Comparative Study by Using a Greedy Approach and Advanced Bio-Inspired Strategies in the Context of the Traveling Thief Problem

Julia Garbaruk, Doina Logofătu, Florin Leon

Comparative Study by Using a Greedy Approach and Advanced Bio-Inspired Strategies in the Context of the Traveling Thief Problem

Julia Garbaruk[1][0000−0002−7039−4901], Doina Logofătu[1][0000−0002−1678−3527], and Florin Leon[2][0000−0002−1370−9145]

[1] Frankfurt University of Applied Sciences, Frankfurt am Main, Germany
j.garbaruk@outlook.de
logofatu@fb2.fra-uas.de
[2] Gheorghe Asachi Technical University of Iaşi, Romania
florinleon@tuiasi.ro

Abstract. Traveling Salesman Problem and Knapsack Problem are perhaps the best-known combinatorial optimization problems that researchers have been racking their brains over for many decades. But they can also be combined into a single multi-component optimization problem, the Traveling Thief Problem, where the optimal solution for each single component does not necessarily correspond to an optimal Traveling Thief Problem solution. The aim of this work is to compare two generic algorithms for solving a Traveling Thief Problem independently of the test instance.

Keywords: Traveling Thief Problem · Combinatorial Optimization · Evolutionary Algorithms.

1 Introduction

In real-world economy, we often have to do with different components that influence each other. For example, if we want to reduce transport costs, we can try to avoid empty runs, enter into collaborations with other companies, join transport networks, combine transports or, for example, only arrange removal on certain days of the week in order to purchase larger quantities. This in turn may have negative effects on the availability of the products or on-time delivery to customers. So you can say that different components are always related and that we usually cannot optimize one component without it having a negative impact on others. It is similar with many areas of our everyday life - for example, when we buy food, we usually make sure that it is of high quality on the one hand and that the costs are as low as possible on the other. If we decrease the cost of spending on a product, it is very likely at the expense of quality. Conversely, if we increase our quality standards, the price usually also deteriorates. As you can see, the principles of multi-objective optimization play an important role in our lives.

The Traveling Thief Problem (TTP), which was presented by Bonyadi et al in 2013, is an NP-hard combinatorial optimization problem that combines two well-known subproblems: the Traveling Salesman Problem (TSP) and the Knapsack Problem (KNP). The motivation behind this was to systematically investigate the interactions between two hard component problems and to use this knowledge to solve real-world problems more efficiently. [8] Research over the past few years has shown that there are many ways to resolve a TTP. Most of the publications aimed to solve it by using different types of algorithms: Heuristic Based, Local Search, Coevolution, Evolutionary Algorithm, Ant Colony Optimization etc. But the success of the algorithm used depends very much on the selected test instance. In our work we wanted to introduce two relatively simple, generic algorithms with which one can solve any TTP and produce results relatively quickly.

2 Problem Description

2.1 Traveling Thief Problem

The Traveling Thief Problem [1], [2] combines the TSP and the KNP in the following way: The traveling thief can collect items from each city he is visiting. The items are stored in a knapsack carried by him. In more detail, each city π_i provides one item, which could be picked by the thief. z is a binary vector, where each element shows if the item at city j where picked or not.

There is an interaction between the subproblems: The velocity of the traveling thief depends on the current knapsack weight w, which is carried by him. It is calculated by considering all cities, which were visited so far, and summing up the weights of all picked items. The weight at city i given π and z is calculated by:

$$w(i, \pi, z) = \sum_{k=1}^{i} \sum_{j=1}^{m} a_j(\pi_k) w_j z_j \tag{1}$$

The function $a_j(\pi_k)$ is defined for each item j and returns 1 if the item could be stolen at city π_k and 0 otherwise. The current weight of the knapsack has an influence on the velocity. When the thief picks an item, the weight of the knapsack increases and therefore the velocity of the thief decreases. The velocity v is always in a specific range $v = [v_{min}, v_{max}]$ and could not be negative for a feasible solution. Whenever the knapsack is heavier than the maximum weight Q, the capacity constraint is violated. However, to provide also the traveling time for infeasible solutions the velocity is set to v_{min}, if $w > Q$:

$$v(w) = \begin{cases} v_{max} - \frac{w}{Q} \cdot (v_{max} - v_{min}) & \text{if } w \leq Q \\ v_{min} & \text{otherwise} \end{cases} \tag{2}$$

