Research-Thesis: Disaster Response Improvement Using Heuristic Methods

Auteur : Cajot, Bastien
Promoteur(s) : Khayyati, Siamak
Faculté : HEC-Ecole de gestion de l'Université de Liège
Diplôme : Master en ingénieur de gestion, à finalité spécialisée en digital business
Année académique : 2025-2026
URI/URL : http://hdl.handle.net/2268.2/25196

HEC LIÈGE
Management School - Liège Université

DISASTER RESPONSE IMPROVEMENT USING HEURISTIC METHODS

Jury:
Supervisor:
Siamak KHAYYATI
Reader:
Bernard FORTZ

Master thesis presented by
Bastien CAJOT
To obtain the degree of
MASTER IN BUSINESS ENGINEERING
with a specialization in
Digital Business
Academic year 2025/2026

Ecole de Gestion de l'Université de Liège

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Professor S. Khayyati, for giving me the opportunity to carry out this work and for making this project possible. His guidance, availability, and advice supported me throughout all stages of this research and the writing of this thesis.

My sincere thanks also go to my reader, Professor B. Fortz, for according his time to this thesis and for carefully reviewing this work.

I would also like to thank all my colleagues at EVS for their advice, encouragement and understanding throughout the undertaking of this thesis.

Finally, I would like to express my deepest gratitude to my girlfriend and to my family for their constant support, patience and encouragement during this past year. Their presence and motivation were essential to the completion of this work.

Contents

1 Introduction **7**

2 Literature Review **8**
2.1 Humanitarian Operations Research (HOR) . . . 8
2.2 Routing Problems in humanitarian logistics . . . 9
2.3 Heuristic and Meta-heuristic methods for Humanitarian Routing . . . 11
2.3.1 Introduction to heuristic methods in computer science . . . 11
2.3.2 Type and categories of heuristics methods . . . 11
2.3.3 Metaheuristic models . . . 12
2.3.4 Application of heuristic method in Humanitarian routing problem 14
2.4 Robust Optimization . . . 16
2.4.1 Uncertainty in disaster response . . . 16
2.4.2 Foundation of robust optimization . . . 16
2.4.3 Time sensitivity and decreasing rewards under uncertainty . . . 17
2.4.4 Robust routing models in humanitarian logistics . . . 17
2.4.5 Adaptative and hybrid robust approaches . . . 18
2.5 Synthesis and Research gap . . . 18

3 Methodology **20**
3.1 Transition . . . 20
3.2 Conceptual Framework . . . 20
3.3 Research Objective . . . 21
3.4 Research Hypotheses . . . 22
3.5 Methodological Approach . . . 23
3.5.1 Step 1: Problem Definition . . . 24
3.5.2 Step 2: Incorporation of Decreasing Rewards . . . 27
3.5.3 Step 3: Integration of Robust Optimization Elements . . . 27
3.5.4 Step 4: Algorithm Development . . . 30
3.5.5 Models inputs . . . 40
3.5.6 Models outputs . . . 41
3.5.7 Step 5: Performance evaluation . . . 42

4 Numerical Experiment **49**
4.1 Context . . . 49
4.2 Experimental setup . . . 49
4.2.1 Instance generation policy . . . 49
4.2.2 Evaluation metrics . . . 50
4.3 Reference experiment . . . 50
4.3.1 Reference experiment configuration . . . 50
4.3.2 Reference experiment analysis . . . 51
4.4 Second stage experiment . . . 55
4.4.1 Runtime limit . . . 55
4.4.2 Gamma Fraction . . . 56
4.4.3 Time constraint . . . 57
4.4.4 Prices . . . 59
4.4.5 Decay rate . . . 61
4.5 Conclusion of the numerical experiment . . . 63

5 Scientific takeaways **64**

6 Limitation and future works **65**
6.1 Limitations 65
6.2 Future research directions 65

7 Conclusion **67**

A Appendix **68**
A.1 Implementation 68
A.2 Experimental results 92
A.2.1 Reference results 92
A.2.2 Runtime limit 92
A.2.3 Gamma fraction 93
A.2.4 Time constraint 93
A.2.5 Prices 94
A.2.6 Decay rates 94

B Bibliography **95**

List of Acronyms

ACO Ant Colony Optimization. 14

ARO Adjustable Robust Optimization. 18, 19

CPTP Capacitated Profitable Tour Problem. 10, 18

CTOP Capacitated Team Orienteering Problem. 10, 15, 18, 20, 21, 28

CV Coefficient of Variation. 53, 57, 59

DOM Disaster Operations Management. 8

GA Genetic Algorithm. 14, 15, 31, 32, 35, 40, 42, 43, 45

HOR Humanitarian Operations Research. 8, 9

HRDP Humanitarian Relief Distribution Problem. 18

ILS Iterated Local Search. 12

MILP Mixed-Integer Linear Program. 7, 27, 40, 42, 50

OP Orienteering Problem. 10, 17, 18

OPTW Orienteering Problem with Time Windows. 10

OR/MS Operations Research and Management Science. 8

PSO Particle Swarm Optimization. 15

RO Robust Optimization. 16, 19

RT-CTOP Robust Time-sensitive Capacitated Team Orienteering Problem. 21–23, 29, 30, 34, 35, 39, 45

RTOP Robust Team Orienteering Problem. 17

SA Simulated Annealing. 13, 15

SP Stochastic Programming. 19

TDOP Time-Dependent Orienteering Problem. 10

TOP Team Orienteering Problem. 10

TOP-DP Team Orienteering Problem with Decreasing Profits. 27

TS Tabu Search. 13, 15

TSP Traveling Salesman Problem. 8–10, 36

VLSI Very large scale integration. 13

VNS Variable Neighborhood Search. 12, 13, 16, 35, 36, 40–42, 45

VRP Vehicle Routing Problem. 8–10, 14, 15, 17, 18, 36

VRPPDTW Vehicle Routing Problem with Pickup and Delivery and Time Windows. 15

1 Introduction

Disaster response operations require rapid and reliable decision-making under severe time pressure and high levels of uncertainty. Efficiently dispatching emergency response teams and allocating limited resources are critical tasks that directly affect the effectiveness of humanitarian interventions and the mitigation of human and economic losses. These challenges naturally give rise to complex optimization problems, particularly vehicle routing problems (VRPs), where decisions must be made regarding the routing and scheduling of response teams under operational constraints and uncertain conditions.

Many disaster response planning problems can be formulated as combinatorial optimization problems, often modeled using Mixed-Integer Linear Program (MILP). Such problems involve discrete decision variables and large finite search spaces which are typically NP-hard, meaning that solving them to optimality requires computational effort that grows exponentially with problem size.

While exact optimization methods can guarantee optimal solutions, their computational cost often becomes extremely high for realistically sized or time-critical disaster scenarios. Moreover, classical deterministic formulations do not adequately account for unplanned events that can significantly degrade the effectiveness of response teams.

To overcome these limitations, heuristic and metaheuristic methods are widely employed. These computational intelligence approaches are designed to efficiently explore large solution spaces and produce high-quality feasible solutions within acceptable computational times. Although heuristic methods do not guarantee optimality, they are particularly well suited for large-scale and real-time decision problems where near-optimal solutions are sufficient and rapid responsiveness is crucial. In addition, robust optimization techniques can be incorporated to explicitly account for uncertainty by anticipating potential disruptions, such as time losses and guarantee solution feasibility and performance under adverse conditions.

This study proposes a comprehensive and technical framework for disaster response planning based on a vehicle routing formulation combined with robust optimization to explicitly account for uncertainty. The resulting robust model is solved using both exact MILP techniques and heuristic approaches for a systematic comparison between traditional combinatorial optimization methods and metaheuristic strategies. The objective is to analyze the extent to which heuristic methods can outperform exact approaches in terms of scalability, computational efficiency, and solution quality when dispatching emergency response teams under uncertainty.

2 Literature Review

This literature review aims to provide a comprehensive overview of existing studies on disaster response optimization and to identify potential research gaps. It is organized following a funnel-shaped structure to progressively narrow the focus from broad conceptual foundations to the specific problem addressed in this research. The review begins with Humanitarian Operations Research (HOR), which frames the unique challenges of disaster-response decision-making, including urgency, uncertainty, and resource scarcity. It then focuses on routing problems in humanitarian logistics, moving from classical models such as the Traveling Salesman Problem (TSP) and the Vehicle Routing Problem (VRP) to reward-based formulations such as the Orienteering Problem and its capacitated, multi-vehicle extensions.

Eventually, the review examines heuristic and metaheuristic approaches used to solve large-scale, NP-hard routing problems under time pressure, followed by robust optimization frameworks that explicitly address uncertainty and disrupted operating conditions. This progression culminates in the Robust Time-sensitive Capacitated Team Orienteering Problem (RT-CTOP), which integrates uncertainty modeling, decreasing rewards, and scalable solution methods and sets the foundation for the research presented in the following sections.

2.1 Humanitarian Operations Research (HOR)

HOR has become a central analytical discipline in managing disaster response and humanitarian logistics. It uses quantitative decision-support tools to enhance the efficiency, responsiveness and adaptability of relief operations conducted under extreme uncertainty and time pressure (Altay and Green III (2006), Kovács and Spens (2007)). Unlike commercial logistics, humanitarian operations must contend with disrupted infrastructure, volatile demand, and scarce resources, which necessitate advanced modeling approaches capable of delivering rapid yet reliable decisions (Van Wassenhove (2006)).

Disaster Operations Management (DOM) encompasses all activities conducted before, during, and after disasters to minimize human and economic losses and restore normalcy (Lamos et al. (2019)). Since the 1980s, Operations Research and Management Science (OR/MS) have provided a rich toolkit to support these decisions through optimization, simulation, and data-driven modeling, enabling the improvement of preparedness, response and recovery processes.

The unpredictability of disasters and the need for rapid response have led to the increasing adoption of quantitative models in humanitarian logistics, particularly during the response and recovery phases.OR/MS models are used to manage resource allocation, facility location, transportation and inventory under uncertainty (Van Wassenhove (2006), Balcik et al. (2010)).

Lamos et al. (2019) note a strong trend toward integrating stochastic and robust optimization methods, reflecting a shift from deterministic modeling to approaches that explicitly account for uncertainty and humanitarian objectives. Their review of 117 studies between 2010–2015 highlights a marked increase in transportation and logistics-

related research, especially those employing mathematical programming and stochastic optimization for relief distribution.

Moreover, disaster management research has evolved beyond pure optimization to include data-driven and machine learning approaches, supporting predictive assessment and decision-making under dynamic conditions. This expansion demonstrates the multidisciplinary nature of HOR, which now bridges logistics, artificial intelligence, and operations research to provide more adaptive and resilient solutions for emergency response.

Complementing these findings, in "A Review of Emergency Response in Disasters: Present and Future Perspectives" Feng and Cui (2021) emphasizes the growing global movement toward technological integration and resilience-oriented management. The study identifies the convergence of digital technologies, geospatial systems and real-time communication tools as critical enablers for next-generation humanitarian logistics.

According to the authors, effective disaster response increasingly depends on interagency coordination, information sharing, and the integration of intelligent decision-support systems capable of handling big data in real time. Moreover, the paper argues for a transition from reactive to anticipatory emergency management, where predictive analytics and simulation models enable pre-emptive mobilization of resources before disasters strike.

These global insights complement the OR/MS-focused literature by introducing a systemic perspective that links quantitative optimization with strategic governance, social resilience, and digital innovation. Both strands of research converge on the recognition that robust, flexible, and equitable decision-making is essential to strengthen humanitarian operations in the face of increasingly frequent and complex crises (Kunz and Reiner (2012)).

Recent studies emphasize that effective humanitarian operations depend not only on efficiency but also on equity and ethical considerations, making sure that relief efforts reach the most vulnerable populations fairly and effectively (Lamos et al. (2019)). Thus, HOR now focuses increasingly on designing systems that are robust, equitable, and sustainable, capable of adapting to the inherent volatility of disaster environments.

2.2 Routing Problems in humanitarian logistics

Routing and scheduling decisions represent some of the most critical optimization challenges in humanitarian logistics, where the timely and efficient delivery of relief goods directly impacts human survival and welfare (Laporte (2009)). Foundational routing models such as the TSP and the VRP provide the mathematical backbone for many disaster relief distribution frameworks. These classical problems have inspired a rich body of extensions that better capture the unique characteristics of humanitarian contexts, including uncertain demands, multiple depots, and capacity constraints.

The TSP, one of the earliest combinatorial optimization models, seeks the shortest route visiting each node exactly once and returning to the origin. In humanitarian logistics, it serves as a simplified abstraction for single-vehicle relief missions or reconnaissance

tours (Applegate et al. (2006)). However, TSP-based formulations quickly become unrealistic when multiple vehicles, depots, or heterogeneous resources are involved, which leads to more general models such as the VRP, which allocates deliveries among several vehicles while minimizing total cost or travel time. Both TSP and VRP are well-known NP-hard problems, implying that exact algorithms fail to scale to large or real-time disaster environments (Lenstra and Kan (2006)).

To address operational priorities beyond pure distance minimization, researchers have proposed profit-based routing models. The Orienteering Problem (OP), first introduced by Golden et al. (1987), combines features of the TSP and the Knapsack Problem: it seeks to maximize the total reward (or humanitarian benefit) obtained by visiting a subset of locations within a limited time or distance budget. This formulation aligns naturally with disaster response operations, where only a subset of affected areas can be served immediately, and priorities are assigned based on urgency or vulnerability. The OP thus provides a framework to determine which locations to serve first under resource and time constraints.

Subsequent research has extended the OP to consider multiple vehicles, time windows, stochastic conditions, and capacity limits. According to Gunawan et al. (2016), the OP has evolved into a family of routing problems with profits, encompassing variants such as the Team Orienteering Problem (TOP), the Orienteering Problem with Time Windows (OPTW)) and the Time-Dependent Orienteering Problem (TDOP). These models have been applied to contexts ranging from tour planning to emergency response, and they increasingly integrate real-world features such as stochastic travel times, multi-agent coordination, and time-dependent congestion. The TOP, originally introduced by Chao et al. (1996), generalizes the OP by allowing several routes (or teams) to be executed simultaneously, each constrained by time or capacity. The TOP and its derivatives are directly applicable to humanitarian convoy routing, where multiple vehicles must serve disjoint sets of affected communities within operational limits.

Building upon these, the Capacitated Team Orienteering Problem (CTOP) and the Capacitated Profitable Tour Problem (CPTP) add explicit capacity constraints, reflecting limited vehicle load or supply availability. Archetti et al. (2009) formalized and analyzed these models, showing that they combine profit maximization with resource feasibility, making them highly relevant for disaster response logistics where both payload and time are scarce. The authors proposed branch-and-price algorithms and metaheuristics, including Tabu Search and Variable Neighborhood Search, demonstrating that these methods deliver near-optimal solutions efficiently even for large-scale problem instances.

In these formulations, the objective shifts from minimizing distance to maximizing mission value, often measured in terms of lives saved, supplies delivered, or priority scores achieved. The CTOP, for instance, optimizes the selection of locations to visit under vehicle capacity and time restrictions, while the CPTP balances profit collection and travel cost, a trade-off analogous to balancing aid effectiveness versus operational expenditure in humanitarian operations.

Recent surveys highlight that metaheuristic algorithms, such as Genetic Algorithms, Simulated Annealing, Tabu Search, and Variable Neighborhood Search have proven

particularly effective in solving OP-based models under real-time constraints (Gunawan et al. (2016)). Moreover, their hybridization with robust or stochastic optimization frameworks is an emerging research direction.

2.3 Heuristic and Meta-heuristic methods for Humanitarian Routing

2.3.1 Introduction to heuristic methods in computer science

Heuristic methods are experience-based problem-solving techniques that employ approximate strategies, simplifications, or rules of thumb to find satisfactory (but not necessarily optimal) solutions efficiently. The term *heuristic* originates from the Greek *heuriskein*, meaning "to find" or "discover," reflecting their fundamental purpose: to uncover good solutions when exact computation is impractical.

In computer science, heuristics are widely applied across combinatorial optimization, graph theory, algorithmic complexity, and artificial intelligence, where many problems are NP-hard, for which their exact solution requires exponential computational effort (Garey and Johnson (1982)). Heuristic algorithms thus trade off optimality for efficiency, producing near-optimal solutions within acceptable computational limits (Gendreau and Potvin (2010)). Their power lies in their flexibility, adaptability and problem-specific design, which allows the rapid generation of usable solutions even in large-scale, dynamic or uncertain environments (Blum and Roli (2003)).

2.3.2 Type and categories of heuristics methods

Heuristics underpin several key domains of computational research and practice, forming the foundation of metaheuristics, hyper-heuristics and hybrid algorithms, which extend their capabilities to handle complex and evolving problem landscapes.

Constructive heuristics: Constructive heuristics build solutions incrementally, starting from an empty or partial configuration and progressively adding components until a complete and feasible solution is obtained. Examples include:

- i) Greedy algorithms, which iteratively select the best local option (e.g., the nearest neighbor in routing problems);
- ii) Savings algorithms, which merge routes or clusters to minimize overall cost;
- iii) Insertion heuristics, which determine the best position to insert new elements (e.g., cities, tasks) in a partial solution.

Constructive heuristics are computationally fast and effective for initial solution generation, particularly in single-objective or bi-objective optimization problems. However, they often risk premature convergence to suboptimal solutions and may perform poorly in distributed or large-scale dynamic systems where the search landscape evolves over time (Nekoueian et al. (2023)).

Improvements heuristics: Improvements heuristics, also known as local search or neighborhood search methods, begin with a complete solution and seek iterative enhancements by applying small modifications (called moves) within a defined neighborhood structure. Common examples include: 2-opt, 3-opt, and Or-opt procedures for route optimization, swap, insertion or reallocation operators for scheduling or assignment problems.
These heuristics are considered solution intensification techniques, designed to exploit promising areas of the search space. They are the foundation for more sophisticated frameworks like Iterated Local Search (ILS) and Variable Neighborhood Search (VNS), which systematically alter neighborhood definitions to escape local optima. Although improvement heuristics effectively refine solutions, they can be trapped in local minima without diversification strategies or stochastic components (Burke et al. (2007)).

Meta-heuristics: Meta-heuristics are high-level, problem-independent algorithmic frameworks designed to guide and enhance the performance of subordinate heuristics in exploring large and complex search spaces (Blum and Roli (2003), Gendreau and Potvin (2010)). Unlike problem-specific heuristics, metaheuristics employ stochastic and adaptive mechanisms to balance exploration (searching new areas of the solution space) and exploitation (intensively refining promising solutions). Their goal is not to guarantee optimality, but rather to find near-optimal and robust solutions efficiently, even for NP-hard and combinatorial problems where exact algorithms are computationally infeasible (Talbi and Talbi (2009)).

Hyper-heuristics: Hyper-heuristics represent a higher level of abstraction in heuristic design. Rather than directly searching within the solution space, they search within a space of heuristics, that is, they learn to select, generate, or adapt low-level heuristics based on performance feedback (Burke et al. (2010)).
There are two main categories:

- Heuristic selection hyper-heuristics, which dynamically choose the most suitable heuristic from a predefined pool during search (reinforcement learning or adaptive operator selection methods are typical)
- Heuristic generation hyper-heuristics, which evolve new heuristics through techniques such as genetic programming or machine learning.

Hyper-heuristics aim to create general-purpose, reusable search methodologies, reducing the need for problem-specific tuning. They are especially promising for multi-objective and real-time optimization, where they can adaptively control the balance between exploitation and exploration.

2.3.3 Metaheuristic models

Metaheuristics can be broadly categorized into two main paradigms: trajectory-based methods, which evolve a single solution over time and population-based methods, which evolve multiple solutions simultaneously.

Trajectory-based metaheuristics: Trajectory-based metaheuristics explore the search space by iteratively modifying a single candidate solution. The algorithm moves through the solution landscape by applying transformation operators and acceptance criteria that determine whether new solutions replace the current one

i) Simulated Annealing (SA))
SA is a well-established stochastic optimization technique inspired by the physical annealing process in metallurgy, where controlled cooling allows a system to settle into a stable state (Kirkpatrick et al. (1983)). In the optimization context, SA explores the solution space by occasionally accepting worse solutions, particularly in the early stages of the search, which helps avoid premature convergence to local optima. As the search progresses, the probability of accepting such deteriorating moves gradually decreases, leading to increased solution refinement.

Simulated Annealing has been successfully applied in numerous fields, such as Very large scale integration (VLSI) design (Kirkpatrick et al. (1983)), job-shop scheduling (van Laarhoven et al. (1992)), and vehicle routing (Osman (1993)), demonstrating its capacity to escape local optima through controlled randomization.

ii) Tabu Search (TS)
Tabu Search (Glover and Laguna (1999)) is a metaheuristic that extends local search by using memory-based mechanisms to guide the exploration of the solution space. Rather than repeatedly accepting only improving moves, TS records recently visited solutions or solution attributes in a tabu list, temporarily forbidding them in order to prevent cycling and encourage exploration beyond local optima.

This controlled use of memory allows TS to balance intensification, by focusing on promising regions of the search space, and diversification, by directing the search toward unexplored areas. Aspiration criteria further enhance flexibility by permitting tabu moves when they lead to globally improved solutions. Owing to these features, TS has proven particularly effective for complex combinatorial optimization problems, including routing and scheduling applications, where structured exploration and solution stability are essential(Glover and Laguna (1999), Burke and Newall (2003)).

iii) Variable Neighborhood Search
VNS is a metaheuristic framework that systematically explores multiple neighborhood structures to overcome local optimalityHansen et al. (2010)). Unlike classical local search methods that rely on a single neighborhood, VNS changes the neighborhood during the search process, allowing the algorithm to escape local optima through structured diversification rather than purely random moves.

The underlying principle of VNS is that a solution that is locally optimal with respect to one neighborhood structure may not be optimal with respect to another. By progressively examining neighborhoods of increasing size and returning to smaller neighborhoods when improvements are found, VNS achieves an effective balance between intensification and diversification. Owing to its flexibility and relatively simple parameterization, VNS has been successfully applied to a wide range of combinatorial optimization problems, including vehicle routing (Kytöjoki et al. (2007)) and orienteering problems (Cacchiani et al. (2014)), where it has demonstrated strong performance in terms of both solution quality and computational efficiency.

Population-based metaheuristics: maintain and evolve a set of candidate solutions, which allows parallel exploration of multiple regions in the search space. They often draw inspiration from biological evolution, collective intelligence, or natural systems that exhibit emergent problem-solving behavior:

i) Genetic Algorithm (GA)
Genetic Algorithms (Holland (1992)) simulate the process of natural selection and evolution. A population of solutions (chromosomes) undergoes genetic operations:

1. Selection, where fitter individuals are chosen for reproduction;
2. Crossover (recombination), which combines features of parent solutions to produce offspring;
3. Mutation, introducing random variations to maintain diversity.

The fitness of each individual is evaluated by an objective function, and successive generations evolve toward improved solutions. GA's success lies in its balance of diversification through crossover and intensification via selection.

ii) Ant Colony Optimization (ACO)
Ant Colony Optimization (Dorigo et al. (1996)) is inspired by the foraging behavior of ants, which collectively discover optimal paths between their nest and food sources through pheromone trails. Artificial ants probabilistically construct solutions guided by:

- Pheromone concentration (representing learned desirability)
- Heuristic information (problem-specific knowledge)

After each iteration, pheromone trails are updated, reinforced on successful paths and evaporated elsewhere to balance exploration and exploitation.

2.3.4 Application of heuristic method in Humanitarian routing problem

Heuristic and metaheuristic algorithms have become indispensable for solving large-scale routing problems in disaster response and humanitarian logistics. Their appeal lies in their ability to generate near-optimal solutions within tight time limits, where traditional exact optimization methods are computationally infeasible due to NP-hard complexity (Lenstra and Kan (2006)). While heuristics apply problem-specific constructive or improvement rules to quickly produce good solutions, metaheuristics provide higher-level adaptive frameworks that guide the search process across diverse regions of the solution space to avoid premature convergence and allow scalability under uncertainty.

- **Genetic Algorithm**

  Among metaheuristics, GA have shown remarkable versatility in addressing VRP, including their humanitarian variants such as multi-vehicle, capacitated, and time-constrained relief routing (Prins (2004)).

Ibrahim et al. (2021) proposed an Improved GA for the Vehicle Routing Problem with Pickup and Delivery and Time Windows (VRPPDTW), incorporating nearest-neighbor heuristics for population initialization, and fitness functions penalizing time-window and capacity violations. Their results demonstrated a 20.6% reduction in total travel distance and 0% constraint violation compared to company baseline routes. Similarly, Ochelska-Mierzejewska et al. (2021) presented a comparative study of GA operator configurations (selection, crossover, mutation) for VRP, emphasizing the impact of population diversity and crossover strategy (e.g., order crossover, edge recombination) on convergence quality. Their extensive experiments (spanning 340 GA runs across benchmark instances) showed that edge-based crossover operators consistently outperformed vertex-based ones in terms of solution stability and global search efficiency.

The flexibility of GAs makes them particularly well-suited for multi-objective humanitarian routing, where trade-offs exist between total cost, response time, and service equity. For example, recent studies integrate GA with robust optimization to maintain performance across uncertain post-disaster scenarios (Marinakis and Marinaki (2019)). Hybrid GA approaches have also combined Particle Swarm Optimization (PSO) and SA components improve search diversification and convergence speed (Ibrahim et al. (2021); Ochelska-Mierzejewska et al. (2021)).

- **Simulated Annealing**

  SA, inspired by the physical process of metal annealing, allows controlled acceptance of worse solutions to escape local optima, thereby balancing exploration and exploitation (Kirkpatrick et al. (1983)).

  In humanitarian routing, SA has been employed to optimize time-sensitive and resource-constrained routing models, particularly where solution robustness under uncertainty is paramount (Gendreau and Potvin (2010)). Comparative studies reveal that GA performs well for time-windowed or stochastic routing tasks but tends to converge slower than GA when solution spaces are highly irregular or when dynamic re-routing is required (Ochelska-Mierzejewska et al. (2021)). Nevertheless, hybrid GA–SA frameworks have shown improved adaptability for real-time disaster response logistics, dynamically adjusting routes based on updated field data (Wang et al. (2025)).

