

# Optimization methods

## Starting project: Traveling Thief Problem

Murad Shahbazov  
251213

## Theoretical Introduction:

The Traveling Thief Problem (TTP) is a complex issue that arises from combining two simpler, yet challenging problems often encountered in the world of optimization and decision-making. Before we delve into the TTP, let's briefly understand its components: the Traveling Salesman Problem (TSP) and the Knapsack Problem (KP).

### The Traveling Salesman Problem (TSP):

Imagine you're a salesman who needs to visit a list of cities. The goal is to figure out the shortest possible route that takes you through all the cities once and brings you back to where you started. This problem might sound straightforward, but finding the best route quickly becomes very tricky as the number of cities increases. This is what the TSP is all about.

### The Knapsack Problem (KP):

Now, consider you have a bag (or "knapsack") with a weight limit, and you're faced with a variety of items, each with its own weight and value. The challenge here is to pick the items that give you the highest value without exceeding the weight limit of your bag. Like the TSP, the KP is easy to understand but hard to solve, especially with a lot of items to choose from.

### The Traveling Thief Problem (TTP):

The TTP brings these two problems together in a unique way. It imagines a scenario where a thief travels through various cities (like in the TSP) and at each city, has the chance to steal some items (like in the KP). However, there's a catch: carrying stolen items slows the thief down, making the travel between cities longer. The thief's goal is to maximize the total value of stolen items while considering the impact of their weight on travel time. This problem is fascinating because the best path to take and the best items to steal depend on each other. The more items the thief carries, the slower they move, affecting the overall journey. Solving this problem involves finding the right balance between the route taken, the items selected, and the total travel time.


Both the TSP and KP are known to be NP-hard problems. This means that as the size of the problem grows (more cities or items), it becomes exponentially harder to find the exact solution in a reasonable amount of time.

## Optimizer Description :

In addressing the Traveling Thief Problem (TTP), several algorithmic strategies can be employed, each with its own set of strengths and weaknesses. For this particular exploration, four distinct approaches were chosen: the Random Search Algorithm, the Greedy Algorithm, the Simulated Annealing, and the Evolutionary Algorithm. These methods range from simple to complex, offering a broad perspective on potential solutions. Let's start by discussing the Random Search Algorithm.

### Random Search Algorithm

The Random Search Algorithm stands as the most straightforward among the three. It operates on the principle of chance, generating solutions randomly in every iteration. Here's how it works:



```
1  def generate_random_solution(self):
2      route = list(range(1, len(self.nodes) + 1))
3      random.shuffle(route)
4
5      items_selected = [-1] * len(self.items)
6      total_weight = 0
7
8      for node_id in route:
9          node_items = [item for item in self.items if item[3] == node_id]
10         for item in node_items:
11             if random.random() < 0.5:
12                 item_index, _, item_weight, _ = item
13                 new_total_weight = total_weight + item_weight
14
15                 if ksp_is_valid(new_total_weight, self.params["knapsack_capacity"]):
16                     items_selected[item_index - 1] = item_index
17                     total_weight = new_total_weight
18                     break
19
20         return Solution(route, items_selected)
21
```

1. Route Generation: A list of all the cities is shuffled to create a random route for each iteration. This randomness ensures that we explore a wide variety of routes without any bias.

2. Item Selection: For every city on the route, the algorithm randomly decides whether to take any items available in that city, making sure not to exceed the knapsack's weight limit. This decision is made independently for each item, adding an additional layer of randomness to the solution.

The Random Search Algorithm is pretty straightforward: in each iteration, it generates a random solution and adds it to a list of solutions. After many iterations, we can sift through this list to pick out the best solution the algorithm found. It's easy to implement, offering a direct approach to exploring solutions, though it might not always lead to the most optimal outcomes.

## Greedy Algorithm

Next up, we have the Greedy Algorithm. This method takes a more calculated approach compared to the Random Search. Using the nearest neighbor strategy, it aims to maintain the best possible route by choosing the closest next city to visit. Once the route is determined, the algorithm then selects the most valuable item available at each node, taking the knapsack size into account to ensure the total weight doesn't exceed the limit. To explore more possibilities and diversify the routes considered, the algorithm generates a starting node randomly in each iteration. After determining a route and selecting items for each iteration, the solution is added to the list of solutions. At the end of all iterations, the algorithm evaluates this list to identify the best solution based on the fitness value.

```
1  def run(self, iterations=100):
2      for _ in range(iterations):
3          solution = self.generate_greedy_solution()
4          fitness_value = fitness(
5              self.params, self.nodes, self.items, solution, greedy=True
6          )
7          self.fitness_values.append(fitness_value)
8
9      self.best_solution = best_fitness(self.fitness_values)
10     self.worst_solution = worst_fitness(self.fitness_values)
11     self.avg_solution = avg_fitness(self.fitness_values)
12     return self.best_solution
```

