardest instances are the ones where the profit and weight are bounded and strongly correlated ($T = 2$) and the ones with no correlation ($T = 0$), while the instances with similar weights ($T = 1$) are the easiest to tackle.

It is interesting to see how the item value drop dependency results in significant gaps in the performance of the algorithm. When it is deactivated (standard TTP), the instances can be solved relatively fast, and the velocity change dependency has only a small impact on the runtimes. When it is activated, the impact of the velocity change dependency is dominated by the capacity and correlation features.

A possible explanation of the stronger impact of $\mathcal{D}_r$ compared to v_{min} can be attributed to the formulation of dependencies. Specifically, in the dropping rate dependency (see Equation 2), the updated item values are dependent on both the previous item value and the time the item has been carried. On the other hand, in the velocity change dependency (see Equation 1, the thief's updated velocity only depend on the knapsack load, and its previous velocity is completely omitted.

These findings show that the impact of different dependency relationships can differ significantly when formulating a problem with multiple components. In our case, the analysis shows that adding the item value drop dependency makes the TTP a much more challenging problem compared to the standard model, leading to harder instances, which is reflected by the expected runtime. Therefore, this aspect should be given more attention when investigated the TTP in particular, or other capacitated routing problems in general, especially because the impact of these features evolves based on the problem size.

Recognising that different dependencies can have a significantly different impact on the problem's difficulty is important. However, one can go beyond and seeks ways to use these results to improve the way these problems are tackled.

As it is possible to estimate how much time would be needed to find a good approximation given a specific problem instance, based only on known features. One way to directly use the results reported here is to make an informed and automated decision on the maximum number of iterations (or time budget) needed to achieve near-optimality, given a specific problem instance. It is also possible to revisit the model and simplify it to reduce the impact of particular dependencies. This should be done carefully as it can lead to undesirable outcomes due to model oversimplification. Another possibility is to use these results as domain knowledge included in the solution methods. Such an approach can be clustering instances into categories that can be tackled with different approaches based on their difficulty, or tuning the heuristic parameters based on these instances clusters.

5 Conclusion

In this paper, we investigated the difficulty of the Travelling Thief Problem by considering two types of dependency on the fitness landscape using performance prediction models (regression analysis). We introduce a tunable model allowing to control the velocity change and item value drop dependencies using associated problem features. The impact of these features on the expected runtime is then evaluated to understand how these features impact the search landscape for a Memetic Algorithm.

This study allowed us to better understand what makes instances harder, and gave us new insights on the impact of the problem features. The analysis shows that the inclusion of the item value drop dependency leads to harder instances, which is reflected by the expected runtime. The results also showed how the impact of these features evolves based on the problem size. In particular, the dropping rate tends to be more important as the problem grows and its impact is stronger than the velocity change constraint.

A continuation of this study is to map these features to the choice of algorithm parameter. Besides, as future work, we intend to explore other fitness landscape techniques such as fitness cloud, auto-correlation, time to local optimum, distance to global optimum, information analysis: entropy, information stability, partial information content and density basin information.

Declarations

Compliance with Ethical Standards

None of the authors of this paper have a financial or personal relationship with other people or organisations that could inappropriately influence or bias the content of the paper. This paper does not contain any studies with human participants or animals performed by any of the authors. This manuscript is the authors' original work and has not been published nor has it been submitted simultaneously elsewhere. All authors have checked the manuscript and have agreed to the submission.

Funding and Competing interests

No funding was received to assist with the preparation of this manuscript. Besides, the authors have no competing interests to declare that are relevant to the content of this article.

Appendix A Additional results

In this appendix, we show the results for additional experiments on different sets of instances for $n = 6$ and $n = 7$. The same process defined in the earlier sections was used to create them, but just different seeds of the random number general were used. The goal is to show that different sets of instances generate roughly the same regression model, i.e., regression trees with a similar logic as shown in Figure A1.

When comparing these two with the corresponding trees in Figures 7 and 8, we can see that the conditions at the inner nodes are almost always identical (i.e. for 10 of 13 inner nodes) or very similar, and the respective errors (shown in Table A1) and sample numbers are very close matches, too.