- **Tabu Search**

  TS introduces adaptive memory mechanisms to avoid revisiting recently explored solutions, allowing more systematic exploration of the solution space (Glover and Laguna (1999)). As a local search–based metaheuristic, TS is particularly effective in fine-tuning route structures after initial GA or heuristic solutions. In comparative metaheuristic analyses, TS achieved faster convergence toward local optima than GA but was more prone to stagnation in highly dynamic or uncertain contexts (Ochelska-Mierzejewska et al. (2021)). Hybrid GA–TS models leverage GA–'s population diversity and TS's intensification capacity to balance exploration and exploitation, resulting in improved convergence speed and stability in VRP– and CTOP– applications.

- **Variable Neighborhood Search**

  Variable Neighborhood Search systematically changes the neighborhood structure during optimization to diversify exploration and escape local minima (Hansen et al. (2010)). Its simplicity and flexibility make it suitable for multi-vehicle routing problems under stochastic and time-dependent conditions. Empirical results show that VNS often outperforms single-operator heuristics in large-scale CTOP and VRP variants by quickly finding high-quality feasible solutions (Cacchiani et al. (2014)). In humanitarian logistics, VNS has been used to reoptimize vehicle dispatch schedules following sudden disruptions or infrastructure damage, ensuring continuity of aid delivery under uncertainty.

2.4 Robust Optimization

2.4.1 Uncertainty in disaster response

Disaster environments are characterized by profound uncertainty in demand levels, infrastructure accessibility, travel times, and resource availability (Holguín-Veras et al. (2016)). Road blockages, damaged bridges, congestion caused by mass population movements, and evolving hazard conditions can drastically alter routing feasibility in real time. Under such circumstances, deterministic optimization models built on the assumption of known and fixed parameters often produce fragile solutions that quickly become infeasible when actual conditions deviate from forecasts. In humanitarian operations, where timely and reliable decisions are critical, such fragility can significantly deteriorate response effectiveness. These realities motivate the need for modeling approaches that explicitly account for uncertainty and provide protection against adverse operational conditions.

2.4.2 Foundation of robust optimization

To address the limitations of deterministic optimization under uncertainty, the operations research literature has increasingly turned to Robust Optimization (RO) frameworks, which explicitly model uncertainty within optimization formulations and seek solutions that remain feasible and near-optimal across a predefined range of adverse realizations (Ben-Tal and Nemirovski (2002), Bertsimas and Sim (2004)). Unlike stochastic optimization, RO does not rely on precise probability distributions, making it particularly well suited to disaster response contexts where data is scarce, incomplete or unreliable.

At the core of RO lies the trade-off between optimality and robustness, termed the "price of robustness" by Bertsimas and Sim (2004). Their pioneering work introduced a controllable uncertainty budget parameter (Γ) that adjusts the conservatism level in optimization models. By tuning Γ, decision-makers can balance solution protection and nominal performance to certify feasibility even when a subset of uncertain coefficients deviates from expected values. This paradigm has been instrumental in developing tractable formulations for linear and discrete optimization problems that would otherwise be intractable under uncertainty. Moreover, subsequent studies by Bertsimas et al. (2011) expanded the theoretical landscape of RO by demonstrating its flexibility, computational tractability, and adaptability across diverse domains, from logistics and finance to engineering.

2.4.3 Time sensitivity and decreasing rewards under uncertainty

Beyond uncertainty in parameters such as travel times and accessibility, humanitarian routing problems are also inherently time-sensitive, as the value of serving a location often decreases as response is delayed. Delayed intervention may result in worsening health conditions, declining survival probabilities, escalating hazards or increased operational risk. Classical routing formulations with static rewards fail to capture this temporal degradation of benefit, potentially producing solutions that are efficient in terms of cost or distance but misaligned with humanitarian priorities.

This limitation has been explicitly addressed through the introduction of time-dependent or decreasing reward structures, most notably in the Robust Team Orienteering Problem (RTOP) proposed by Yu et al. (2022). In their formulation, rewards associated with service locations decrease as a function of arrival time, capturing the loss of humanitarian value caused by delayed response. By integrating decreasing rewards with robust travel-time uncertainty, the RTOP provides a more realistic representation of disaster-response conditions, where both uncertainty and urgency jointly influence routing decisions.

The RTOP framework highlights that incorporating time decay fundamentally alters routing behavior. Solutions tend to prioritize early service to high-impact locations via shorter and more reliable routes, even if this reduces total nominal reward. Yu et al. (2022) demonstrate that ignoring either time-dependent rewards or robustness can lead to routing plans that appear optimal under nominal assumptions but perform poorly when delays occur. Their results show that robustness and time sensitivity are complementary modeling dimensions, both of which are essential for effective disaster-response routing.

2.4.4 Robust routing models in humanitarian logistics

In the broader context of humanitarian logistics, uncertainty in travel times, accessibility, and service conditions poses severe challenges for routing and resource allocation. Classical deterministic formulations such as the VRP or OP often fail to guarantee feasibility once infrastructure disruptions or information gaps arise. As a result, several studies have integrated robust optimization techniques directly into routing models for disaster response.
Balcik and Yanıkoğlu (2019) proposed one of the first robust optimization frameworks specifically designed for humanitarian needs assessment routing under travel time uncertainty. By formulating a robust counterpart of a mixed-integer program and employing ellipsoidal uncertainty sets, their model guarantees route feasibility under worst-case travel conditions while minimizing mission duration. Their results demonstrate that robust solutions significantly improve operational reliability compared to deterministic plans, without incurring excessive cost increases, highlighting the practical value of robust optimization in humanitarian missions.

Building on this line of research, Yanıkoğlu and Yavuz (2021) extended robust optimization techniques to multi-agent and multi-machine scheduling problems affected by uncertain processing times. Although developed in a scheduling context, their methodological contributions are directly relevant to humanitarian routing. In particular, they

show that ellipsoidal uncertainty sets often lead to less conservative and more realistic solutions than simple interval-based models, and that advanced solution techniques such as branch-and-price can efficiently solve large-scale robust formulations. These insights inform the design of scalable robust routing models for disaster response operations.

2.4.5 Adaptative and hybrid robust approaches

More recent advances have moved beyond static robust formulations toward Adjustable Robust Optimization (ARO), which allows certain decision variables to adapt dynamically as uncertainty is progressively revealed. Avishan et al. (2023) introduced an adjustable robust framework for the Humanitarian Relief Distribution Problem (HRDP), explicitly modeling uncertain travel times and allowing service times and routing decisions to be adjusted during execution. Their adjustable robust distribution model improves both efficiency and equity in relief delivery, achieving measurable gains in objective value and service coverage compared to static robust approaches.

From a theoretical perspective, robust optimization offers tractable linear and conic reformulations that provide deterministic feasibility guarantees without requiring probabilistic assumptions (Bertsimas et al. (2011)). This property is especially valuable in humanitarian contexts, where data limitations often prevent reliable stochastic modeling. At the same time, robust formulations typically increase problem complexity, making exact solution approaches impractical for large-scale or time-critical disaster scenarios.

To overcome these computational challenges, recent studies have combined robust optimization with metaheuristic solution methods, giving rise to hybrid robust–metaheuristic frameworks. Liberatore and Luo (2010) demonstrate that integrating robust optimization principles with heuristic search methods can produce high-quality, resilient solutions for uncertain routing and allocation problems within acceptable computational times. These hybrid approaches are particularly promising for humanitarian logistics, as they balance robustness, scalability and rapid decision-making principles that directly motivate the modeling and algorithmic choices adopted in this study.

2.5 Synthesis and Research gap

The evolution of the literature on disaster-response optimization and humanitarian logistics reveals a progressive methodological shift over the past two decades. Early works predominantly relied on deterministic mathematical programming to model vehicle routing, resource allocation, and facility location problems (Laporte (2009), Balcik et al. (2008)). While these approaches provided solid theoretical frameworks, they were limited by their inability to accommodate uncertainty, time pressure, and the scale of real-world disaster operations.

As the field matured, researchers increasingly adopted heuristic and metaheuristic algorithms to overcome computational intractability in NP-hard routing problems such as the VRP, OP, CTOP, and CPTP (Golden et al. (1987), Archetti et al. (2009), Gunawan et al. (2016)). These algorithms, including Genetic Algorithms (GA), Simulated Annealing (SA), Tabu Search (TS) and Variable Neighborhood Search (VNS) proved

effective in generating high-quality solutions within strict time limits. However, while these heuristic approaches improved computational efficiency they typically assumed static and deterministic environments, leaving their solutions vulnerable to disruptions and uncertainty common in post-disaster contexts.

In response, a new generation of research integrated RO and Stochastic Programming (SP) paradigms into humanitarian routing (Bertsimas and Sim (2004), Balcik and Yanıkoğlu (2019), Avishan et al. (2023)). These studies emphasized solution resilience, focusing on protecting operations against uncertain demand, travel times and accessibility conditions. More recent work introduced ARO allowing adaptive decision-making as new information unfolds during response operations (Avishan et al. (2023)). Yet, such robust and adjustable models often suffer from computational complexity and rely on exact or mixed-integer formulations that struggle to scale to real-time, large-scale disaster environments.

Parallel to this line of research, a smaller but growing body of work has highlighted the importance of time-dependent or decreasing rewards, recognizing that the humanitarian value of serving a location diminishes as response is delayed. Models such as the Robust Team Orienteering Problem (Yu et al. (2022)) demonstrate that incorporating time decay fundamentally alters routing priorities and favor early service to high-impact locations rather than maximizing static profit. Nevertheless, such time-sensitive reward structures remain largely absent from heuristic and metaheuristic disaster-routing frameworks and are rarely combined with robust optimization in a scalable manner.

Taken together, the literature demonstrates a three-stage methodological evolution:

1. deterministic mathematical programming for idealized routing
2. heuristic and metaheuristic approaches for computational feasibility
3. robust and hybrid frameworks for uncertainty resilience

Despite these advances, a significant research gap persists at the intersection of three critical dimensions:

- computational efficiency
- robustness to uncertainty
- time-sensitive prioritization through decreasing rewards

Current studies tend to prioritize either solution quality and speed (in heuristic models) or stability under uncertainty (in robust optimization frameworks) while often neglecting the temporal degradation of humanitarian value that is central to disaster response and rarely in a unified methodology. Few works provide scalable, hybrid heuristic–robust algorithms that simultaneously account for uncertainty, urgency, and real-time operational constraints in disaster environments characterized by limited data and rapidly evolving conditions.

3 Methodology

3.1 Transition

The literature review highlighted the limitations of purely deterministic optimization in disaster-response contexts, as well as the complementary strengths and weaknesses of metaheuristic and robust optimization approaches. While metaheuristics offer scalability and rapid solution generation, they typically rely on nominal parameter values. Conversely, robust optimization explicitly accounts for uncertainty but often leads to increased computational burden.

This methodology is positioned at the intersection of these two paradigms. Rather than treating robustness and heuristic efficiency as competing objectives, the proposed framework integrates robust modeling elements directly into heuristic and metaheuristic search procedures. In doing so, the study addresses three key methodological requirements:

- Realistic routing formulations aligned with disaster-response operations
- Time-sensitive and priority-based reward modeling
- Explicit management of uncertainty through robust optimization

3.2 Conceptual Framework

The conceptual framework adopted in this research is organized around 3 interrelated components that collectively characterize the decision-making challenges of disaster-response routing:

1. **Disaster-response routing as a CTOP**, which models the selection and routing of emergency teams under strict time and capacity constraints.
2. **Uncertainty management through robust optimization with time-dependent rewards**, accounting for operational disruptions and the decreasing value of delayed service.
3. **Metaheuristic search strategies**, for scalable and rapid solution generation.

The interaction among these components constitutes the foundation of the proposed robust time-sensitive metaheuristic disaster routing, which integrates realistic modeling, robustness and algorithmic efficiency to support effective disaster-response decision-making.

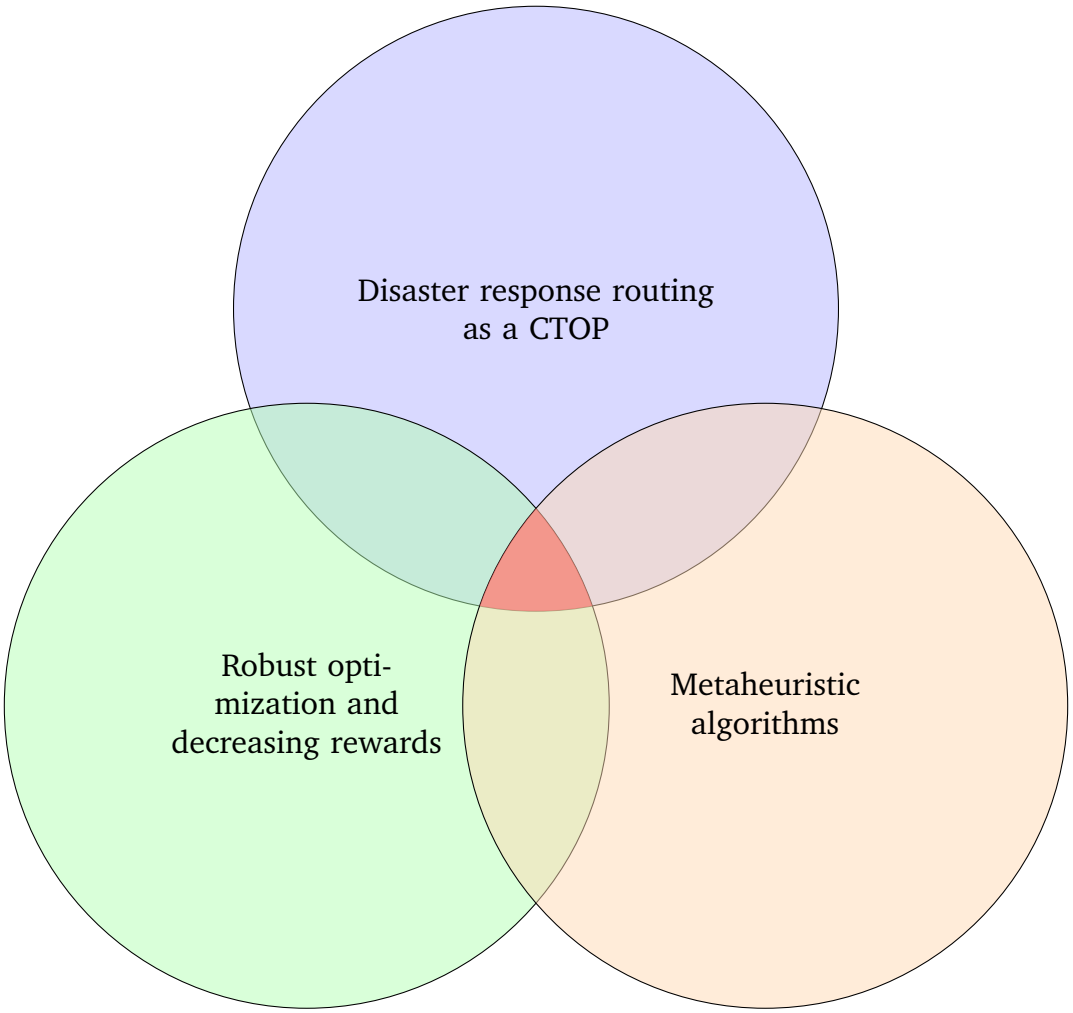


Figure 1: Representation of the conceptual framework

3.3 Research Objective

Drawing from the gap identified in the literature review, this study pursues the following objectives:

1. To formulate a robust, time-sensitive version of CTOP that models real disaster-response constraints including decaying node rewards and uncertain travel times.

2. To design and implement heuristic and metaheuristic algorithms adapted to the Robust Time-sensitive Capacitated Team Orienteering Problem (RT-CTOP).

3. To evaluate algorithmic performance in terms of solutions quality, computational efficiency, robustness to uncertainty and operational relevance.

4. To analyze trade-offs between efficiency, robustness and humanitarian responsiveness to provide insights for decision-makers in real world relief planning.

3.4 Research Hypotheses

Based on the conceptual framework and literature synthesis, the following hypotheses guide this research:

- **H1** Integrating robust optimization into metaheuristic algorithms yields more feasible and stable routing solutions under uncertainty than conventional heuristics.
- **H2** The proposed hybrid robust-metaheuristics achieve comparable or superior solution quality to deterministic models while significantly reducing computation time.
- **H3** Incorporating time-sensitive (decreasing) rewards improve the prioritization of high-impact nodes without significantly degrading overall solution quality.
- **H4** Algorithms tuned for RT-CTOP maintain higher solution feasibility across multiple uncertainty levels than their non-robust counterparts.

3.5 Methodological Approach

This research adopts a model-driven, computational experimentation methodology grounded in OR modeling and heuristic design The methodological structure follows 5 steps

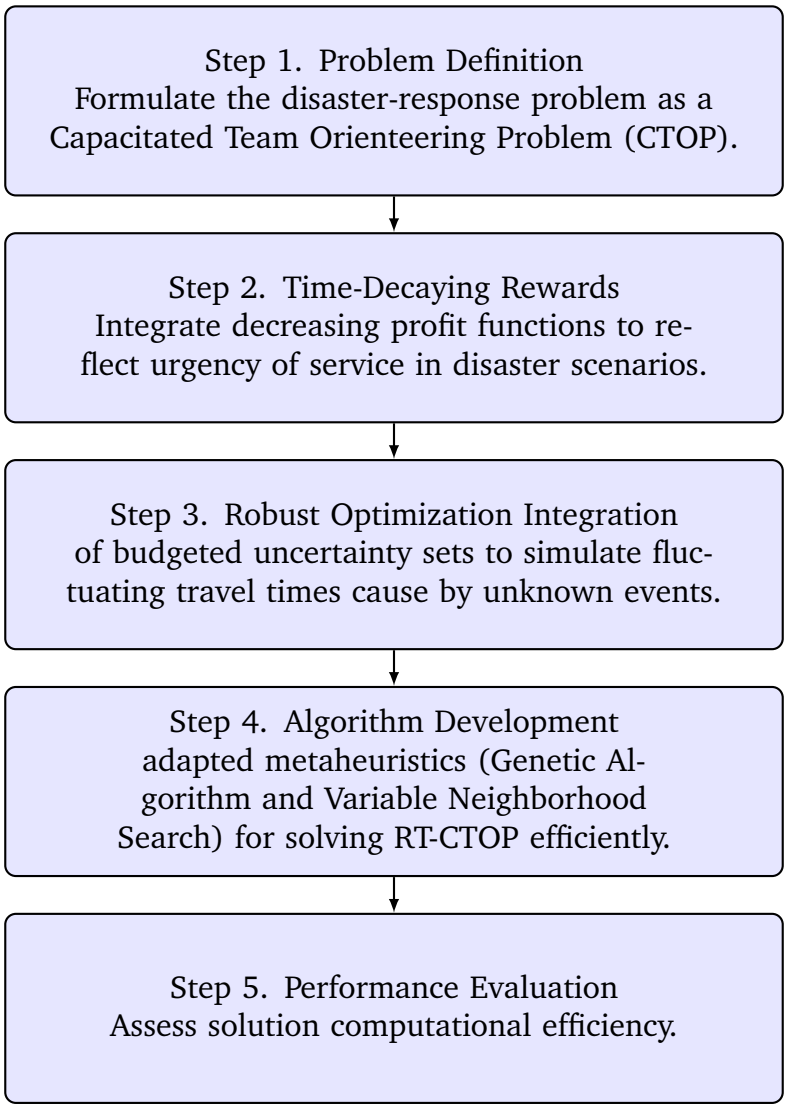


Figure 2: Methodological process of the research, illustrating the sequence from modeling R-CTOP to algorithm design.

3.5.1 Step 1: Problem Definition

This implementation uses the orienteering and capacitated team orienteering literature as a theoretical foundation for defining the disaster-response routing problem. In particular, the model builds upon the Capacitated Team Orienteering Problem (CTOP) formulations proposed by Archetti et al. (2009), as well as more recent extensions that address uncertainty and incomplete information (Yu et al. (2022)). These works provide a rigorous framework for routing problems in which limited resources prevent full coverage and decisions must balance node selection, route feasibility and operational constraints.

The disaster environment is represented as a directed graph $G = (V, E)$, following standard OP/TOP/CTOP formulations. Nodes correspond to a depot $0 \in V$ and a set of candidate service locations $N = V \backslash \{0\}$, while arcs represent feasible travel between locations with associated travel times. A fleet of vehicles $K = \{1, \ldots, K\}$ departs from and returns to the depot, with each vehicle k constructing a route that selects and serves a subset of locations. This formulation directly reflects the structure of CTOP, where multiple vehicles must jointly maximize the total collected reward under route-level constraints (Archetti et al. (2009)). The model also enforces a maximum mission duration T for each vehicle to prevent unrealistic or unsafe deployment plans.

Another modeling aspect is node selection under resource scarcity. Unlike classical VRP formulations, which assume that all customer nodes must be served, CTOP explicitly recognizes that limited time and resources make full coverage infeasible in most disaster scenarios. Consequently, the model allows only a subset of locations to be visited, reflecting the logic of humanitarian triage, where high-impact or high-urgency areas must be prioritized (Balcik et al. (2008)). These priorities are formalized through a profit-maximization objective, in which each node is associated with a reward p_i representing factors such as the number of affected individuals, the severity of needs or the overall humanitarian benefit of service. Profit-based routing formulations, in particular the Team Orienteering Problem, have long been used to model situations in which not all tasks can be completed and decision-makers must select those with the highest impact (Vansteenwegen and Souffriau (2011)).

The formulation further supports multi-vehicle coordination, which is essential in disaster-response logistics where several response teams operate simultaneously across different regions. The CTOP structure allows the simultaneous planning of multiple routes and ensures that each selected location is served by at most one vehicle to prevent redundant visits. Route feasibility and continuity are enforced through classical routing constraints, including flow conservation and subtour elimination implemented via Miller–Tucker–Zemlin (MTZ) variables, which prevent the formation of disconnected cycles within individual vehicle routes. The importance of coordinated multi-team deployment has been emphasized in humanitarian logistics as a key factor for rapid response and minimal duplication of effort (Yi and Ozdamar (2007)).

Mixed inter linear programming

The Capacitated Team Orienteering Problem (CTOP) is first formulated as a mixed-integer linear programming (MILP) model , which serves as a baseline representatioon of the disaster response routing problem. The CTOP seeks to maximize the total collected profit of a fleet of vehicles subject to time and capacity constraints. The model is implemented in Python using the DOcplex modeling framework and solved with the IBM ILOG CPLEX optimizer within a Jupyter Notebook environment (see Appendix A.1). The corresponding mathematical programming formulation is presented below.

Sets and indices

- $V = \{0, 1, ..., n\}$: set of nodes
- $0 \in V$: depot nodes
- $N = V \backslash \{0\}$: set of customer nodes
- $K = \{1, ..., K\}$: set of vehicles
- $i, j \in V$: node indices
- $k \in K$: vehicle index

Parameters

- p_i: initial reward at node $i \in N$
- α_i: decay rate of reward at node i
- d_{ij}: travel time (distance) between node i and j
- T: maximum mission duration (time capacity) per vehicle i
- w_T: travel cost weight
- M: large constant for time linearization

Decision Variables

To model routing, time propagation, reward decay and adversarial robustness the MILP uses the following variables:

- Routing variables
  Equal 1 if vehicle kv travels from node i to node j: $x_{ijk} \in \{0, 1\}\ \forall i, j \in V, \forall k \in K$
- Node-visit variables
  Equal 1 if node i is served by any vehicle: $y_i \in \{0, 1\}\ \forall i \in V$
- Time variables
  Arrival time at node i on vehicle k: $a_{ik} \in [0, T]\ \forall i \in V, \forall k \in K$
- Linearization variables
  $\theta_{ik} \in [0, T]\ \forall i \in N, \forall k \in K$

- Subtour-elimination variables
  MTZ ordering variable for vehicle k: u_{ik} $\forall i \in V, \forall k \in K$

Routing constraints

- Visit linking
  $\sum_{j \in V, j \neq i} x_{jik} = z_{ik} \quad \forall i \in N, \forall k \in K$
- $\sum_{k \in K} z_{ik} = y_i \quad \forall i \in N$
- Flow conservation
  $\sum_{i \in V, i \neq h} x_{ihk} = \sum_{j \in V, j \neq h} x_{hjk} \quad \forall h \in N, \forall k \in K$
- Subtour elimination (MTZ)
  $u_{ik} - u_{jk} + (n-1)x_{ijk} \leq n-2 \quad \forall i \in N, \forall j \in N \forall i \neq j, \forall k \in K$

Vehicle constraints

- $\sum_{j \in N} x_{0jk} \leq 1 \ \forall k \in K$
- $\sum_{i \in N} x_{i0k} \leq 1 \ \forall k \in K$
- $\sum_{j \in N} x_{0jk} = \sum_{i \in N} x_{i0k} \ \forall k \in K$

Time constraints

- Big-M tightening
  $M = T + \max_{i,j \in V, i \neq j} d_{ij}$
- Depot
  $a_{0k} \quad \forall k \in K$
- Arrival time propagation along active arcs
  $a_{ik} + d_{ij} \leq a_{jk} + M(1 - x_{ijk}) \quad \forall i \in V, \forall j \in N, j \neq i, \forall k \in K$
- Per vehicle route duration
  $\sum_{i \in V} \sum_{j \in V, j \neq i} d_{ij} x_{ijk} \leq T \quad \forall k \in K$
- Return to depot feasibility
  $a_{ik} + d_{i0} \leq T + M(1 - x_{i0k}) \quad \forall i \in V, \forall k \in K$
- Mission duration
  $0 \leq a_{ik} \leq T \quad \forall i \in V, \forall k \in K$

Objective function

$$max \sum_{i \in N} p_i y_i - w_T \sum_{k \in K} \sum_{i,j \in V} d_{ij} x_{ijk}$$

3.5.2 Step 2: Incorporation of Decreasing Rewards

To capture the time-sensitive nature of disaster-response operations, the model incorporates decreasing rewards, where the value of servicing a location diminishes as arrival is delayed. This modeling choice is directly inspired by theTeam Orienteering Problem with Decreasing Profits (TOP-DP) and its robust extensions, in which customer profits are assumed to decrease linearly with time and timely service is critical for preserving value (Yu et al. (2022)).

Following the linear profit-decay structure commonly adopted in the TOP-DP literature, the collected profit at node i is expressed as the difference between an initial reward and a time-dependent penalty proportional to the arrival time. To preserve linearity within a MILP framework, an auxiliary variable θ_i is introduced to represent the effective arrival time used in the reward decay function. A similar linearization strategy is employed by Yu et al. (2022), where arrival-time variables are explicitly modeled to compute time-dependent profits while maintaining tractability.