If the knapsack is empty the velocity is equal to v_{max}. Contrarily, if the current knapsack weight is equal to Q the velocity is v_{min}. Furthermore, the traveling time of the thief is calculated by:

$$f(\pi, z) = \sum_{i=1}^{n-1} \frac{d_{\pi_i, \pi_{i+1}}}{v(w(i, \pi, z))} + \frac{d_{\pi_n, \pi_1}}{v(w(n, \pi, z))} \tag{3}$$

The calculation is based on TSP, but the velocity is defined by a function instead of a constant value. This function takes the current weight, which depends on the index i of the tour. The current weight, and therefore also the velocity, will change on the tour by considering the picked items defined by z. In order to calculate the total tour time, the velocity at each city needs to be known. For calculating the velocity at each city the current weight of the knapsack must be given. Since both calculations are based on z and z is part of the knapsack subproblem, it is very challenging to solve the problem to optimality. In fact, such problems are called interwoven systems as the solution of one subproblem highly depends on the solution of the other subproblems.

Here, we leave the profit unchanged to be calculated as in the KNP problem. Finally, the TTP problem is defined by

$$\begin{aligned}
&min\ f(\pi, z) && traveling\,time \\
&max\ g(z) && profit \\
f(\pi, z) &= \sum_{i=1}^{n-1} \frac{d_{\pi_i, \pi_{i+1}}}{v(w(i, \pi, z))} + \frac{d_{\pi_n, \pi_1}}{v(w(n, \pi, z))} \\
g(z) &= \sum_{j=1}^{m} z_j b_j \\
s.t.\ &\pi = (\pi_1, \pi_2, ..., \pi_n) \in P_n \\
&\pi_1 = 1 \\
&z = (z_1, ..., z_m) \in \mathbb{B}^m \\
&\sum_{j=1}^{m} z_j w_j \leq Q
\end{aligned} \tag{4}$$

2.2 Test Data (Input)

For our research we have used the test instances of the TTP provided by the organizators of GECCO 2019 (Bi-objective Traveling Thief Competition). They are very versatile and provide all the information needed to construct a Traveling Thief Problem. First, the general parameters such as number of cities or items in the test instance are described. This is followed by a list with the coordinates of all cities as well as a list of profit, weight and the city of each item.
The 9 test instances differ in the following aspects:

- Number of cities (smallest value: 280 cities, highest value: 33810 cities)

- Number of items (smallest value: 279 items, highest value: 338090 items)
- Capacity of Knapsack (smallest value: 25936, highest value: 153960049)
- Knapsack data type (bounded strongly correlated, uncorrelated and similar weights or uncorrelated)

Moreover we have determined that we consider a maximum of 100 solutions in the Pareto front for each test instance.

2.3 Result Format (Output)

For each problem two output files are generated: One file containing the tour and packing plan (*test.x*) and one file containing the time and profit for each solution (*test.f*).

test.x contains for each solution two lines, where the first represents the permutation vector and the second line the packing plan encoded by 0 and 1. The output might look like this:

```
1 2 3 4
0 0 0

1 4 3 2
0 0 0

1 2 3 4
0 0 1

1 4 3 2
1 0 0

1 4 3 2
0 1 0

1 3 2 4
1 0 1

1 2 3 4
0 1 1

1 4 3 2
1 1 0
```

test.f contains the corresponding time and profit separated by space. An example output might look like this:

```
20.0000000000000000 0.0000000000000000
```

```
20.0000000000000000 0.0000000000000000
20.9279869067103130 25.0000000000000000
22.0377358490566020 34.0000000000000000
27.3636363636363630 40.0000000000000000
28.5852929784761830 59.0000000000000000
33.1072075335023540 65.0000000000000000
38.9144385026737900 74.0000000000000000
```

