Socially Inspired Algorithms for the Traveling Thief Problem

Mohammad Reza Bonyadi
School of Computer Science
The University of Adelaide
Adelaide, Australia
mrbonyadi@cs.adelaide.edu.au

Zbigniew Michalewicz
School of Computer Science
The University of Adelaide
Adelaide, Australia
zbyszek@cs.adelaide.edu.au

Michał Roman Przybyłek
Polish-Japanese Institute of Information Technology
Warsaw, Poland
mrp@pjwstk.edu.pl

Adam Wierzbicki
Polish-Japanese Institute of Information Technology
Warsaw, Poland
adamw@pjwstk.edu.pl

ABSTRACT

Many real-world problems are composed of two or more problems that are interdependent on each other. The interaction of such problems usually is quite complex and solving each problem separately cannot guarantee the optimal solution for the overall multi-component problem. In this paper we experiment with one particular 2-component problem, namely the Traveling Thief Problem (TTP). TTP is composed of the Traveling Salesman Problem (TSP) and the Knapsack Problem (KP). We investigate two heuristic methods to deal with TTP. In the first approach we decompose TTP into two sub-problems, solve them by separate modules/algorithms (that communicate with each other), and combine the solutions to obtain an overall approximated solution to TTP (this method is called CoSolver). The second approach is a simple heuristic (called density-based heuristic, DH) method that generates a solution for the TSP component first (a version of Lin-Kernighan algorithm is used) and then, based on the fixed solution for the TSP component found, it generates a solution for the KP component (associated with the given TTP). In fact, this heuristic ignores the interdependency between sub-problems and tries to solve the sub-problems sequentially. These two methods are applied to some generated TTP instances of different sizes. Our comparisons show that CoSolver outperforms DH specially in large instances.

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2576768.2598367.

Categories and Subject Descriptors

G.1.6 [**Optimization**]: Constrained optimization
; I.2.8 [**Problem Solving, Control Methods, and Search**]: Scheduling

General Terms

Algorithms

Keywords

Traveling Thief Problem, Co-evolution, Heuristics, Metaheuristics, Multi-objective optimization, Non-separable problems, Real-world optimization problems

1. INTRODUCTION

In contemporary business enterprises the complexity of real-world problems has to be perceived as one of the greatest obstacles in achieving effectiveness. Even relatively small companies are frequently confronted — in their daily routines — with problems of very high complexity. There have been some studies to investigate sources of complexity (hardness) in real-world combinatorial optimization problems [9, 23, 15, 16]. For example, in [23] several reasons were identified that make optimization problems hard to solve. However, the reasons listed and discussed in that article (premature convergence, robustness, etc) were more related to the issues of the solvers rather than the hardness of the problems. Also, in [16], five main reasons behind the hardness of real-world problems were discussed: the size of the problem, modeling issues, noise, constraints, and some psychological pressures on the designer when problems are big. In [15], real-world problems were categorized into two groups: (1) design/static problems, and (2) operational/dynamic problems. The first category includes a variety of problems, such as traveling salesman problem (TSP), Vehicle Routing Problem (VRP), knapsack problem (KP), etc. The second category includes problems presented in the now-a-days industries (e.g. real-world supply chain). The author argued that, although some of the design/static problems (first category)

are really hard and most of the current research has been concentrated on those problems, the problems from the second category (i.e. operational/dynamic problems) are much harder and represent a huge opportunity for Evolutionary Algorithms. It was also stated that the value of addressing the problems in the first category does not have significant influence on solving the problems in the second category [15].

The reason behind the aforementioned minimal influence was investigated in [9] where the authors claimed that the sources of complexity in real-world problems in the first group is somehow different from the ones in the second group. It was argued that most of real-world problems are multi-component problems (several potentially NP-Hard problems interact with each other) while most of the current researches have been concentrated on single component problems (traveling salesman problem, Knapsack problem, job shop scheduling, vehicle routing problem, etc). Also, it was stated that *interdependency* among components in operational/dynamic problems plays a key rule in the complexity of the problems. However, this interdependency is not found in design/static problems [9]. To illustrate the complexity that arises by interdependency in multi-component problems, a new test problem called Traveling Thief Problem (TTP) was introduced [9]. TTP was a composition of two well-known problems, TSP and KP (see section 2.3 for more details).

In this paper, a new algorithm (called CoSolver) for dealing with multi-component problems is proposed. CoSolver mimics the behavior of a corporation that has separate departments that specialize in solving some aspects of a multi-component problem. Each department is good in solving one component, like choosing a route or a schedule. On the other hand, departments need to work together to solve a real problem. They do so by communicating their partial solutions and negotiating with other departments. Also, we propose a simple heuristic (called Density-based Heuristic, DH) that first generates a solution for the TSP component of a given TTP and then it generates a solution for the found TSP so that the objective value is maximized. Note that DH ignores all interdependencies between the components. These two methods are compared through some generated benchmark TTPs. Results show that the performance of CoSolver is better than DH, which means that considering the interdependency between components in solving this multi-component problem (TTP) is beneficial.

The paper has been organized as follows. Section 2 gives some background information about combinatorial optimization problems, multi-component vs single component problems, and the description of the traveling thief problem. In section 3, we propose two methods to solve TTP. Section 4 gives some details for generating instances, and in section 5, the results of application of the proposed methods to the test problems are discussed. Section 6 concludes the paper.