The collected profit at node i is then expressed as

$$CP_i = p_i y_i - \alpha_i \sum_{k \in K} \theta_i \quad \forall i \in N$$

where p_i is the initial reward, α_i the decay rate and y_i indicates whether the node i is visited. The auxiliary variable θ_i is linked to the actual arrival times a_{ik} through a set of linear constraints that ensure θ_i takes the arrival time of the vehicle serving node i and is zero otherwise. This construction mirrors the arrival-time–based profit computation used in TOP-DP and RTOP-DP formulations, where arrival times are explicitly tracked to evaluate decreasing profits and enforce feasibility (Yu et al. (2022)). The relationship is enforced through these constraints:

- $\theta_i \leq a_{ik} \quad \forall i \in V, \forall k \in K$
- $\theta_i \leq T z_{ik} \quad \forall i \in V, \forall k \in K$
- $\theta_i \geq a_{ik} - T(1 - ik) \quad \forall i \in V, \forall k \in K$

3.5.3 Step 3: Integration of Robust Optimization Elements

The integration of robustness into the proposed model is grounded in the budgeted uncertainty framework introduced by Bertsimas and Sim, which has become a cornerstone of modern robust optimization. In their seminal work, Bertsimas and Sim (2004) propose a cardinality-constrained uncertainty set that limits the number of parameters allowed to deviate simultaneously from their nominal values. This approach introduces a budget of uncertainty, denoted by Γ, which enables the decision-maker to explicitly control the trade-off between robustness and solution conservatism. Rather than protecting against the unrealistic worst case in which all parameters deviate simultaneously, the budgeted uncertainty model assumes that only a limited number of adverse deviations occur at the same time, resulting in solutions that are both robust and practically meaningful.

Following this principle, the robust formulation in this thesis protects routing decisions against worst-case delays affecting a limited number of visited nodes, while avoiding

overly conservative solutions. The uncertainty budget Γ restricts the number of nodes that can experience their maximum deviation simultaneously, thereby allowing the robustness level to be tuned according to the perceived severity of operational uncertainty. This modeling choice is consistent with the interpretation of Γ as a measure of protection intensity in the Bertsimas and Sim framework and provides a controlled balance between performance and robustness.
Rather than explicitly formulating a bi-level max–min problem between a planner and an adversary, the model adopts the dual reformulation of the Bertsimas–Sim robust counterpart, which yields a tractable single-level mixed-integer linear program. As shown by Bertsimas and Sim (2004), this dualization replaces the inner maximization over uncertainty realizations with additional decision variables and linear constraints, preserving computational tractability while enforcing worst-case protection. In the present model, this reformulation introduces a global protection variable λ, that represents the shared robustness budget, along with node-specific slack variables s_i , which capture localized worst-case deviations.

To adapt the budgeted uncertainty concept to the time-dependent reward structure of the CTOP, additional insights from the robust orienteering and robust routing literature are incorporated. In particular, recent studies on robust orienteering problems with uncertain or time-sensitive rewards demonstrate how uncertainty can be embedded directly into profit terms rather than travel costs or feasibility constraints (Yu et al. (2022)). Following this line of work, uncertainty is modeled through node-specific deviation parameters δ_i, which represent the maximum increase in effective reward decay due to adverse travel conditions.

For each node $i \in N$, the robust counterpart enforces dual feasibility through linear constraints that ensure the potential worst-case loss $\delta_i \theta_i$ is covered either by the global robustness budget λ or by the local slack variable s_i. The corresponding robustness penalty is incorporated directly into the objective function via the terms $-\lambda\Gamma_{\text{eff}}$ and $-\sum_{i \in N} s_i$, in line with the Bertsimas–Sim formulation. This structure guarantees protection against the worst-case realization within the prescribed uncertainty set, while maintaining a linear and tractable optimization model.

Robustness via budgeted uncertainty
For each customer node $i \in N$ an uncertainty width δ_i is defined, representing the maximum increase in effective decay due to adverse travel conditions. The dual feasibility conditions of the robust counterpart are enforced through the following constraints:

- $\lambda \geq \delta_i \sum_{k \in K} \theta_{ik} \quad \forall i \in N, \forall k \in K$
- $\lambda + s_i \geq \delta_i \theta_{ik} \quad \forall i \in N$

The complete objective function is :

$$max \sum_{i \in N} CP_i - \lambda\Gamma_{eff} - \sum_{i \in N} s_i - w_T \sum_{k} \sum_{ij} d_{ij} x_{ijk}$$

Algorithm 1 MILP Procedure for the RT-CTOP (CP–θ–λ formulation)

Input: Nodes V with depot 0, customers $N = V \setminus \{0\}$, prizes p_i $(i \in N)$, decay rates α_i, vehicles K, time budget T, robustness budget Γ

Input: Parameters: `delta_factor`, `gamma_scale`, minimum coverage fraction ρ, balancing flag

Output: Routes for each vehicle, objective value, profit, travel cost, robust penalty, best bound, relative MIP gap

Compute distances d_{ij} for all $i \neq j$

Compute uncertainty widths $\delta_i \leftarrow \texttt{delta\_factor} \cdot \alpha_i \quad \forall i \in N$

Compute effective robustness budget $\Gamma_{\text{eff}} \leftarrow \max\{0, \text{round}(\texttt{gamma\_scale} \cdot \Gamma)\}$

Set big-M constant $M \leftarrow T + \max_{i \neq j} d_{ij}$

Build MILP model:

Create decision variables:

- $x_{ijk} \in \{0,1\}$ for $i \neq j$ (routing)
- $y_i \in \{0,1\}$ for $i \in N$ (node visited)
- $z_{ik} \in \{0,1\}$ for $i \in N, k \in K$ (node assigned to vehicle)
- $a_{ik} \in [0,T]$ for $i \in V, k \in K$ (arrival time)
- u_{ik} for $i \in N, k \in K$ (MTZ order)
- $CP_i \geq 0$ for $i \in N$ (collected profit)
- $\theta_{ik} \in [0,T]$ for $i \in N, k \in K$ (effective arrival time)
- $\lambda \geq 0$, $s_i \geq 0$ for $i \in N$ (robustness variables)

Set objective:

$$\max \sum_{i \in N} CP_i \; - \; \lambda \, \Gamma_{\text{eff}} \; - \; \sum_{i \in N} s_i \; - \; \sum_{k \in K} \sum_{\substack{i,j \in V \\ i \neq j}} d_{ij} \, x_{ijk}.$$

Add routing and assignment constraints:

- *Visit linking (aggregate):* $\sum_{k \in K} \sum_{\substack{i \in V \\ i \neq j}} x_{ijk} = y_j \quad \forall j \in N$
- *Flow conservation:* $\sum_{\substack{i \in V \\ i \neq h}} x_{ihk} = \sum_{\substack{j \in V \\ j \neq h}} x_{hjk} \quad \forall h \in N, \forall k \in K$
- *Depot degree / vehicle usage:*

$$\sum_{j \in N} x_{0jk} \leq 1, \quad \sum_{i \in N} x_{i0k} \leq 1, \quad \sum_{j \in N} x_{0jk} = \sum_{i \in N} x_{i0k} \quad \forall k \in K;$$

- *MTZ subtour elimination:* $u_{ik} - u_{jk} + (|V|-1)x_{ijk} \leq |V| - 2 \quad \forall i \neq j, \; \forall i, j \in N, \; \forall k \in K$
- *Assignment via incoming flow:* $\sum_{\substack{j \in V \\ j \neq i}} x_{jik} = z_{ik} \quad \forall i \in N, \forall k \in K$
- *Unique assignment:* $\sum_{k \in K} z_{ik} = y_i \quad \forall i \in N$

Add time feasibility constraints:

- *Depot time:* $a_{0k} = 0 \quad \forall k \in K$
- *Time propagation:* $a_{ik} + d_{ij} \leq a_{jk} + M(1 - x_{ijk}) \quad \forall k \in K, \forall i \in V, \forall j \in N, \; j \neq i$
- *Return-to-depot feasibility:* $a_{ik} + d_{i0} \leq T + M(1 - x_{i0k}) \quad \forall i \in N, \forall k \in K$
- *Route duration (arc-based):* $\sum_{\substack{i,j \in V \\ i \neq j}} d_{ij} \, x_{ijk} \leq T \quad \forall k \in K$

Add decaying-profit linearisation:
$\theta_{ik} \le a_{ik},\ \theta_{ik} \le T z_{ik},\ \theta_{ik} \ge a_{ik} - T(1 - z_{ik}) \quad \forall i \in N, \forall k \in K$
$CP_i = p_i y_i - \alpha_i \sum_{k \in K} \theta_{ik} \quad \forall i \in N$

Add robust (Bertsimas–Sim) constraints:
$\lambda \ge \delta_i \sum_{k \in K} \theta_{ik}, \quad \lambda + s_i \ge \delta_i \sum_{k \in K} \theta_{ik} \quad \forall i \in N$

if $\rho > 0$ **then**
  Add minimum coverage constraint: $\sum_{i \in N} y_i \ge \lceil \rho |N| \rceil$

if **balancing is enabled** **then**
  Add per-vehicle visit cap (simple balancing)

Solve MILP with CPLEX under a time limit **if** **no feasible solution** **then**
  return **None**

Extract solution:
For each vehicle k, extract arcs with $x_{ijk} = 1$ and reconstruct the route by following arcs from depot 0
Compute: profit $\sum_{i \in N} CP_i$, travel cost $\sum_k \sum_{i \ne j} d_{ij} x_{ijk}$, robust penalty $\Gamma_{\text{eff}} \lambda + \sum_{i \in N} s_i$
Read solver statistics (best bound, relative MIP gap)
return **routes, objective, profit, travel cost, robust penalty, best bound, relative gap**

This pseudo-code is based on the full mixed integer linear programming implementation for the RT-CTOP presented in **Appendix A.1**

3.5.4 Step 4: Algorithm Development

Once the CTOP is formally defined and extended to incorporate decreasing rewards and robust optimization, resulting in the RT-CTOP, the model is adapted to fit two metaheuristic solution approaches: a genetic algorithm and a variable neighborhood search:

1. Genetic Algorithm (GA)

This genetic algorithm implements a population-based, mutation-only metaheuristic specifically designed to construct vehicle routes and evaluate them under a robustness-aware objective function. At each generation, the algorithm maintains a population of candidate solutions, where each individual represents a complete routing plan for the entire fleet.

Representation and fitness

This component of the genetic algorithm is designed following the concept of budgeted uncertainty introduced by Bertsimas and Sim (2004), in which uncertainty is controlled through a parameter Γ that limits the number of coefficients allowed to deviate simultaneously in a worst-case scenario.

Each candidate solution g is encoded as a set of routes, one per vehicle, with every route starting and ending at the depot. The internal sequence of customers within each

route determines both the total travel cost and the arrival time a_i at each visited node $i \in N$.

Given these arrival times , node profits are computed using a time-dependent, linearly decaying reward function. The nominal collected reward at node i defined as

$$CP_i = max\{p_i - \alpha_i a_i, 0\}$$

where p_i denotes the initial reward associated with node i and α_i is the corresponding decay rate. The nominal total profit of solution g is then given by

$$P^{nom}(g) = \sum_{i \in \mathcal{V}(g)} CP_i$$

where $\mathcal{V}(g) \subseteq N$ denotes the set of nodes visited by solution g.

Adversarial evaluation and robustness

Robustness is incorporated through an adversarial evaluation mechanism applied during the fitness computation. Under a budgeted uncertainty scheme, an attacker is assumed to have a limited budget Γ that represents the maximum number of visited nodes that may be adversely affect by deteriorated conditions (e.g. unexpected delays or increased decay effects).
For each visited node i, the adversary evaluates the potential loss in reward by increasing the decay effect. The attacked reward is defined as

$$CP_i^{att} = max\{p_i - (\alpha_i + \delta)a_i, 0\}$$

where δ is the adversarial intensity parameter. The resulting adversarial loss at node i is therefore

$$\Delta_i = CP_i - CP_i^{att}$$

The adversary then selects the subset $\mathcal{I}_\Gamma(g) \subseteq \mathcal{V}(g)$ of at most Γ nodes with the largest losses Δ_i. This selection corresponds to a worst-cases scenario in which the most vulnerable nodes are impacted simultaneously. The total adversarial penalty associated with solution g is defined as

$$Penalty(g) = \sum_{i \in \mathcal{I}_\Gamma(g)} \Delta_i$$

Importantly, the adversary does not modify the routing structure itself. Instead, it acts purely as an evaluation oracle that assigns a pessimistic score to each candidate solution based on its sensitivity to adverse conditions.

Robust fitness function

The robust fitness value used for selection and replacement in the GA is defined as

$$\text{fitness}(g) = P^{nom}(g) - Penalty(g) - w_T \cdot \text{TravelCost}(g)$$

where w_T denotes the travel cost weight and $TravalCost(g)$ represents the total distance (travel time) incurred by solution g.

This fitness value represents the worst-case achievable objective value under the assumed adversarial conditions. As a consequence, the evolutionary process naturally favors routing solutions that are less sensitive to delays, better balanced in arrival times, and more robust with respect to reward degradation, without explicitly enforcing additional constraints on the route construction.

Initialization

The population size is user-defined. At creation, each individual is generated by randomly inserting customers into the vehicle routes and then pruning the solution until it satisfies the time-budget and routing constraints. This yields a set of feasible but structurally diverse initial solutions. Optionally, one trivial individual with empty routes [0,0] for all vehicles may be included as a baseline, but it is not the only feasible solution in the initial population.

Variation operators

The variation mechanism of the proposed genetic algorithm is designed following classical principles of evolutionary search, where mutation operators play a central role in exploring the solution space and maintaining population diversity (Goldberg and Deb, 1991). In contrast to traditional crossover-based genetic algorithms, this implementation relies exclusively on mutation operators to modify and improve routing structures, a design choice commonly adopted in routing and permutation-based problems where localized changes are more effective than recombination.

Four mutation operators are used, selected randomly:

- Insert-unvisited mutation: selects a customer that is not yet visited in the current solution and inserts it into a randomly chosen route at a randomly chosen feasible position (before the last depot). This operators allow the GA to "grow" solutions from sparse or even empty routes by adding new customers.
- Insert mutation: removes a randomly chosen visited customer from its current position and reinserts it into a randomly chosen route and insertion index. This operator changes the assignment and ordering of already served customers.
- Swap-within mutation: within a randomly selected route, swaps the positions of two internal customers.
- Move-between mutation: moves a randomly chosen internal customer from one route to another.

The operator is chosen by sampling $u \sim U[0,1]$. If $u < 0.25$, the insert-unvisited mutation is applied; if $0.25 \leq u < 0.50$ the insert (relocate) mutation is used; if $0.50 \leq u < 0.75$ the swap-within mutation is triggered; otherwise, the move-between mutation is applied.

Selection and replacement

After all individuals have been evaluated using the robust fitness function, the algorithm proceeds to the selection and replacement phase. This step determines how

the next generation is formed and is crucial both for guiding the search toward high-quality routing solutions and for maintaining sufficient diversity to avoid premature convergence in line with the theoretical foundations of evolutionary algorithms (Back (1996)).
To process involves 4 main steps :

1. Evaluation

   Each individual is evaluated using the robust fitness function defined above.

2. Sorting

   Individuals are sorted in descending order of fitness:

   $$g_1, g_2, \ldots, g_{\text{pop}} \quad \text{with} \quad f(g_1) \geq f(g_2) \geq \ldots$$

   This ranking identifies the best solutions, the weakest ones to be replaced and the candidates for parent selection.

3. Elitism

   To prevent the loss of high-quality solutions due to stochastic effects, an elitist strategy is applied. The number of elite individuals is defined as

   $$E = \max\left(1, \frac{\text{pop\_size}}{20}\right)$$

   The top E individuals are copied directly into the next generation without modification. This guarantees solution quality and prevents regressions.

4. Creation of remaining population

   The remaining individuals of the next generation are produced via binary tournament selection followed by mutation. For each spring:

   4.1 Binary tournament selection

   (a) Two individuals g_a and g_b are selected uniformly at random.

   (b) Their fitness value are compared, and the parent is chosen as

   $$\text{parent} = \begin{cases} g_a & \text{if } f(g_a) \geq f(g_b) \\ g_b & \text{otherwise} \end{cases}$$

   This mechanism offers several advantages : Better individuals with better solution quality have higher probability of being selected, weaker individuals can still be selected the minority of times (to maintain diversity) and the cost is very cheap because it compares only 2 individuals.

   4.2 Mutation-only offspring generation

   Unlike classical genetic algorithms, this implementation does not use crossover. Instead, exploration is driven exclusively by the mutation operators described earlier.

A child is created from the selected parent as follows: with probability equal to the mutation rate, one of the mutation operators is applied; otherwise, the parent is copied unchanged. The resulting child is always inserted into the new population. Feasibility is encouraged indirectly: infeasible offspring receive a fitness of $-\infty$ under the robust objective and are therefore very unlikely to be selected in subsequent generations. This mutation-driven evolutionary process continues until the new population reaches full size.

Algorithm 2 Genetic Algorithm for RT-CTOP

Input: Cities, prices, decay, K, T, Γ, δ, population size, generations, mutation rate
Output: Best solution g^*

```
Precompute distances between all cities
Initialize population P of size pop_size with random routing plans
foreach g ∈ P do
    while FEASIBLE(g) = false do
        Remove a random internal customer from g and CANONICALIZE(g)
    fitness(g) ← ROBUSTOBJECTIVE_GA(g)
for gen = 0 to generations do
    Sort P by fitness in descending order
    Record statistics (best, mean, median)
    Apply elitism: copy top E individuals into P_new
    while |P_new| < pop_size do
        Select g1, g2 uniformly at random from P
        parent ← arg max{fitness(g1), fitness(g2)}
        child ← parent
        if rand() < mutation_rate then
            child ← MUTATE(child) // Insert-unvisited / insert / swap-within /
              move-between
        Add child to P_new
    P ← P_new
    foreach g ∈ P do
        fitness(g) ← ROBUSTOBJECTIVE_GA(g)
g* ← best individual in final population  return g*
```

This pseudo-code is based on the full genetic algorithm implementation for the RT-CTOP presented in **Appendix A.1**

2. Variable Neighborhood Search (VNS)

The VNS algorithm implemented in this research is a metaheuristic designed to solve the RT-CTOP by systematically exploring multiple neighborhood structures around the current best solution. The method alternates between diversification phases, achieved through controlled random perturbations (shaking), and intensification phases, achieved through deterministic local search. To address long stagnation phases, a large perturbation mechanism based on ruin-and-recreate is additionally incorporated. This general framework follows the principles of VNS introduced by Hansen et al. (2010) and is consistent with successful applications of VNS to vehicle routing problems, such as the work of Cacchiani et al. (2014).

The fundamental principle of VNS is that a solution that is locally optimal with respect to one neighborhood structure may not be locally optimal with respect to another. Consequently, the algorithm explores neighborhoods of growing complexity in a structured manner to increase to the probability of escaping local optima and identifying high-quality robust routing solutions.

Robust objective evaluation

As in the GA, robustness within the VNS is incorporated directly through the objective evaluation, rather than by enforcing additional constraints on the routing structure. The fitness of each candidate solution is therefore computed under a robustness-aware objective inspired by the budgeted uncertainty framework of Bertsimas and Sim (2004). For a given routing solution, arrival times determine the nominal reward at each visited node, after which an adversary selects up to Γ nodes with the highest vulnerability. The corresponding worst-case penalty is subtracted from the nominal profit.

Each candidate solution g encodes a set of routes (one per vehicle). From the route structure, arrival times a_i are computed for all visited nodes $i \in \mathcal{V}(g) \subseteq N$. The nominal collected reward at node i is

$$CP_i = max\{p_i - \alpha_i a_i, 0\}$$

and the nominal total profit is

$$P^{nom}(g) = \sum_{i \in \mathcal{V}(g)} CP_i$$

Robustness is incorporated through a budgeted adversary with budget Γ. Under adverse conditions, the reward at node i becomes

$$CP_i^{att} = max\{p_i - (\alpha_i + \delta)a_i, 0\}$$

leading to a node-wise loss

$$\Delta_i = CP_i - CP_i^{att}$$

The adversary selects the subset $\mathcal{I}_\Gamma(g) \subseteq \mathcal{V}(g)$ of size at most Γ with the largest losses Δ_i. The total worst-case penalty is

$$Penalty(g) = \sum_{i \in \mathcal{I}_\Gamma(g)} \Delta_i$$

Finally the robust objective value (fitness) used throughout VNS is

$$f(g) = P^{nom}(g) - Penatly(g) - w_T \cdot TravelCost(g)$$

All neighborhood moves (shaking, local search, and ruin-and-recreate) are evaluated using this same robust objective. As a result, the VNS naturally favors solutions that are less sensitive to delays and better balanced in arrival times, without explicitly modifying the routing structure to enforce robustness.

Neighborhood Structure
A neighborhood of a solution s, denoted $\mathcal{N}(s)$, is defined as the set of all solutions that can be obtained from s by applying a single move of a given type. The VNS implemented in this work uses five neighborhood structures (Hansen et al. (2010), Cacchiani et al. (2014)), each induced by one of the following move operators:

1. **Remove**

   The goal of this move is to remove unhelpful customers from the solution in order to simplify the routing plan. Removing a customer reduces route length, frees time budget for more important customers, and can decrease the adversarial penalty by eliminating high-vulnerability nodes. This operator is particularly useful when a route contains customers whose service cost is high relative to their decayed profit.

2. **Insert unvisited**

   This operator aims to include currently unvisited customers. It considers inserting an unvisited node into all feasible positions across all routes. This is important because the initial solution often leaves many customers unserved, inserting new customers can increase profit when their remaining reward is sufficiently high. Moreover, this operator introduces structural diversity into the solution space and helps prevent premature convergence. Only feasible insertions, i.e. those that respect the time budget T, are retained.

3. **Relocate**

   The relocate move improves the route structure by moving customers. It extracts a customer from one route and reinserts it either into another vehicle's route or into a different position within the same route. Relocation is one of the most powerful local operators for VRP-type problems: it redistributes the load among vehicles, changes arrival times (thus altering the decay impact on rewards) and can simultaneously reduce travel cost and increase total profit.

4. **2-opt inside a route**

   This operator locally improves a route by modifying the order of customers. Specifically, it reverses a contiguous subsequence of nodes inside a single route. The 2-opt move is a classical VRP/TSP improvement technique that shortens routes, improves arrival times, and removes inefficient detours. Since the reward is time-sensitive due to decay, even small reductions in travel time can have a positive effect on the overall fitness.

5. **Swap**

   The swap move exchanges the positions of two customers, potentially located in different routes. By swapping two customers, this operator can reduce travel cost, improve the balance and fairness between vehicles, and reduce adversarial sensitivity by altering which nodes are visited earlier or later in time.

Steepest descent local search

The local search component of the VNS follows a steepest descent (best-improvement) strategy. This approach systematically explores the neighborhood of the current solution and always applies the move that yields the largest improvement in the objective function among all evaluated neighbors (Hansen et al., 2010).

Given a current solution s, a finite set of neighboring solutions $\mathcal{N}(s)$ is generated by applying all neighborhood operators, subject to feasibility constraints. Since the complete neighborhood may be prohibitively large a representative subset of neighbors is sampled for each move type.

Each neighbor $s' \in \mathcal{N}(s)$ is evaluated using the robust fitness function $f(\cdot)$, which account for time decayed reward, adversarial penalties and travel costs. Among all sampled neighbors , the steepest descent rule selects the one with the highest fitness value:

$$s^+ = \arg\max_{s' \in \mathcal{N}s} f(s')$$

If the best neighbor improves upon the current solution that is

$$f(s^+) > f(s)$$

the move is accepted and the current solution is updated according to

$$s \leftarrow s^+$$

The neighborhood exploration then restarts from the new solution to guarantee that each accepted move yields the maximum possible improvement at stage. The process ends when no sampled neighbor provides an improvement over the current solution :

$$f(s') \leq f(s) \quad \forall s' \in \mathcal{N}(s)$$

At this point the solution s is said to be a local optimum with respect to the explored neighborhoods.

Shaking and neighboring change

In the VNS outer loop, neighborhood are explored sequentially using an index $k \in \{1, \ldots, k_{max}\}$. Before local search, a diversified solution is generated by shaking:

$$s_{shake} = Shake(s_{best}, k)$$

where $Shake(\cdot, k)$ applied k random moves (mutation operators). Local search then yields

$$s_{local} = LS(s_{shake})$$

If the new solution improves the incumbent best, it is accepted and the neighborhood index is reset, if no the algorithm proceeds to the next neighborhood.

$$\begin{cases} s_{best} \leftarrow s_{local}, k \leftarrow 1 & \text{if } f(s_{local}) > f(s_{best}) \\ k \leftarrow k+1 & \text{otherwise} \end{cases}$$

Ruin-and-recreate

To mitigate prolonged stagnation during the search process, a diversification mechanism inspired by the ruin-and-recreate principle proposed by Schrimpf et al. (2000) is employed. The algorithm tracks the number of iterations without improvement, denoted c. If no improvements occurs in an outer iteration the counter is incremented $c \leftarrow c+1$ and reset to zero whenever an improvement is found.

When stagnation reaches a threshold $patience$, a larger pertubation is applied:

$$s_{kick} = RuinRecreate(s_{best}, q)$$

where q customers are removed and then reinserted greedily into feasible positions (lowest insertion cost while satisfying the time constraint). The stagnation counter is then rest.

Algorithm 3 Variable Neighbourhood Search for RT-CTOP

Input: Cities, prices, decay, K, T, Γ, travel_cost_weight, $k_{\max}$, iterations
Output: Best solution g^*

```
Precompute distances
g ← RandomInitialSolution()
Repair g to satisfy feasibility constraints (if needed)
best ← g
best_val ← RobustObjective_VNS(best)
since_best ← 0
for it = 1 to iterations do
    k ← 1
    improved_outer ← false
    while k ≤ k_max do
        s_shake ← Shake(best, k)
        s_local ← LocalSearchDescent(s_shake)
        v_local ← RobustObjective_VNS(s_local)
        if v_local > best_val then
            best ← s_local
            best_val ← v_local
            k ← 1
            improved_outer ← true
            since_best ← 0
        else
            k ← k + 1
    if not improved_outer then
        since_best ← since_best + 1
    if since_best ≥ patience then
        best ← RuinAndRecreate(best)
        best_val ← RobustObjective_VNS(best)
        since_best ← 0
    Record (it, best_val) in trace
return best
```

This pseudo-code is based on the full implementation of the genetic algorithm for the RT-CTOP presented in **Appendix A.1**

3.5.5 Models inputs

All three solution approaches (MILP, GA, and VNS) are implemented and executed within a single unified computational framework. The algorithms are grouped in a single notebook cell to ensure identical execution conditions and consistent data handling. A wide range of parameters can be configured, as detailed in **Appendix A.1**. To facilitate systematic experimentation, the parameters common to all models are defined as arrays and allows the program to automatically iterate over multiple instance configurations.