3 Algorithms

3.1 Greedy Algorithm

In this approach we solve the two sub-problems separately or independently of each other. First of all, we need a TSP solver. To find the shortest path, we chose the Nearest Neighbor Algorithm, which is a heuristic method from the graph theory. Starting from a node as a starting point, the minimum weighted adjacent edge is selected to the next node. This will be continued successively until all nodes have been combined into a Hamiltonian circle. The result of this is a tour π^*, in our case starting at city 1. As long as the maximum capacity constraint and maximum number of solutions are not violated, items are picked. The *getNextItemGreedily* method returns the next item which is not picked so far and provides the best $\frac{b_i}{w_i}$ rate. Then the picking decision vector z is combined with π^* and $sym(\pi^*)$ and is added to the set. Finally the resulting Pareto front is returned.

Algorithm 1 Greedy Algorithm

Require: $P \leftarrow$ Traveling Thief Problem
  $\pi^* \leftarrow$ nearestNeighborHeuristic
  $z \leftarrow z^{empty}$
  front $\leftarrow \{F(\pi^*, z), F(\text{sym}(\pi^*), z)\}$
  while $\sum_{j=1}^{m} w_j z_j \leq Q$ **and** front.size $<$ max. number of solutions **do**
    $i \leftarrow$ getNextItemGreedily(P, z) ▷ Based on the best profit/weight rate
    $z_i \leftarrow 1$
    add(front, $F(\pi^*, z)$)
    add(front, $F(\text{sym}(\pi^*), z)$
  end while
  return front

3.2 Evolutionary Algorithm (NSGA-II)

Evolutionary algorithms are popular approaches to generate Pareto optimal solutions. The main advantage of evolutionary algorithms, when applied to solve

Algorithm 2 Nearest Neighbor Heuristic

```
route ← 1                                          ▷ route starts at city 1
j ← 1                                              ▷ next city
l ← j                                              ▷ current city
W ← {1, 2, ..., n}\{j}
while W ≠ ∅ do
    Let j ∈ W such that c_lj = min{c_li | j ∈ W}
    Connect l to j and update the route
    W ← W\{j}
    l ← j
end while
return route
```

multi-objective optimization problems, is the fact that they typically generate sets of solutions, allowing computation of an approximation of the entire Pareto front. The algorithm we used for solving the Traveling Thief problem is the NSGA-II.
As usual for evolutionary algorithms all individuals in NSGA-II are factored and added to the population in the beginning. Then each individual gets a rank based on its level of domination and a crowding distance which is used as a density estimation in the objective space. The binary tournament selection compares rank and crowding distance of randomly selected individuals in order to return individuals for the recombination. After executing recombination and mutation the offspring is added to the population. In the end of each generation the population is truncated after sorting it by domination rank and crowding distance.

Non-dominated Sort The-non dominated sort represents the core component of the whole algorithm. It splits the population into sets of non-dominated solutions and orders them into a hierarchy of non-dominated Pareto fronts. Based on this, the solutions in a Pareto front are assigned a rank. The rank is needed to calculate the fitness of a solution. Solutions with the highest rank (smallest rank number) are considered the fittest.

Crowding Distance The Crowding Distance provides an estimate of the density of solutions surrounding a solution. The Crowding Distance of a particular solution is the average distance of the two adjacent solutions (along all objective functions (in this case time and profit)). Solutions with higher Crowding Distance are considered good, since it guarantees diversity and spread of solutions.