## Evolutionary Algorithm

Next on our list is the Evolutionary Algorithm, a unique approach compared to the earlier ones. It begins by creating a pool of possible solutions, where 80% are crafted using the Greedy Algorithm's smart decisions, and the remaining 20% come from the Random Search's guesses. This strategy provides a solid starting point with a rich variety of options :

```
1  def run(self):
2      self.initialize_population()
3
4      for generation in range(self.generations):
5          new_population = []
6          for _ in range(self.population_size):
7              # Selection
8              parent1, parent2 = (
9                  self.tournament_selection(),
10                 self.tournament_selection(),
11             )
12
13             # Cross-over
14             if random.random() < self.crossover_rate:
15                 child = self.crossover_ox(parent1, parent2)
16             else:
17                 child = random.choice([parent1, parent2])
18
19             # Mutation
20             if random.random() < self.mutation_rate:
21                 child = self.swap_mutation(child)
22             fitness(self.params, self.nodes, self.items, child)
23             new_population.append(child)
24
25             # Elitism
26             self.population = sorted(
27                 self.population + new_population, key=lambda x: x.fitness, reverse=True
28            )[: self.population_size]
29             self.
```

The detailed process for each population:

1. **Tournament Selection:** This is where the algorithm picks two "parent" solutions that are doing well. It's like a mini-competition where the strongest solutions are more likely to become parents for the next generation.
2. **OX Crossover (Ordered Crossover):** With a certain chance, this step mixes the parents' routes to create a "child." This method carefully selects a portion from one parent and fills in the rest from the other, trying to keep the best traits of both. For deciding which items the child should carry, it simply picks the items list from one of the parents at random. This way, it hopes to inherit good item choices without complicating the crossover step.
3. **Swap Mutation:** Sometimes, to shake things up, the algorithm swaps two cities in the child's route. This is like making a tiny change to see if it can stumble upon an even better route.
4. **Elitism:** After creating a new generation, the algorithm only keeps the best solutions from both the old and new groups. This means that no matter what, the top performers stick around for the next round, ensuring that the quality of solutions only goes up.

The cycle of generating new solutions, mixing them, making slight changes, and then picking the best to keep going repeats several times. Each "generation" aims to be slightly better than the last, gradually improving the pool of solutions.

### **Simulated annealing:**

The Simulated Annealing algorithm starts with an initial solution obtained through a Greedy Algorithm. This technique iteratively refines the solution by introducing minor alterations to the route, inspired by metallurgy's annealing process. As the "temperature" lowers, the algorithm becomes less likely to accept suboptimal solutions, enabling a wide search at first, which narrows down to the best solutions discovered.

```

1  def run(self, iterations=100):
2      while (
3          self.temperature > self.stopping_temperature and self.iteration < iterations
4      ):
5          candidate_solution = self.mutate_solution(self.current_solution)
6          candidate_fitness = fitness(
7              self.params, self.nodes, self.items, candidate_solution
8          )
9          current_fitness = fitness(
10             self.params, self.nodes, self.items, self.current_solution
11         )
12
13         if random.random() < self.accept_probability(
14             candidate_fitness, current_fitness
15         ):
16             self.current_solution = candidate_solution
17             if candidate_fitness > fitness(
18                 self.params, self.nodes, self.items, self.best_solution
19             ):
20                 self.best_solution = candidate_solution
21
22             self.temperature *= self.alpha
23             self.iteration += 1
24
25     return self.best_solution.fitness

```

A crucial feature is the *mutate\_solution* function, which perturbs the current solution in hopes of finding a better one. The algorithm evaluates each new solution's fitness, comparing it to the current one. If the new solution is better, or if it passes a probability check influenced by the temperature, it becomes the new current solution. This process repeats until the temperature reaches a predefined stopping point or a set number of iterations are completed, aiming to balance exploration and exploitation throughout the search

### Fitness calculation:

All solutions are rated using the fitness method, which judges a solution based on the profit from items collected and the journey's duration. The goal is to increase profit while reducing the time spent traveling. Solutions with higher fitness scores are better, as they effectively balance these factors, making them the top choices.

```

1  current_speed = (
2      max_speed - total_weight * (max_speed - min_speed) / knapsack_capacity
3  )
4
5      total_distance += distance
6      traveling_time = distance / current_speed
7      total_traveling_time += traveling_time
8  fitness = total_profit - total_traveling_time

```

## Results:

The tables below display experiments with the parameters of the Evolutionary Algorithm and their impact on the solutions.