2. BACKGROUND

In this section, some background information about Traveling Salesman Problem (TSP) and Knapsack Problem (KP) are given. Also, the Traveling Thief Problem (TTP) formulation is reviewed.

2.1 Traveling salesman and knapsack problems

Most of the real-world optimization problems are NP-complete[1], which means that they are *verifiable* in polynomial time, and every problem verifiable in polynomial time polynomially reduces to any of these problems. The question whether every polynomial-time-verifiable problem is polynomial-time-*solvable* is an old famous question in theoretical computer science [11, 3]. In particular, there are no known sub-exponential algorithms for any known NP-complete problem.

Two well-studied NP-complete problems in the literature are the Knapsack Problem (KP) and the Traveling Salesman Problem (TSP). In light of the above, there are no known polynomial algorithms to either of these problems.

State-of-the-art algorithm for TSP runs in $2^{n-\Omega(\sqrt{n/\log n})}$ time and exponential space [7], which is a huge improvement over classical algorithm described in [13] running in 2^n time[2]. Not only TSP is hard to solve, it is also hard to *approximate* — approximating TSP with a ratio bounded by a constant is polynomially equivalent to solving TSP. Moreover, almost every restriction of TSP that makes it non-trivial is also NP-complete. There are known, however, better algorithms and approximations in some special cases. For bounded-degree graphs with bounded weights there is an algorithm running in $(2-\epsilon)^n$ time and polynomial space [8]. For Euclidean TSP with bounded double dimension, there is a polynomial-time approximation scheme [6, 5]. The classical Christofides algorithm [10] computes a 1.5-approximation for metric TSP. Variety of heuristics were proposed to solve general TSP — this includes approaches based on genetic algorithms and ant colony optimization [12]. Two best available heuristics base on reformulation of TSP as Mixed Integer Linear Programming [1], and a generalization of k-opt heuristic [2].

Knapsack Problem can be solved in pseudo-polynomial time (i.e. polynomial in the unary coding of natural numbers) by using two obvious dynamic programming approaches: one with respect to weights of items, the other with respect to their profits. Also, the second approach gives rise to a fully-polynomial-time approximation scheme by simple truncation of profits. A survey on methods of solving Knapsack Problem found in Pisinger's PhD dissertation [17].

2.2 Multi vs single component problems

There have been many well-studied combinatorial optimization problems during the past years such as Vehicle Routing problem (VRP) and Capacitate VRP (CVRP) [22], TSP [14], and KP[18]. Most of these problems were inspired by some problems in industries such as transportation industry, mining industry, and supply chains. In fact, an efficient algorithm that is able to solve each of these problems were in a great value because it could be beneficial for some operations in an industry. However, during the past years, the need for problem solving in industries has been changed so that solving these problems have become in a less interest by industrial companies. In fact, none of these problems can be found as a single problem in an industry, so that solving each of these problems (even to optimality) is not that beneficial. This point was clarified in [15] and [9] where the main reason behind this lost of interest was introduced as new industries are after comprehensive solutions which covers the whole

[1]To be precise, the decision counterpart of these problems are NP-complete

[2]Up to a polynomial.

operation and not only a single part of the operation. As an example, international delivery companies are not only after finding the best routes for transportation (a VRP), but also they are after the best time-table for scheduling drivers for those vehicles, scheduling maintenance for the vehicles, and packing of items into vehicles at the same time. Note that all of these sub-problems (components) interdependent from each other, i.e. changing the transportation plan affects the best achievable solution for the scheduling drivers. Thus, it seems that the new industries are after optimization methods for multi-component problems and the methods that can only deal with single-component problems are not in interest.

This point was first investigated in [9] and a benchmark problem called Traveling Thief Problem (TTP) for multi-component optimization problems was introduced. TTP was a composition of two other well-studied problems (TSP and KP). It was shown [9] that, in TTP, solving each sub-problem to optimality does not guarantee locating the overall optimal solution.

2.3 Traveling thief problem

In the Traveling Thief Problem (TTP), there are n cities (i.e. nodes of a graph) together with a function d assigning to every pair i, j of cities a non-negative distance $d(i, j)$ from i to j ($d(i, j)$ plays the role of a weighted edge from a node i to a node j in the graph; $d(i, j) = +\infty$ means that there is no edge from i to j), and m items (each has a profit p_i and a weight w_i) distributed in the cities. The availability of items is given by a function $a_i = \{c_1, \ldots, c_{k_i}\}$, which shows that item i is available at cities $c_1, c_2, \ldots, c_{k_i}$, where c_i belongs to the set of cities $\{1, \ldots, n\}$. It follows from our notation, that k_i copies of item i can be found over all cities. Additionally, we are given the following constants:

- a non-negative real number R — the rent ratio;
- a natural number W — the capacity of "the knapsack";
- two positive real numbers $v_{min} \leq v_{max}$ — the minimal and maximal speed of "the traveler".

