Approximate Approaches to the Traveling Thief Problem

Hayden Faulkner, Sergey Polyakovskiy, Tom Schultz, Markus Wagner
Optimisation and Logistics
School of Computer Science
University of Adelaide, Australia

ABSTRACT

This study addresses the recently introduced Traveling Thief Problem (TTP) which combines the classical Traveling Salesman Problem (TSP) with the 0-1 Knapsack Problem (KP). The problem consists of a set of cities, each containing a set of available items with weights and profits. It involves searching for a permutation of the cities to visit and a decision on items to pick. A selected item contributes its profit to the overall profit at the price of higher transportation cost incurred by its weight. The objective is to maximize the resulting profit.

We propose a number of problem-specific packing strategies run on top of TSP solutions derived by the Chained Lin-Kernighan heuristic. The investigations provided on the set of benchmark instances prove their rapidity and efficiency when compared with an approximate mixed integer programming based approach and state-of-the-art heuristic solutions from the literature.

Categories and Subject Descriptors

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

Keywords

Traveling thief problem; local search; multi-component problems

1. INTRODUCTION

Recently, a new benchmark problem called the Traveling Thief Problem (TTP)has been introduced [3] in an attempt to provide an abstraction of problems with multiple interdependent components. The underlying idea of TTP is to combine the Traveling Salesman Problem (TSP) and the Knapsack Problem (KP). They both have been intensively studied for many years and are the core problems in the field of optimization. The TTP comprises a thief stealing items with weights and profits from a number of cities. The thief uses a knapsack of limited capacity and pays rent for it depending on the overall travel duration. To make the two components interdependent, the speed of the thief is made dependent (non-linearly) on the weight of the items picked so far. The thief has to visit all cities once and collect items so that the overall profit is maximized. While it is possible to generate solutions for the TTP by combining solutions for the individual components, such approaches do not necessarily result in near-optimal solutions: (1) each solution for the TSP hinders the best quality that can be achieved in the KP component because of the impact on the profit that is a function of travel time, and vice versa (2) each solution for the KP component influences the tour time for TSP as different items impact the speed of travel differently due to the variability of the weights of items.

A range of different approaches to the problem have been developed recently. For example, Mei et al. [6] focus on large TTP instances with relatively simple approaches. They analyze the computational complexity of different algorithms and suggest asymptotic speed-ups. The resulting algorithm significantly outperforms the three iterative and constructive algorithms originally presented in [10]. Bonyadi et al. [4] are the first to attempt to solve both components of the TTP in parallel, instead of sequentially. The authors design a co-evolutionary approach called CoSolver in which modules responsible for either the TSP part or the KP part communicate with each other. The proposed CoSolver outperforms a heuristic similar to the constructive heuristic SH from [10]. Mei et al. [7] also investigate a co-evolutionary approach and compare it to a memetic algorithm, with the result of the memetic algorithm outperforming the co-evolutionary one.

Recently, Polyakovskiy et al. [11] investigated a non-linear knapsack problem that occurs when packing items along a fixed route and taking into account travel time. They address the complexity of the problem and show that even the capacity unconstrained version is $\mathcal{NP}$-hard. Subsequently, they propose exact and approximate mixed integer programming (MIP) solutions that are able to produce (near-) optimal results for instances of a moderate size.

Despite the state-of-the-art techniques there remains potential for TTP-specific algorithms based on different paradigms. In this article, we provide a set of structurally diverse algorithms to effectively solve the entire range of existing TTP instances.

This article is set out as follows. In Section 2 we outline the important features of the Traveling Thief Problem. In Section 3 we describe our three local search routines, based on which we construct heuristic searches in Section 4. Sec-

GECCO '15, July 11 - 15, 2015, Madrid, Spain

 ISBN 978-1-4503-3472-3/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2739480.2754716

tion 5 adopts the MIP based approximate approach of [11] and yields another high-quality technique. In Section 6, we present and compare their results with the state-of-the-art solutions. We finish with some concluding remarks.

2. TRAVELING THIEF PROBLEM

In the following, we first describe the TTP as defined in [10]. Given is a set of cities $N = \{1, \ldots, n\}$ and a set of items $M = \{1, \ldots, m\}$ distributed among the cities. The distance d_{ij} between any pair of cities $i, j \in N$ is known. Every city i, except the first one, contains a set of items $M_i = \{1, \ldots, m_i\}$, $M = \bigcup_{i \in N} M_i$. Each item k positioned in the city i is characterized by its profit p_{ik} and weight w_{ik}, thus the item $I_{ik} \sim (p_{ik}, w_{ik})$. The thief must visit each of the cities exactly once starting from the first city and returning back to it in the end. Any item may be selected in any city as long as the total weight of collected items does not exceed the capacity W. A renting rate R is to be paid per each time unit taken to complete the tour. Respectively, v_{max} and v_{min} denote the maximal and minimum speeds that the thief can move. The goal is to find a tour, along with a packing plan, that results in the maximal profit.

To set up the objective function precisely, let $y_{ik} \in \{0, 1\}$ be a binary variable equal to one when the item k is selected in the city i. In addition, let W_i denote the total weight of collected items when the thief leaves the city i. Therefore, the objective function for a tour $\Pi = (x_1, \ldots, x_n)$, $x_i \in N$ and a packing plan $P = (y_{21}, \ldots, y_{nm_i})$ has the following form:

$$Z(\Pi, P) = \sum_{i=1}^{n} \sum_{k=1}^{m_i} p_{ik} y_{ik} - R \left(\frac{d_{x_n x_1}}{v_{max} - \nu W_{x_n}} + \sum_{i=1}^{n-1} \frac{d_{x_i x_{i+1}}}{v_{max} - \nu W_{x_i}} \right) \quad (1)$$

where $\nu = \frac{v_{max} - v_{min}}{W}$ is a constant value defined by input parameters. The minuend is the sum of all packed items' profits and the subtrahend is the amount that the thief pays for the knapsack's rent equal to the total traveling time along Π multiplied by R.