Tested instance: eil51\_n150\_uncorr-similar-weights\_01.ttp

population\_experiment

Parameter Value	Best	Worst	Avg
<b>50</b>	9417.875972487305	7014.677981434995	9306.219630391131
<b>100</b>	9763.10708829592	7023.240227520699	9679.441555132966
<b>200</b>	10126.777990078825	7022.327541079832	9982.4066053948
<b>400</b>	10179.480834193213	6960.688081112847	10043.82877133814
<b>500</b>	10212.632798575007	6995.9258210897515	10102.596653708018

mutation\_experiment

Parameter Value	Best	Worst	Avg
<b>0.01</b>	9788.359318542707	6994.59076602907	9713.6741420468
<b>0.1</b>	9905.540030222664	6988.065320446634	9776.254990694875
<b>0.2</b>	9769.413180456751	7090.493573401178	9606.496449917062
<b>0.3</b>	9898.747301025223	6870.87162557735	9786.201025628556
<b>0.5</b>	9831.320109885226	6824.490714696758	9648.25762872222

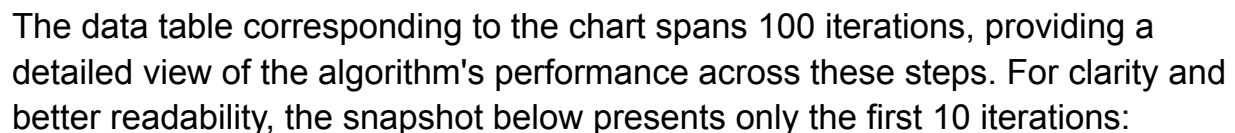
crossover\_experiment

Parameter Value	Best	Worst	Avg
<b>0.3</b>	10037.571254058896	6929.856395148442	9917.90293057651
<b>0.4</b>	9521.745065507996	6908.175436235742	9452.307587794015
<b>0.5</b>	9782.62891835877	7022.327541079832	9707.512467356097
<b>0.7</b>	9778.699981440574	6984.01181112803	9678.549187708313
<b>0.9</b>	9540.026594654104	7091.915663053095	9505.188342315236



The graph displays the 'Best Fitness' value on the y-axis (ranging from 9200 to 9900) against 'Iterations' on the x-axis (ranging from 0 to 1000). The fitness value starts at approximately 9420 at iteration 0, rises sharply to 9500 by iteration 50, then to 9700 by iteration 100. It remains at 9700 until iteration 700, where it jumps to 9810. It then rises to 9820 at iteration 800, 9830 at iteration 900, and finally reaches 9840 at iteration 1000.

Iterations	Best Fitness
0	9420
50	9500
100	9700
700	9700
800	9810
900	9830
1000	9840



Iteration	Best Fitness	Average Fitness	Worst Fitness
1	9421,276527517140	7735,731756839950	7056,408266754670
2	9421,276527517140	8657,335246026720	7942,711781639180
3	9490,324833011500	9207,395228634100	8804,602000830830
4	9490,324833011500	9399,044484390200	9333,018938755670
5	9490,324833011500	9447,989168688260	9421,276527517140
6	9704,100641469170	9472,575879908270	9454,034311578440
7	9704,100641469170	9500,933420236340	9490,324833011500
8	9704,100641469170	9541,531842394830	9490,324833011500
9	9704,100641469170	9683,294593652950	9588,935449539790
10	9704,100641469170	9704,100641469170	9704,100641469170

The results, derived from the experiments detailed above and from running five additional other instances of the problem, have been compiled and are stored in the project's 'results' folder. The tests were conducted using the following parameters: population size of 100, 100 generations, a crossover rate of 0.7, a mutation rate of 0.01, and a tournament size of 5.