The task is to find a complete tour Π visiting each city exactly once (i.e. a permutation Π of the initial segment of n positive natural numbers; abusing notation, we shall also write Π_{n+1} for Π_1; furthermore we assume that the permutation is stable on the first city — i.e. $\Pi_1 = 1$) and pick items from the cities x_i in a way that the total weight of the picked items does not exceed W, and the total profit (formulated in Eq. 1) is maximized.

$$P = \sum_{i=1}^{m} p_i \min(x_i, 1) - R \sum_{i=1}^{n} t_{i,i+1} \tag{1}$$

where P is the total profit, $x_i \in a_i \cup \{0\}$ represents the city that the item i should be picked from (0 refers to not picking the item at all), and $t_{i,j}$ is the time to travel from Π_i to Π_j assuming the weight of all picked items by city Π_i is W_{Π_i}, which is given by the formula:

$$t_{i,j} = \frac{d(\Pi_i, \Pi_j)}{v_{max} - W_{\Pi_i} \frac{v_{max} - v_{min}}{W}} \tag{2}$$

where W_{Π_i} is the total weight of the picked items from cities $\{\Pi_2, \Pi_3, \ldots, \Pi_i\}$ (we assume that items from city $\Pi_1 = 1$ are picked at the end of the tour). Note that, if more items are picked while their total weight is smaller than W, the value of $t_{i,j}$ grows which causes reducing the value of P in Eq. 1. Also, by taking better tours in terms of total distance, some possibly high quality items (items which have a high profit) might only be available at the beginning of the tour and, hence, by picking those items, the travel time increases (items should be carried for a longer time), which causes reduction in the value of P. This shows that the interdependency between the two problems influences the optimum solutions for the whole problem. Also, solving each component in isolation does not necessarily lead to the optimum of the problem [9].

3. PROPOSED ALGORITHMS

In this section, we propose two algorithms to deal with TTP. The first algorithm (called Density-based Heuristic, DH) is a simple heuristic which ignores the interdependency between the components. The second algorithm is a socially inspired algorithm which mimics the behavior of groups of experts in solving multi-component problems in real-world.

3.1 Simple/DH heuristic

We propose a heuristic algorithm, called density-based heuristic, DH, for TTP. DH starts with generating a tour for the TSP (Asymmetric TSP, ATSP) that is as short as possible (according to the distances of the cities) and then, it tries to find the best picking plan for the found tour.

The Chained-Lin-Kernighan [4] method is used to generate TSP (ATSP)[3]. Once the tour Π is fixed, a $\overline{p}_i$ is calculated for each item i that estimates how good i is according to the tour. We propose the following function to calculate $\overline{p}_i$ for each item:

$$\overline{p}_i = p_i - Rt_i \tag{3}$$

where p_i is the value of item i, R is the rent rate, and t_i is given by:

$$t_i = \frac{L_i}{v_{max} - w_i \frac{v_{max} - v_{min}}{W}} \tag{4}$$

In Eq. 4, L_i is the total distance from the city where item i is picked from to the end of the tour, i.e.:

$$L_i = \begin{cases} 0 & \textit{if } i = 1 \\ \sum_{k=i}^{n+1} d(\Pi_k, \Pi_{k+1}) & \textit{if } i \neq 1 \end{cases} \tag{5}$$

In fact, the value of $\overline{p}_i$ is the total profit for the thief if only item i is picked during the whole tour [4]. Note that, to improve the time complexity of the algorithm, the L array is pre-calculated so that it is not recalculated for each item separately. After calculating $\overline{p}_i$ for all items, the items are sorted (descending according to their $\overline{p}_i$ for all i) and processed one by one in the sorted order. For each item i, if i fits into the knapsack and picking item i does not decrease the objective value then the item i is picked, otherwise, next item is processed. Algorithm 1 shows step by step process for DH.

[3]A C++ based implementation for this algorithm can be found in [2].

[4]Note that, in the case of repetitions of an item in different cities, the repetitive items can be assumed as another item and the $\overline{p}_i$ value is calculated for that as a new item

Algorithm 1: Simple Heuristic (SH)

1: $\Pi \leftarrow$ solve TSP (ATSP) with given distances
2: Fill the array L
3: **for** i=1 to m **do**
4: Calculate t_i by using Eq. 4
5: $\overline{p}_i \leftarrow p_i - Rt_i$
6: $sortedItems \leftarrow sort$ items according to $\overline{p}_i$
7: $W_c = 0, pickedItems = \{\}$
8: $P \leftarrow calcObj(pickedItems, \Pi)$
9: **for** i=1 to m **do**
10: **if** $W_c + w_{sortedItems_i} < W$ **then**
11: $P' \leftarrow calcObj(pickedItems \cup \{sortedItems_i\}, \Pi)$
12: **if** $P' > P$ **then**
13: $W_c = W_c + w_{sortedItems_i}$
14: $pickedItems = pickedItems \cup \{sortedItems_i\}$
15: $P \leftarrow P'$
16: **if** $W_c \geq W$ **then**
17: break
18: **return** $pickedItems, \Pi$

Algorithm 2: CoSolver

1: $d_k \leftarrow 0$
2: $W_k \leftarrow 0$
3: $P \leftarrow -\infty$
4: **for** $r \leftarrow 1$ **to** *MaxIter* **do**
5: $pickedItems' \leftarrow$ solve KRP with p_k, w_k, d_k and parameter W_k
6: $W_k \leftarrow \sum_{i \in pickedItems'} w_i[0 < x_i \leq k]$
7: $\Pi' \leftarrow$ solve TSKP with W_k, d
8: $P' \leftarrow calcObj(pickedItems', \Pi')$
9: **if** $P' > P$ **then**
10: $P \leftarrow P'$
11: $\Pi \leftarrow \Pi'$
12: $pickedItems \leftarrow pickedItems'$
13: $d_k \leftarrow d(\Pi_k, \Pi_{k+1})$
14: **else**
15: break
16: **return** $pickedItems, \Pi$