Common parameters: The following parameters are shared by the three models and remain identical across algorithms for a given configuration:

- N: Number of customers
- Gamma frac (γ): Instead of having an absolute Γ, we adopt a fractional formulation defined as $\Gamma = \lceil \gamma \cdot N \rceil$ where $\gamma \in [0;1]$ is the gamma fraction. This allows the robustness budget to scale with the instance size and stay consistent as the number of customers increases
- K: Number of vehicles in the fleet
- T: Time capacity (route duration limit) for each vehicle
- Price range: Interval defining the uniform distribution used to sample customer rewards
- Decay range: Range defining the random distribution of reward decay rates associated with each customer
- Travel cost weight: Weight applied to travel distance in the objective function, controlling the trade-off between collected profit and routing cost
- Setups: Number of distinct customer spatial distributions
- Repetitions: Number of repetitions per configuration, change only sampled for prices and decay rates

Model-specific parameters: In addition to the common parameters, each algorithm relies on a set of method-specific tuning parameters:

- MILP parameters
  - delta_factor =1: Scaling factor used in the robust penalty formulation
  - gamma_scale =0.8: Adjustment parameter controlling the effective robustness budget within the MILP model.
  - runtime limit =30 seconds: Runtime limit imposed on the MILP
- GA parameters
  - delta_ga =1: Robustness-related penalty coefficient
  - pop_size =150: Population size

- generation =800: Maximum number of generations
- time_limit=30 seconds: Runtime limit imposed on the algorithm
- high_priority_frac =0.2: Fraction of high-priority customers emphasized during selection and evaluation

- VNS parameters
  - delta_vns =1 : Robustness-related penalty coefficient
  - K_max =6: Maximum number of neighborhood structures explored.
  - Local search sampling: Fixed sample sizes for neighborhood operators (insert=50, remove=50, reloc=50, twoopt=50, swap=50).
  - patience =50: Maximum number of iterations without improvement before diversification
  - kick_size =10: Magnitude of the shaking phase
  - time_limit =30 seconds: Runtime limit imposed on the algorithm
  - high_priority_frac =0.2: Fraction of high-priority customers emphasized during evaluation

3.5.6 Models outputs

At the end of each experimental run, the outputs of all models are systematically collected and stored in a single consolidated results table. Each row of the table corresponds to one algorithm execution for a specific configuration defined by the setup, instance parameters, and repetition index. The table is composed of two categories of fields: instance descriptors and performance metrics.

Table 1: Example of the output table obtained after a run with the instance parameters N=[15,25,30], Γ=0.3, K=3, T=250, Runtime=30 seconds, Setup=1, Prices range=[25;250], Decay rates range[0.01;0.1]

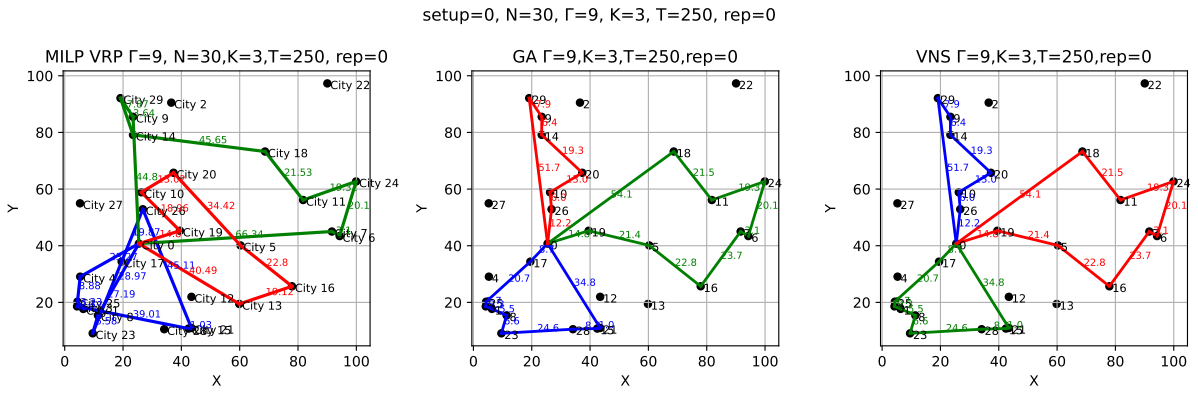


Figure 3: Example of the output solutions for MILP , GA and VNS
Source: Table 1

3.5.7 Step 5: Performance evaluation

1. Motivation and context
To complete the algorithm description presented in the previous steps we analyze the asymptotic computational complexity of the three solution framework considered in this work. MILP, GA and VNS.

The analysis is conducted using the Big-Theta (Θ) asymptotic notations, which gives a tight bound that describes the exact asymptotic growth when upper and lower bound coincide.

The objective of this discussion is not to obtain exact execution time , which depend on parameters setup, hardware and solver heuristics, but rather to quantify how each method scales with the number of customer N. Such complexity bounds help to explain the behavior observed in the numerical experiments and reveal wether an algorithm can scale to real-world problem sizes, which components of the method dominate computational cost and where bottleneck or inefficiencies are most likely to occur.

2. Time and space complexity analysis

2.1 Computational complexity of MILP

The exact formulation proposed in this work is expressed as a Mixed-Integer Linear Program (MILP). It is well established that MILP is NP-hard, even when restricted to binary decision variables (Garey and Johnson (1982)). The computational difficulty of solving a MILP exactly stems from the presence of binary variables, which induce a combinatorial search space.

In the proposed formulation, the main source of combinatorial complexity arises from the binary routing variables x_{ijk}, which indicate whether vehicle k travels from node i to node j, as well as from the auxiliary binary variables z_{ik} and y_i. For n customers and K vehicles, the total number of binary variables is dominated by the routing variables and scales as

$$B = |x| + |y| + |z| = \Theta(Kn^2) + \Theta(n) + \Theta(Kn) = \Theta(Kn^2)$$

Modern MILP solvers, such as CPLEX, rely on branch-and-bound or branch-and-cut algorithms. In the worst case, these methods may explore a number of nodes in the search tree that grows exponentially with the number of binary variables. Consequently, the worst-case time complexity of solving the MILP is exponential in B.
Worst-case time:

$$T_{MLIP} = O(2^B) = O(2^{\Theta(Kn^2)}$$

2.2 Computational complexity of the Genetic Algorithm

To analyze the computational effort required by the GA described above, we refer both to its pseudo-code (**Algorithm 2**) and to the implementation in **appendix A.1**

Let

- n: number of customer (non-depot nodes)
- P: Population size
- G: number of generations
- K: number of vehicles (assumed constant)
- Γ: robustness parameter ($\Gamma \leq n$)

Since the genetic algorithm consists of three main phases : preprocessing, fitness evaluation and evolutionary iterations, we will analyze each component separately.

1. Preprocessing

(a) Distance matrix construction

The algorithm first computes pairwise distances between the n+1 nodes (customers + depot). This produces a $(n+1)*(n+1)$ matrix:

$$\Theta(n^2)$$

(b) Customer list construction

Creating the list of customers requires linear time:

$$\Theta(n)$$

(c) Initial population generation

The population consists of P individuals (genes), each encoding routes that collectively visit all customers:

$$\Theta(Pn)$$

(d) Initial scoring and sorting

Each individual is evaluated once:

- Fitness evaluation: $\Theta(n \log n)$
- Total scoring of population: $\Theta(Pn \log n)$
- Sorting population by fitness $\Theta(P \log P)$

The overall preprocessing time is :

$$T_{pre}(n, P) = \Theta(n^2 + Pn + Pn \log n + P \log P)$$

Since n^2 (distance matrix) and $Pn \log n$ (initial fitness) grow fastest, the dominant contributions are :

$$T_{pre}(n, P) = \Theta(n^2 + Pn \log n)$$

2. Fitness evaluation

Each call of the robust function performs:

- Canonicalization and arrival time computation: $\Theta(n)$
- Construction of reward and adversarial reward tables: $\Theta(n)$
- Creation and sorting of the losses list: $\Theta(n \log n)$
- Slicing of top Γ and penalties computation: $\Theta(\Gamma) \leq \Theta(n)$
- Travel cost computation $\Theta(n)$

The fitness evaluation is dominated by the sort of losses list $n \log n$

$$T_{fitness}(n) = \Theta(n \log n)$$

3. Mutation operators

Each mutation:

- copies a full gene:$\Theta(n)$
- scans routes to locate a node:$\Theta(n)$
- applies a local modification:$O(1)$
- canonicalizes the result:$\Theta(n)$

$$T_{mutate}(n) = \Theta(n)$$

4. Main evolution loop

For each generation (from 0 to generations)

1. Compute population statistics
2. Create new population
3. Re-score population
4. Sort by fitness

Total time per generation

$$T_{gen}(n, P) = \Theta(Pn \log n + Pn + P \log P + P)$$

Dominant term:

$$T_{gen}(n, P) = \Theta(Pn \log n)$$

Overall complexity of GA

$$T_{GA}(n, P, G) = \Theta(n^2 + GPn \log n)$$

If P and G are treated as fixed parameters , the complexity reduces asymptotically to:

$$T_{GA}(n) = \Theta(n^2)$$

In the proposed GA for the RT-CTOP, the dominant computational cost arises from two components: (i) the precomputation of the all-pairs distance matrix and (ii) the repeated evaluation of the robust fitness function over the population. The distance matrix requires quadratic time and space $\Theta(n^2)$ where n denotes the number of customers. This preprocessing step is unavoidable for routing-based algorithms and is common to both heuristic approaches considered in this work.

Each individual solution (gene) encodes a set of vehicle routes whose total size is linear in the number of customers. Evaluating a gene requires computing arrival times, route costs, and robust penalties. In particular, the robust objective function involves sorting a list of at most n adversarial profit losses in order to extract the Γ worst-case contributions. This sorting step dominates the fitness evaluation and leads to a per-individual complexity of $\Theta(n \log n)$.

At each generation, the algorithm evaluates P individuals and applies mutation operators that run in linear time with respect to n. Over G generations, this results in a total time complexity of $\Theta(n^2 + GPn \log n)$ where the quadratic term corresponds to distance preprocessing and the remaining term reflects repeated fitness evaluations. Since the population size P and the number of generations G are fixed hyperparameters of the algorithm and do not scale with the instance size, the asymptotic complexity of the GA with respect to the number of customers simplifies to $\Theta(n^2)$.

2.3 Computational complexity of the Variable Neighborhood Search

To analyse the computational complexity of the VNSh described above, we refer to both the pseudo-code (**Algorithm 3**) and the actual implementation (**Appendix A.1**).

Let:

- n: number of customers (non-depot nodes)
- K: number of vehicles
- I: maximum number of outer iterations (iterations)
- k_{max}: maximum shaking amplitude

- q: number of customers removed during the diversification "kick"
- S: total number of sampled neighbors per local search pass

  $S = S_{remove} + S_{insert} + S_{reloc} + S_{2opt} + S_{swap}$
- D: number of improving descent passes performed by the steepest-descent loop inside the local search

1.Preprocessing phase

(a) Distance matrix construction
As in GA, VNS begins by precomputing all pairwise distances for $n+1$ nodes (customers + depot), leading to a complete matrix:

$$\Theta(n^2)$$

(b) Initial solution generation
A random initial gene is constructed in:

$$\Theta(n)$$

2.Fitness (robust objective evaluation

- route canonicalisation and arrival computation: $\Theta(n)$
- construction of profit tables and adversarial profit tables: $\Theta(n)$
- computation of profit gaps and sorting to extract the top Γ: $\Theta(n \log n)$
- travel computation: $\Theta(n)$

$$T_{fitness}(n) = \Theta(n \log n)$$

3.Shaking phase

The shaking procedure applies k random moves (insert unvisited/relocate/swap). Each move involves

- locating customers in routes
- applying a move that copies and canonicalises the gene

In the worst-case:

$$T_{shake}(n, k) = \Theta(k_{max} n)$$

4.Local search by steepest descent

Evaluation of multiple neighborhoods (remove,insert unvisited, relocate, 2-opt, swap). Samples at most S neighbors per descent pass.

(a) Cost per neighbor evaluation

Generating a neighbor (copying the gene and applying move): $\Theta(n)$
Evaluating its fitness requires: $\Theta(n \log n)$

Per neighbor:

$$T_{neighbor}(n) = \Theta(n \log n)$$

(b) Cost per descent pass

With at most S sampled neighbors:

$$T_{LS-pass}(n) = \Theta(S.n \log n)$$

(c) Total local search cost

Steepest descent repeats passes while improvements occur:

$$T_{LS}(n) = \Theta(D \cdot S \cdot n \log n)$$

5.Ruin-and-recreate diversification

Apply when stagnation persist:

- Remove q customers: $\Theta(n)$
- Reinsert each removed customer by testing all insertion (scan insertion position and check feasibility): $\Theta(n^2)$

$$T_{kick}(n, q) = \Theta(qn^2)$$

6. Overall complexity of VNS

$$T_{VNS}(n) = \Theta(n^2 + I(k_{max}n + k_{max}DSn \log n + qn^2))$$

If k_{max},S,q, and I are treated as fixed design parameters, the complexity reduces asymptotically to

$$T_{VNS}(n) = \Theta(n^2)$$

As with the GA, the VNS requires an initial preprocessing step to compute the all-pairs distance matrix, which takes $\Theta(n^2)$ time for n customers and is common to all routing-based methods considered in this work. Solution evaluation relies on the robust objective function, which computes arrival times, travel costs and worst-case penalties by sorting at most $n$adversarial profit losses, resulting in a per-evaluation cost of $\Theta(n \log n)$.

During each outer iteration, VNS applies a sequence of shaking operations followed by an intensification phase based on steepest-descent local search. Each shaking step consists of a small number of random local modifications and runs in linear time with respect to n. The local search explores multiple neighborhood structures using a fixed number of sampled neighbors per neighborhood; since each neighbor evaluation requires a robust fitness computation, the overall cost of the local search phase is proportional to the number of sampled evaluations times $\Theta(n \log n)$.

To mitigate stagnation, VNS occasionally applies a ruin-and-recreate diversification mechanism. Although this operator has quadratic worst-case complexity due to the evaluation of multiple reinsertion positions, it removes only a small, fixed number of customers and is triggered infrequently, limiting its impact in practice.

Aggregating these components, the overall time complexity of the VNS can be expressed as

$$\Theta(n^2 + I(n^2 + n \log n))$$

where I denotes the number of outer iterations. When I and all neighborhood parameters are treated as fixed design choices, the asymptotic complexity with respect to the number of customers simplifies to $\Theta(n^2)$.

4 Numerical Experiment

4.1 Context

The numerical experiments are organized in two complementary stages. In the first stage, we consider a reference experiment designed to highlight the general performance trends of the three models (MILP, GA and VNS). These experiment relies on relatively optimistic settings that allow all methods to perform under favorable conditions. The goal of this initial analysis is not to stress the algorithms, but rather to provide an intuitive and controlled comparison of their behavior as the problem size increases. In particular, we examine how solution quality, runtime, and robustness evolve with the number of customers, and how the exact MILP approach compares to the heuristic methods in terms of scalability and optimality gap.

In the second stage, we perform a series of targeted experimental runs with modified parameter configurations. Each setup is designed to isolate a specific source of difficulty, such as tighter runtime limits, bigger robust budget or higher decay rates and price distributions. By varying one group of parameters at a time while keeping all others fixed, we are able to systematically assess how each method responds to increasing problem hardness and to identify their respective strengths and limitations.

4.2 Experimental setup

4.2.1 Instance generation policy

All experimental runs are performed on instances sharing the same core parameters, namely the number of customers N, the robustness budget Γ, the time horizon T, the number of vehicles K and the runtime limit. In addition, controlled stochasticity is introduced through the use of multiple customer spatial distributions and repeated randomization of economic parameters.

- Setups

  Each setup corresponds to a distinct customer spatial distribution (i.e., a different "city" configuration) generated using a different random seed.

- Customer spatial distribution

  Customer locations (nodes) are sampled uniformly within a two-dimensional 100×100 region. For each setup, the N customers are randomly distributed over the region using a fixed seed. When the number of customers increases, the existing customers remain unchanged and additional customers are appended to the original distribution, ensuring nested instances and consistent scalability analysis.

- Repetitions

  For a given configuration defined by $(N, \Gamma, T, K, runtime, setup)$, each model is executed for a fixed number of repetitions. Across repetitions, customer rewards (prices) and decay rates are randomly sampled within predefined ranges, while all other parameters remain unchanged. The same sampled prices are used for all models within each repetition.

4.2.2 Evaluation metrics

At the end of each experimental run, several performance metrics are collected to evaluate solution quality, robustness, and computational efficiency:

- **Objective value**: Final value of the objective function, defined as total collected profit minus travel costs and robustness-related penalties.
- **Travel cost**: Total distance traveled by the entire fleet multiplied by the unit travel cost.
- **Robust penalty**: Aggregate penalty incurred due to robustness constraints, reflects protection against adverse realizations as defined in the robust optimization formulation.
- **Optimality gap (MILP only)**: Percentage gap between the best feasible solution value obtained within the runtime limit and the optimal (or best known lower/upper bound) solution value reported by the MILP solver.
- **Runtime**: Wall-clock computational time required to reach the final solution, capped by a predefined runtime limit applied to all models.
- **Average route length**: Mean total distance traveled per vehicle, computed over all active routes in the solution.
- **Number of visited customers**: Total number of distinct customers served by the fleet.

4.3 Reference experiment

4.3.1 Reference experiment configuration

- $N = [6, 11, 21, 31, 41, 51]$
- $\Gamma = 0.3$
- $T = 250$
- $K = 3$
- Runtime limit $= 30$ seconds
- Number of different setups $= 3$
- Prices $= [25; 250]$
- Decay rate $= [0.01; 0.5]$
- Number of repetition $= 3$

4.3.2 Reference experiment analysis

Objective value: **Figure 4** illustrates the evolution of the objective value for the MILP, GA and VNS as a function of the number of customers. For small and medium-sized instances, all three methods exhibit a steady increase in objective value under favorable parameter settings. For larger instances, however, the behavior of the methods diverges. Starting from approximately $n = 30$, the MILP objective growth becomes less pronounced and then decreases sharply, whereas both GA and VNS continue to improve steadily and reach objective values in the range of $[3800; 4100]$. In this regime, the MILP, GA, and VNS display closely aligned objective value and confirms that the heuristic approaches are able to approximate the optimal solutions obtained by the exact formulation.

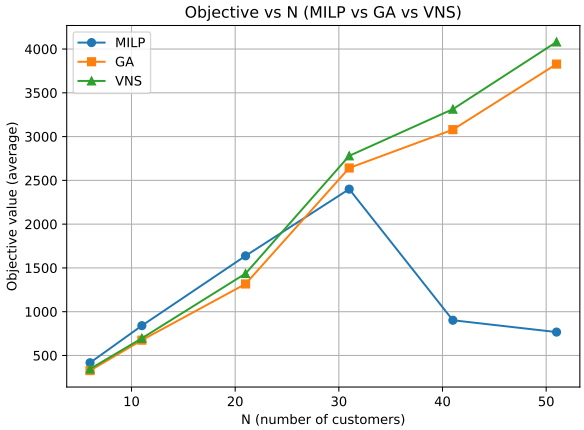


Figure 4: MILP-GA-VNS Objective value per customers
Source: Table 2

Runtime: **Figure 5** presents the average runtime per instance as a function of the number of customers for each method. From a computational perspective, the MILP runtime increases sharply with problem size. For instances exceeding $N = 30$ customers, the solver consistently reaches the imposed time limit, at which point the achieved objective value begins to decrease. This behavior is further confirmed in **Figure 6**, which shows a corresponding increase in the MILP optimality gap starting at $N = 30$ suggesting growing difficulty in closing the gap within the available runtime limit.

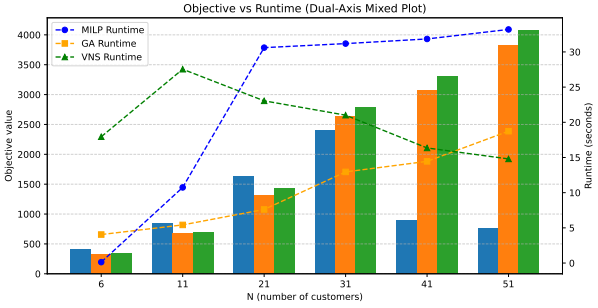


Figure 5: MILP-GA-VNS Objective value and runtime per customers
Source: Table 2

In contrast, the runtime of the GA increases steadily with the number of customers but remains well below the imposed runtime limit for all tested instances. The VNS exhibits a different pattern, also visible in**Figure 5**. Its runtime is initially higher than that of GA for small and medium-sized instances due to the cost of systematic neighborhood exploration and repeated local descent procedures. However, as the number of customers increases, the VNS runtime gradually decreases. This suggests that for larger instances, the search process converges more rapidly toward stable local optima, with fewer improving moves available in each neighborhood. As a result, the algorithm terminates earlier despite the increased problem size.

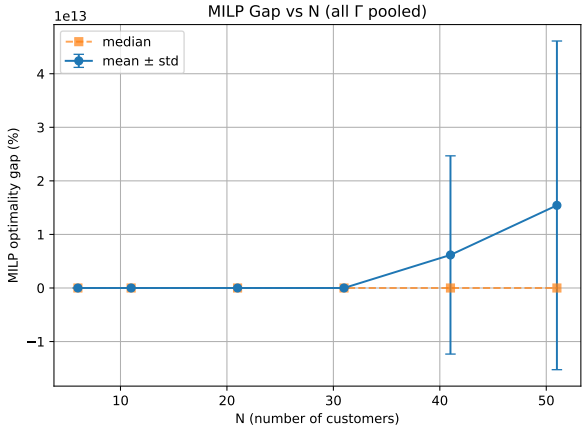


Figure 6: MILP optimality gap per customers
Source: Table 2

Routes: **Figure 7** presents the evolution of the average route length as a function of the number of customers for each method. A similar pattern emerges when analyzing route structure indicators. For small and medium-sized instances, the MILP, GA and VNS achieve relatively high coverage, with the MILP producing longer routes thanks to its ability to optimally exploit available customers when sufficient computational resources are available. However, once the number of customers exceeds approximately $N = 30$, a sharp drop in average route length and coverage is observed for the MILP.

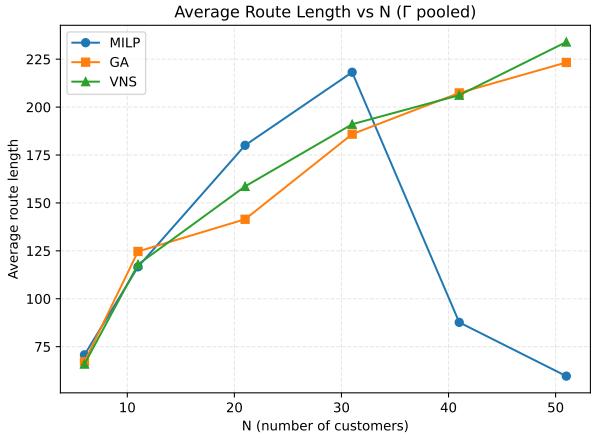


Figure 7: MILP-GA-VNS Average route length per customers
Source: Table 2

Robustness: To analyze the robustness of the models to uncertainty, we use the Coefficient of Variations (CVs), defined as the ratio between the standard deviation and the mean objective value. This metric provides a normalized measure of variability and allows direct comparison of stochastic sensitivity across models and instance sizes.

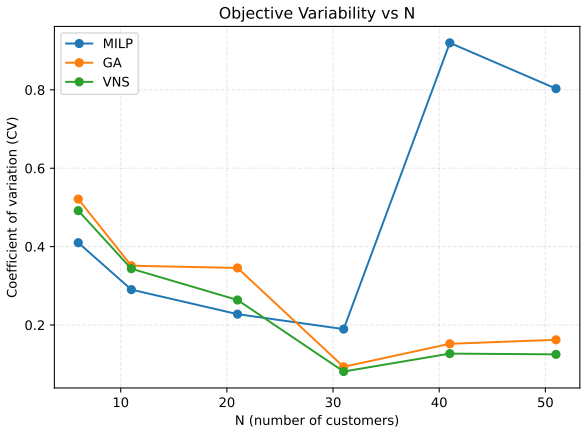


Figure 8: MILP-GA-VNS Coefficient of variation of the objective value
Source: Table 2

As shown in **Figure 8**, the CV of the MILP objective is relatively low for small instances but increases sharply once the number of customers exceeds $N = 30$. This indicates that, beyond this threshold, the MILP becomes increasingly sensitive to stochastic effects, largely due to the interaction between randomness and tight computational limits. In contrast, GA and VNS exhibit consistently low CV values across all instance sizes, with only a small increase of approximately 0.05 as N grows.

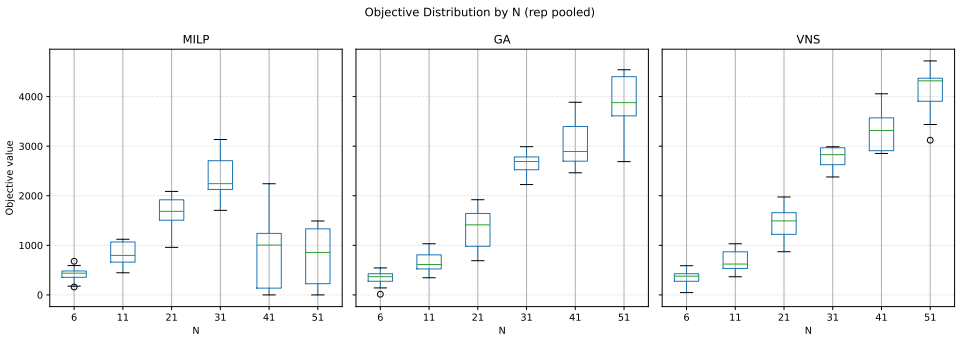


Figure 9: MILP - GA - VNS Objective value distribution
Source: Table 2

Figure 9 further supports this observation by showing box plots of the objective value distributions for each model. The MILP displays the widest distributions, characterized by long boxes and pronounced tails, which reflects high variability across repetitions. GA exhibits moderate dispersion, while VNS shows the narrowest distributions, particularly for larger instances. This indicates that VNS converges more consistently to similar-quality solutions and is the most robust to stochasticity among the evaluated methods.