Tournament Selection The tournament selection selects randomly two parents from the population. From these two parents, a winner is determined. With the probability p, the parent with the higher fitness will be selected. Accordingly, the worse solution is selected with the probability $1 - p$. It is important to give a worse solution a chance to witness offspring, because otherwise there is a risk

of premature convergence because of too strong selective pressure towards best solution.
If the Tournament Selection was done twice, we have two winners, which then produce the offspring.

Recombination We have to keep in mind that each city in the path may only appear once. This condition applies both to parents and to newly created paths. If a city occurs several times as a result of the recombination and some another city does not occur at all, this must be repaired. We used a simple recombination method that ensures from the outset that each city occurs exactly once. One of the parents ($p1$) inherits a randomly selected area of its path to the child. The rest of the path is populated with the values from the other parent ($p2$). For this, the algorithm iterates through the entire path of $p2$ and check whether the currently viewed city is already included in the child path. If not, it will be added to the next free place in the child path. Of course, the packing plan must also be recombined. To do this, we performed a simple crossover operation at a random point of the packing list. After the recombination of the packing plan, the maximum capacity of the knapsack may have been exceeded. To fix this, randomly selected items are removed from the packing list until the knapsack has reached a permissible weight again.

Mutation Since each city must be visited exactly once, it makes sense to simply swap two randomly chosen cities. This is possible without any problem, since there is a connection from every city to every other city.
A mutation can also occur in the packing list. We opted for a simple mutation method in which a randomly determined bit that represents an item or rather its status (collected or not collected) is toggled. After the packing plan has been mutated, the maximum capacity of the knapsack may have been exceeded. To fix this, randomly selected items are removed from the packing list until the knapsack has reached a permissible weight again.

4 Experimental Results

We show our test results using the test instance with the following parameters:
- 280 cities
- 279 Items
- Knapsack capacity of 25936

The profit and the weight of the items correlate strongly (the correlation is ignored here because the methods are generic).

Since NSGA-II has more additional parameters, such as mutation rate and population rate, we have also tried various possible combinations of these parameters in order to better compare the results with the greedy approach.

Algorithm 3 Fast Non-dominated Sort

Require: Popualtion P
  for each $p \in P$ **do**
    $S_p = \emptyset$ ▷ S_p: set of solutions that the solution p dominates
    $n_p = 0$ ▷ n_p: nr. of solutions which dominate the solution p
    for each $q \in P$ **do**
      if $p < q$ **then** ▷ if p dominates q
        $S_p = S_p \cup \{q\}$ ▷ add q to the set of solutions dominated by p
      else if $q < p$ **then**
        $n_p = n_p + 1$ ▷ increment the domination counter of p
      end if
    end for
    if $n_p = 0$ **then** ▷ p belongs to the first front
      $p_{rank} = 1$
      $F_1 = F_1 \cup \{p\}$
    end if
  end for
  $i = 1$ ▷ initialize the front counter
  while $F_i \neq \emptyset$ **do**
    $Q = \emptyset$ ▷ used to store the members of the next front
    for each $p \in F_i$ **do**
      for each $q \in S_p$ **do**
        $n_q = n_q - 1$
        if $n_q = 0$ **then** ▷ q belongs to the next front
          $q_{rank} = i + 1$
          $Q = Q \cup \{q\}$
        end if
      end for
    end for
    $i = i + 1$
    $F_i = Q$
  end while
  return $\{F_1, F_2, ...\}$ ▷ return all Pareto fronts in the population

Algorithm 4 Crowding Distance Assignment

Require: Front F
  $l \leftarrow |F|$ ▷ number of solutions in the front F
  for each i **do**
    $F[i]_{distance} = 0$ ▷ set the distance to 0 at the beginning
  end for
  for each objective m **do** ▷ for time and profit
    $F = sort(F, m)$ ▷ sort using each objective value
    $F[0]_{distance} = \infty$ ▷ so that boundary points are always selected
    $F[l-1]_{distance} = \infty$ ▷ so that boundary points are always selected
    for $i = 1$ to $l - 2$ **do** ▷ for all other points
      $F[i]_{distance} = F[i]_{distance} + (F[i+1].m - F[i-1].m)/(f_m^{max} - f_m^{min})$

