A Genetic Algorithm Approach to the Travelling Thief Problem

Havarneau Matei
Gheorghe Purci

13.03.2025

1 Introduction

This document presents the theoretical foundations of three classical combinatorial optimisation problems — the Travelling Salesman Problem (TSP), the 0/1 Knapsack Problem, and their coupled hybrid variant known as the Travelling Thief Problem (TTP). Particular attention is given to the interaction mechanisms that emerge when two NP–hard subproblems are not solved independently, but jointly contribute to a single global objective.

In addition to the theoretical formulation, this document also provides a detailed description of how these concepts were operationalized in our implementation. Specifically, we describe how a Genetic Algorithm (GA) was adapted from a baseline TSP solver and extended to support TTP semantics by integrating time-dependent profit evaluation and weight-dependent travel velocity into the fitness model.

2 The Travelling Salesman Problem (TSP)

The TSP is a classical NP-hard routing problem. Given n cities and a fully-connected distance matrix $D = \{d_{ij}\}$ the objective is to find a tour that:

- visits each city exactly once
- returns to the starting city
- minimizes total travelling cost

A tour is represented as a permutation:

$$x = (x_1, x_2, \ldots, x_n) \tag{1}$$

The objective function (minimization) is:

$$f(x) = \sum_{i=1}^{n-1} \frac{d_{x_i x_{i+1}}}{v_c} + \frac{d_{x_n x_1}}{v_c} \tag{2}$$

where v_c is the speed (constant in TSP).
TSP is a pure routing problem.

3 0/1 Knapsack Problem

We have m items. Each item k has:

- profit p_k
- weight w_k

We have a maximum capacity W. We must choose items $z_k \in \{0, 1\}$.
Objective (maximization):

$$\max \sum_{k=1}^{m} p_k z_k \quad \text{subject to} \quad \sum_{k=1}^{m} w_k z_k \leq W \tag{3}$$

Knapsack is a pure packing/selection problem.

4 TTP: Interdependency between TSP and KP

The Travelling Thief Problem is a realistic benchmark problem introduced to **bridge the gap between artificial benchmarks and real-world optimization**. (See Bonyadi & Michalewicz).
TTP combines:

- the TSP sub-problem (tour order)
- the knapsack sub-problem (which items to take)

The key point: **these are not independent**. Taking heavy items slows the speed.
Velocity model:

$$v = v_{max} - (v_{max} - v_{min}) \cdot \frac{W_c}{W} \tag{4}$$

Where W_c is the current-carrying weight.
Time accumulates over edges. The objective (maximization):

$$F = \sum p(t) - R \cdot T \tag{5}$$

where R is rent rate per time.

4.1 Profit decay models

In the TTP formulation the thief's reward is not static: arriving later at a city makes the collected items less valuable. This introduces a temporal coupling between route planning and packing decisions. In our implementation we tested two different decay formulations:

Linear decay

$$p(t) = p_0 - \alpha t \tag{6}$$

This model assumes that value decreases uniformly over time. The penalty is proportional to arrival delay. This makes the model easy to interpret and produces a smooth trade–off between "take more items" vs. "arrive earlier".

Exponential decay

$$p(t) = p_0 e^{-\lambda t} \tag{7}$$

Exponential decay penalises late arrival more aggressively. It reflects a realistic scenario where profits degrade slowly in the beginning but collapse rapidly once travel time becomes large. In our experiments this model amplifies the interdependence: greedy item–collecting strategies that look locally optimal ("pick everything") quickly become globally inefficient because later arrival destroys most of the profit.

5 Genetic Algorithm Implementation

We used a classical GA with:

- **population of permutations (routes)** — every individual in the GA is one possible TSP route, represented as an ordered list of city indices
- **selection proportional to fitness** — individuals with higher fitness values have a higher probability of being selected as parents (roulette wheel)
- **custom crossover for permutations** — we cut a middle segment of Parent 1 and fill the remaining cities from Parent 2 in order, ensuring the child is still a valid permutation (no duplicates)
- **mutation for diversity** — we apply small random edits (swap two cities or replace one gene) to avoid early convergence to a poor local optimum
- **elitism** — the best individuals (highest fitness) are copied directly into the next generation to ensure the global best solution is not accidentally destroyed

5.1 Parent selection

Parents are chosen using **fitness-proportional random sampling**. We compute the fitness array and convert to probabilities:

$$P(i) = \frac{fitness(i)}{\sum fitness(j)} \tag{8}$$

High-scoring tours have higher probability of being selected.