Reference experiment conclusion: The results indicate that, as the number of customers increases, the heuristic methods are able to effectively exploit the larger pool of candidate locations to construct increasingly profitable routes while maintaining feasibility under robust constraints. In contrast, the MILP approach becomes increasingly constrained by its computational requirements, limiting its ability to scale to larger instances within the imposed time limits.

Moreover, in the problem regimes where the MILP can be solved effectively, the close alignment of objective values across all methods provides strong validation of both the proposed MILP formulation and the correctness and effectiveness of the heuristic implementations. This agreement confirms that the heuristics are capable of producing high-quality solutions that closely approximate the optimal solutions when sufficient computational resources are available.

4.4 Second stage experiment

4.4.1 Runtime limit

300 seconds: **Figure 10** illustrates the impact of increasing the maximum runtime to 300 seconds. Under this extended computational budget, the MILP objective value improves for instances with $N = 40$, indicating that additional time allows the solver to further refine solutions in this range. However, the MILP objective begins to decrease again for $N = 50$, which shows that even substantial increases in runtime cannot fully overcome the combinatorial complexity of larger instances. In practice, the MILP reaches the 300-seconds time limit already for instances with approximately $N = 20$ customers. In contrast, the objective values obtained by GA and VNS remain essentially unchanged when increasing the runtime limit. Both heuristics converge to high-quality solutions well before reaching 300 seconds.

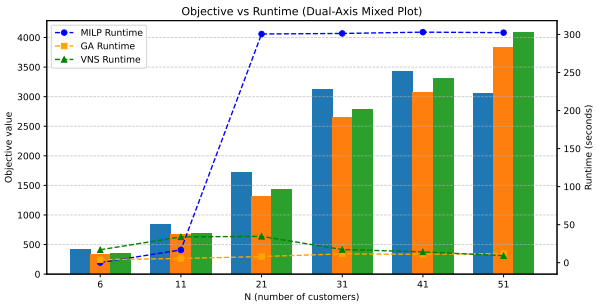


Figure 10: MILP-GA-VNS Objective value with runtime limit =300 seconds
Source: Table 3

3 seconds: **Figure 11** presents the results obtained when reducing the runtime limit to 3 seconds. In this extreme time-constrained setting, the performance of the MILP degrades significantly. The solver fails to construct meaningful solutions and does not reach objective values above 1000 even for instances with $N = 20$ customers. GA and VNS are also affected by the stricter runtime constraint, as they systematically reach the imposed time limit. However, their performance degradation remains moderate: on average, GA and VNS experience objective reductions of approximately 30% and 31%, respectively. These results demonstrate the ability of heuristic methods to still generate reasonably good solutions under severe time constraints, highlighting their robustness in time-critical environments.

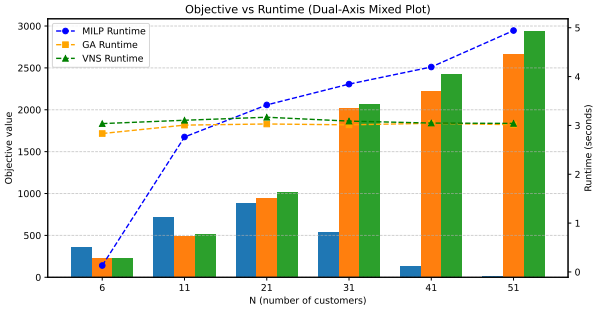


Figure 11: MILP-GA-VNS Objective value with runtime limit = 3 seconds
Source: Table 4

4.4.2 Gamma Fraction

To assess how the models behave under more or less uncertainty assumptions,we evaluate different robustness levels expressed as fraction of the number of customers, $\Gamma/N = \{0.1, 0.3, 0.5\}$. By employing a fractional robustness budget rather than an absolute value of Γ, the level of conservatism scales proportionally with the instant size and allow a meaningful comparison across different value of N. Lower fractions correspond to more optimistic environments with limited protection against uncertainty while higher fractions represent more conservative settings.

Objective value: **Figure 12** shows the evolution of the objective value across the different robustness fractions considered, the overall performance trends observed in the reference experiment remain largely unchanged. In particular, the objective values associated with each robustness fraction follow the same general pattern: for all $\Gamma/N \in \{0.1, 0.3, 0.5\}$, the MILP objective increases for small and medium-sized instances but drops once the number of customers exceeds approximately $N = 30$, whereas both GA and VNS continue to show a steady and nearly linear increase in objective value as the problem size grows. It should be noted that, for the MILP, the absence of a strict monotonic ordering of objective values with respect to Γ is primarily due to computational limitations. For larger instances, the MILP frequently reaches the imposed time limit and is not solved to proven optimality. Under exact optimality, higher robustness levels would always yield weakly lower objective values.

A comparison relative to the reference robustness level $\Gamma/N = 0.3$ further refines this observation. For GA and VNS, adopting a more optimistic setting ($\Gamma/N = 0.1$) consistently leads to higher objective values across all instance sizes. For medium and large instances, the average improvement reaches approximately 23% for GA and 12% for VNS. Increasing conservatism to $\Gamma/N = 0.5$ results in a systematic reduction in objective value for both heuristic methods, with average decreases of about 9% for GA and 7% for VNS.

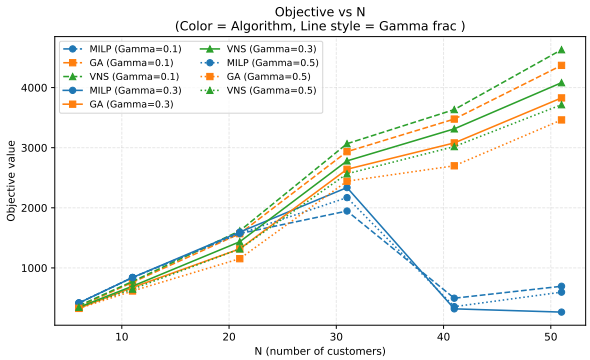


Figure 12: MILP-GA-VNS Objective value per gamma fraction
Source: Table 5

Runtime: From a computational view (**figure13**), increasing the robustness has a noticeable but limited impact on runtime. The effect is most pronounced for the MILP: higher robustness levels lead to larger optimality gaps and cause the solver to reach the imposed runtime limit already for instances of sizes $N = 20$ customers. For GA and

VNS, runtime is also affected by increasing robustness but the impact remains negligible with the execution times remaining well below the imposed time limits.

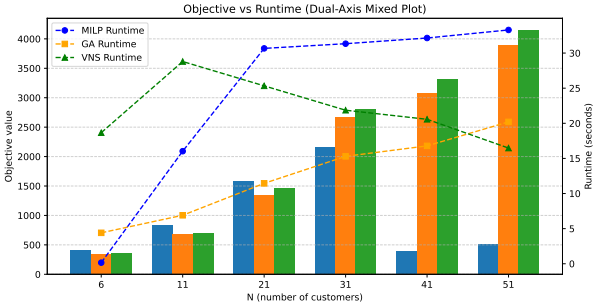


Figure 13: Average objective value and runtime for gamma fraction $\{0.1, 0.3, 0.5\}$
Source: Table 5

Robustness: The evolution of the coefficient of variation of the objective value across different robustness fractions, shown in **figure 14**, indicates that GA and VNS exhibit stable behavior. For both methods, lower robustness budgets are associated with lower objective CV values, reflecting reduced variability in more optimistic settings. As the robustness fraction increases, the CV rises slightly, but the overall trend remains consistent and the variability stays limited. Importantly, for both GA and VNS, the CV follows the same general pattern observed in previous analyses and remains low across all values of Γ/N. This indicates that their performance is not strongly affected by stochasticity induced by changes in the robustness level.

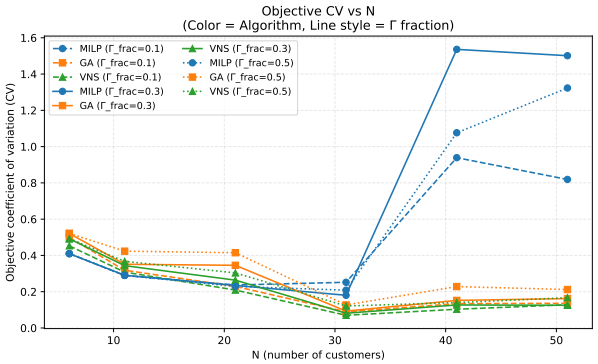


Figure 14: MILP-GA-VNS Coefficient of variation of the objective value per gamma fraction
Source: Table 5

4.4.3 Time constraint

To evaluate the behavior of the MILP, GA, VNS under time-critical window conditions, we analyze their performance when the available time window is reduced $T = \{150, 200, 250\}$. These values are chosen to represent increasingly restrictive operational settings but remaining sufficiently large to allow all methods to produce feasible solutions.

Objective value: Reducing the available time budget has a systematic and measurable impact on the performance of all three methods. As shown in **figure 15**, shorter time

windows lead to a decrease in objective value across all models. Reducing the time budget to $T = 200$ results in a relatively modest loss in objective value typically below 10%, even for larger instances, whereas a more restrictive budget of $T = 150$ leads to substantially larger losses, in the range of 20–30%. With less time available per route, feasible solutions are necessarily constrained to shorter tours and a smaller number of visited customers, which directly limits the attainable reward.

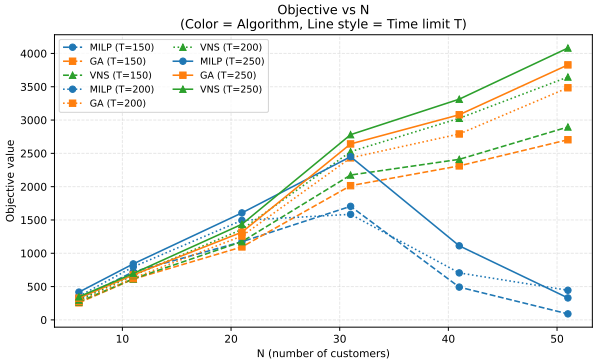


Figure 15: MILP-GA-VNS Objective value per time constraint
Source: Table 6

Runtime: The impact of reduced time budgets reflected in computational runtime is shown in **figure 16**. The runtime of the MILP remains approximately unchanged across different values of T, as the solver continues to reach its imposed computational limit regardless of the available route time. For the GA, a modest decrease in runtime of about 4% is observed for $T = 150$, while a slight increase of approximately 2% occurs for $T = 200$ when compared to the reference case $T = 250$. In contrast, the VNS benefits significantly from tighter time budgets. Its runtime decreases by approximately 35% for $T = 150$ and by about 16% for $T = 200$. This improvement is likely due to the reduced number of feasible solutions and neighborhood moves that need to be evaluated under stricter time constraints, which accelerates convergence and leads to earlier termination of the search.

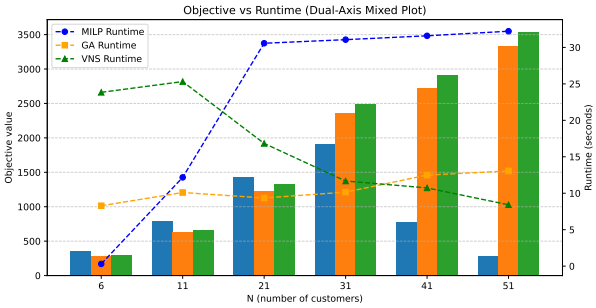


Figure 16: MILP-GA-VNS Average objective value and runtime per customers
Source: Table 6

Routes: The impact of reduced time budgets is also reflected in route structure indicators. All methods exhibit lower coverage and serve fewer customers as the time budget decreases, as shorter routes force more selective customer choices. For all algorithms, reducing the time budget from $T = 250$ to $T = 200$ or $T = 150$ leads to a

systematic decrease in average route length, with relative reductions that grow with the number of customers. The effect is particularly pronounced for the most restrictive setting ($T = 150$) where average route lengths are reduced by approximately 27% for GA and VNS and by 40% for the MILP.

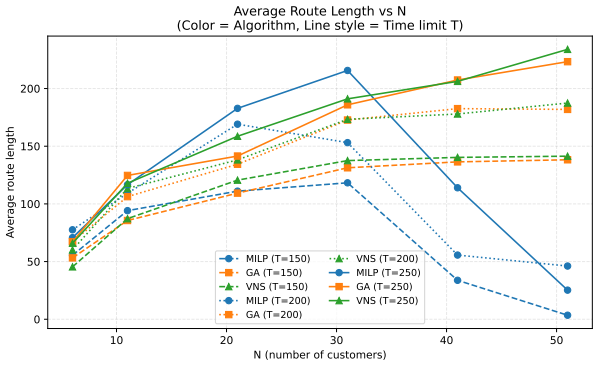


Figure 17: MILP-GA-VNS Average route length per time constraints
Source: Table 6

Robustness: As shown in **figure 18**, the coefficient of variation does not exhibit behavior different from what was previously observed. In particular, stricter time constraints are associated with higher CV values, indicating increased variability in solution quality as the available time budget decreases.

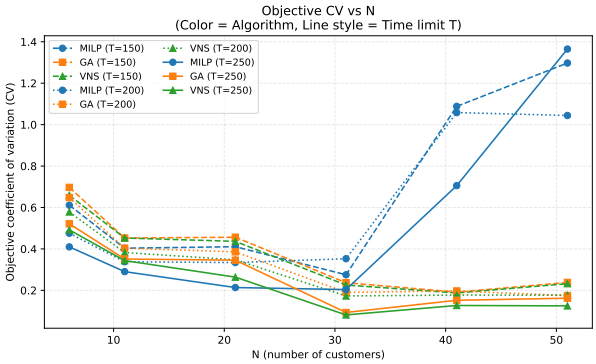


Figure 18: MILP-GA-VNS Coefficient of variation of objective value per time constraints
Source: Table 6

4.4.4 Prices

To analyze how the models react to different levels of reward randomness, we vary the range of node prices while keeping all other parameters fixed. The reference configuration uses prices sampled uniformly in the range [25;250]. Two alternative ranges are tested: a narrower and higher-value range [200;250] and a wider range [10,300], which introduces greater variability.

Objective value: **Figure 19** shows that, for all models, the achieved objective values are higher under the alternative price ranges than under the reference setting. This behavior can be explained by the differences in the expected price levels. Under uniform sampling, the theoretical mean price is approximately 137.5 for the reference

range [25;250], 225 for the high-value range [200;250], and 155 for the wide range [10;300]. As a result, instances generated with higher expected rewards naturally allow the construction of routes with larger total profit.

Across all models, the highest objective values are obtained for the [200;250] range, which reflects both the higher mean reward and the reduced variability in prices and favor the consistent selection of high-value customers. Conversely, the reference range yields the lowest objective values, as it combines a lower average reward with moderate variability. The wide range [10;300] produces intermediate results: although its expected mean price is higher than that of the reference, the increased dispersion introduces more low-value customers and limits the achievable objective compared to the concentrated high-price setting.

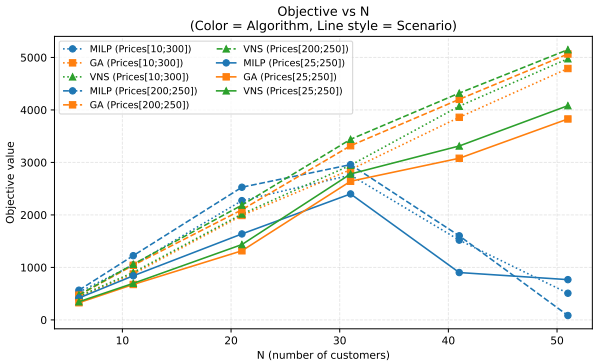


Figure 19: Objective value per prices ranges
Source: Table 2, Table 8, Table 7

Robustness: The coefficient of variation (**figure 20**) further illustrates the impact of price distributions on solution stability. As expected, the widest price range [10;300] exhibits the highest CV values across all models, reflecting the increased variability induced by the larger dispersion of customer rewards. In contrast, the concentrated high-value range [200;250] yields the lowest CV values, indicating more stable and repeatable solution quality due to the consistent availability of high-reward customers.

Despite these differences in magnitude, the overall trends observed in the reference analysis remain unchanged. For both GA and VNS, the CV decreases as the number of customers increases, demonstrating that heuristic methods remain robust to stochastic variability even under higher reward dispersion. In contrast, the MILP exhibits a sharp increase in CV as the problem size grows, particularly for larger instances.

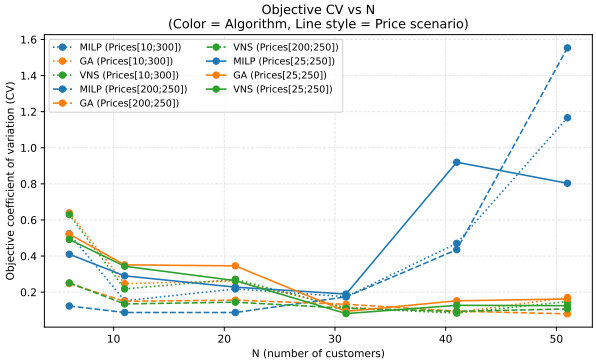


Figure 20: MILP-GA-VNS Coefficient of variation of objective value per prices ranges
Source: Table 2, Table 8, Table 7

4.4.5 Decay rate

By varying the range of decay rates, we analyze how the models behave as rewards become time-sensitive and progressively lose value, thereby simulating environments with different levels of urgency. The reference range [0.01;0.1] corresponds to a low-urgency setting in which rewards decay slowly. The range [0.1;3] represents a highly urgent environment with very fast reward decay, while the intermediate range [0.1;0.5] captures a moderately urgent scenario.

Objective value: In **figure 21** The high-decay setting produces the lowest objective values across all models, with important reductions of approximately 63% for GA and 77% for VNS relative to the reference. In this extreme regime, the problem becomes so restrictive that the MILP, GA, and VNS converge to similarly low objective levels. This indicates that the dominant limitation is no longer the quality of the search process, but the rapid loss of reward value itself. The fact that GA is less impacted than VNS suggests that its solution construction and evolutionary mechanisms are slightly more resilient under extreme urgency.

In contrast, the moderate-decay setting yields the second-highest objective values and reveals a different sensitivity ranking between the heuristic methods. In this case, VNS is less affected by the increase in decay (approximately –9%) than GA (approximately –19%). This suggests that when decay is significant but not catastrophic, the systematic intensification of VNS through local search becomes particularly effective, allowing better reordering of visits to prioritize early high-value customers and limit the loss induced by decay. GA remains competitive, but appears more sensitive to this intermediate level of urgency.

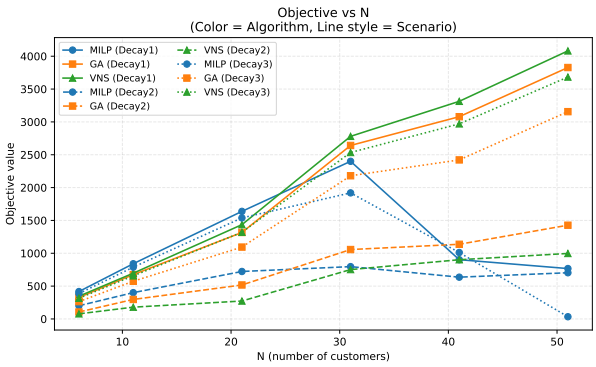


Figure 21: MILP-GA-VNS Objective value per decay rates range
Source: Table 9, Table 10

Robustness: Regarding robustness against uncertainty, the coefficient of variation shown in **figure 22** increases as the decay rate becomes higher, indicating greater sensitivity to stochastic effects in more urgent environments. In the high-decay setting, VNS exhibits the highest CV among the heuristic methods, suggesting that its performance becomes more variable when rewards decay very rapidly. This increased variability reflects the greater difficulty of consistently identifying and ordering high-value customers when urgency is extreme. In contrast, when decay rates are lower, the pattern changes: GA becomes the most affected heuristic in terms of variability, exhibiting higher CV values than VNS. This indicates that under less urgent conditions, GA's stochastic search dynamics lead to greater dispersion in solution quality across repetitions, while VNS benefits from its systematic local search to maintain more consistent performance. The MILP is excluded from this comparison, as its variability is dominated by computational effects and runtime saturation rather than by decay-related uncertainty.

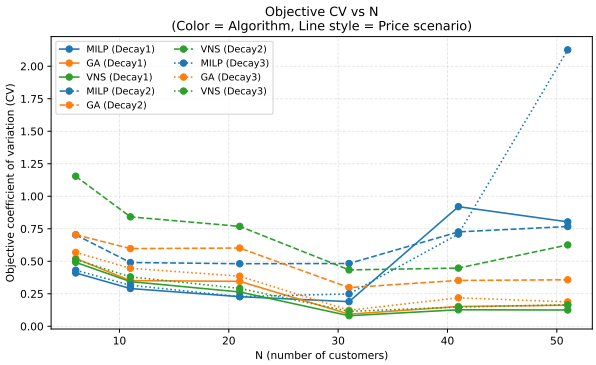


Figure 22: MILP-GA-VNS Coefficient of variation of the objective value per decay rates range

Source: Table 2 Table 9, Table 10

4.5 Conclusion of the numerical experiment

A first key conclusion concerns **scalability**. The MILP formulation performs well and provides strong optimality guarantees for small and medium-sized instances when sufficient computational time is available. In these regimes, the objective values achieved by the MILP closely match those obtained by GA and VNS, confirming both the correctness of the formulation and the effectiveness of the heuristic implementations. However, as the number of customers increases, the MILP rapidly reaches its computational limits. Runtime saturation leads to growing optimality gaps, stagnation or degradation of objective values, reduced coverage and less effective route construction. In contrast, GA and VNS exhibit significantly better scalability and continue to improve solution quality as problem size grows while maintaining low and stable runtimes.

A second major finding relates to **time-critical** settings. Reducing the operational time budget or imposing strict runtime limits strongly penalizes the MILP, whose performance deteriorates once the available computation time falls below a critical threshold. Heuristic methods, by contrast, demonstrate robust anytime behavior: they generate high-quality feasible solutions early in the search and degrade gracefully under tighter time constraints. Among the heuristics, VNS consistently achieves the highest objective values, while GA offers slightly lower solution quality with particularly stable and predictable performance.

Regarding **robustness to uncertainty**, varying the robustness budget expressed as a fraction of the number of customers does not alter the qualitative performance trends. Increasing conservatism shifts objective values downward but does not fundamentally change scalability behavior. Importantly, GA and VNS remain stable across robustness levels, whereas the MILP becomes increasingly sensitive to robustness as it accelerates the onset of computational saturation. The analysis of stochastic variability using the coefficient of variation further confirms that heuristic methods exhibit low and stable variability across repetitions, while the MILP shows high and erratic variability once time limits become binding.

The numerical experiment also highlight the influence of **instance characteristics**, such as price distributions and decay parameters. Higher average rewards and more concentrated price ranges lead to systematically higher objective values across all methods, while increased randomness primarily affects achievable profit through structural effects rather than algorithmic instability. Route structure analyses consistently show that tighter constraints reduce coverage and average route length, while improving route balance as a structural consequence of shorter routes.

In conclusion the results show a clear trade-off between exactness and practical scalability. While the MILP is well suited for small instances and settings where optimality guarantees are required, heuristic approaches offer a more robust and scalable alternative for large-scale, time-critical, or uncertainty-intensive environments. These findings strongly support the use of advanced heuristic methods for real-world robust routing problems, where solution quality, stability and computational efficiency are often more critical than exact optimality.

5 Scientific takeaways

- **Heuristic methods can reliably approximate optimal robust solutions in solvable regimes**

  In problem sizes where the MILP can be solved (or nearly solved), GA and VNS produce solutions with almost identical objective trends. This demonstrates that well-designed heuristics can closely approximate optimal robust solutions while avoiding the computational burden of exact optimization.

- **Anytime behavior is a decisive advantage in time-critical robust routing**

  GA and VNS generate high-quality feasible solutions early and degrade smoothly under tighter runtime or operational constraints. In contrast, the MILP exhibits threshold behavior: once computational time becomes insufficient, solution quality deteriorates rapidly and unpredictably.

- **Exactness–scalability trade-off is unavoidable in large-scale robust routing**

  Exact methods are best suited for small instances or offline planning with generous computational resources, whereas heuristic approaches offer a more effective balance between solution quality, robustness and computational efficiency in large-scale or real-time settings.

- **Robustness primarily shifts solution quality rather than altering scalability dynamics**

  Increasing the robustness budget reduces objective values by enforcing conservatism, but does not fundamentally change how algorithms scale with problem size. For heuristic methods, robustness is absorbed smoothly, but for the MILP, it accelerates the onset of computational saturation.

- **Stochastic stability distinguishes heuristic and exact approaches under time limits**

  The coefficient of variation reveals that GA and VNS maintain low and stable variability across repetitions, even under increased robustness and tighter time budgets. In contrast, the MILP becomes highly unstable once time-limited, with performance strongly dependent on early solver decisions.

- **Instance parameter distributions critically influence achievable performance**

  Variations in price ranges and decay parameters have a significant impact on objective values, highlighting the importance of accurate parameter specification. While robust optimization effectively mitigates uncertainty in operational parameters, it does not eliminate sensitivity to reward modeling choices. Robustness provides protection against adverse deviations around assumed values but it cannot compensate for poorly designed or too unrealistic parameters.

6 Limitation and future works

6.1 Limitations

Despite the promising results obtained in this study, several limitations must be acknowledged.

First, only Genetic Algorithm (GA) and Variable Neighborhood Search (VNS) were implemented and evaluated. While these methods provide strong performance and representative behavior, many alternative metaheuristics exist, such as tabu search or simulated annealing, which could potentially achieve better solution quality or different trade-offs between runtime and robustness. The conclusions drawn in this work are therefore limited to the specific heuristic considered.