In our first example we compare the results of the Greedy Algorithm with NSGA-II depending on the population size of the NSGA-II. The appropriate population size is still the subject of discussion. Many different approaches are known: For example, there is the 10x*Dimensions* rule, according to which you get the right number of individuals for the population by multiplying the number of objectve functions by 10. Some others recommend large populations, which should be divided into subpopulations. The population size also does not always have to be static, but could, for example, shrink over time. However, the best way is always to test different population sizes and get a solution tailored to the specific problem and its parameters.
We decided to test different static population sizes, from 20 up to 500. We have set the number of generations to 10,000, since from there the changes take place slower and you can still see the tendency. The mutation rate remained constant at 4%. As you can see in Fig. 1, the results of the NSGA-II increase significantly with increasing population size, but only up to a certain limit. Up to a population size of 100, the rule "the more the better" seems to apply. At a population size of 200, however, it turns out that the results are not getting much better and the range of values is also shrinking as the solutions seem to converge more and more. With a population size of 500, this problem becomes even clearer - the curve that forms the Pareto front is getting very short, which means is that the tours and packing plans are becoming increasingly similar. We therefore consider a population size of ca. 100 to be recommended in this case. Nonetheless, NSGA-II scores significantly worse here than the simple Greedy Algorithm. Not only in relation to the approximation of the Pareto curve to the axes, but also in relation to the range of the solutions. This is noteworthy because Greedy Algorithm only creates a single route using the Nearest Neighbor Algorithm which, as we know, is almost never optimal, while NSGA-II is able to create many different paths. It seems that a single path that has a high probability of getting close to optimum is better than trying to optimize several random paths.

We now consider how the results change when we change the mutation rate of the NSGA-II algorithm. From now on, we use the population size that has been found to be optimal for the respective test case (100).

The present results show that it is definitely advisable to play with the mutation rate. Increasing the mutation rate has an effect similar to increasing the population rate. Above a certain value, however, this has an adverse effect because too many good individuals are rejected, which leads to an overall deterioration in the gene pool. In this test case, the mutation rate should not be more than approx. 30%. We were surprised that such a high mutation rate leads to improved results. Usually a single digit mutation rate is the rule. But here, too, the result of the NSGA-II does not come close to the result of the Greedy Algorithm, which is very astonishing.
In all other test instances, it was also observed that the greedy algorithm delivers better results on balance, regardless of the setting of the population size and the mutation rate.

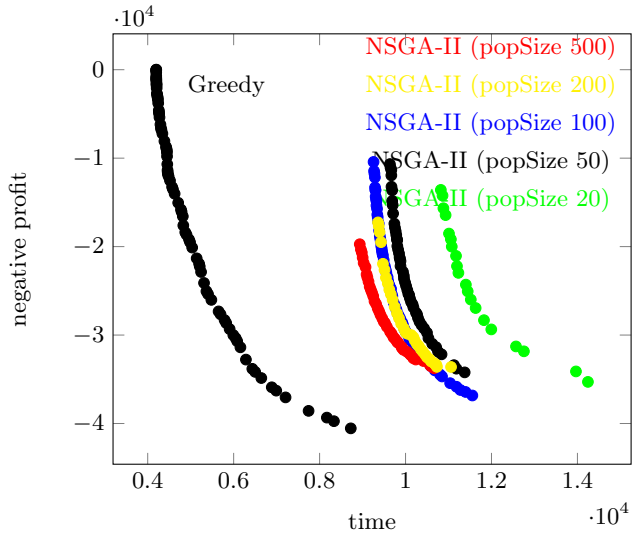

Fig. 1. Results of the NSGA-II for the Traveling Thief Problem depending on the population size after 10,000 generations in contrast to the Greedy Algorithm (test case 1)