A brief numeric example of the TTP problem is provided in Figure 1 [10]. Each node but the first has an assigned set of items. Let us assume that the maximum weight $W = 3$, the renting rate $R = 1$ and v_{max} and v_{min} are set as 1 and 0.1, respectively. Then the optimum objective value $Z(\Pi, P) = 50$ for $\Pi = (1, 2, 4, 3)$ and $P = (0, 0, 0, 1, 1, 0)$. Specifically, the thief collects no items traveling from city 1 to city 3 via cities 2 and 4. Therefore, this part of the tour has a cost of 15. In the city 3 only items I_{32} and I_{33} are picked up, resulting in a total profit of 80. However, on the way from city 3 back to city 1 the thief's knapsack has a weight of 2. This reduces the speed and results in an increased cost of 15. Consequently, the final objective value is $Z(\Pi, P) = 80 - 15 - 15 = 50$.

3. LOCAL SEARCH ROUTINES

In this section, we discuss a number of routines used to elaborate a packing plan for a given permutation of the cities Π. We assume that a TSP tour Π is the outcome of the call of the Chained Lin-Kernighan heuristic [2][1]. A packing routine is applied to construct an initial packing which is further enhanced by a set of local search procedures.

[1]As available at http://www.tsp.gatech.edu/concorde/downloads/downloads.htm

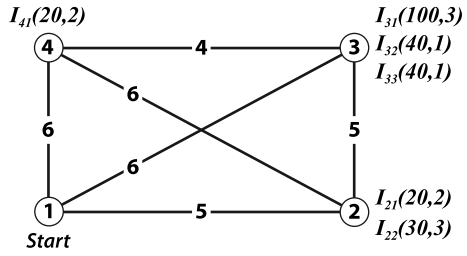

Figure 1: Illustrative Example [10]

3.1 Packing Routine

For greedy heuristics, it is common to utilize a goodness measure of the elements of a problem to make a decision. Here, we provide our packing heuristic algorithm with a scoring function which generates a score s_{ik} for item k placed in city i as:

$$s_{ik} = \frac{p_{ik}}{w_{ik} \times d_i},$$

where d_i is the distance from city i to the end of a given tour Π. The function incorporates a trade-off between a distance that item I_{ik} is to be carried over, its weight and its profit. Depending on a particular instance, it might be beneficial to strengthen the influence of one or more variables of the function. In this scope, we introduce exponents applied to the weight and the profit of an item to manage their impact. Thus, the function takes the following form:

$$s_{ik} = \frac{p_{ik}^{\alpha}}{w_{ik}^{\alpha} \times d_i}.$$

Our preliminary study shows that keeping the exponent of a distance low relative to the exponents of other two variables results in the best objective values. Therefore, we omit using an exponent for the distance variable. Moreover, using the same value for the exponents of the profit and weight variables often achieves the higher objective values when compared to varying exponents between the variables.

Algorithm 1 depicts our knapsack packing routine. The constructing process starts by giving a score to each item, and then subsequently sorting the entire set M of the items in non-decreasing order based on their scores. Then, the algorithm instantiates the control parameters. Since evaluating the objective function (2) is time-consuming, especially when complicated by the size of an instance, we introduce a parameter μ controlling the frequency of the objective value recomputation. Thus, the objective value is only evaluated each time μ items have been included for a possible packing. The value of μ is initialized in relation to the number of items m as $\mu = \lfloor m/\tau \rfloor$, where τ is a given integer constant. In addition, the current and the best packing plans P and P^* respectively, are initialized as empty sets. Finally, the best objective value found so far is set to $-\infty$.

Having been initialized, the algorithm starts the iterative solution construction process by considering the first item $I_1 \in M$ with the largest score. At each iteration k, item k is added to P if there remains enough free space in the knapsack. Every time μ items have been considered, the routine computes the objective value Z. When Z is improved, μ remains the same and the best objective value Z^* is updated

Algorithm 1
Packing Routine PACK (Π,α)

compute score for each of the items $I_m \in M$
sort the items of M in non-decreasing order of their scores
set the frequency $\mu = \lfloor m/\tau \rfloor$
set the current packing plan $P = \emptyset$ and $W' = 0$
set the best packing plan $P^* = \emptyset$
set best objective value $Z^* = -\infty$
set the counter $k = 1$ and set $k^* = 1$
while $(W' < W)$ and $(\mu > 1)$ **do**
  if $(W' + w_k \leq W)$ **then**
    add item $I_k \in M$ to the packing plan $P = P \cup \{I_k\}$
    set $W' = W' + w_k$
    if $(k \; mod \; \mu = 0)$ **then**
      compute the objective value $Z = Z(\Pi, P)$
      if $(Z < Z^*)$ **then**
        restore the packing plan $P = P^*$
        set $k = k^*$ and update W'
        set $\mu = \lfloor \mu/2 \rfloor$
      else
        update the best packing plan with $P^* = P$
        set $k^* = k$ and $Z^* = Z$
  set $k = k + 1$
return P

accordingly. However, as soon as the packing plan P yields $Z < Z^*$, the previous packing plan P^* is restored and the algorithm returns back to consider the items again starting with item $I_k \in M$. In addition, the frequency μ is halved. This strategy helps to identify the moment when no further solution improvement is possible while keeping the algorithm computationally fast. The heuristic terminates either when no further improvement is possible or the limit value of 1 is reached for μ.