Second, the performance of all models in this study is strongly influenced by instance parameters such as node prices, decay rates, and algorithmic settings. In practice, these quantities are often difficult to estimate accurately and may vary across operational contexts or over time. While robust optimization mitigates uncertainty in travel times, it does not address uncertainty or misspecification in reward-related parameters, meaning that poorly calibrated inputs can still lead to suboptimal solutions.
Moreover, the numerical experiments rely on synthetic instances with controlled distributions of prices and decay rates. Although this helps for a systematic and reproducible evaluation, it may not fully capture the complexity of real-world data, where rewards and decay dynamics may exhibit correlations, non-stationary behavior and domain-specific constraints. As a result, both parameter uncertainty and data simplification limit the direct generalization of the results to real operational environments.

Finally, the heuristic approaches rely on fixed parameter settings determined through preliminary experimentation. Although GA and VNS demonstrate stable and robust performance across a wide range of scenarios, their effectiveness may still depend on problem-specific tuning. Adaptive or instance-aware parameter configurations could further improve performance and robustness, but were not explored in this study.

6.2 Future research directions

A first direction concerns learning-based parameters estimation and tuning. Machine learning techniques could be employed to infer prices, decay rates or algorithmic parameters directly from historical data. For example, reinforcement learning or Bayesian optimization could be used to adapt heuristic parameters dynamically to reduce reliance on manual tuning. Recent advances at the intersection of machine learning and metaheuristics have demonstrated the effectiveness of such approaches in combinatorial optimization, showing that data-driven parameter control can significantly enhance both solution quality and algorithmic stability (Karimi Mamaghan et al. (2021)).

A second direction involves adaptive and data-driven robustness. Rather than fixing the robustness budget Γ in advance, future models could adjust the level of conservatism dynamically based on observed variability, confidence in parameter estimates or real time information. Such adaptive robustness implementation could provide a better

balance between protection and performance and avoid unnecessary conservatism in understood environments while increasing robustness under high uncertainty. This perspective aligns with research in data-driven robust optimization, where uncertainty sets and robustness parameters are learned from data rather than specified ex ante and allow models to adapt their level of protection as information evolves (Bertsimas et al. (2013)). Similarly, Esfahani and Kuhn (2015) introduce a distributionally robust framework based on the Wasserstein metric, which provides probabilistic performance guarantees while accounting for ambiguity in the underlying data-generating process.

Third direction to complement this work is the integration of hybrid optimization strategies by combining exact methods and heuristics, for instance using the MILP to guide heuristic initialization or employing decomposition technique. Doing this could help leverage the strength of both approaches and improve solution quality while preserving scalability. Maniezzo et al. (2010) demonstrate that metaheuristic frameworks can achieve superior solution quality while preserving scalability in complex combinatorial optimization problems. Integrating such strategies into the robust routing framework considered in this work represents a promising opportunity for future research.

Finally, extending this work to real-world data and dynamic scenarios represents an important step forward in assessing the practical capabilities of the proposed methods. Incorporating real operational data, time-dependent information and dynamically evolving customer sets would allow a more realistic evaluation of algorithm performance and robustness, and would help bridge the gap between controlled experimental settings and real-world deployment.

7 Conclusion

This work investigated a robust routing framework for time-sensitive disaster response operations, addressing the inherent trade-off between solution quality, robustness and computational scalability. By combining a robust MILP formulation with tailored heuristic solution approaches, this thesis demonstrates that high-quality and stable solutions can be obtained even under significant uncertainty and tight computational constraints.

The numerical results highlight that while exact methods remain valuable for small-scale or offline planning, heuristic approaches such as GA and VNS provide a more suitable balance for large-scale and time-critical applications. Robust optimization was shown to improve solution reliability without fundamentally altering algorithmic scalability, reinforcing the relevance of heuristic methods in uncertain operational environments.

In conclusion, this work contributes to bridging robust optimization and metaheuristic design in disaster response routing and provides a solid foundation for future research on adaptive, data-driven and hybrid optimization strategies aimed at real-world deployment.

A Appendix

A.1 Implementation

Models implementation

In [39]:
```python
from __future__ import annotations
from typing import List, Dict, Tuple, Set, Iterable, Optional
import math
import random
import os
import time
import numpy as np

try:
    import matplotlib.pyplot as plt
except Exception:
    plt = None

try:
    import pandas as pd
except Exception:
    pd = None

from docplex.mp.model import Model

# ======= Types =======
CityDict = Dict[int, Tuple[float, float]]
PriceDict = Dict[int, float]
DecayDict = Dict[int, float]
Route = List[int]
Gene = Dict[int, Route]


# ============================================================
#   SHARED GEOMETRY / GENE UTILITIES
# ============================================================

def euclid(a: Tuple[float, float], b: Tuple[float, float]) -> float:
    return round(((a[0] - b[0])**2 + (a[1] - b[1])**2) ** 0.5, 4)

def build_distances(cities: CityDict) -> Dict[Tuple[int, int], float]:
    keys = list(cities.keys())
    return {(i, j): (0.0 if i == j else euclid(cities[i], cities[j])) for i in keys for j in keys}

def new_empty_gene(K: int, depot: int = 0) -> Gene:
    return {k: [depot, depot] for k in range(K)}

def canonicalize(g: Gene) -> Gene:
    """Remove duplicates in a row, ensure routes start/end at depot."""
    depot = 0
    hh = {}
    for k, r in g.items():
        if len(r) < 2:
            hh[k] = [depot, depot]
            continue
        clean = [r[0]]
        for x in r[1:]:
            if x != clean[-1]:
                clean.append(x)
        if clean[0] != depot:
            clean = [depot] + clean
        if clean[-1] != depot:
            clean = clean + [depot]
        if len(clean) < 2:
            clean = [depot, depot]
        hh[k] = clean
    return hh

def plot_gene_routes(
    g: Gene,
    cities: CityDict,
    distances: Dict[Tuple[int,int], float],
    ax,
```

Models implementation

```
    title: str = ""
):
    """Plot a Gene (set of routes) on the given Matplotlib axis."""
    if ax is None or plt is None:
        return

    g = canonicalize(g)
    city_list = list(cities.keys())

    # scatter cities
    for i in city_list:
        xi, yi = cities[i]
        ax.scatter(xi, yi, color='black', s=25)
        ax.text(xi + 1, yi - 2, f"{i}", fontsize=8)

    # colors for vehicles
    colors = ['blue', 'green', 'red', 'orange', 'purple', 'brown']

    for k, route in g.items():
        color = colors[k % len(colors)]
        for i, j in zip(route[:-1], route[1:]):
            xi, yi = cities[i]
            xj, yj = cities[j]
            ax.plot([xi, xj], [yi, yj], color=color, linewidth=2.0)
            # optional: show distance on edge
            mx, my = (xi + xj) / 2, (yi + yj) / 2
            ax.text(mx, my, f"{distances[(i, j)]:.1f}", fontsize=7, color=color)

    ax.set_title(title)
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.grid(True)

def route_length(route: Route, distances: Dict[Tuple[int,int], float]) -> float:
    if len(route) < 2:
        return 0.0
    total = 0.0
    for i in range(len(route)-1):
        total += distances[(route[i], route[i+1])]
    return total

def gene_travel_cost(g: Gene, distances: Dict[Tuple[int,int], float]) -> float:
    return sum(route_length(r, distances) for r in g.values())

def compute_arrivals(g: Gene, distances: Dict[Tuple[int,int], float]) -> Dict[int, float]:
    """Arrival time t_i (cumulative length) at each visited customer."""
    a: Dict[int, float] = {}
    for r in g.values():
        t = 0.0
        for i in range(len(r)-1):
            u, v = r[i], r[i+1]
            t += distances[(u, v)]
            if v != 0 and v not in a:
                a[v] = t
    return a

def feasible(g: Gene, T: float, distances: Dict[Tuple[int,int], float], depot: int=0) -> bool:
    # routes start/end at depot and respect per-route budget
    for r in g.values():
        if not r or r[0] != depot or r[-1] != depot:
            return False
        if route_length(r, distances) > T:
            return False
    # no duplicate customers
    seen = set()
    for r in g.values():
        for u in r[1:-1]:
            if u in seen:
                return False
            seen.add(u)
```

Models implementation

```
    return True

def all_visited(g: Gene, depot: int=0) -> Set[int]:
    S: Set[int] = set()
    for r in g.values():
        for u in r[1:-1]:
            if u != depot:
                S.add(u)
    return S

def route_insert(route: Route, u: int, idx: int) -> Route:
    return route[:idx] + [u] + route[idx:]

def remove_customer(route: Route, u: int) -> Route:
    r = route[:]
    r.remove(u)
    return r

def random_gene(customers: List[int], K: int, rng: random.Random, depot: int=0) -> Gene:
    g: Gene = {k: [depot, depot] for k in range(K)}
    order = customers[:]
    rng.shuffle(order)
    for u in order:
        k = rng.randrange(K)
        r = g[k]
        idx = rng.randrange(1, len(r))
        g[k] = route_insert(r, u, idx)
    return canonicalize(g)

def compute_solution_metrics(
    g: Gene,
    cities: CityDict,
    prices: PriceDict,
    decay: DecayDict,
    T: float,
    Gamma: int,
    distances: Dict[Tuple[int,int], float],
    high_priority_frac: float = 0.2,
    attack_mode: str = "time",
    delta: float = 0.25,
    beta: float = 0.25,
    travel_cost_weight: float = 0.0,
) -> Dict[str, float]:
    """
    Compute per-solution metrics for Step 6.
    """
    g = canonicalize(g)
    arrivals = compute_arrivals(g, distances)
    visited = set(arrivals.keys())
    non_depot = [i for i in cities if i != 0]

    num_visited=len(visited)
    num_non_depot=len(non_depot)

    # --- 1) Total collected reward (nominal, no adversary) ---
    CP = {}
    for i in visited:
        t_i = arrivals[i]
        CP[i] = max(prices[i] - decay[i] * t_i, 0.0)
    total_reward = sum(CP.values())

    # --- 2) High-priority zones served ---
    # Define high-priority as top q% by initial price
    sorted_by_price = sorted(
        [i for i in non_depot if i in prices],
        key=lambda i: prices[i],
        reverse=True
    )
    cutoff = max(1, int(high_priority_frac * len(sorted_by_price)))
    high_priority_set = set(sorted_by_price[:cutoff])
```

Models implementation

```
    high_priority_served = len(visited & high_priority_set)
    hp_coverage = high_priority_served / len(high_priority_set)

    # --- 3) Coverage + fairness between vehicles ---
    coverage_ratio = len(visited) / max(1, len(non_depot))

    # reward per vehicle
    rewards_per_vehicle = []
    for k, r in g.items():
        r_reward = 0.0
        t = 0.0
        for i, j in zip(r[:-1], r[1:]):
            t += distances[(i, j)]
            if j in prices:
                r_reward += max(prices[j] - decay[j]*t, 0.0)
        rewards_per_vehicle.append(r_reward)

    if len(rewards_per_vehicle) > 0:
        mean_reward = float(np.mean(rewards_per_vehicle))
        std_reward  = float(np.std(rewards_per_vehicle))
        fairness = 1.0 - (std_reward / (mean_reward + 1e-9))  # 1 = perfectly balanced
    else:
        fairness = 0.0

    # --- 4) Mission duration ---
    route_lengths = [route_length(r, distances) for r in g.values()]
    avg_route_length = float(np.mean(route_lengths)) if route_lengths else 0.0
    max_route_length = float(np.max(route_lengths)) if route_lengths else 0.0

    # --- 5) Robust penalty & objective (time attack, consistent with VNS) ---
    fit = robust_objective_respecting_instruction(
        g, prices, decay, distances, T, Gamma,
        travel_cost_weight=travel_cost_weight,
        attack_mode=attack_mode,
        delta=delta,
        beta=beta,
    )
    travel_cost = gene_travel_cost(g, distances)

    # Approximate penalty from difference between nominal reward and robust objective
    penalty = total_reward - (fit + travel_cost_weight * travel_cost)

    return {
        "total_reward": total_reward,
        "high_priority_served": high_priority_served,
        "high_priority_coverage": hp_coverage,
        "coverage_ratio": coverage_ratio,
        "fairness": fairness,
        "avg_route_length": avg_route_length,
        "max_route_length": max_route_length,
        "robust_obj": fit,
        "robust_penalty": penalty,
        "travel_cost": travel_cost,
        "num_visited": num_visited,
        "num_non_depot": num_non_depot,
    }

# ============================================================
#   GA: ROBUST VRP
# ============================================================

def robust_objective_ga(
    g: Gene,
    prices: PriceDict,
    decay: DecayDict,
    distances: Dict[Tuple[int,int], float],
    T: float,
    Gamma: int,
    delta: float,
    travel_cost_weight: float = 1.0,
```

Models implementation

```
):
    g = canonicalize(g)
    arrivals = compute_arrivals(g, distances)
    visited = set(arrivals.keys())
    CP = {}
    CP_att = {}
    for i in visited:
        a_i = arrivals[i]
        cp = max(prices[i] - a_i * decay[i], 0.0)
        cp_att = max(prices[i] - a_i * (decay[i] + delta), 0.0)
        CP[i] = cp
        CP_att[i] = cp_att
    orig_total = sum(CP.values())
    losses = [(i, CP[i] - CP_att[i]) for i in visited]
    losses.sort(key=lambda t: t[1], reverse=True)
    top = losses[:Gamma] if Gamma > 0 else []
    penalty = sum(v for _, v in top)
    travel_cost = gene_travel_cost(g, distances)
    fitness = orig_total - penalty - travel_cost_weight * travel_cost
    return fitness, CP, CP_att, penalty, travel_cost, top, orig_total

def mutate_insert(g: Gene, customers: List[int], rng: random.Random, depot: int=0) -> Gene:
    g = {k: r[:] for k, r in g.items()}
    visited = list(all_visited(g, depot))
    if not visited:
        return g
    u = rng.choice(visited)
    for k, r in g.items():
        if u in r[1:-1]:
            g[k] = remove_customer(r, u)
            break
    k2 = rng.randrange(len(g))
    r2 = g[k2]
    idx = rng.randrange(1, len(r2))
    g[k2] = route_insert(r2, u, idx)
    return canonicalize(g)

def mutate_insert_unvisited(
    g: Gene,
    customers: List[int],
    rng: random.Random,
    depot: int = 0
) -> Gene:
    g = {k: r[:] for k, r in g.items()}
    visited = all_visited(g, depot)
    unvisited = [c for c in customers if c not in visited]
    if not unvisited:
        return g

    u = rng.choice(unvisited)
    k = rng.randrange(len(g))
    r = g[k]
    # insert before the last depot
    idx = 1 if len(r) <= 2 else rng.randrange(1, len(r))
    g[k] = route_insert(r, u, idx)
    return canonicalize(g)


def mutate_swap_within(g: Gene, rng: random.Random, depot: int=0) -> Gene:
    g = {k: r[:] for k, r in g.items()}
    for k, r in g.items():
        if len(r) > 3:
            i = rng.randrange(1, len(r)-1)
            j = rng.randrange(1, len(r)-1)
            if i != j:
                r[i], r[j] = r[j], r[i]
                g[k] = r
                return canonicalize(g)
    return g
```

Models implementation

```
def mutate_move_between(g: Gene, rng: random.Random, depot: int=0) -> Gene:
    g = {k: r[:] for k, r in g.items()}
    if len(g) < 2:
        return g
    ks = list(g.keys())
    k_from, k_to = rng.sample(ks, 2)
    r_from = g[k_from]
    r_to = g[k_to]
    if len(r_from) <= 2:
        return g
    i = rng.randrange(1, len(r_from)-1)
    u = r_from[i]
    del r_from[i]
    j = rng.randrange(1, len(r_to))
    r_to.insert(j, u)
    g[k_from] = r_from
    g[k_to] = r_to
    return canonicalize(g)

def mutate(g: Gene, customers: List[int], rng: random.Random,
           distances=None, T=None, feasible_only=False, depot: int = 0) -> Gene:
    op = rng.random()
    if op < 0.25:
        child = mutate_insert_unvisited(g, customers, rng)
    elif op < 0.50:
        child = mutate_insert(g, customers, rng)
    elif op < 0.75:
        child = mutate_swap_within(g, rng)
    else:
        child = mutate_move_between(g, rng)

    if feasible_only and distances is not None and T is not None:
        if not feasible(child, T, distances, depot):
            return g

    return child

def plot_convergence_ga(trace, ax=None, title="Convergence"):
    """trace is a list of (generation, best_value)."""
    it = [t["iter"] for t in trace]
    best = [t["best"] for t in trace]
    mean = [t["mean"] for t in trace]
    med  = [t["median"] for t in trace]
    if ax is None and plt is not None:
        fig, ax = plt.subplots(1,1, figsize=(8,5))
        ax.plot(it, best, label="best", linewidth=2)
    if ax is not None:
        ax.plot(it, mean, label="mean")
        ax.plot(it, med,  label="median")
        ax.plot(it, best, label="best")
        ax.set_xlabel("Iteration")
        ax.set_ylabel("Objective")
        ax.set_title(title)
        ax.grid(True)
        ax.legend()
    return ax

def genetic_algorithm_robust_vrp(
    cities: CityDict,
    prices: PriceDict,
    decay: DecayDict,
    K: int = 3,
    T: float = 400.0,
    Gamma: int = 3,
    delta: float = 0.25,
    pop_size: int = 60,
    generations: int = 200,
    mutation_rate: float = 0.35,
    travel_cost_weight: float = 1.0,
    rng_seed: int = 42,
```

Models implementation

```
    verbose: bool = True,
    feasible_only: bool = False,
    plot_interval: int = 0,
    time_limit_sec: float | None=None,
):
    rng = random.Random(rng_seed)
    depot = 0
    distances = build_distances(cities)
    customers = [i for i in cities if i != depot]

    start_time= time.perf_counter()

    population: List[Gene] = [new_empty_gene(K, depot)] + [
        random_gene(customers, K, rng, depot) for _ in range(pop_size-1)
    ]

    def fitness(g):
        val, *_ = robust_objective_ga(
            g, prices, decay, distances, T, Gamma, delta, travel_cost_weight
        )
        if feasible_only:
            return val if feasible(g, T, distances, depot) else float('-inf')
        return val

    scored = [(fitness(g), g) for g in population]
    scored.sort(key=lambda t: t[0], reverse=True)
    trace = []

    for gen in range(generations+1):

        elapsed = time.perf_counter() - start_time
        if time_limit_sec is not None and elapsed >= time_limit_sec:
            if verbose:
                print(f"[GA] Time limit reached ({elapsed:.1f}s) at generation {gen}; stopping.")
            break

        vals = [s[0] for s in scored if s[0] != float('-inf')]
        if vals:
            trace.append({
                "iter": gen,
                "best": np.max(vals),
                "mean": np.mean(vals),
                "median": np.median(vals)
            })
        else:
            trace.append({"iter": gen, "best": -np.inf, "mean": -np.inf, "median": -np.inf})

        best_val, best_gene = scored[0]

        if verbose and gen % max(1, generations//10) == 0:
            mean_val = float(np.mean(vals)) if vals else float('-inf')
            print(f"[GA] Gen {gen:4d} | best={best_val:.3f} | mean={np.mean(vals):.3f}")

        if plot_interval > 0 and plt is not None and gen % plot_interval == 0 and gen > 0:
            # optional plotting (disabled in experiments by setting plot_interval=0)
            pass

        if gen == generations:
            break

        # next generation
        new_pop = []
        elites = max(1, pop_size // 20)
        new_pop.extend([g for _, g in scored[:elites]])

        while len(new_pop) < pop_size:
            a, b = rng.sample(scored[:max(5, len(scored))], k=2)
            parent = a[1] if a[0] >= b[0] else b[1]
            child = parent
            if rng.random() < mutation_rate:
```

Models implementation

```
                child = mutate(child, customers, rng)
            new_pop.append(child)

        scored = [(fitness(g), g) for g in new_pop]
        scored.sort(key=lambda t: t[0], reverse=True)

    best_val, best_gene = scored[0]
    fit, CP, CP_att, penalty, travel_cost, top,orig_total = robust_objective_ga(
        best_gene, prices, decay, build_distances(cities), T, Gamma, delta, travel_cost_weight
    )
    return {
        "best_gene": canonicalize(best_gene),
        "best_value": fit,
        "penalty": penalty,
        "travel_cost": travel_cost,
        "top_adversarial_losses": top,
        "profit": orig_total,
        "trace": trace,
    }


# ============================================================
#   VNS: ROBUST VRP
# ============================================================

def robust_objective_vns_full(
    g: Gene,
    prices: PriceDict,
    decay: DecayDict,
    distances: Dict[Tuple[int,int], float],
    T: float,
    Gamma: int,
    travel_cost_weight: float = 1.0,
    attack_mode: str = "rate",
    delta: float = 0.25,
    beta: float = 0.25
):
    """
    Returns:
        fitness, orig_total_profit, penalty, travel_cost
    """
    g = canonicalize(g)
    arrivals = compute_arrivals(g, distances)
    visited = set(arrivals.keys())

    CP, CP_att = {}, {}
    for i in visited:
        t_i = arrivals[i]
        a_i = decay[i]
        p0  = prices[i]
        p_cur = max(p0 - a_i * t_i, 0.0)
        if attack_mode == "time":
            p_att = max(p0 - a_i * ((1.0 + beta) * t_i), 0.0)
        else:
            p_att = max(p0 - (a_i + delta) * t_i, 0.0)
        CP[i] = p_cur
        CP_att[i] = p_att

    gaps = {i: max(CP[i] - CP_att[i], 0.0) for i in visited}
    top = sorted(gaps.items(), key=lambda kv: kv[1], reverse=True)[:max(0, Gamma)]
    penalty = sum(v for _, v in top)

    orig_total = sum(CP.values())
    travel_cost = gene_travel_cost(g, distances)
    fitness = orig_total - penalty - travel_cost_weight * travel_cost
    return fitness, orig_total, penalty, travel_cost

def robust_objective_respecting_instruction(
    g: Gene,
    prices: PriceDict,
```

Models implementation

```
    decay: DecayDict,
    distances: Dict[Tuple[int,int], float],
    T: float,
    Gamma: int,
    travel_cost_weight: float = 1.0,
    attack_mode: str = "rate",
    delta: float = 0.25,
    beta: float = 0.25
):
    # keep old signature/behaviour for the local search
    fitness, _, _, _ = robust_objective_vns_full(
        g, prices, decay, distances, T, Gamma,
        travel_cost_weight=travel_cost_weight,
        attack_mode=attack_mode,
        delta=delta,
        beta=beta
    )
    return fitness

def apply_reloc(g: Gene, u: int, k_from: int, i_from: int, k_to: int, j_to: int) -> Gene:
    h = {k: r[:] for k, r in g.items()}
    del h[k_from][i_from]
    if j_to > len(h[k_to]):
        j_to = len(h[k_to])
    h[k_to].insert(j_to, u)
    return canonicalize(h)

def apply_swap(g: Gene, k1: int, i1: int, k2: int, i2: int) -> Gene:
    h = {k: r[:] for k, r in g.items()}
    h[k1][i1], h[k2][i2] = h[k2][i2], h[k1][i1]
    return canonicalize(h)

def apply_2opt_intra(g: Gene, k: int, i: int, j: int) -> Gene:
    """Reverse subsegment (i..j) within route k; i<j; exclude depots."""
    h = {kk: rr[:] for kk, rr in g.items()}
    r = h[k]
    if not (1 <= i < j <= len(r)-2):
        return g
    r[i:j+1] = reversed(r[i:j+1])
    h[k] = r
    return canonicalize(h)

def locate_all(g: Gene) -> Dict[int, Tuple[int,int]]:
    """Map customer -> (vehicle, index in that route)."""
    pos = {}
    for k, r in g.items():
        for idx in range(1, len(r)-1):
            pos[r[idx]] = (k, idx)
    return pos

def enumerate_remove_neighbors(g, rng, max_samples=200):
    out = []
    for k, r in g.items():
        for idx in range(1, len(r)-1):
            u = r[idx]
            h = {kk: rr[:] for kk, rr in g.items()}
            del h[k][idx]
            if len(h[k]) < 2:
                h[k] = [0,0]
            out.append(canonicalize(h))
            if len(out) >= max_samples:
                return out
    return out

def enumerate_reloc_neighbors(
    g: Gene, rng: random.Random, K: int, max_samples: int = 200
) -> Iterable[Gene]:
    """Sample up to max_samples Relocate(1) neighbors."""
    pos = locate_all(g)
    if not pos:
```

Models implementation

```
        return []
    customers = list(pos.keys())
    out = []
    for _ in range(max_samples):
        u = rng.choice(customers)
        k_from, i_from = pos[u]
        k_to = rng.randrange(K)
        r_len = len(g[k_to])
        j_to = 1 if r_len <= 1 else rng.randrange(1, r_len)
        if (k_to == k_from) and (j_to == i_from):
            continue
        out.append(apply_reloc(g, u, k_from, i_from, k_to, j_to))
    return out

def enumerate_swap_neighbors(
    g: Gene, rng: random.Random, max_samples: int = 200
) -> Iterable[Gene]:
    """Sample up to max_samples Swap(1,1) neighbors."""
    pos = locate_all(g)
    cust = list(pos.keys())
    if len(cust) < 2:
        return []
    out = []
    for _ in range(max_samples):
        u, v = rng.sample(cust, 2)
        k1, i1 = pos[u]
        k2, i2 = pos[v]
        if k1 == k2 and i1 == i2:
            continue
        out.append(apply_swap(g, k1, i1, k2, i2))
    return out

def enumerate_2opt_neighbors(
    g: Gene, rng: random.Random, max_samples: int = 200
) -> Iterable[Gene]:
    """Sample up to max_samples intra-route 2-opt neighbors."""
    out = []
    ks = list(g.keys())
    if not ks:
        return []
    for _ in range(max_samples):
        k = rng.choice(ks)
        r = g[k]
        if len(r) <= 3:
            continue
        i = rng.randrange(1, len(r)-2)
        j = rng.randrange(i+1, len(r)-1)
        out.append(apply_2opt_intra(g, k, i, j))
    return out

def enumerate_insert_unvisited_neighbors(
    g: Gene,
    cities: CityDict,
    distances: Dict[Tuple[int,int], float],
    T: float,
    rng: random.Random,
    max_samples: int = 200,
    depot: int = 0
) -> Iterable[Gene]:
    visited = all_visited(g, depot)
    all_customers = [i for i in cities if i != depot]
    candidates = [i for i in all_customers if i not in visited]
    if not candidates:
        return []
    out = []
    K = len(g)
    for _ in range(max_samples):
        u = rng.choice(candidates)
        k = rng.randrange(K)
```