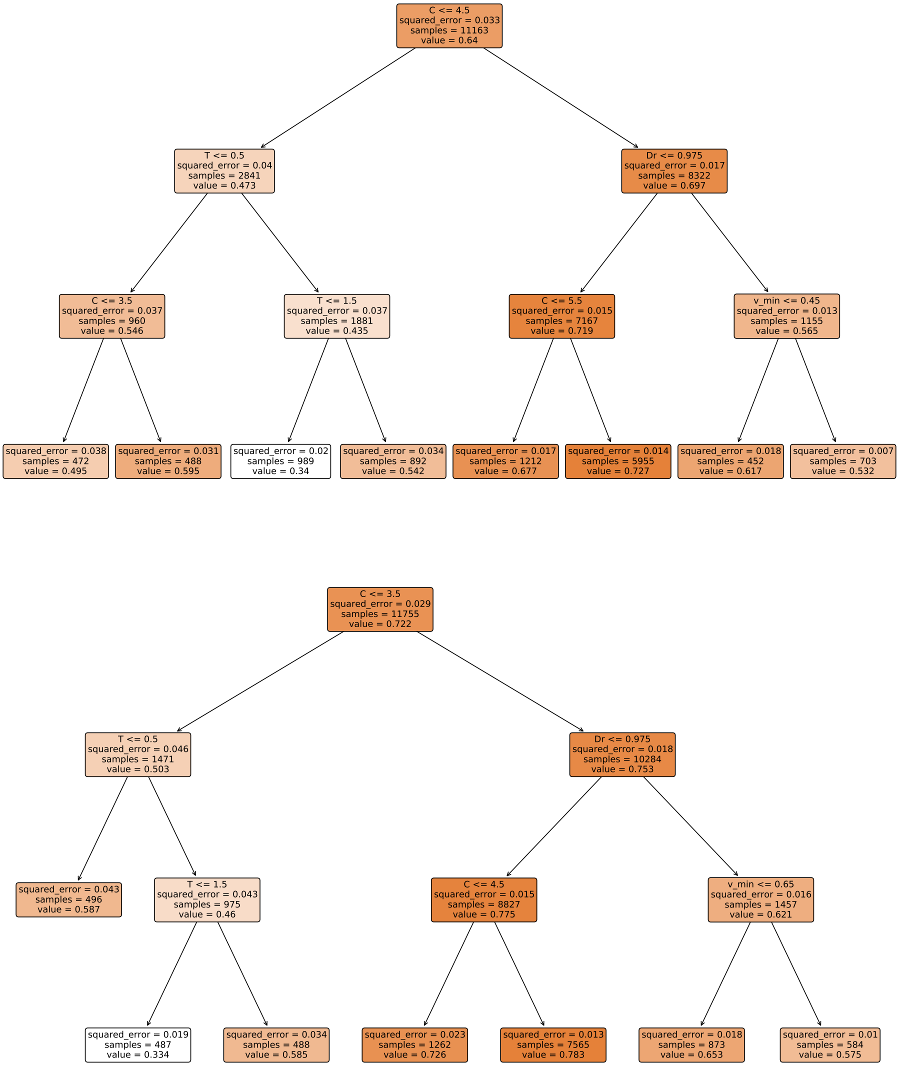


Figure A1: Regression tree $RT^{6*}_{3,380}$ for a new dataset with $n = 6$ (top) and $RT^{7*}_{3,380}$ with $n = 7$ (bottom).

Table A1: Evaluation metrics of the regression trees.

Model	$RT^{6*}_{3,380}$	$RT^{7*}_{3,380}$
MAE	0.1	0.1
MSE	0.13	0.13
RMSE	0.02	0.02
R^2	0.45	0.4

Hence, we conclude that even though the methodology is based on randomly created instances and even though it employs a memetic algorithm as a randomised search heuristic, the achievable insights at a high level (i.e. when reasoning about the effects of dependencies) are unaffected.

References

[1] Applegate D, Cook W, Rohe A (2003) Chained lin-kernighan for large traveling salesman problems. INFORMS Journal on Computing 15(1):82–92

[2] Bonyadi MR, Michalewicz Z, Barone L (2013) The travelling thief problem: The first step in the transition from theoretical problems to realistic problems. In: 2013 IEEE Congress on Evolutionary Computation, IEEE

[3] Bonyadi MR, Michalewicz Z, Przybylek MR, et al (2014) Socially inspired algorithms for the travelling thief problem. In: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation

[4] Bonyadi MR, Michalewicz Z, Wagner M, et al (2019) Evolutionary computation for multicomponent problems: opportunities and future directions. In: Optimization in Industry. Springer

[5] Chand S, Wagner M (2016) Fast heuristics for the multiple traveling thieves problem. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, pp 293–300

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

[7] Darestani SA, Hemmati M (2019) Robust optimization of a bi-objective closed-loop supply chain network for perishable goods considering queue system. Computers & industrial engineering 136:277–292

[8] Eiben AE, Smith JE (2015) Introduction to evolutionary computing. Springer

[9] El Yafrani M, Ahiod B (2016) Population-based vs. single-solution heuristics for the travelling thief problem. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016

[10] El Yafrani M, Ahiod B (2018) Efficiently solving the traveling thief problem using hill climbing and simulated annealing. Information Sciences 432

[11] El Yafrani M, Martins MS, Krari ME, et al (2018) A fitness landscape analysis of the travelling thief problem. In: Proceedings of the Genetic and Evolutionary Computation Conference

[12] El Yafrani M, Scoczynski M, Delgado M, et al (2022) On the fitness landscapes of interdependency models in the travelling thief problem. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion

[13] Faulkner H, Polyakovskiy S, Schultz T, et al (2015) Approximate approaches to the traveling thief problem. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation

[14] Ghomi SF, Asgarian B (2019) Development of metaheuristics to solve a transportation inventory location routing problem considering lost sale for perishable goods. Journal of Modelling in Management