In Algorithm 1, *calcObj* is a function that calculates the objective value according to a given set of items and tour. The time complexity for the algorithm for solving the knapsack part is in $O(nm^2)$. In fact, calculations for array L is in $O(n)$, calculations of $\overline{p}_i$ for all i is in $O(m)$, sorting the $\overline{p}_i$ for all i is in $O(m \log(m))$, and calculating objective value for each item is in $O(mn)$ in the worst case scenario (all items are picked).
A method similar to DH (called Simple Heuristic, SH) has been also proposed in [19]. In SH, it was proven that if picking a particular item i by itself results in worse objective value than picking no item, then picking item i is always disadvantageous if it is picked with any other combination of items. Thus, in SH, a measure u_i was calculated for each item that represented whether item i is disadvantageous. Then, rather than recalculating the objective value, the value of u_i was used in the condition for the *if* statement in line 11 of Algorithm 1. Note that calculating u_i is in $O(n)$, which can be done prior to the for loop in line 8. Thus, the time complexity of SH is in $O(mn)$, which is faster than DH. However, the solutions that SH was generating could still become even worse than the case that no item is picked. Also, the main aim of [19] was to propose a comprehensive benchmark set for TTP.

3.2 CoSolver: A Socially Inspired Algorithm

The main idea behind CoSolver is to decompose TTP into sub-problems, solve the sub-problems separately, with some communication between the sub-problems, and then compose the solutions back to obtain a solution to TTP. The general idea is shown on Fig 1.

We have identified the following sub-problems of TTP — one corresponding to a generalization of TSP, which we shall call Traveling Salesman with Knapsack Problem (TSKP), and another corresponding to a generalization of KP, which we shall call Knapsack on the Route Problem (KRP).

The Traveling Salesman with Knapsack Problem (TSKP) consists of n cities, a distance function d, a positive integer W (the capacity of the knapsack), a non-negative real number R (the rent rate) and a function w assigning to every node $i \in \{1, 2, \ldots, n\}$ the total weight w_i of items picked at i, and two positive real numbers $v_{min} \leq v_{max}$ corresponding to the minimal and the maximal speed of the traveler. The task is to find a complete tour Π visiting each city exactly once, such that the following is minimized:

$$T = -R \sum_{i=1}^{n} t_{i,i+1} \tag{6}$$

where $t_{i,i+1}$ is defined in the same way as in Eq. 2 from Section 2.

The Knapsack on the Route Problem (KRP) consists of n cities, a function d assigning to a city i the cost $d(i)$ of traveling from city i to city $i+1$ for $i < n$ and from city n to city 1 otherwise, and m items (each has a profit p_i and a weight w_i) distributed in the cities. The availability of items is given by a function $a_i = \{c_1, \ldots, c_{k_i}\}$. Additionally, we are given a natural number W (the capacity of the knapsack), a non-negative real number R (the rent ratio) and two positive real numbers $v_{min} \leq v_{max}$ corresponding to the minimal and the maximal speed of the traveler. The task is to find a function assigning to an item i a city $x_i \subseteq a_i \cup \{0\}$ where item i is picked (0 refers to not picking the item at all) such that the collective total weight of items does not exceed W, and the following is maximized:

$$P = \sum_{i=1}^{m} p_i \min(x_i, 1) - R \sum_{i=1}^{n} t_i \tag{7}$$

where:

$$t_i = \frac{d(i)}{v_{max} - W_i \frac{v_{max} - v_{min}}{W}} \tag{8}$$

and W_i is the total weight of the picked items from cities $\{2, \ldots, i\}$ (we assume that items from city 1 are picked at the end of the tour).

An iterative heuristic based on the above observation is presented as Algorithm 2.

Given an instance of TTP, CoSolver starts by creating an instance of KRP that consists of all items from all nodes and distances $d(k)$ equal zero. After finding a solution K' for this instance, it creates an instance of TSKP by assigning to each city a weight equal to the total weights of items picked at the city by KRP. A solution for TTP at the initial

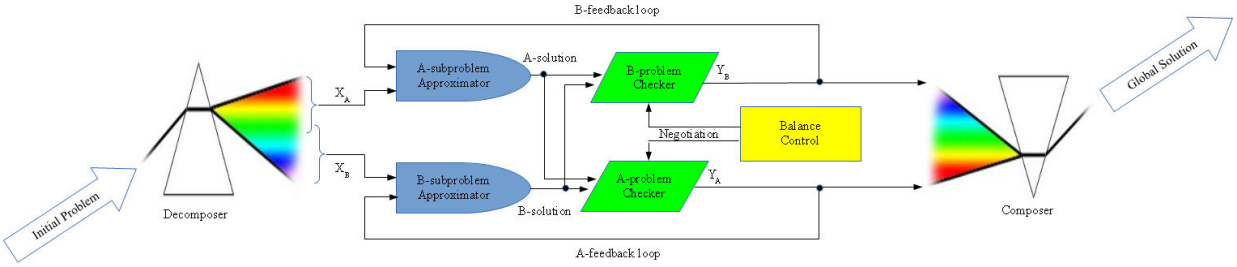

Figure 1: Decomposition of a problem on two sub-problems: A and B.

step consists of a pair K', Π', where Π' is the route found as a solution to the instance of TSKP. Then the profit P' of the solution is calculated. If profit P' is better than the best profit P that has been found so far, the process repeats with distances between nodes adjusted along tour Π'.