Models implementation

```
        base = g[k][:]
        if len(base) < 2:
            base = [depot, depot]

        if len(base) == 2:
            j = 1
        else:
            j = rng.randrange(1, len(base))

        h = {kk: rr[:] for kk, rr in g.items()}
        h[k] = route_insert(base, u, j)
        h = canonicalize(h)
        if feasible(h, T, distances, depot):
            out.append(h)
            if len(out) >= max_samples:
                break
    return out

def shake(
    g: Gene,
    k: int,
    rng: random.Random,
    K: int,
    cities: CityDict,
    distances: Dict[Tuple[int,int], float],
    T: float
) -> Gene:
    """Apply k random simple moves (insert-unvisited / relocate / swap) to diversify."""
    h = {kk: rr[:] for kk, rr in g.items()}
    for _ in range(k):
        pos = locate_all(h)

        if rng.random() < 0.33:
            ins = list(enumerate_insert_unvisited_neighbors(
                h, cities, distances, T, rng, max_samples=1
            ))
            if ins:
                h = ins[0]
                continue

        if pos and rng.random() < 0.5:
            u = rng.choice(list(pos.keys()))
            k_from, i_from = pos[u]
            k_to = rng.randrange(K)
            r_len = len(h[k_to])
            j_to = 1 if r_len <= 1 else rng.randrange(1, r_len)
            if not (k_to == k_from and j_to == i_from):
                h = apply_reloc(h, u, k_from, i_from, k_to, j_to)
        else:
            cust = list(pos.keys())
            if len(cust) >= 2:
                u, v = rng.sample(cust, 2)
                k1, i1 = pos[u]
                k2, i2 = pos[v]
                h = apply_swap(h, k1, i1, k2, i2)

    return canonicalize(h)

def local_search_descent(
    g: Gene,
    fitness_fn,
    rng: random.Random,
    K: int,
    cities: CityDict,
    distances: Dict[Tuple[int,int], float],
    T: float,
    samples: Dict[str, int] = None,
    feasible_only: bool = True
) -> Gene:
    if samples is None:
```

Models implementation

```
        samples = {"insert":150, "remove": 200, "reloc": 200, "twoopt": 200, "swap": 200}

    s = g
    s_val = fitness_fn(s)

    improved = True
    while improved:
        improved = False
        best_neighbor: Optional[Gene] = None
        best_val = s_val

        for n in enumerate_remove_neighbors(s, rng, max_samples=samples["remove"]):
            v = fitness_fn(n)
            if v > best_val:
                best_val, best_neighbor = v, n

        for n in enumerate_insert_unvisited_neighbors(
            s, cities, distances, T, rng, max_samples=samples["insert"]
        ):
            v = fitness_fn(n)
            if v > best_val:
                best_val, best_neighbor = v, n

        for n in enumerate_reloc_neighbors(s, rng, K, max_samples=samples["reloc"]):
            v = fitness_fn(n)
            if v > best_val:
                best_val, best_neighbor = v, n

        for n in enumerate_2opt_neighbors(s, rng, max_samples=samples["twoopt"]):
            v = fitness_fn(n)
            if v > best_val:
                best_val, best_neighbor = v, n

        for n in enumerate_swap_neighbors(s, rng, max_samples=samples["swap"]):
            v = fitness_fn(n)
            if v > best_val:
                best_val, best_neighbor = v, n

        if best_neighbor is not None:
            s, s_val = best_neighbor, best_val
            improved = True

    return s

def ruin_and_recreate(
    g: Gene,
    rng: random.Random,
    q: int,
    cities: CityDict,
    distances: Dict[Tuple[int,int], float],
    T: float,
    depot: int = 0
) -> Gene:
    visited = list(all_visited(g, depot))
    if not visited:
        return g
    q = min(q, len(visited))
    remove_set = set(rng.sample(visited, q))

    h = {k: r[:] for k, r in g.items()}
    for k, r in h.items():
        h[k] = [u for u in r if (u == depot or u not in remove_set)]
        if len(h[k]) < 2:
            h[k] = [depot, depot]
    h = canonicalize(h)

    for u in remove_set:
        best = None
        best_delta = float('inf')
        for k, r in h.items():
```

Models implementation

```
            if len(r) < 2:
                r = [depot, depot]
            for j in range(1, len(r)):
                a, b = r[j-1], r[j]
                add = distances[(a,u)] + distances[(u,b)] - distances[(a,b)]
                if add < best_delta:
                    trial = {kk: rr[:] for kk, rr in h.items()}
                    trial[k] = route_insert(r, u, j)
                    trial = canonicalize(trial)
                    if feasible(trial, T, distances, depot):
                        best = (k, j)
                        best_delta = add
        if best:
            k, j = best
            h[k] = route_insert(h[k], u, j)
            h = canonicalize(h)
    return h

def vns_vrp(
    cities: CityDict,
    prices: PriceDict,
    decay: DecayDict,
    K: int = 3,
    T: float = 1000.0,
    Gamma: int = 0,
    travel_cost_weight: float = 1.0,
    attack_mode: str = "time",
    delta: float = 0.25,
    beta: float = 0.25,
    kmax: int = 4,
    iterations: int = 250,
    rng_seed: int = 43,
    feasible_only: bool = True,
    verbose: bool = True,
    observer=None,
    ls_samples: Dict[str, int] = None,
    patience: int = 60,
    kick_size: int = 5,
    time_limit_sec: float | None=None,
):
    rng = random.Random(rng_seed)
    depot = 0
    distances = build_distances(cities)
    customers = [i for i in cities if i != depot]

    start_time = time.perf_counter()

    def fitness(g: Gene) -> float:
        if feasible_only and not feasible(g, T, distances, depot):
            return float("-inf")
        return robust_objective_respecting_instruction(
            g, prices, decay, distances, T, Gamma,
            travel_cost_weight=travel_cost_weight,
            attack_mode=attack_mode, delta=delta, beta=beta
        )

    curr = random_gene(customers, K, rng, depot)
    if feasible_only and not feasible(curr, T, distances, depot):
        cur = {k: r[:] for k, r in curr.items()}
        while not feasible(cur, T, distances, depot):
            all_pos = [(k, idx) for k, r in cur.items() for idx in range(1, len(r)-1)]
            if not all_pos:
                break
            k, idx = rng.choice(all_pos)
            del cur[k][idx]
            if len(cur[k]) < 2:
                cur[k] = [depot, depot]
            cur = canonicalize(cur)
        curr = canonicalize(cur)
```

```
    best = curr
    best_val = fitness(best)
    trace = [{"iter": 0, "best": best_val}]
    since_best = 0

    if verbose:
        print(f"[VNS] init best = {best_val:.3f}")

    for it in range(1, iterations + 1):
        elapsed = time.perf_counter() - start_time
        if time_limit_sec is not None and elapsed >= time_limit_sec:
            if verbose:
                print(f"[VNS] Time limit reached ({elapsed:.1f}s) at iteration {it}; stopping.")
            break

        k = 1
        improved_this_outer = False

        while k <= kmax:
            s_shake = shake(best, k, rng, K, cities, distances, T)
            s_local = local_search_descent(
                s_shake, fitness_fn=fitness, rng=rng, K=K,
                cities=cities, distances=distances, T=T,
                samples=ls_samples or {"insert":150, "remove": 200, "reloc": 200, "twoopt": 200, "swap": 200},
                feasible_only=feasible_only
            )
            v_local = fitness(s_local)

            if v_local > best_val:
                best, best_val = s_local, v_local
                k = 1
                improved_this_outer = True
                since_best = 0
                if verbose:
                    print(f"[VNS] it={it:4d} improved → {best_val:.3f} (reset k=1)")
            else:
                k += 1

        if not improved_this_outer:
            since_best += 1

        if since_best >= patience:
            kicked = ruin_and_recreate(best, rng, q=kick_size, cities=cities, distances=distances, T=T)
            if feasible(kicked, T, distances):
                best = kicked
                best_val = fitness(best)
                if verbose:
                    print(f"[VNS] it={it:4d} stagnation kick (q={kick_size}) → best={best_val:.3f}")
            since_best = 0

        trace.append({"iter": it, "best": best_val})

        if observer:
            observer(it, best, best_val, trace)

    fitness_best, profit_best, penalty_best, travel_cost_best = robust_objective_vns_full (
        best,
        prices,
        decay,
        distances,
        T,
        Gamma,
        travel_cost_weight = travel_cost_weight,
        attack_mode = attack_mode,
        delta = delta,
        beta= beta
    )

    return {
        "best_gene": canonicalize(best),
```

Models implementation

```
        "best_value": best_val,
        "profit": profit_best,
        "travel_cost": travel_cost_best,
        "penalty": penalty_best,
        "trace": trace
    }


# ============================================================
#   MILP: ROBUST VRP WITH CP/∂/λ (DOCPLEX)
# ============================================================

def solve_vrp_with_CP_theta(
    cities: CityDict,
    prices: PriceDict,
    decay_rates: DecayDict,
    K: int = 3,
    T: float = 400.0,
    Gamma: int = 9,
    with_balancing: bool = False,
    ax=None,
    title: str = "VRP with CP/theta",
    delta_factor: float= 0.5,
    gamma_scale: float= 0.5,
    min_coverage_frac: float = 0.7
):
    num_cities = len(cities)
    city_list = list(cities.keys())
    non_depot = [i for i in city_list if i != 0]

    def distance(i, j):
        xi, yi = cities[i]
        xj, yj = cities[j]
        return round(math.hypot(xi - xj, yi - yj), 2)

    distances = {(i, j): distance(i, j) for i in city_list for j in city_list if i != j}
    delta = {i: delta_factor * decay_rates[i] for i in non_depot}

    Gamma_eff = int(round(gamma_scale * Gamma))
    Gamma_eff = max(0, Gamma_eff)

    maxd = max(distances.values())
    M = T + maxd

    mdl = Model("VRP_CP_theta")

    x = mdl.binary_var_dict([(i, j, k)
                             for i in city_list for j in city_list if i != j
                             for k in range(K)], name="x")
    u = mdl.continuous_var_dict([(i, k) for i in non_depot for k in range(K)],
                                lb=1, ub=num_cities - 1, name="u")
    a = mdl.continuous_var_dict([(i, k) for i in city_list for k in range(K)],
                            lb=0, ub=T, name="a")
    y = mdl.binary_var_dict(non_depot, name="y")
    CP = mdl.continuous_var_dict(non_depot, lb=0, name="CP")
    theta = mdl.continuous_var_dict([(i, k) for i in non_depot for k in range(K)],
                                lb=0, ub=T, name="theta")
    lambd = mdl.continuous_var(lb=0, name="lambda")
    s = mdl.continuous_var_dict(non_depot, lb=0, name="s")
    z = mdl.binary_var_dict([(i, k) for i in non_depot for k in range(K)], name="z")

    mdl.maximize(
        mdl.sum(CP[i] for i in non_depot)
        - lambd * Gamma_eff
        - mdl.sum(s[i] for i in non_depot)
        - mdl.sum(distances[i, j] * x[i, j, k] for (i, j) in distances for k in range(K))
    )

    for j in non_depot:
        mdl.add_constraint(
```

Models implementation

```
            mdl.sum(x[i, j, k] for i in city_list if i != j for k in range(K)) == y[j]
        )

    for k in range(K):
        for h in non_depot:
            mdl.add_constraint(
                mdl.sum(x[i, h, k] for i in city_list if i != h)
                == mdl.sum(x[h, j, k] for j in city_list if j != h)
            )
    for k in range(K):
        mdl.add_constraint(mdl.sum(x[0, j, k] for j in non_depot) <= 1)
        mdl.add_constraint(mdl.sum(x[i, 0, k] for i in non_depot) <= 1)
        mdl.add_constraint(
            mdl.sum(x[0, j, k] for j in non_depot) ==
            mdl.sum(x[i, 0, k] for i in non_depot)
    )

    for k in range(K):
        for i in non_depot:
            mdl.add_constraint(a[(i,k)] + distances[i,0] <= T + M*(1 - x[i,0,k]))

    # MTZ subtour elimination
    for k in range(K):
        for i in non_depot:
            for j in non_depot:
                if i != j:
                    mdl.add_constraint(
                        u[i, k] - u[j, k] + (num_cities - 1) * x[i, j, k] <= num_cities - 2
                    )

    # Time continuity
    for k in range(K):
        mdl.add_constraint(a[(0, k)] == 0)

    for k in range(K):
        for i in city_list:
            for j in non_depot:
                if i != j:
                    mdl.add_constraint(
                        a[(i, k)] + distances[i, j] <= a[(j, k)] + M * (1 - x[i, j, k])
                    )
    for k in range(K):
        mdl.add_constraint(mdl.sum(distances[i,j]*x[i,j,k] for (i,j) in distances) <= T)

    for i in non_depot:
        for k in range(K):
            mdl.add_constraint(
                mdl.sum(x[j, i, k] for j in city_list if j != i) == z[i, k]
            )

    for i in non_depot:
        mdl.add_constraint(mdl.sum(z[i, k] for k in range(K)) == y[i])

    for i in non_depot:
        for k in range(K):
            mdl.add_constraint(theta[i, k] <= a[i, k])
            mdl.add_constraint(theta[i, k] <= T * z[i, k])
            mdl.add_constraint(theta[i, k] >= a[i, k] - T * (1 - z[i, k]))

    for i in non_depot:
        pi = prices[i]
        bar_alpha = decay_rates[i]  # nominal alpha_i
        delta_i = delta_factor*decay_rates[i]
        mdl.add_constraint(CP[i] == pi * y[i] - bar_alpha * mdl.sum(theta[i, k] for k in range(K)))

    for i in non_depot:
        delta_i = 0.25 * decay_rates[i]
        expr = delta_i * mdl.sum(theta[i, k] for k in range(K))
        mdl.add_constraint(lambd >= expr)
        mdl.add_constraint(lambd + s[i] >= expr)
```

Models implementation

```
if min_coverage_frac > 0.0:
    min_cov = math.ceil(min_coverage_frac * len(non_depot))
    mdl.add_constraint(mdl.sum(y[i] for i in non_depot) >= min_cov)


#Optional
if with_balancing:
    max_customers = (len(non_depot)) / K
    for k in range(K):
        mdl.add_constraint(
            mdl.sum(x[i, j, k] for i in city_list for j in non_depot if i != j) <= max_customers
        )

mdl.context.cplex_parameters.emphasis.mip = 1
solution = mdl.solve(log_output=False, time_limit=30)


if not solution:
    print(f"No solution found for: {title}")
    return None

routes = {k: [] for k in range(K)}
for k in range(K):
    arcs = [(i, j) for (i, j) in distances if x[i, j, k].solution_value > 0.5]
    current = 0
    while arcs:
        for arc in arcs:
            if arc[0] == current:
                routes[k].append(arc)
                current = arc[1]
                arcs.remove(arc)
                break
        else:
            current = 0

if ax is not None and plt is not None:
    colors = ['blue', 'green', 'red', 'orange', 'purple', 'brown']
    for i in city_list:
        xi, yi = cities[i]
        ax.scatter(xi, yi, color='black', s=25)
        ax.text(xi + 1, yi - 2, f"City {i}", fontsize=8)

    for k in range(K):
        color = colors[k % len(colors)]
        for i, j in routes[k]:
            xi, yi = cities[i]
            xj, yj = cities[j]
            ax.plot([xi, xj], [yi, yj], color=color, linewidth=2.25)
            mx, my = (xi + xj) / 2, (yi + yj) / 2
            ax.text(mx, my, f"{distances[i, j]}", fontsize=7, color=color)

    ax.set_title(title)
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.grid(True)

total_profit = sum(CP[i].solution_value for i in CP)
travel_cost = sum(
    distances[i, j] * x[i, j, k].solution_value
    for (i, j) in distances for k in range(K)
)
objective_value = mdl.objective_value
details = mdl.solve_details

# These are the key numbers:
#best_obj   = details.best_obj
best_bound = details.best_bound
rel_gap    = details.mip_relative_gap
bs_penalty = Gamma_eff * lambd.solution_value + sum(s[i].solution_value for i in non_depot)
```

Models implementation

```
    return {
        "objective": objective_value,
        "total_profit": total_profit,
        "travel_cost": travel_cost,
        "lambda": lambd.solution_value,
        "routes": routes,
        "best_bound": best_bound,
        "relative_gap": rel_gap,
        "bs_penalty": bs_penalty,
    }


# ============================================================
#   MASTER EXPERIMENT DRIVER
#   (Gamma, N, rep loops: MILP, GA, VNS, write table)
# ============================================================

def run_experiments():
    gamma_fracs = [0.3]
    Ns = [15,25,30]
    Ks = [3]
    Ts = [250]

    NUM_REP = 3
    NUM_SETUPS = 1

    delta_ga = 1.00
    delta_vns = 1.00
    travel_cost_weight = 1.0

    time_limit_ga= 30.0
    time_limit_vns = 30.0

    all_results = []

    base_seed_map =4321
    base_seed_pd = 44

    maxN = max(Ns)
    master_cities = {}
    for setup_id in range(NUM_SETUPS):
        rng_map = random.Random(base_seed_map + setup_id)
        cities_full = {
            i: (rng_map.uniform(0, 100), rng_map.uniform(0, 100))
            for i in range(maxN)
        }
        master_cities[setup_id] = cities_full

    for N in Ns:
        for setup_id in range(NUM_SETUPS):
            # ---------- FIXED MAP FOR THIS (N, setup_id) ----------
            cities_full = master_cities[setup_id]
            cities = {i: cities_full[i] for i in range(N)}
            rng_map = random.Random(base_seed_map + 1000 * N + setup_id)
            distances = build_distances(cities)

            print(f"\n=== MAP: setup={setup_id}, N={N} (cities 0..{N-1}) ===")

            for rep in range(NUM_REP):
                # ---------- PRICES & DECAY VARY WITH rep (BUT SAME FOR ALL Γ,K,T) ----------
                rng_pd = random.Random(
                    base_seed_pd + 100000 * N + 1000 * setup_id + 10 * rep
                )
                prices = {i: rng_pd.randint(25,250) for i in cities if i != 0}
                decay  = {i: rng_pd.uniform(0.01, 1) for i in prices.keys()}

                print(f"\n--- Instance econ: setup={setup_id}, N={N}, rep={rep} ---")

                for gamma_frac in gamma_fracs:
```

Models implementation

```
                Gamma = max(1, int(round(gamma_frac * (N - 1))))
                for K in Ks:
                    for T in Ts:
                        Gamma = max(1, int(round(gamma_frac * (N - 1))))
                        # seed for GA/VNS randomness
                        seed_alg = (
                            1000000 + 10000 * N + 1000 * setup_id
                            + 100 * rep + 10 * Gamma + K + int(T)
                        )

                        print(f"  >> Γ={Gamma} (≈{gamma_frac*100:.0f}% of {N-1}), K={K}, T={T}, rep={rep}")

                        row = {
                            "setup": setup_id,
                            "GammaFrac": gamma_frac,
                            "Gamma": Gamma,
                            "N": N,
                            "K": K,
                            "T": T,
                            "rep": rep,
                        }

                        # Create figure / axes
                        if plt is not None:
                            fig, axes = plt.subplots(1, 3, figsize=(12, 4))
                        else:
                            axes = [None, None, None]

                        # ---------- MILP ----------
                        t0 = time.perf_counter()
                        milp_res = solve_vrp_with_CP_theta(
                            cities=cities,
                            prices=prices,
                            decay_rates=decay,
                            K=K,
                            T=T,
                            Gamma=Gamma,
                            with_balancing=False,
                            ax=axes[0],
                            delta_factor=1,
                            gamma_scale=0.8,
                            min_coverage_frac=0.0,
                            title=f"MILP VRP Γ={Gamma}, N={N},K={K},T={T}, rep={rep}",
                        )
                        t1 = time.perf_counter()

                        if milp_res is not None:

                            milp_gene: Gene = {}
                            for k in range(K):
                                route = [0]
                                arcs = milp_res["routes"][k]
                                current = 0
                                while True:
                                    nxt = [j for (i, j) in arcs if i == current]
                                    if not nxt:
                                        break
                                    current = nxt[0]
                                    route.append(current)
                                    if current == 0:
                                        break
                                if route[-1] != 0:
                                    route.append(0)
                                milp_gene[k] = route

                            # MILP metrics: no delta, no beta
                            milp_metrics = compute_solution_metrics(
                                milp_gene,
                                cities, prices, decay,
                                T, Gamma,
```

Models implementation

```
                distances,
                high_priority_frac=0.2,
                attack_mode="rate",
                delta=0.10,
                beta=0.0,
                travel_cost_weight=travel_cost_weight,
            )

            row.update({
                "obj_MILP": milp_metrics["robust_obj"],
                "profit_MILP": milp_metrics["total_reward"],
                "travel_cost_MILP": milp_metrics["travel_cost"],
                "penalty_MILP": milp_res["bs_penalty"],
                "time_sec_MILP": t1 - t0,
                "total_reward_MILP": milp_metrics["total_reward"],
                "high_priority_served_MILP": milp_metrics["high_priority_served"],
                "high_priority_coverage_MILP": milp_metrics["high_priority_coverage"],
                "coverage_ratio_MILP": milp_metrics["coverage_ratio"],
                "fairness_MILP": milp_metrics["fairness"],
                "avg_route_length_MILP": milp_metrics["avg_route_length"],
                "max_route_length_MILP": milp_metrics["max_route_length"],
                "gap_MILP": milp_res["relative_gap"],
                "num_visited_MILP": milp_metrics["num_visited"],
                "num_non_depot_MILP": milp_metrics["num_non_depot"],
            })
        else:
            # in case MILP fails: fill with None so table has the columns
            row.update({
                "obj_MILP": None,
                "profit_MILP": None,
                "travel_cost_MILP": None,
                "penalty_MILP": None,
                "time_sec_MILP": t1 - t0,
                "total_reward_MILP": None,
                "high_priority_served_MILP": None,
                "high_priority_coverage_MILP": None,
                "coverage_ratio_MILP": None,
                "fairness_MILP": None,
                "avg_route_length_MILP": None,
                "max_route_length_MILP": None,
                "gap_MILP": None,
                "num_visited_MILP": None,
                "num_non_depot_MILP": None,
            })

        # ---------- GA ----------
        t0 = time.perf_counter()
        ga_res = genetic_algorithm_robust_vrp(
            cities=cities,
            prices=prices,
            decay=decay,
            K=K,
            T=T,
            Gamma=Gamma,
            delta=delta_ga,
            pop_size=150,
            generations=800,
            travel_cost_weight=travel_cost_weight,
            rng_seed=seed_alg,
            verbose=False,
            feasible_only=True,
            plot_interval=0,
            time_limit_sec=time_limit_ga,
        )
        t1 = time.perf_counter()
        ga_gene = ga_res["best_gene"]
        ga_metrics = compute_solution_metrics(
            ga_gene, cities, prices, decay, T, Gamma, distances,
            high_priority_frac=0.2,
            attack_mode="rate",
```