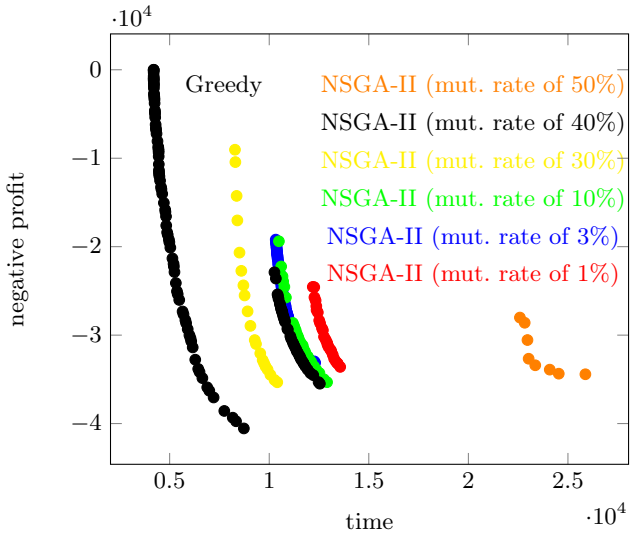

Fig. 2. Results of the NSGA-II for the Traveling Thief Problem depending on the mutation rate after 10,000 generations in contrast to the Greedy Algorithm (test case 1)

5 Conclusion

In this work we compared two different approaches to solving the Traveling Thief Problem - a simple greedy algorithm and an advanced evolutionary algorithm. Surprisingly, the greedy algorithm performed far better, although it can only create a single route for the thief. The advantage of the evolutionary algorithm of creating different paths is likely not to be fully effective because the paths are generated randomly. Given their size, they cannot achieve a satisfactory result, regardless of how the population size and the mutation rate are set.

This is definitely an interesting observation, because it is generally assumed that an evolutionary algorithm can predict the outcome much better than simple heuristic methods. One idea for the future would be to work with an optimized initial population in order to give the algorithm a head start and to take full advantage of its special features. The result of the greedy algorithm could be used for this. In order to generate different routes, one could, for example, choose the second best path from one node to the next at a random point while the path is being created. As result, the following route would change every time and differ from others, but at the same time it would still be much better than a purely randomly generated route.

References

1. J. Blank, K. Deb, and S. Mostaghim, "Solving the bi-objective traveling thief problem with multi-objective evolutionary algorithms" In International Conference on Evolutionary Multi-Criterion Optimization, pp. 46–60, Springer, 2017.
2. M. R. Bonyadi, Z. Michalewicz, and L. Barone, "The travelling thief problem: The first step in the transition from theoretical problems to realistic problems", 2013 IEEE Congress on Evolutionary Computation, pp. 1037–1044.IEEE, 2013.
3. K. Deb, A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-2.IEEE Trans. Evol. Comput., 6(2):182–197, 2002.
4. Y. Mei, X. Li, and X. Yao, "On investigation of interdependence between subproblems of the travelling thief problem", Soft Computing, 20(1):157–172, 2016.
5. N. Srinivas and K. Deb, "Muiltiobjective optimization using non-dominated sorting in genetic algorithms", Evolutionary computation, 2(3):221-248,1994.
6. A. Zamuda and J. Brest, "Population Reduction Differential Evolution with Multiple Mutation Strategies in Real World Industry Challenges", Artificial Intelligence and Soft Computing - ICAISC 2012, 2012, vol. 7269/2012, pp. 154–161.
7. J.Garbaruk and D. Logofatu, Convergence Behaviour of Population Size andMutation Rate for NSGA-II in the Context of the Traveling Thief Problem
8. M. Wagner, M. Lindauer, M. Mısır, S. Nallaperuma, F. Hutter, A case study of algorithm selection for the traveling thief problem