The current implementation of CoSolver uses an exact algorithm for TSKP, and the following heuristic for KRP. Let W_i denote the total weight of items picked at cities $\{2, \ldots, i\}$ according to some picking plan. We create an instance of KP with "relaxed profits" in the following way:

$$\overline{p}_i = p_i - R(t - t')$$

where t and t' are given by:

$$t_i = \frac{L_i}{v_{max} - (W_{i-1} + w_i)\frac{v_{max} - v_{min}}{W}} \quad (9)$$

$$t'_i = \frac{L_i}{v_{max} - W_{i-1}\frac{v_{max} - v_{min}}{W}} \quad (10)$$

and:

$$L_i = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{k=i}^{n+1} d(k) & \text{if } i \neq 1 \end{cases} \quad (11)$$

Instances of KP are solved exactly by the usual dynamic programming method.

Note that the idea of decomposition and negotiation between modules was also mentioned [9]. In fact, the general idea of CoSolver is very similar to co-operational co-evolution proposed in [20] where a problem is divided to sub-problems and different parts of the problem are solved by different modules (potentially, each module is a population-based method). Then, the final solution is combined to create the solution for the original problem.

4. GENERATION OF INSTANCES

To compare performance of the proposed algorithms, we have prepared a generic framework for generating classes of TTP-instances. Each class is composed of three independent components: *meta*, *TSP* and *KP* that are explained in the rest of this section. Depending on parameters configuration of these components one is able to create separate classes of TTP-instances.

4.1 Meta-component

This subsection describes all parameters that are independent from the TSP graph and KP items — i.e. knapsack-capacity, rent rate, minimal-velocity and maximal-velocity. One can alter the importance of TSP or KP components in a TTP by tunning these parameters. As an example, a

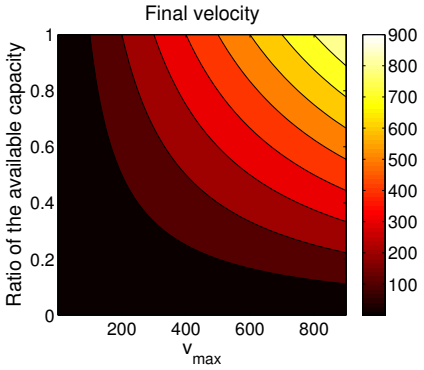

Figure 2: Impact of different values of the maximum velocity and ratio of the available capacity of the knapsack on the final velocity for a specific W.

reducing the value of the rent rate (w.r.t the profit of items) in fact reduces the contribution of the TSP component to the final objective value. In an extreme case, the TSP component is completely ignored if the rent rate is zero. Also, by increasing the value of the rent rate, total profit of items becomes less important in the final objective value and the speed of travel become more important (contribution of TSP in the final objective is increased). Consequently, the solution for the KP component have smaller impact on the total objective. This is also the same for the value of $v_{max} - v_{min}$. As an example, lets assume that v_{min} is a constant (0.1). If the value of $v_{max} - v_{min}$ becomes zero, the value of velocity, calculated by $v_{max} - (v_{max} - v_{min})\frac{W_c}{W}$ (W_c is the total weight of the picked items) becomes v_{max} for any $\frac{W_c}{W}$. Thus, picking any item has no impact on the speed of the travel, and hence, one can completely decompose TTP instance to two independent sub-problems (TSP component and KP-component) and solve them separately. Note that, with this setting, the optimal solution in isolation for each component in fact constructs the overall optimal solution for any given rent rate (see Fig 2). By making v_{max} larger (for a fixed v_{min}), the variability of the final velocity increases according to the ratio of the available capacity ($1 - \frac{W_c}{W}$). In fact, for a large v_{max}, the more items the thief pick, the faster the speed drops comparing to a smaller value for v_{max}. If the value of v_{max} is set to a very large value, the final velocity drops very fast and, for fixed values of other parameters (fixed profits and weights for items, rent rate, and distances), it would be better to have more available capacity in the knapsack to avoid significant drop of the travel speed. In fact, in this case, solving KP component to optimality is more harmful.

Algorithm 3: Generation of Instances

1: $V = \{0, 1, \ldots, n-1\}$
2: $E = \{\}$
3: **for** given number of iteration **do**
4: perm ← random permutation of V
5: add to E edges of the Hamiltonian cycle, that is defined by perm
6: return $G = (V, E)$

4.2 Methods of Generating *TSP-component*

By the *TSP-component* of a TTP-instance we define a weighted directed graph $G = (V, E, w)$, where V denotes the set of vertices, E denotes the set of edges and $w\colon E \to \mathbb{R}$ that assigns length (distance) to each of the edges.

Below we describe the classes of underlying graphs in greater depth. Both the class of *simple random graphs* and the class of *Euclidean random graphs* are based on the basic $G(n, p)$ model[5]; where $\boldsymbol{n}$ denotes the number of vertices in the graph G and $\boldsymbol{p}$ denotes probability that an edge e is — independently from other edges — included in $E(G)$.

Simple Random Graphs In this class we extend $G(n, p)$-model by an option of setting the length l of each edge $e \in E(G)$. Instances of this class are characterized by relatively high difficulty of the *TSP-component*. *Parameters:* $\boldsymbol{l_{min}, l_{max}}$ — boundary values of an uniform distribution, from which lengths of edges are drawn.