5.2 Crossover

We select two random cut points $a < b$. Child is built as:

$$child_i = \begin{cases} parent1_i & i \in [a, b] \\ parent2_i & \text{otherwise} \end{cases} \tag{9}$$

This preserves permutation validity.

5.3 Mutation

Random small operations prevent stagnation:

- **swap mutation (swap two cities)** — we randomly choose two different positions in the permutation and exchange their city indices; this preserves all cities but changes local route ordering, which may shorten or lengthen that part of the path
- **replacement mutation (replace one city index randomly)** — we randomly select one position in the permutation and overwrite its city index with another randomly chosen city index; this introduces a sudden topological change and increases exploration capability

These keep diversity and allow escaping local extrema.

5.4 Fitness implementation

In our implementation, the fitness function is the *core place where the TTP logic is injected*. The GA does not know anything about TSP, Knapsack, or TTP itself — the only thing the GA sees is a vector of fitness values (one scalar per individual). Therefore, the fitness function is the boundary between the problem domain and the evolutionary engine.

In our code, `fitness(population)` receives the whole population matrix. Each row of `population` is one candidate route (one permutation of the cities) generated by the GA.

For each individual, we simulate the behavior of the thief:

1. starts with an empty knapsack ($current\_weight = 0$)
2. travel city-by-city in the order given by the individual
3. when arriving in the city i:
   - collect all items assigned to the city i
   - compute the current profit of each item using time-dependent decay (linear: $p(t) = p_0 - \alpha t$, exponential: $p(t) = p_0 e^{-\lambda t}$)
   - add their value to `total_profit`
   - add their weight to `current_weight`
4. compute the speed of travel as defined in the TTP model:

$$v(t) = v_{max} - (v_{max} - v_{min}) \cdot \frac{current\_weight}{W}$$

5. compute the travel time from this city to the next

$$\Delta t = \frac{d_{ij}}{v(t)}$$

6. accumulate time: $total\_time+ = \Delta t$

In the end, we close the route by returning to the initial city (last $\rightarrow$ first) and compute the final travel time using the same speed dynamics.

When this simulation is finished for a single individual, the final fitness is computed as:

$$fitness = total\_profit - R \cdot total\_time$$

This formula encodes the complete TTP trade–off: we want to take many profitable items, but we want not to become too slow (because weight reduces speed and time is penalized).

Important remark: In the GA framework, the fitness must return one single scalar per individual. Therefore, our final implementation returns a vector of floating-point values with a length equal to the population size. Example shape:

$$fitness(population) \rightarrow \mathbb{R}^{POPULATION\_SIZE}$$

This vector is then used by:

- parent selection (probabilities proportional to fitness)
- elitism extraction (best individuals)
- generation statistics (printing best distance / best profit)

In other words, the whole GA evolution pressure is driven by this numeric fitness value.

6 Conclusion

In this work, we started from a classical evolutionary TSP solver and progressively enriched it towards a Traveling Thief Problem instance. The essential contribution was the replacement of the TSP-only distance–based fitness with a TTP fitness model that explicitly accounts for item profit, time–dependent profit decay, knapsack loading dynamics, and velocity reduction as the knapsack becomes heavier. The GA engine (population, selection, crossover, mutation, elitism) remained structurally unchanged — this demonstrates the modularity and portability of genetic operators once the fitness function is properly defined.

Through this integration, the route is no longer evaluated purely by geometric efficiency, but by the global trade–off between "picking more items" and "arriving sooner". This is the core TTP interdependence described in the literature: routing decisions affect the value of the knapsack (later arrival → less profit), and picking decisions affect the routing cost (more weight → slower → more time penalty). By embedding these two opposing forces directly into the fitness function, we produced a single scalar objective that drives the evolutionary pressure towards balanced solutions.

The result is not just a GA that solves TTP, but a concrete demonstration of how metaheuristics should be designed when the problem is not a single isolated NP-hard problem, but a coupled composition of two NP-hard components. This supports the argument of Bonyadi et al. (2013) that future benchmark problems must reflect realistic interdependencies rather than idealized single-component models.

7 Bibliography

References

[1] M. R. Bonyadi, Z. Michalewicz, and L. Barone, *The travelling thief problem: the first step in the transition from theoretical problems to realistic problems.*

School of Computer Science, The University of Adelaide, Australia.