To reach a better packing performance, we run the packing routine routine PACK (Π,α) iteratively for different values of the exponent α. Algorithm 2 explains the structure of the solution process. The algorithm PACKITERATIVE (Π,c,δ,q) is fed by the starting exponent value $c \in \mathbb{R}_+$, and a deviation value $\delta \in \mathbb{R}_+$ which defines the current studying interval for potential exponent values as $(c - \delta, c + \delta)$. In addition, an integer q is given as a limit on the number of iterations. The algorithm starts with evaluating objective values obtained by PACK (Π,α) in case of $\alpha = c$ and in cases of two terminal values $\alpha = c - \delta$ and $\alpha = c + \delta$. At each iteration, the algorithm compares the corresponding objective values Z_l, Z_m and Z_r, and selects the further interval for investigation. Specifically, it switches its attention to smaller values if $(Z_l > Z_m) \wedge (Z_l > Z_r)$ or to larger values if $(Z_r > Z_m) \wedge (Z_r > Z_l)$. Subsequently, the routine updates the best packing plan P^* found so far. In addition, it narrows the potential interval for a search setting $\delta = \delta/2$ and changes the current exponent value c to $c - \delta$ or $c + \delta$, depending on which produces the largest objective value. The searching process continues until the number of iterations performed reaches its limit or improvement in the objective value is less than ϵ, where ϵ is a small threshold constant.

3.2 Bitflip

The previously presented packing approach is not guaranteed to find a globally optimal TTP solution as (1) it does not modify the tours, and (2) the packing plan it finds may

Algorithm 2
Iterative Packing Routine PACKITERATIVE (Π,c,δ,q)

obtain P_l by PACK (Π,$c - \delta$) and compute $Z_l = Z(\Pi, P_l)$
obtain P_m by PACK (Π,c) and compute $Z_m = Z(\Pi, P_m)$
obtain P_r by PACK (Π,$c+\delta$) and compute $Z_r = Z(\Pi, P_r)$
set the best packing plan $P^* = \emptyset$
set the counter $i = 1$
while $i \leq q$ **do**
  if $(Z_l > Z_m)$ and $(Z_r > Z_m)$ **then**
    if $(Z_l > Z_r)$ **then**
      set $Z_m = Z_l$, $c = c - \delta$ and $P^* = P_l$
    else
      set $Z_m = Z_r$, $c = c + \delta$ and $P^* = P_r$
  else if $(Z_l > Z_m)$ **then**
    set $Z_m = Z_l$, $c = c - \delta$ and $P^* = P_l$
  else if $(Z_r > Z_m)$ **then**
    set $Z_m = Z_r$, $c = c + \delta$ and $P^* = P_r$
  $\delta = \delta/2$;
  obtain P_l by PACK (Π,$c-\delta$) and compute $Z_l = Z(\Pi, P_l)$
  obtain P_r by PACK (Π,$c+\delta$) and compute $Z_r = Z(\Pi, P_r)$
  set $i = i + 1$
  if $(Z_l - Z_m < \epsilon)$ and $(Z_r - Z_m < \epsilon)$ **then** break
return P^*

Algorithm 3
BITFLIP (Π,P)

set $Z^* = Z(\Pi, P)$
set $P^* = P$
for each item $I_m \in M$ **do**
  if $I_m \notin P^*$ **then**
    add item I_m, $P = P^* \cup \{I_m\}$
  else
    remove item I_m, $P = P^* \setminus \{I_m\}$
  compute the objective value $Z(\Pi, P)$
  if $Z > Z^*$ **then**
    set $Z^* = Z$
    set $P^* = P$
return P^*

not necessarily be optimal for a given tour. Therefore, we present two local search operators to complement the packing plan generation from Section 3.1.

The BitFlip operator is a simple greedy hillclimber with a low runtime. Its pseudocode is shown in Algorithm 3. The operator iteratively evaluates the outcome of flipping each bit position corresponding to item $I_m \in M$ in the packing plan P. If flipping a bit improves the objective value ($Z > Z^*$), the change is kept, otherwise P remains unchanged. A single iteration of the operator ends when all bit flips have been attempted once. Multiple passes can result in further improvement of the objective value.

3.3 Insertion

The insertion operator takes advantage of the situation where a valuable item at a particular city is picked up early and it is worth trying to delay the item pickup by modifying the tour Π. Figure 2 shows an example where city x_2 is inserted later in the tour, after city x_7. In this case the result is a slightly longer tour, however the items at x_2 are now later in the tour and hence contribute a lower carrying cost.

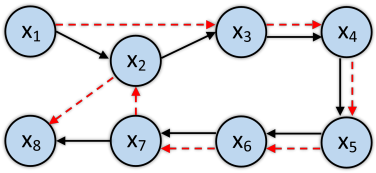

Figure 2: A graphical example of insertion: The solid black line represents the initial tour option, while the red dashed line represents a modified tour where city x_2 has been inserted after city x_7. The red tour can have a higher objective score due to a valuable (but heavy) picked item at city x_2.

Algorithm 4
INSERTION (Π,P)

```
for a = n → 1 do
    compute Z = Z(Π, P)
    for b = a − 1 → 1 do
        set Π* = Π
        insert city x_a between x_b and x_{b−1} in tour Π*
        compute Z* = Z(Π*, P)
        if Z* > Z then
            set Z = Z*
            set Π** = Π*
        if b = 1 then
            set Π = Π**
return Π
```

There is a tradeoff between increasing the tour length and hence travel time and carrying cost for all items before the insertion point, and decreasing the carrying cost of the items of the inserted city. The insertion pseudocode is shown in Algorithm 4. The operator searches over all cities in reverse tour order ($x_n \rightarrow x_1$), evaluating the effect of inserting each city at all positions before its own position in the tour Π^*. If an insertion for a city x_a is found that produces a greater objective value than (1) the current tour Π and (2) all other insertions for Π^*_a, then the tour is stored as Π^{**}. Once the tour has been completely checked for better positions for x_a then Π^{**} becomes the new tour Π and the process begins with the next city x_{a-1}. Again, multiple passes can result in further improvement of the objective value.