Euclidean Random Graphs Vertices of the underlying graph G are embedded in some m-dimensional Euclidean space; i.e. there exists a function $\iota\colon V(G) \to \mathbb{R}^m$, such that for each $(n_1, n_2) \in E(G)$, the length of (n_1, n_2) is equal to Euclidean distance between $\iota(n_1)$ and $\iota(n_2)$. Although we allow m to be arbitrary large, up-to-now we have been concentrating mainly on cases for which $m = 2$, as graphs of this type resembles in highest degree most of the real-world problems and are the easiest to visualize. *Parameters:* m — number of dimensions of the Euclidean space; l_{max} — for each point from $\mathbb{R}^m$, that corresponds to some vertex from $V(G)$, values alongside each dimension are between 0 and l_{max} (in our case we draw them from a uniform distribution); $dist_{min}$ — minimal length of an edge (n_1, n_2).

Hamiltonian-Dense Graphs The main motivation behind this class of graphs is to make finding *Hamiltonian-cycles* relatively easy. Instances of this class are generated by Algorithm 3. *Parameters:* i — number of iterations; l_{min}, l_{max} — boundary values of the uniform distribution, from which lengths of edges are drawn.

4.3 Methods of Generating *KP-component*

KP-component of a TTP-instance is characterized by a set of all items I, each item being a pair $(weight, value)$ and a function $avail\colon I \times V(G) \to \{0, 1\}$ such that:

$$avail(i, v) = \begin{cases} 1 & \text{if item } i \text{ is available at vertex } v \\ 0 & \text{otherwise} \end{cases}$$

Uncorrelated Weights and Values Since *actual/global* valuableness of an item does strongly depend on the wider context in which the item is being considered[6] any sophisticated methods of generating the KP-component are rather hard to conceptualize. Therefore, we decided to generate *weight* and *value* of each item independently from one another, and control the hardness of the KP-component mainly by its size and the value of knapsack-capacity. *Parameters:* n is the number of items; $value_{min}, value_{max}$ are boundary values of the uniform distribution, from which *values* of items are drawn; $weight_{min}, weight_{max}$ are boundary values of an uniform distribution, from which *weights* of items are drawn; p is the probability with which each item is available at each vertex.

Greedy-proof Due to the fact that KP-instance in which values and weights of items are uncorrelated, are — in general — considered to be rather easy [18] and a greedy algorithm can solve them to high quality solutions, we propose a method of generating KP-component instances that are resistant against the greedy approach. As the first step of this *greedy-proof*-method we generate a number of items of relatively low weight and relatively high $\frac{value}{weight}$-ratio, whose total weight from all vertices constitutes some small fraction f of the knapsack-capacity W. As the second step we generate a relatively high amount of items, whose weights are drawn from the interval $[w - \frac{f}{2}, w]$. *Parameters: big* — number of heavy items; p — probability with which each heavy-item is available at each vertex; *small* — total number of small-items in all vertices; *fraction* — control parameter for bounds of total weight of small-items and weights of heavy-items.

4.4 Parameters of Benchmark Instances

The set of our benchmark instances counts 45 TTP-instances of different sizes[7] and levels of difficulty. The instances can be grouped into the following 6 classes according to the methods by which they were generated. Number in brackets indicates the actual number of instances of given type.

Euclidean (9) *meta-component*: $rent \in [1, 400]$, $capacity \in [200, 500]$, $velocity_{min} \in [25, 50]$, $velocity_{max} \in [60, 75]$; *TSP-component — Euclidean*: $n \in [15, 25]$, $p \in [0.1, 0.3]$, $l_{max} = 5000$, $dist_{min} = 10$; *KP-component — Uncorrelated*: $n \in [5, 10]$, $p \in [0.1, 0.3]$, $value \in [1, 10000]$, $weight \in [50, 100]$

Dense (3) *meta-component*: $rent \in [0.01, 100]$, $capacity \in [300, 600]$, $velocity_{min} \in [20, 40]$, $velocity_{max} \in [50, 90]$; *TSP-component — Hamiltonian-Dense*: $n \in [30, 35]$, $cycles = 3$; *KP-component — Uncorrelated*: $n \in [7, 10]$, $value \in [1, 10000]$, $weight \in [50, 100]$, $p = 0.47$

Random (8) *meta-component*: $rent \in [1, 1000]$, $capacity \in [200, 400]$, $velocity_{min} \in [25, 50]$, $velocity_{max} \in [60, 75]$; *TSP-component — Simple*: $n \in [15, 25]$, $p \in [0.1, 0.2]$, $length \in [1, 500]$; *KP-component — Uncorrelated*: $n \in [5, 10]$, $value \in [1, 500]$, $weight \in [30, 80]$, $p \in [0.1, 0.2]$

Small (5) *meta-component*: $rent \in [2, 5]$, $capacity \in [400, 800]$, $velocity_{min} \in [10, 100]$, $velocity_{max} \in [100, 200]$; *TSP-component — Simple*: $n \in [3, 6]$, $p = 1$, $length \in [1000, 5000]$; *KP-component — Uncorrelated*: $n \in [5, 10]$, $value \in [1, 1000]$, $weight \in [20, 80]$, $p \in [0.1, 0.3]$

Greedy-Proof (10) *meta-component*: $rent \in [0.05, 0.1]$, $capacity \in [10000, 20000]$, $velocity_{min} \in [25, 50]$, $velocity_{max} \in [60, 75]$; *TSP-component — Hamiltonian-Dense*: $n \in [4, 8]$,

[5]Introduced in [21], but named later after Erdős and Rényi.
[6]The main factors are: distance to last vertex on the tour, knapsack-rent, current load of knapsack, minimal and maximal velocity
[7]Because we were mainly interested in the question, how solutions obtained by our algorithms differ from the exact solutions, the sizes of instances are rather mediocre, as the latter need *exhaustive-search*.