[15] Hansen N, Auger A, Ros R, et al (2010) Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009. In: Proceedings of the 12th annual conference companion on Genetic and evolutionary computation, pp 1689–1696

[16] Işıklı E, Aydın N, Bilgili L, et al (2020) Estimating fuel consumption in maritime transport. Journal of Cleaner Production 275:124,142

[17] Janardhanan M, Li Z, Nielsen P (2019) Model and migrating birds optimization algorithm for two-sided assembly line worker assignment and balancing problem. Soft Computing 23(21):11,263–11,276. https://doi.org/10.1007/s00500-018-03684-8

[18] Janković A, Doerr C (2019) Adaptive landscape analysis. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp 2032–2035

[19] Jensen JLWV (1906) Sur les fonctions convexes et les inégalités entre les valeurs moyennes. Acta mathematica 30(1):175–193

[20] Jost L (2006) Entropy and diversity. Oikos 113(2):363–375

[21] Li W, Meng X, Huang Y (2021) Fitness distance correlation and mixed search strategy for differential evolution. Neurocomputing 458:514–525

[22] Liefooghe A, Verel S, Aguirre H, et al (2013) What makes an instance difficult for black-box 0–1 evolutionary multiobjective optimizers? In: International Conference on Artificial Evolution (Evolution Artificielle), Springer, pp 3–15

[23] Liefooghe A, Verel S, Daolio F, et al (2015) A feature-based performance analysis in evolutionary multiobjective optimization. In: International Conference on Evolutionary Multi-Criterion Optimization, Springer, pp 95–109

[24] Lu H, Zhou R, Fei Z, et al (2019) Spatial-domain fitness landscape analysis for combinatorial optimization. Information Sciences 472:126–144

[25] Martins MS, El Yafrani M, Santana R, et al (2018) On the performance of multi-objective estimation of distribution algorithms for combinatorial problems. In: 2018 IEEE congress on evolutionary computation (CEC), IEEE, pp 1–8

[26] Mei Y, Li X, Yao X (2014) Improving efficiency of heuristics for the large scale traveling thief problem. In: Asia-Pacific Conference on Simulated Evolution and Learning, Springer

[27] Merz P, Freisleben B (2000) Fitness landscape analysis and memetic algorithms for the quadratic assignment problem. IEEE transactions on evolutionary computation 4(4):337–352

[28] Michalewicz Z, Fogel DB (2013) How to solve it: modern heuristics. Springer Science & Business Media

[29] Nikfarjam A, Neumann A, Neumann F (2022) On the use of quality diversity algorithms for the traveling thief problem. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp 260–268

[30] Ochoa G, Malan K (2019) Recent advances in fitness landscape analysis. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp 1077–1094

[31] Polyakovskiy S, Bonyadi MR, Wagner M, et al (2014) A comprehensive benchmark set and heuristics for the traveling thief problem. In: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation

[32] Przybylek MR, Wierzbicki A, Michalewicz Z (2018) Decomposition algorithms for a multi-hard problem. Evolutionary computation 26(3)

[33] Rahman H, Nielsen I (2019) Scheduling automated transport vehicles for material distribution systems. Applied Soft Computing Journal 82. https://doi.org/10.1016/j.asoc.2019.105552

[34] Rahman H, Janardhanan M, Nielsen P (2020) An integrated approach for line balancing and agv scheduling towards smart assembly systems. Assembly Automation 40(2):219–234. https://doi.org/10.1108/AA-03-2019-0057

[35] Richter H (2008) Coupled map lattices as spatio-temporal fitness functions: Landscape measures and evolutionary optimization. Physica D: Nonlinear Phenomena 237(2):167–186

[36] Richter H (2013) Dynamic fitness landscape analysis. In: Evolutionary computation for dynamic optimization problems, Springer, pp 269–297

[37] Sung I, Nam H, Lee T (2013) Scheduling algorithms for mobile harbor: An extended m-parallel machine problem. International Journal of Industrial Engineering : Theory Applications and Practice 20(1-2):211–224

[38] Wagner M (2016) Stealing items more efficiently with ants: a swarm intelligence approach to the travelling thief problem. In: International Conference on Swarm Intelligence, Springer

[39] Wagner M, Lindauer M, Mısır M, et al (2018) A case study of algorithm selection for the traveling thief problem. Journal of Heuristics 24(3):295–320

[40] Wang Jj, Wang L (2022) A cooperative memetic algorithm with feedback for the energy-aware distributed flow-shops with flexible assembly scheduling. Computers & Industrial Engineering 168:108,126

[41] Watson JP (2010) An introduction to fitness landscape analysis and cost models for local search. In: Handbook of metaheuristics. Springer

[42] Wu J, Polyakovskiy S, Neumann F (2016) On the impact of the renting rate for the unconstrained nonlinear knapsack problem. In: Genetic and Evolutionary Computation Conference, ACM

[43] Wuijts RH, Thierens D (2019) Investigation of the traveling thief problem. In: Proceedings of the Genetic and Evolutionary Computation Conference

[44] Zou F, Chen D, Liu H, et al (2022) A survey of fitness landscape analysis for optimization. Neurocomputing 503:129–139