Note that in typical good solutions to the TTP many items are picked up towards the end of the tour. The intuition is that it is not too harmful to the overall objective value if items are picked up late, since the speed of the thief is typically low at that point in time. We begin our insertions at the end of the tour since this is a time consuming search operation for very large instances.

4. LOCAL SEARCH ALGORITHMS

In order to isolate the effects of the different search routines, we first define a set of straight-forward local search algorithms. All algorithms use the Chained Lin-Kernighan heuristic (CLK) to generate an initial TSP tour:

- Heuristic S1: run CLK, then PACKITERATIVE;
- Heuristic S2: run CLK, then PACKITERATIVE, then repeat BITFLIP until converged;
- Heuristic S3: run CLK, then PACKITERATIVE, then (1+1)-EA;
- Heuristic S4: run CLK, then PACKITERATIVE, then repeat INSERTION until converged;
- Heuristic S5: repeat S1 until time is up.

Because BITFLIP and INSERTION are deterministic, we can stop the heuristics S2 and S4 once the objective value does not change.

(1+1)-EA is a simple alternative to the greedy BITFLIP, which was previously used in [10]: given a packing plan, we flip each item with probability $1/m$. If the TTP solution with the new packing plan has a higher objective value, we use it as the next starting point. We terminate it once the CPU time is used up.

Since we expect the different routines to be able to escape each others local optima, we also define the following slightly more complex heuristics:

- Heuristic C1: run CLK, then PACKITERATIVE, then repeat "one BITFLIP pass, one INSERTION pass" until converged;
- Heuristic C2: run CLK, then PACKITERATIVE, then repeat "one BITFLIP pass, one (1+1)-EA pass, one INSERTION pass";
- Heuristic C3: run CLK until 10% of the time is used, then apply PACKITERATIVE, select the best of those, then "one BITFLIP pass, one INSERTION pass";
- Heuristic C4: run CLK until 10% of the time is used, then apply PACKITERATIVE, select the best of those, then "one BITFLIP pass, one (1+1)-EA pass, one INSERTION pass";
- Heuristic C5: repeat C1 until time is up;
- Heuristic C6: repeat C2 until time is up.

The heuristics C1 and C2 are straight-forward extensions of S2 and S4, with the hope that the two routines complement each other. Heuristics C3 and C4 use 10% of their CPU time to sample better starting points. This allows us to investigate the benefits of a better starting point over performing more iterations of the routines. Lastly, C5 and C6 are the restart variants of C1 and C2, with which we can observe the advantage of restarts to make the best use of the available computation time.

Note that we limit (1+1)-EA to 10,000 iterations in these complex heuristics, whereas it is unlimited in the simple ones. In addition, note that we decided to execute only single passes of BITFLIP and INSERTION per iteration due to their computational complexity. With this interleaving of the two routines, none of them will use a disproportionate amount of time. This is important particularly for the instances in which there are a large number of cities and items.

5. PACKING ITEMS WITH A MIP BASED APPROACH

When a tour is given, the two MIP based approaches of Polyakovskiy and Neumann [11] are able to solve the residuary knapsack packing part either exactly or approximately. Obtaining the optimal solution is costly in regard to running time as it requires the use of a linearization technique to handle the non-linear terms in the objective function. As an alternative, the piecewise linear approximation technique can be used. In this case, each of the non-linear terms is represented by the function $t(v) = 1/v$ which is subsequently

approximated by a set of straight line segments. The computational experiments presented in [11] show that the approximate approach yields a tight approximation within only 1% of the optimal packing plan.

For our MIP-based approach, we adopt the approximative technique and run it on top of the Chained Lin-Kernighan heuristic [2] as a tour generator. The final solution for a particular instance results from the following iterative process. Each iteration starts with computing a distinct tour Π as a result of the TSP heuristic. Then, we run the approximative solver for the KP component. Firstly, the primal order $\Pi = (x_1, x_2, \ldots, x_n)$ is considered. Specifically, the set of items M is refined to exclude any existing unprofitable and compulsory items according to the pre-processing scheme as prescribed in [11]. Then the resulting non-linear knapsack problem on the reduced set of items is solved as the linear mixed 0-1 program named NKP^a_τ, where τ is the number of equal-sized line segments and is used to manage the precision of the approximation. Secondly, the reversed order $\overline{\Pi} = (x_1, x_n, \ldots, x_2)$ is investigated for which the pre-processing scheme and NKP^a_τ are performed again. Thus, we obtain through each iteration two feasible solutions based on either Π or $\overline{\Pi}$. Afterwards, we start the next iteration with a new tour. We consider the best solution found by this iterative process within a given time limit to be the output of this approach.

6. EXPERIMENTAL INVESTIGATIONS

In this section, we investigate the effects of the different local search routines and compare our approaches with existing ones.

6.1 Experimental Setup

As mentioned above, we use the comprehensive set of TTP instances from Polyakovskiy et al. [1, 10] for our investigations. The two components of the problem are balanced in such a way that the near-optimal solution of one sub-problem does not dominate over the optimal solution of another sub-problem.

The characteristics of the 9,720 instances vary widely, and we outline the most important ones in the following:[2]

- The instances have 51 to 85,900 cities, based on instances from the TSPlib by Reinelt [12];
- For each TSP instance, there are three different types of knapsack problems: *uncorrelated*, *uncorrelated with similar weights* and *bounded strongly correlated* types, where the latter has been shown to be difficult for different types of knapsack solvers in [5, 10];
- For each TSP and KP combination, the number of items per city (referred to as an *item factor*) is $F \in \{1, 3, 5, 10\}$. Note that all cities of a single TTP instance have the same number of items, except for the first city (which is also the last city), where no items are available;
- For each instance, the renting rate R that links both subproblems is chosen in such a way that at least one TTP solution with zero negative objective value exists;
- Lastly, for each TTP configuration 10 different instances exist where the knapsack capacity is varied.