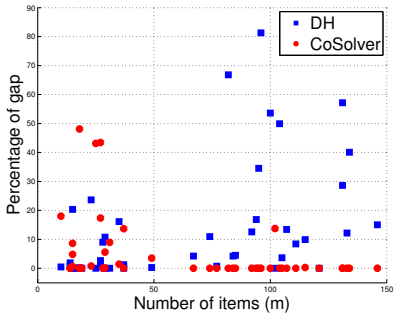

Figure 3: Performance of DH and CoSolver in solving the generated benchmark problems with regards to different number of items.

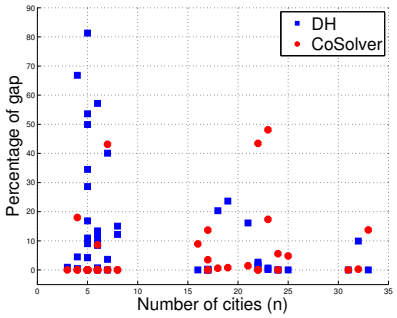

Figure 4: Performance of DH and CoSolver in solving the generated benchmark problems with regards to the different number of cities.

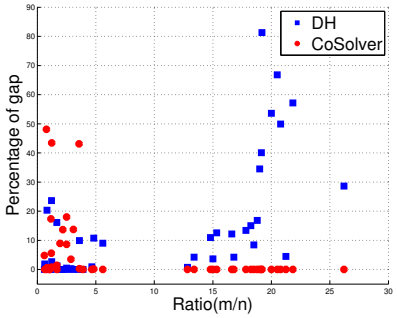

Figure 5: Performance of DH and CoSolver in solving the generated benchmark problems with regards to the number of items/number of cities.

cycles $= 3$; *KP-component — Greedy-Proof*: *big* $\in [50, 70]$, $p \in [0.2, 0.25]$, *small* $\in [20, 25]$, *fraction* $\in [0.2, 0.25]$

KP-Centric (10) *meta-component*: *rent* $\in [1, 20]$, *capacity* $\in [5000, 10000]$, $velocity_{min} \in [25, 50]$, $velocity_{max} \in [60, 75]$; *TSP-component — Euclidean*: $n \in [4, 6]$, $p = 1$, $l_{max} = 100$, $dist_{min} = 10$; *KP-component — Uncorrelated*: $n \in [50, 100]$, *value* $\in [1, 10000]$, *weight* $\in [1, 10000]$, $p \in [0.2, 0.3]$

5. EXPERIMENTAL RESULTS

In order to compare DH and CoSolver, we generated 45 instances with different number of cities (from 3 to 33) and items (from 10 to 146) with the parameters mentioned in section 4.4. Both methods are applied to this set of benchmark problems and their results are compared. We also designed an exhaustive search that solves the generated benchmark set to optimality. In all comparisons, we use the percentage of the gap between the generated solution by each heuristic algorithm and the exact solution by $gap = 100\frac{z-z^*}{z^*}$, where z^* is the solution found by the exact solver and z is the solution found by a heuristic (DH or CoSolver). Note that the better the solution is, the lower the value of the gap will be.

The overall performance of the methods has been reported in Table 1. Clearly, CoSolver has a better performance than DH in average and also it has smaller standard deviation from the exact solution.

Table 1: Average and standard deviation of the gap for DH and CoSolver over all 45 instances.

	DH	CoSolver
Average	13.70	**5.16**
Std	20.03	**11.76**

Fig. 3 shows the scatter plot of the average value of the gap for CoSolver and DH from the exact solutions versus the number of items. It is clear in the figure that the CoSolver outperforms DH (smaller value for the gap is better) when the number of items is large. This was in fact expected as the strategy for picking items in DH for a given tour is very simple and intuitive. However, it seems that CoSolver has a good performance when the number of items is large (the value of gap is close to zero in most of the cases). This confirms that the negotiation between modules improves the ability of the method to make better decision to solve multi-component problems.

Fig. 4 shows scatter plot of the percentage of the gap versus the number of cities. It is clear from the figure that for a small number of cities, the performance of DH is much worse than CoSolver. The reason is that, when the number of cities is small, the contribution of the TSP part becomes minimal (the total distance for the whole tour is small). Thus, the picking plan for the items becomes very important. However, DH only uses a very simple rule to pick the items, which makes the algorithm ineffective. CoSolver however is equally effective when the number of cities is small or large.

Fig. 5 shows the gap vs ratio ($ratio = m/n$) for DH and CoSolver. Figure indicates that the higher this ratio is, the less effective DH is. In fact, when the number of cities is small and the number of items is large, DH is not that useful as the strategy for picking items in DH is not that effective in finding good solutions for the KP component to match with the found tour. However, CoSolver is way more effective especially for instances in which ration is high. This in fact confirms that the negotiation between modules can effectively address the multi-component problems. Note also that the algorithm for generating the tour in DH is one of the most effective known methods for solving TSP. However, CoSolver only utilizes problem knowledge to solve the given TTPs and no complex strategy in the algorithm were used.