Below is a comparison of the results from all algorithms for the given problem, with the tables illustrating the outcomes. The parameters for the Evolutionary Algorithm were consistent across all cases, ensuring a fair comparison.

overall\_results

File Name	Algorithm	Best Fitness	Worst Fitness	Average Fitness
berlin52_n51_bounded-strongly-corr_01.ttp	Random Search	-119781,96960744200	-289616,515018515	-189966,77874114800
berlin52_n51_bounded-strongly-corr_01.ttp	Greedy Algorithm	-27260,8431251218	-74855,94845438750	-52260,07113197120
berlin52_n51_bounded-strongly-corr_01.ttp	Evolutionary Algorithm	-8321,835806940290	-70000,77763668840	-9146,750481268160
berlin52_n51_bounded-strongly-corr_01.ttp	Simulated Annealing	-25427,843696113100	-	-
eil51_n50_uncorr_01.ttp	Random Search	-2395,402634277340	-13265,680989980700	-8103,6565216424700
eil51_n50_uncorr_01.ttp	Greedy Algorithm	2901,7209278935000	-2640,311169326880	492,007411315472
eil51_n50_uncorr_01.ttp	Evolutionary Algorithm	4414,726931360120	-765,9296198948660	4291,108362878870
eil51_n50_uncorr_01.ttp	Simulated Annealing	1984,42404889738	-	-
berlin52_n51_uncorr_01.ttp	Random Search	-111109,61169543500	-277700,3485218280	-199949,42421800000
berlin52_n51_uncorr_01.ttp	Greedy Algorithm	-35552,77153139290	-76287,03965222660	-55079,60811997170
berlin52_n51_uncorr_01.ttp	Evolutionary Algorithm	-11732,739785076500	-75442,21307457680	-13162,059348952600
berlin52_n51_uncorr_01.ttp	Simulated Annealing	-44093,80454171530	-	-
eil51_n50_uncorr-similar-weights_01.ttp	Random Search	-3420,254778835420	-6929,952267638450	-5126,554713635750
eil51_n50_uncorr-similar-weights_01.ttp	Greedy Algorithm	1541,5710994608500	-1307,3905560032400	392,10103406219400
eil51_n50_uncorr-similar-weights_01.ttp	Evolutionary Algorithm	2216,8332637450900	-190,6192755054630	2161,940401932910
eil51_n50_uncorr-similar-weights_01.ttp	Simulated Annealing	210,90268641275100	-	-
eil51_n150_uncorr-similar-weights_01.ttp	Random Search	-107,78450294446700	-5686,839482564250	-2785,598422402480
eil51_n150_uncorr-similar-weights_01.ttp	Greedy Algorithm	9406,92218830654	7243,733242672120	8280,79071802457
eil51_n150_uncorr-similar-weights_01.ttp	Evolutionary Algorithm	9840,044785582600	7056,408266754670	9732,31884542675
eil51_n150_uncorr-similar-weights_01.ttp	Simulated Annealing	7124,443746383010	-	-
eil51_n50_bounded-strongly-corr_01.ttp	Random Search	-471,7347154591850	-10898,739861354100	-5607,761533870570
eil51_n50_bounded-strongly-corr_01.ttp	Greedy Algorithm	4578,823451221630	1724,090709833140	3276,325199890770
eil51_n50_bounded-strongly-corr_01.ttp	Evolutionary Algorithm	5068,222545243110	1658,529950456580	4944,222498606660
eil51_n50_bounded-strongly-corr_01.ttp	Simulated Annealing	3191,7052647559200	-	-
berlin52_n51_uncorr-similar-weights_01.ttp	Random Search	-97453,87056862910	-140148,41380015200	-121837,28177048700
berlin52_n51_uncorr-similar-weights_01.ttp	Greedy Algorithm	-18460,128537535700	-40049,685307831700	-28232,18949054830
berlin52_n51_uncorr-similar-weights_01.ttp	Evolutionary Algorithm	-8901,295041578260	-38563,89995361450	-9414,692248518240
berlin52_n51_uncorr-similar-weights_01.ttp	Simulated Annealing	-15970,868687393200	-	-

## Fitness Value Comparison Across Algorithms for the Traveling Thief Problem



### Conclusions:

In summary, when tackling the Traveling Thief Problem, the Evolutionary Algorithm gave the best results. The Random Search was the least effective, as expected, while the Greedy Algorithm did surprisingly well, even better than Simulated Annealing in some cases. Looking at the Evolutionary Algorithm's settings, changing the mutation and crossover rates didn't make much difference. However, having a larger population size clearly led to better outcomes. This shows that for complex problems like this, starting with more options to choose from can really help find better solutions.

**The source code of the project is available at:**

**<https://github.com/murad-sh/traveling-thief-problem>**