[2]For a more detailed description, we refer the interested reader to [1, 10].

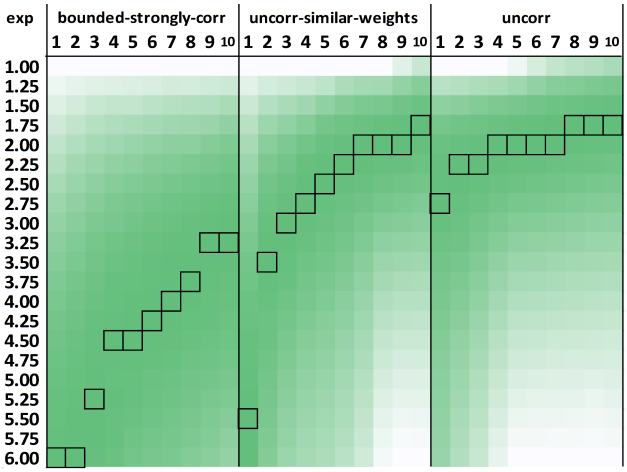

Figure 3: Heat-map of objective values for different values of α, averaged over the same 50 tours for each of the thirty instances of pcb3038_n9111. Greener (darker) squares indicate higher objective values and boxes represent the best.

From the set of 9,720 TTP instances, we select 72 as a representative subset to cover small, medium, and large size instances with different characteristics:

- six different numbers of cities (spread out roughly logarithmically): 195, 783, 3038, 11849, 33810, 85900;
- two different numbers of items per city: 3, 10;
- all three different types of knapsacks;
- two different sizes of knapsacks (capacities): 3 and 7 times the size of the smallest knapsack.

We run all algorithms for a maximum of 10 minutes per instance, except for the MIP-based approach which we run for 8 hours on the instances with $n \in \{33810, 85900\}$ cities. Due to their randomized nature, we perform 30 independent repetitions of the algorithms on each instance. All computations of heuristics S1–S5 and C1–C6 are performed on machines with Intel Xeon E5430 CPUs (2.66GHz), on Debian Linux 4.7.2, with Java SE RE 1.7.0. All computations of our MIP-based appproach were performed on machines with AMD Opteron 6348 CPUs (2.8GHz) on Red Hat Linux 4.4.7, with IBM ILOG CPLEX 12.6.

We assess the quality of the algorithms using the following approach. For each instance, we consider the best solution found to be a lower bound on the achievable objective value. Then, we take the average of the 30 results produced by an algorithm and then compute the ratio between that average and the best objective value found, which gives us the approximation ratio. This ratio allows us to compare the performances across the chosen set of instances, since the objective values vary across several orders of magnitude.

6.2 Impact of the Exponent in the Packing Routine

In the following, we investigate the performance of the packing approaches from Section 3.1. First, in an attempt to uncover the best performing values for α for varying instances, we carry out an experimental study with objective values being calculated and averaged over 100 unique tours using values $e \in [1, 6]$ at 0.25 intervals.

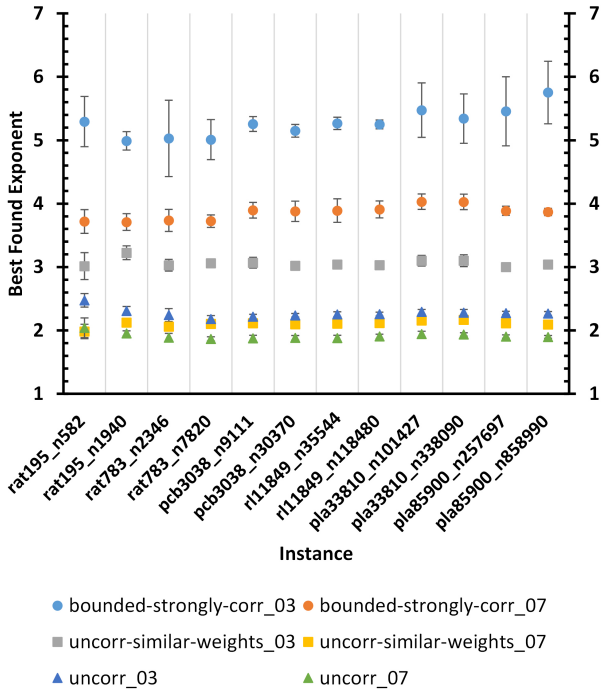

Figure 4: The best performing exponent values averaged over 50 tours for varying instances and instance types.

Figure **??** is a heat-map from the study for the pcb3038_n9111 instances.[3] It is easy to see that (1) there appears to be a single best performing exponent for each instance, and (2) there is also an optimal trend-line curve for instances of the same type that stars at high exponents for small knapsacks and ends at smaller exponents for larger knapsacks. This heat-map is representative of heat-maps for other instances. Across all instances, the trend-lines are dependent on the knapsack types (bounded-strongly, uncorr-similar, uncorr).

In order to quickly determine well performing exponents, we use PackIterative. Based on previous experiments, we use the starting exponent $c = 5$ and the sampling spread $\delta = 2.5$. To limit the runtime of Algorithm 2, we limit the number of iterations $q = 20$ and use $\epsilon = 0.1$. We show the best performing exponents in Figure **??**. What we can observe in Figure **??** is now visible across all 72 instances: the best performing exponents for the uncorrelated ones are significantly smaller than those for the instances with bounded and strongly correlated items.

Interestingly, the best performing exponents for certain classes of TTP instances are in very narrow ranges, e.g. for all instances with uncorrelated weights. For the instances with bounded and strongly correlated items, however, the best performing exponents vary significantly.

Further investigation into these trends is necessary to fully understand why they exist and what particularities of an instance affect the value of the optimal exponent.