Fig. 6 shows the gap vs rent rate for DH and CoSolver methods. Clearly, the performance of DH is lower than the performance of CoSolver when the rent rate is small. The reason is that (as discussed in section 4.1), for smaller values of the rent rate, the significance of the solution for the TSP component becomes smaller. As DH uses a simple heuristic to solve the KP component, it is actually expected that its performance is low when the rent rate is small. however,

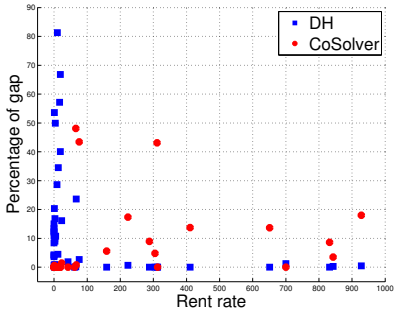

Figure 6: Performance of DH and CoSolver in solving the generated benchmark problems with regards to different rent rates.

when the rent rate is high, the contribution of TSP component becomes larger. As DH uses a good method to deal with the TSP component, its results become better when the rent rate increases.

6. CONCLUSIONS AND FUTURE WORK

In this paper, two methods for solving a multi-component problem (the Traveling Thief Problem, TTP) were proposed. The first method, called Density-based Heuristic (DH), bases on solving the components sequentially. The second method, called CoSolver, which was inspired by co-operational co-evolution, considers the interdependency among components. CoSolver mimics the behavior of group of experts who solve real-world problems in companies. In that method, the problem is decomposed and each component is solved by a module of CoSolver. The algorithm revises the solution through negotiation between its modules. This is done iteratively until no improvement is possible. We also generated some benchmark problems and applied the above two methods to the benchmarks. Results show that CoSolver outperforms DH in most of the cases, especially when the components are not decomposable (i.e. rent rate is larger than zero). This in fact confirms that considering interdependency is beneficial in solving multi-component problems. In future work, we will study other types of negotiation between modules of CoSolver, and develop algorithms based on other decompositions of TTP. Moreover, we will be interested in substituting exact algorithms for sub-problems of TTP with approximation algorithms.

7. REFERENCES

[1] A computer code for tsp. `http://www.math.uwaterloo.ca/tsp/concorde.html`.

[2] An effective implementation of the lk-heuristic. `http://www.akira.ruc.dk/~keld/research/LKH`.

[3] Millennium problems. `http://www.claymath.org/millenium-problems/p-vs-np-problem`.

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

[5] Y. Bartal and L.-A. Gottlieb. A linear time approximation scheme for euclidean tsp. *IEEE Annual Symp. on Found. of Comp. Sci.*, 0:698–706, 2013.

[6] Y. Bartal, L.-A. Gottlieb, and R. Krauthgamer. The traveling salesman problem: Low-dimensionality implies a polynomial time approximation scheme. In *Proc. of Symp. on Theory of Computing*, pages 663–672, New York, NY, USA, 2012. ACM.

[7] A. Björklund. Below all subsets for permutational counting problems. *CoRR*, abs/1211.0391, 2012.

[8] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. The traveling salesman problem in bounded degree graphs. *ACM Trans. on Algorithms*, 8(2):18, 2012.

[9] M. Bonyadi, Z. Michalewicz, and L. Barone. The travelling thief problem: The first step in the transition from theoretical problems to realistic problems. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1037–1044, 2013.

[10] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[11] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[12] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[13] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. In *Proceedings of the 1961 16th ACM National Meeting*, ACM '61, pages 71.201–71.204, New York, NY, USA, 1961. ACM.

[14] E. L. Lawler, J. K. Lenstra, A. R. Kan, and D. B. Shmoys. *The traveling salesman problem: a guided tour of combinatorial optimization*, volume 3. Wiley Chichester, 1985.

[15] Z. Michalewicz. Quo vadis, evolutionary computation? In *Advances in Computational Intelligence*, pages 98–121. Springer, 2012.

[16] Z. Michalewicz and D. B. Fogel. *How to solve it: Modern Heuristics*. Springer New York, 2000.

[17] D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, Dep. of Comp. Sci., Uni. of Copenhagen, 1995.

[18] D. Pisinger. Where are the hard knapsack problems? *Comp. & Op. Res.*, 32(9):2271–2284, 2005.

[19] S. Polyakovskiy, M. R. Bonyadi, M. Wagner, F. Neumann, and Z. Michalewicz. A comprehensive benchmark set and heuristics for the travelling thief problem. In *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2014, To appear.

[20] M. A. Potter and K. A. De Jong. A cooperative coevolutionary approach to function optimization. In *Parallel Problem Solving from Nature—PPSN III*, pages 249–257. Springer, 1994.

[21] A. Rapoprt and R. Solomonof. Connectivity of random nets. *Bulletin of Mathematical Biology*, 13(2):107–117, 1951.

[22] P. Toth and D. Vigo. *The vehicle routing problem*, volume 9. Siam, 2002.

[23] T. Weise, M. Zapf, R. Chiong, and A. Nebro. Why is optimization difficult? In R. Chiong, editor, *Nature-Inspired Algorithms for Optimisation*, volume 193 of *Studies in Computational Intelligence*, pages 1–50. Springer Berlin Heidelberg, 2009.