6.3 Results and Discussion

In the following, we will show and analyse the performance of the different heuristics on the 72 instances. We compare our heuristics S1–S5, C1–C6, and MIP based heuristic with the heuristic MATLS [6] and with the three heuristics presented in [10]. Note that the study in [6] already shows that MATLS outperforms these three on large instances.

In Table 1 we list all achieved average ratios. Due to space constraints, we only show in this table the results of algorithms with an average approximation ratio $\geq 96\%$. Thus, the three approaches from [10] are missing in the table, as are S2–S4. S1 remains as an important baseline for our analyses. In addition, we show in Figure **??** for all 16 approaches the number of times that a heuristic achieved the best, second best, and third best average ratio.

A closer look into the results yields surprising insights. First, let us look into our group of simple heuristics S1–S5. In Figure **??** we show the number of times that a particular approach performed on average either best or second best in this group. We can see that S5 clearly outperforms our other simple heuristics when it comes to the number of times that its average is the highest. The runner-ups are S2–S4, which shows that all three routines Bitflip, (1+1)-EA and Insertion perform well on instances for which the others do not. In particular the performance of S4 shows that modifications of the tours need to be considered in addition to modifications of the packing plans. Note that the dominating performance of S5 is a first indicator of the importance of the initial TSP tour for the final objective value.

Next, for the analysis of the complex heuristics, let us recall that C3 and C4 sample several starting points in comparison to C1 and C2, and that C5 and C6 are the restart variants of C1 and C2. In Figure **??** we show again the number of times that an approach achieved either the best, second best, or third best average objective value in this group. We can see that C3 and C4 perform best, which we attribute to the fact that both select the best from several TTP solution candidates for the subsequent hill-climbing. The restarting algorithms C5 and C6 perform better than the single-iteration ones, however, they do not perform as well as algorithms C3 and C4 that can explore more TSP tours. Again, the performance of the algorithms that sample more tours indicates that the TSP tours are of high importance.

Interestingly, it is possible to directly compare the effects of resampling tours and of hill-climbing based on a good initial solution: C3/C4 contain S5, even though it is limited to just 10% of the overall computation time. A look into Table 1 reveals that the resampling strategy is the more successful one, since S5 achieves an average approximation ratio of 98.2%, whereas C3 and C4 achieve only about 97%.

Since PackIterative requires only a small number of evaluations, it is very quick compared to the traditional iterative TTP approaches that are allowed to use up to 10 minutes. For example, the runtimes of PackIterative ranges from 15-60 milliseconds for the instances with 195 cities and 1.2-13 seconds for instances with 11,849 cities, to 18-110 seconds for the instances with 85,900 cities. Since S1 performs only a single iteration of PackIterative, we can compare the performance of this basic step with the other approaches in Table 1.

As we can observe in the approximation ratios of S1 and S5, the impact of initial tours on the final solution quality can be significant. On many instances, the restart approach of S5 outperforms any other complex approach. This strongly indicates that, even though the local search routines

[3]Only here we consider the wider range of knapsack sizes 1–10 to better show the trends.

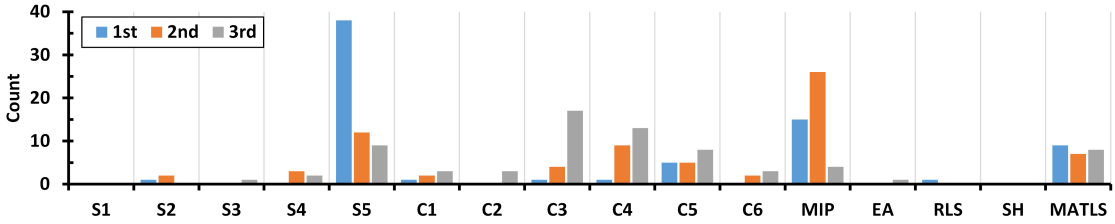

Figure 5: Results for all approaches: ranking across the 72 instances. Shown are the number of times an algorithm's approximation ratio is best, second best, or third best on an instance.

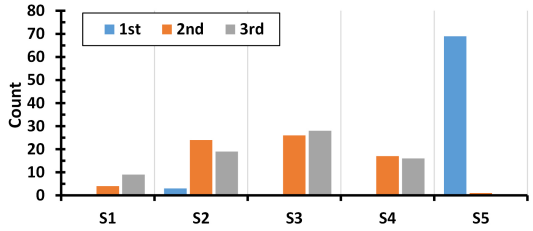

Figure 6: Results of S1–S5: ranking across the 72 instances.

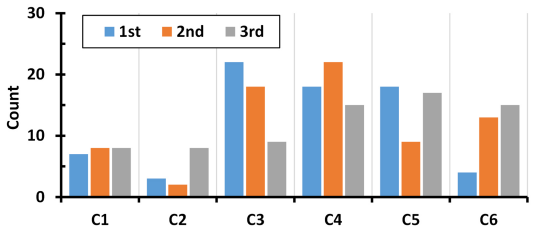

Figure 7: Results of C1–C6: ranking across the 72 instances.

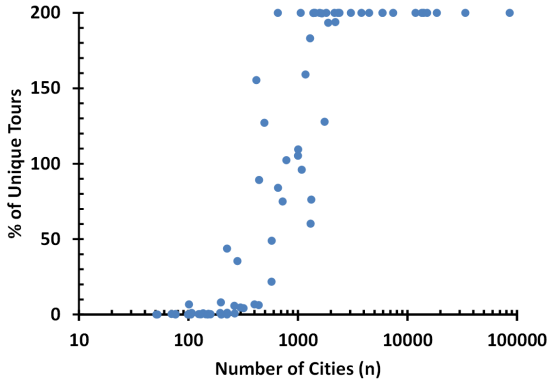

Figure 8: Percentage of unique tours found in repeated Lin-Kernighan runs. Percentages >100 are possible as a single TSP tour can be used in two different directions for TTP purposes.

are problem-specific, their effect is limited. Given our short computation time budget, even the iterative MIP approach achieves just comparable performance.

The results also show that the routines BITFLIP and INSERTION can help to improve the solutions, in particular when we compare the ratios of the complex algorithms C1–C6 with that of S1. The drawback of these routines, however, is their computational complexity. To increase their effectiveness, fast and particular heuristics are necessary to prevent the exhaustive search that both routines currently perform.

Additional noteworthy observations include, amongst others:

- The mixed-integer approach performs second best on average, if we do not consider the instances with n =85,900 cities.
- MATLS cleary outperforms our approaches on a few instances. On average, however, it is not amongst the best-performing algorithms.

Since generating tours can be very time consuming for very large problem instances, it would be an obvious improvement to pre-compute many TSP tours and to randomly pick from these. In practice, the decision to pre-compute can be reasonable, in cases where the locations of delivery or pick-up spots rarely change. In fact, a large number of slightly different tours can be achieved by running CLK multiple times. To investigate this further, we repeatedly run CLK: if the number of cities n <1,000 then we run it 10,000 times, if 1,000 $\leq n <$ 10,000 we run it 1,000 times, and if $n \geq$10,000 we run the heuristic 100 times (see Figure 8). In particular for TSP instances that are based on electronic circuits, which exhibit regular grid like arrangements, often the final tours are identical with the exception of a few nodes that then belong to earlier or later parts of the tour.

Summarizing the results the following conclusion can be made: while problem understanding can help, it is surprising how well a greedy approach with low computational complexity performs compared to more complex ones.

7. CONCLUDING REMARKS

As discovered from the literature and as observed in our experiments, there does not yet seem to be a single best algorithm paradigm for the TTP. So far, constructive heuristics, simple and complex hill-climbers, and also more sophisticated co-evolutionary approaches have been applied to the TTP. With each study, more successful approaches are identified however further investigation is needed to enable understanding to solve the TTP. The interdependence of the TTP is rarely considered, and minimally problem-specific local searches that alternate between solving the TSP and KP components perform best.

It appears that the near future in TTP research will mostly be driven by experiments. Maybe research similar to that in [8, 9, 13], where the difficulty of TSP instances is systematically analyzed, will contribute to the theoretical understanding of the TTP. Computational complexity analyses of simple approaches on simple instances may also provide insights.

n	m	t	F	S1	S5	C3	C4	C5	C6	MIP	MATLS
195	582	bsc	3	88.1	99.6	99.4	99.4	99.6	99.5	100	96.3
			7	88.0	99.2	99.2	99.2	99.2	99.2	99.2	95.6
		unc	3	98.2	99.2	99.3	99.3	99.3	99.3	99.3	99.5
			7	99.2	99.9	100	100	100	100	100	99.8
		usw	3	96.1	98.6	99.0	98.9	99.3	99.0	99.5	98.5
			7	98.2	99.1	99.2	99.2	99.3	99.2	99.3	99.2
	1940	bsc	3	89.6	99.9	99.9	99.9	99.9	99.9	100	98.3
			7	89.2	97.7	97.1	97.5	97.7	97.6	98.1	97.0
		unc	3	96.0	99.1	98.7	98.7	99.3	99.1	99.1	99.0
			7	96.3	98.6	97.7	97.7	98.7	98.5	98.8	99.2
		usw	3	88.4	91.3	91.3	91.6	91.5	91.7	91.4	97.0
			7	92.6	96.3	95.6	95.6	96.4	96.0	96.9	99.4
783	2346	bsc	3	97.8	99.7	99.4	99.4	99.6	99.4	99.7	96.6
			7	95.5	99.3	98.7	98.6	99.1	98.8	99.0	96.6
		unc	3	95.9	98.9	98.5	98.5	98.8	98.5	98.7	98.8
			7	96.3	97.9	97.6	97.7	97.9	97.7	97.8	98.6
		usw	3	95.3	99.1	99.2	99.2	99.5	99.3	99.5	98.7
			7	96.0	99.7	99.6	99.5	99.8	99.5	99.7	98.7
	7820	bsc	3	98.0	99.7	99.6	99.7	99.4	99.5	99.8	94.8
			7	97.0	99.5	99.4	99.4	99.3	99.1	99.5	94.5
		unc	3	96.8	99.3	99.2	99.1	99.2	98.7	99.3	98.5
			7	97.9	99.5	99.4	99.3	99.3	99.1	99.4	99.0
		usw	3	96.0	99.3	99.6	99.6	99.4	99.0	99.6	97.7
			7	95.9	99.3	99.2	99.2	99.2	98.4	99.4	98.1
3038	9111	bsc	3	97.9	99.5	99.1	99.1	98.1	98.0	99.1	93.9
			7	97.6	99.4	99.0	99.0	97.7	97.4	98.8	94.4
		unc	3	98.0	99.7	99.4	99.4	98.5	98.0	99.5	99.0
			7	98.3	99.7	99.5	99.4	98.9	98.5	99.6	99.4
		usw	3	97.0	99.1	99.2	99.1	97.2	97.2	99.2	98.2
			7	97.6	99.6	99.3	99.2	98.4	97.7	99.3	99.0
	30370	bsc	3	98.1	99.6	99.3	99.3	99.0	99.0	99.3	96.7
			7	97.0	99.2	98.7	98.8	98.6	98.2	98.9	95.8
		unc	3	97.1	99.6	99.2	99.1	98.9	98.8	99.3	99.0
			7	97.8	99.5	99.3	99.3	99.2	98.9	99.3	99.2
		usw	3	94.8	98.9	98.2	98.3	97.6	97.6	98.6	98.3
			7	96.2	99.1	98.6	98.4	98.5	97.9	98.6	98.6
11849	35544	bsc	3	97.1	99.2	98.4	98.6	97.3	97.4	97.5	93.5
			7	96.7	98.9	97.9	98.1	96.6	96.7	96.7	93.9
		unc	3	97.4	99.0	98.4	98.5	97.6	97.8	98.0	98.4
			7	97.9	99.5	98.9	99.0	98.0	98.3	98.4	99.3
		usw	3	96.3	98.6	97.8	98.0	96.2	96.5	97.2	97.6
			7	97.1	99.0	98.5	98.5	97.0	97.3	97.3	98.7
	35544	bsc	3	96.7	99.0	98.4	98.3	96.9	97.0	97.9	93.6
			7	96.2	99.2	98.1	98.3	96.2	96.4	97.4	94.0
		unc	3	97.2	99.2	98.6	98.7	97.4	97.6	98.3	98.4
			7	97.4	99.2	98.6	98.4	97.9	97.8	98.5	98.8
		usw	3	95.3	98.3	97.8	97.8	95.6	95.9	97.1	97.6
			7	96.2	98.9	98.1	98.0	96.7	96.7	97.9	98.5
33810	101427	bsc	3	91.3	97.9	95.5	94.0	93.9	92.0	98.3	94.4
			7	91.0	97.9	94.2	95.3	93.8	91.7	96.5	94.1
		unc	3	70.6	73.5	71.4	71.5	71.2	70.9	73.3	75.8
			7	95.1	98.2	96.4	96.0	95.3	95.5	99.9	98.4
		usw	3	90.4	97.5	93.7	93.3	92.5	91.8	96.2	95.9
			7	92.2	98.0	94.7	93.9	93.9	93.5	98.1	97.4
	338090	bsc	3	92.2	97.3	93.8	93.3	92.5	92.4	99.1	93.9
			7	92.6	97.1	94.9	94.3	92.6	93.3	96.9	94.6
		unc	3	94.7	98.3	95.3	95.5	95.8	95.0	97.8	98.0
			7	95.0	98.4	96.0	96.1	96.2	95.7	98.5	98.7
		usw	3	91.3	97.7	93.5	92.8	92.1	92.3	98.3	96.4
			7	93.9	98.3	94.4	95.1	94.1	94.6	99.5	98.3
85900	338090	bsc	3	95.8	98.3	96.3	95.8	95.9	96.4	97.6	-
			7	96.1	97.8	96.8	95.9	96.3	97.1	98.4	-
		unc	3	97.4	98.9	94.2	94.1	97.8	97.7	-	-
			7	97.6	98.6	95.6	95.4	98.0	98.2	98.1	-
		usw	3	95.1	97.6	92.6	92.3	96.1	95.2	-	-
			7	95.8	97.4	93.4	93.3	96.4	96.0	97.9	-
	858990	bsc	3	95.9	96.8	96.2	97.1	95.6	96.3	-	-
			7	96.6	97.2	97.0	97.6	96.6	96.3	-	94.1
		unc	3	97.6	97.5	92.1	92.0	97.6	97.3	-	-
			7	97.9	97.9	94.6	94.5	97.8	97.7	-	98.5
		usw	3	95.7	97.5	91.8	92.7	96.3	95.3	-	-
			7	96.6	96.9	93.2	93.2	96.5	96.3	-	97.9
avg				95.2	98.2	97.0	96.9	96.9	96.8	87.2	84.9
avg^{-85900}				95.0	98.3	97.5	97.4	97.0	96.8	98.1	97.0

Table 1: Approximation ratios achieved. n is the number of cities, m the number of items, t stands for the three KP types, and F stands for the KP size. **avg** is the average ratio achieved across all 72 instances, **avg**$^{-85900}$ across the top 60 instances.

Acknowledgements

This work has been supported by the ARC Discovery Project DP130104395.

References

[1] TTP Test Data. See http://cs.adelaide.edu.au/~optlog/research/ttp.php.

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

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

[4] M. R. Bonyadi, Z. Michalewicz, M. R. Przybylek, and A. Wierzbicki. Socially inspired algorithms for the travelling thief problem. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 421–428, New York, NY, USA, 2014. ACM.

[5] S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424, Mar. 1999.

[6] Y. Mei, X. Li, and X. Yao. Improving efficiency of heuristics for the large scale traveling thief problem. In *Simulated Evolution and Learning (SEAL)*, volume 8886 of *LNCS*, pages 631–643. Springer, 2014.

[7] Y. Mei, X. Li, and X. Yao. On investigation of interdependence between sub-problems of the travelling thief problem. *Soft Computing*, pages 1–16, 2014.

[8] O. Mersmann, B. Bischl, H. Trautmann, M. Wagner, J. Bossek, and F. Neumann. A novel feature-based approach to characterize algorithm performance for the traveling salesperson problem. *Annals of Mathematics and Artificial Intelligence*, 69(2):151–182, 2013.

[9] S. Nallaperuma, M. Wagner, and F. Neumann. Parameter prediction based on features of evolved instances for ant colony optimization and the traveling salesperson problem. In *Parallel Problem Solving from Nature PPSN XIII*, volume 8672 of *LNCS*, pages 100–109. Springer, 2014.

[10] S. Polyakovskiy, M. R. Bonyadi, M. Wagner, Z. Michalewicz, and F. Neumann. A comprehensive benchmark set and heuristics for the traveling thief problem. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 477–484. ACM, 2014.

[11] S. Polyakovskiy and F. Neumann. Packing while traveling: Mixed integer programming for a class of nonlinear knapsack problems. In *Integration of AI and OR Techniques in Constraint Programming*, volume 9075 of *LNCS*, pages 330–344. Springer, 2015.

[12] G. Reinelt. TSPLIB - A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[13] K. Smith-Miles, J. van Hemert, and X. Y. Lim. Understanding TSP difficulty by learning from evolved instances. In *Learning and Intelligent Optimization (LION)*, pages 266–280. Springer, 2010.