Models implementation

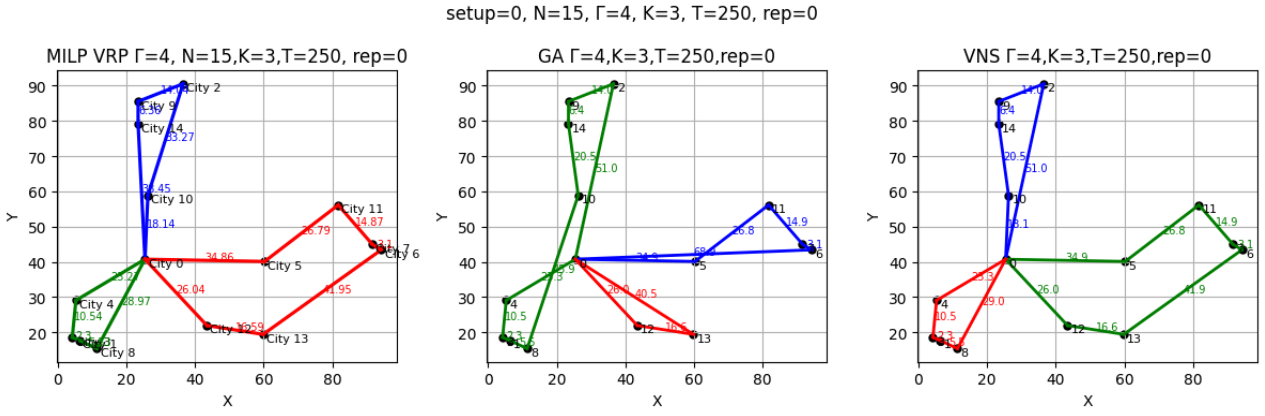
```
                    delta=delta_ga,
                    beta=beta_ga,
                    travel_cost_weight=travel_cost_weight,
                )

                row.update({
                    "obj_GA": ga_metrics["robust_obj"],
                    "profit_GA": ga_metrics["total_reward"],
                    "travel_cost_GA": ga_metrics["travel_cost"],
                    "penalty_GA": ga_metrics["robust_penalty"],
                    "time_sec_GA": t1 - t0,
                    "total_reward_GA": ga_metrics["total_reward"],
                    "high_priority_served_GA": ga_metrics["high_priority_served"],
                    "high_priority_coverage_GA": ga_metrics["high_priority_coverage"],
                    "coverage_ratio_GA": ga_metrics["coverage_ratio"],
                    "fairness_GA": ga_metrics["fairness"],
                    "avg_route_length_GA": ga_metrics["avg_route_length"],
                    "max_route_length_GA": ga_metrics["max_route_length"],
                    "num_visited_GA": ga_metrics["num_visited"],
                    "num_non_depot_GA": ga_metrics["num_non_depot"],
                })


                plot_gene_routes(
                    ga_gene, cities, distances,
                    axes[1],
                    title=f"GA Γ={Gamma},K={K},T={T},rep={rep}",
                )

                # ---------- VNS ----------
                t0 = time.perf_counter()
                vns_res = vns_vrp(
                    cities=cities,
                    prices=prices,
                    decay=decay,
                    K=K,
                    T=T,
                    Gamma=Gamma,
                    attack_mode="rate",
                    delta=delta_vns,
                    beta=beta_vns,
                    travel_cost_weight=travel_cost_weight,
                    kmax=6,
                    iterations=200,
                    rng_seed=seed_alg,
                    feasible_only=True,
                    verbose=False,
                    observer=None,
                    ls_samples={"insert": 50, "remove": 50, "reloc": 50, "twoopt": 50, "swap": 50},
                    patience=50,
                    kick_size=10,
                    time_limit_sec=time_limit_vns,
                )
                t1 = time.perf_counter()

                vns_gene = vns_res["best_gene"]
                vns_metrics = compute_solution_metrics(
                    vns_gene,
                    cities, prices, decay,
                    T, Gamma,
                    distances,
                    high_priority_frac=0.2,
                    attack_mode="time",
                    delta=delta_vns,
                    beta=beta_vns,
                    travel_cost_weight=travel_cost_weight,
                )

                row.update({
                    "obj_VNS": vns_metrics["robust_obj"],
```

```
                            "profit_VNS": vns_metrics["total_reward"],
                            "travel_cost_VNS": vns_metrics["travel_cost"],
                            "penalty_VNS": vns_metrics["robust_penalty"],
                            "time_sec_VNS": t1 - t0,
                            "total_reward_VNS": vns_metrics["total_reward"],
                            "high_priority_served_VNS": vns_metrics["high_priority_served"],
                            "high_priority_coverage_VNS": vns_metrics["high_priority_coverage"],
                            "coverage_ratio_VNS": vns_metrics["coverage_ratio"],
                            "fairness_VNS": vns_metrics["fairness"],
                            "avg_route_length_VNS": vns_metrics["avg_route_length"],
                            "max_route_length_VNS": vns_metrics["max_route_length"],
                            "num_visited_VNS": vns_metrics["num_visited"],
                            "num_non_depot_VNS": vns_metrics["num_non_depot"],
                        })


                        plot_gene_routes(
                            vns_gene, cities, distances,
                            axes[2],
                            title=f"VNS Γ={Gamma},K={K},T={T},rep={rep}",
                        )

                        # figure for this instance
                        if plt is not None:
                            fig.suptitle(f"setup={setup_id}, N={N}, Γ={Gamma}, K={K}, T={T}, rep={rep}")
                            plt.tight_layout()
                            plt.show()

                            fig.suptitle(f"setup={setup_id}, N={N}, Γ={Gamma}, K={K}, T={T}, rep={rep}")
                            plt.tight_layout()

                            # ---- SAVE PDF ----
                            out_dir = "outputs/routes"
                            os.makedirs(out_dir, exist_ok=True)
                            pdf_name = f"routes_setup{setup_id}_N{N}_G{Gamma}_K{K}_T{int(T)}_rep{rep}.pdf"
                            fig.savefig(
                                os.path.join(out_dir, pdf_name),
                                format="pdf",
                                bbox_inches="tight",
                            )

                            plt.show()
                            plt.close(fig)

                        # store the row
                        all_results.append(row)

    # ---------- TABLE OUTPUT ----------
    if pd is not None:
        df = pd.DataFrame(all_results)
        base_cols = ["setup","Gamma","GammaFrac", "N","K","T", "rep","gap_MILP"]
        metrics = [
            "obj",
            "profit",
            "travel_cost",
            "penalty",
            "time_sec",
            "total_reward",
            "high_priority_served",
            "high_priority_coverage",
            "coverage_ratio",
            "fairness",
            "avg_route_length",
            "max_route_length",
            "num_visited",
            "num_non_depot",
        ]
        methods = ["MILP", "GA", "VNS"]

        ordered_cols = base_cols + [
```

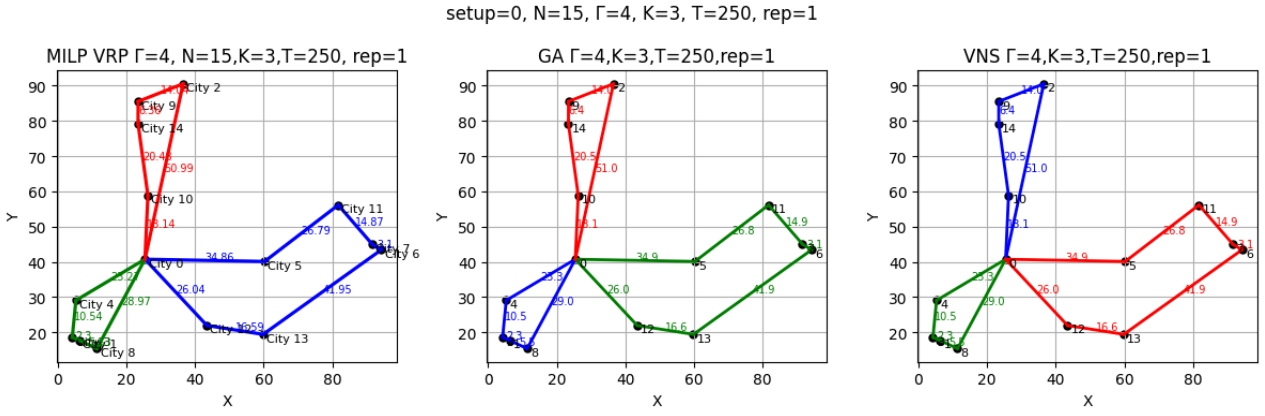
Models implementation

```
            f"{m}_{meth}"
            for m in metrics
            for meth in methods
            if f"{m}_{meth}" in df.columns
        ]

        remaining = [c for c in df.columns if c not in ordered_cols]
        df = df[ordered_cols + remaining]


        print("\n=== RESULTS TABLE ===")
        print(df)

        df.to_csv("vrp_results_Graph3.csv", index=False)
        print("\nResults written to vrp_results.csv")
    else:
        print("\n=== RESULTS (dict rows) ===")
        for row in all_results:
            print(row)

if __name__ == "__main__":
    run_experiments()
```

```
=== MAP: setup=0, N=15 (cities 0..14) ===

--- Instance econ: setup=0, N=15, rep=0 ---
  >> Γ=4 (≈30% of 14), K=3, T=250, rep=0
```




```
<Figure size 640x480 with 0 Axes>
--- Instance econ: setup=0, N=15, rep=1 ---
  >> Γ=4 (≈30% of 14), K=3, T=250, rep=1
```

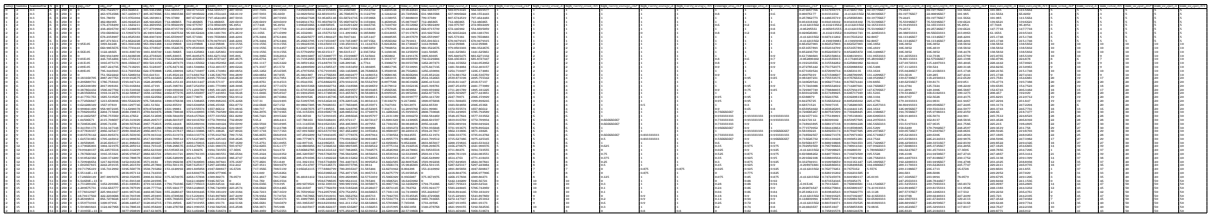



```
<Figure size 640x480 with 0 Axes>
--- Instance econ: setup=0, N=15, rep=2 ---
  >> Γ=4 (≈30% of 14), K=3, T=250, rep=2
```

A.2 Experimental results

A.2.1 Reference results

Table 2: VRP reference experimental results



A.2.2 Runtime limit

Table 3: VRP experimental results for runtime limits = 300 seconds

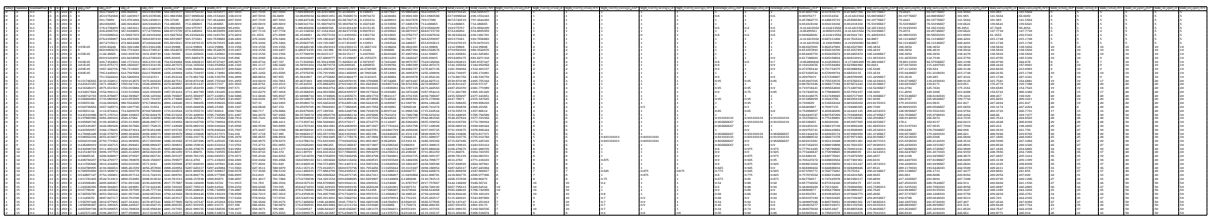
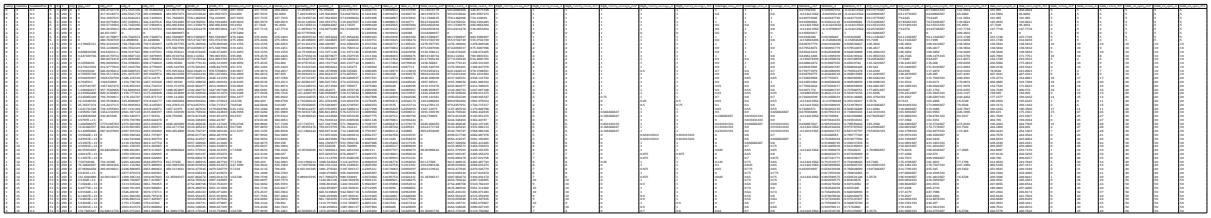
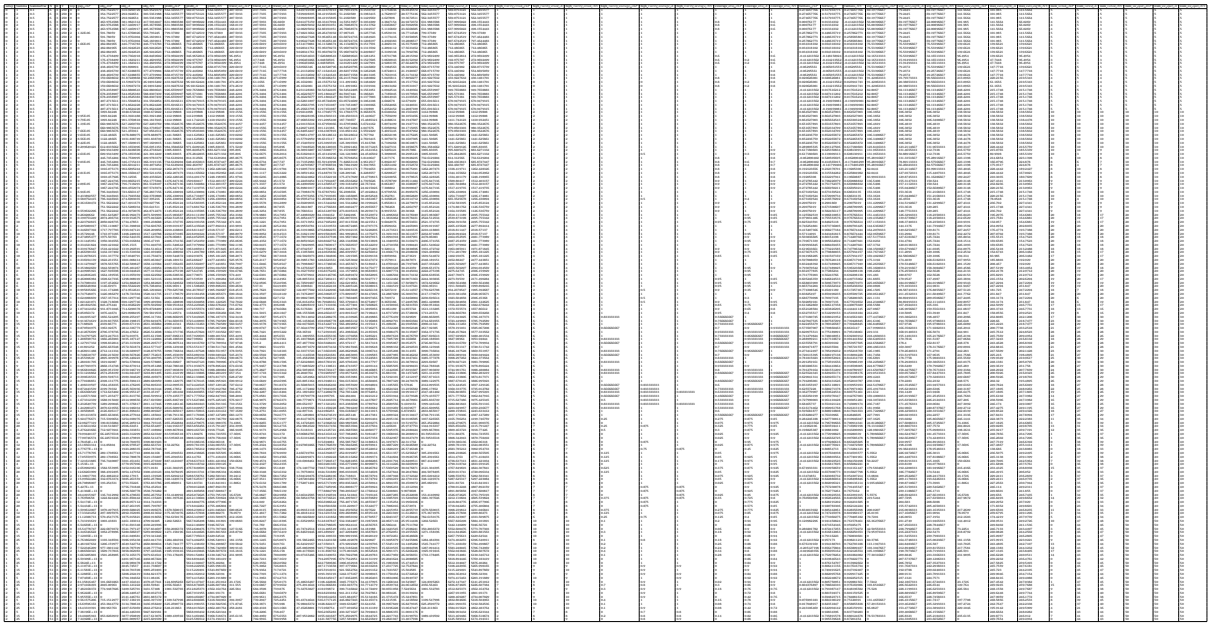


Table 4: VRP experimental results for runtime limits = 3 seconds



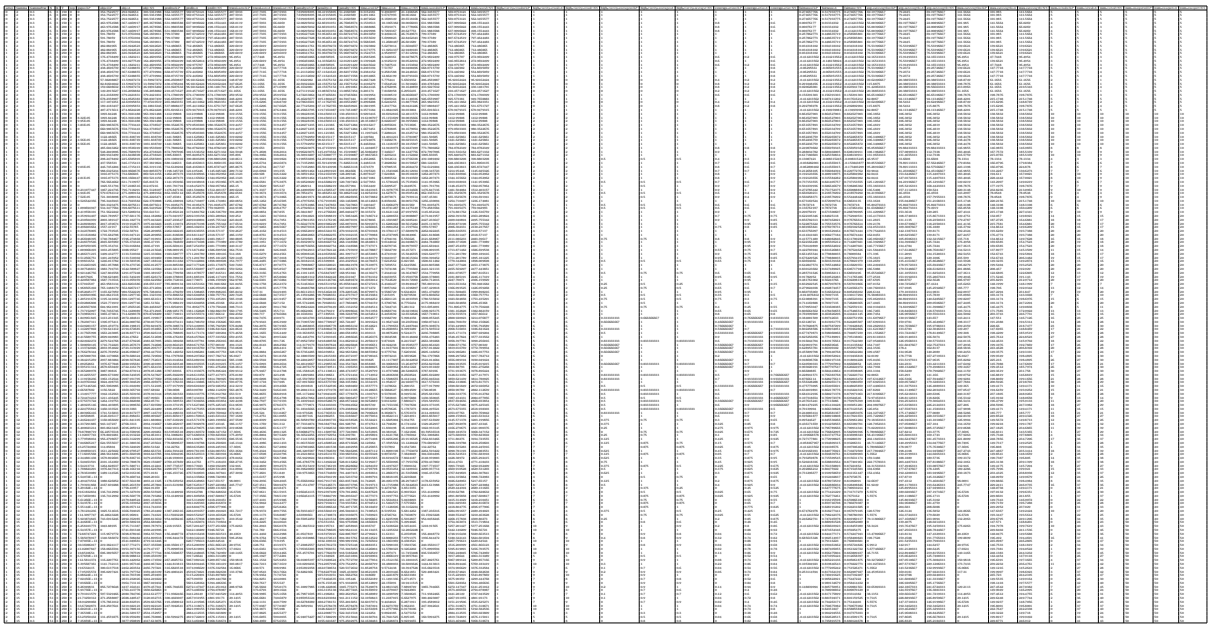
A.2.3 Gamma fraction

Table 5: VRP experimental results for gamma fraction {0.1,0.3,0.5}



A.2.4 Time constraint

Table 6: VRP experimental results for T = {150,200,250}



A.2.5 Prices

Table 7: VRP experimental results for prices ranges [10;300]

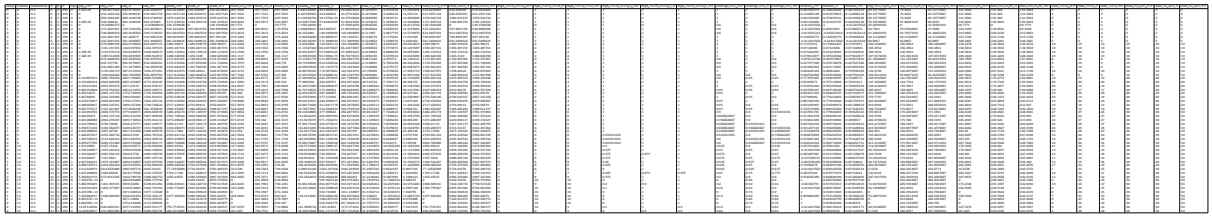
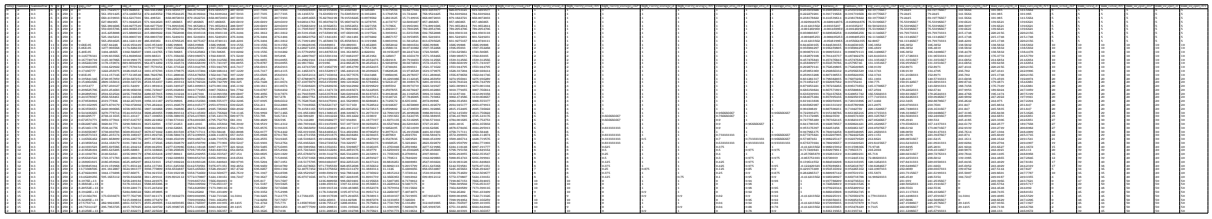


Table 8: VRP experimental results for prices ranges [200;250]



A.2.6 Decay rates

Table 9: VRP experimental results for decay rates ranges [0.1;3]

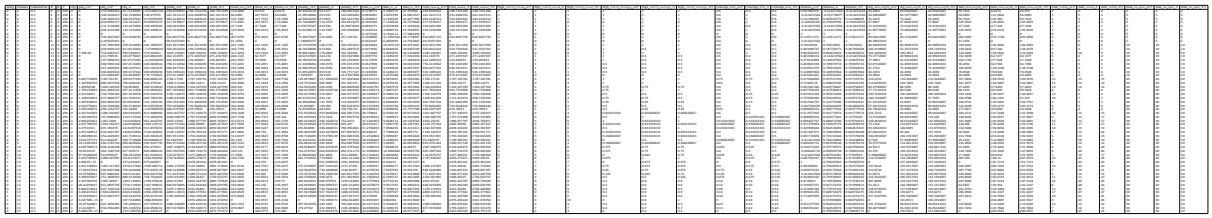


Table 10: VRP experimental results for decay rates ranges [0.1;0.5]

B Bibliography

References

Altay, N. and Green III, W. (2006). Or/ms research in disaster operations management. *European Journal of Operational Research*, 175:475–493. https://doi.org/10.1016/j.ejor.2005.05.016. 8

Applegate, D., Bixby, R., Chvátal, V., and Cook, W. (2006). The traveling salesman problem: A computational study. *The Traveling Salesman Problem: A Computational Study*. 10

Archetti, C., Feillet, D., Hertz, A., and Speranza, M. G. (2009). The capacitated team orienteering and profitable tour problems. *Journal of the Operational Research Society*, 60(6):831–842. https://doi.org/10.1057/palgrave.jors.2602603. 10, 18, 24

Avishan, F., Elyasi, M., Yanıkoğlu, , Ekici, A., and Ozener, O. (2023). Humanitarian relief distribution problem: An adjustable robust optimization approach. *Transportation Science*. https://doi.org/10.1287/trsc.2023.1204. 18, 19

Back, T. (1996). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press. 33

Balcik, B., Beamon, B., Krejci, C., Muramatsu, K., and Ramirez, M. (2010). Coordination in humanitarian relief chains: Practices, challenges and opportunities. *International Journal of Production Economics*, 126:22–34. https://doi.org/10.1016/j.ijpe.2009.09.008. 8

Balcik, B., Beamon, B., and Smilowitz, K. (2008). Last mile distribution in humanitarian relief. *Journal of Intelligent Transportation Systems*, 1818. https://doi.org/10.1080/15472450802023329. 18, 24

Balcik, B. and Yanıkoğlu, (2019). A robust optimization approach for humanitarian needs assessment planning under travel time uncertainty. *European Journal of Operational Research*, 282. https://doi.org/10.1016/j.ejor.2019.09.008. 17, 19

Ben-Tal, A. and Nemirovski, A. (2002). Robust optimization-methodology and applications. *Math. Program.*, 92:453–480. https://doi.org/10.1007/s101070100286. 16

Bertsimas, D., Brown, D., and Caramanis, C. (2011). Theory and applications of robust optimization. *SIAM Review*, 53:464–501. https://doi.org/10.1137/080734510. 16, 18

Bertsimas, D., Gupta, V., and Kallus, N. (2013). Data-driven robust optimization. *Mathematical Programming*, 167. https://doi.org/10.1007/s10107-017-1125-8. 66

Bertsimas, D. and Sim, M. (2004). The price of robustness. *Operations Research*, 52:35–53. https://doi.org/10.1287/opre.1030.0065. 16, 19, 27, 28, 30, 35

Blum, C. and Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308. https://doi.org/10.1145/937503.937505. 11, 12

Burke, E., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., and Woodward, J. (2010). A classification of hyper-heuristic approaches. *Handbook of Metaheuristics*, 146:449–468. https://doi.org/10.1007/978-1-4419-1665-5$_1$5.12

Burke, E., Kendall, G., Mccollum, B., and Mcmullan, P. (2007). Constructive versus improvement heuristics: an investigation of examination timetabling. 12

Burke, E. and Newall, J. (2003). Enhancing timetable solutions with local search methods. volume 2740, pages 195–206. https://doi.org/10.1007/978-3-540-45157-0$_1$3.13

Cacchiani, V., Hemmelmayr, V., and Tricoire, F. (2014). A set-covering based heuristic algorithm for the periodic vehicle routing problem. *Discrete Applied Mathematics*, 163:53–64. https://doi.org/10.1016/j.dam.2012.08.032. 13, 16, 35, 36

Chao, I.-M., Golden, B. L., and Wasil, E. A. (1996). The team orienteering problem. *European Journal of Operational Research*, 88(3):464–474. https://ideas.repec.org/a/eee/ejores/v88y1996i3p464-474.html. 10

Dorigo, M., Maniezzo, V., and Colorni, A. (1996). Ant system: Optimization by a colony of cooperating agents. ieee trans syst man cybernetics - part b. *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, 26:29–41. https://doi.org/10.1109/3477.484436. 14

Esfahani, P. and Kuhn, D. (2015). Data-driven distributionally robust optimization using the wasserstein metric: Performance guarantees and tractable reformulations. *Mathematical Programming*, 171. https://doi.org/10.1007/s10107-017-1172-1. 66

Feng, Y. and Cui, S. (2021). A review of emergency response in disasters: present and future perspectives. *Natural Hazards*, 105. https://doi.org/10.1007/s11069-020-04297-x. 9

Garey, M. and Johnson, D. S. (1982). The np-completeness column: An ongoing guide. *Journal of Algorithms*, 3(3):288–300. https://doi.org/10.1016/0196-6774(87)90021-6. 11, 42

Gendreau, M. and Potvin, J.-Y. (2010). *Handbook of Metaheuristics*, volume 146. https://doi.org/10.1007/978-1-4614-0769-0$_3$0.11,12,15

Glover, F. and Laguna, M. (1999). *Tabu search I*, volume 1. https://doi.org/10.1287/ijoc.1.3.190. 13, 15

Goldberg, D. E. and Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. *Foundations of Genetic Algorithms*. https://doi.org/10.1016/B978-0-08-050684-5.50008-2. 32

Golden, B., Levy, L., and Vohra, R. (1987). The orienteering problem. *Nav Res Logist*, 34:307–318. https://doi.org/10.1002/1520-6750(198706)34:310, 18

Gunawan, A., Lau, H. C., and Vansteenwegen, P. (2016). Orienteering Problem: A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*, 255(2):315–332. https://doi.org/10.1016/j.ejor.2016.03.028. 10, 11, 18

Hansen, P., Mladenovic, N., and Moreno-Pérez, J. (2010). Variable neighbourhood search: Methods and applications. *4OR*, 175:367–407. https://doi.org/10.1007/s10479-009-0657-6. 13, 16, 35, 36, 37

Holguín-Veras, J., Sánchez-Díaz, I., and Browne, M. (2016). Sustainable urban freight systems and freight demand management. *Transportation Research Procedia*, 12:40–52. https://doi.org/10.1016/j.trpro.2016.02.024. 16

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press. 14

Ibrahim, M. F., Putri, M., Farista, D., and Utama, D. (2021). An improved genetic algorithm for vehicle routing problem pick-up and delivery with time windows. *Jurnal Teknik Industri*, 22:1–17. https://doi.org/10.22219/JTIUMM.Vol22.No1.1-17. 14, 15

Karimi Mamaghan, M., Mohammadi, M., Meyer, P., Karimi Mamaghan, A. M., and Talbi, E.-G. (2021). Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. *European Journal of Operational Research*, 296. https://doi.org/10.1016/j.ejor.2021.04.032. 65

Kirkpatrick, S., Gelatt, C., and Vecchi, M. (1983). Optimization by simulated annealing. *Science (New York, N.Y.)*, 220:671–80. https://doi.org/10.1126/science.220.4598.671. 13, 15

Kovács, G. and Spens, K. (2007). Humanitarian logistics in disaster relief operations. *International Journal of Physical Distribution Logistics Management*, 37:99–114. https://doi.org/10.1108/09600030710734820. 8

Kunz, N. and Reiner, G. (2012). A meta-analysis of humanitarian logistics research. *Journal of Humanitarian Logistics and Supply Chain Management*, 2:116–147. https://doi.org/10.1108/20426741211260723. 9

Kytöjoki, J., Nuortio, T., Bräysy, O., and Gendreau, M. (2007). An efficient variable neighborhood search heuristic for very large scale vehicle routing problems. *Computers Operations Research*, 34:2743–2757. https://doi.org/10.1016/j.cor.2005.10.010. 13

Lamos, H., Aguilar Imitola, K., and Amado, R. (2019). Or/ms research perspectives in disaster operations management: a literature review. *Revista Facultad de Ingenieria*, pages 43–59. https://doi.org/10.17533/udea.redin.n91a05. 8, 9

Laporte, G. (2009). Fifty years of vehicle routing. *Transportation Science*, 43:408–416. https://doi.org/10.1287/trsc.1090.0301. 9, 18

Lenstra, J. and Kan, A. (2006). Complexity of vehicle routing and scheduling problems. *Networks*, 11:221 – 227. https://doi.org/10.1002/net.3230110211. 10, 14

Liberatore, M. and Luo, W. (2010). The analytics movement: Implications for operations research. *Interfaces*, 40:313–324. https://doi.org/10.1287/inte.1100.0502. 18

Maniezzo, V., Stützle, T., and Voss, S. (2010). *Matheuristics – Hybridizing Metaheuristics and Mathematical Programming*. 66

Marinakis, Y. and Marinaki, M. (2019). A multi-adaptive particle swarm optimization for the vehicle routing problem with time windows. *Information Sciences*, 481:311–329. https://doi.org/10.1016/j.ins.2018.12.086. 15

Nekoueian, R., Servranckx, T., and Vanhoucke, M. (2023). Constructive heuristics for selecting and scheduling alternative subgraphs in resource-constrained projects. *Computers Industrial Engineering*, 182:109399. https://doi.org/10.1016/j.cie.2023.109399. 11

Ochelska-Mierzejewska, J., Poniszewska-Maranda, A., and Marańda, W. (2021). Selected genetic algorithms for vehicle routing problem solving. *Electronics*, 10:3147. https://doi.org/10.3390/electronics10243147. 15

Osman, I. (1993). Meta-strategy simulated annealing and tabu search algorithms for the vehicle routine problem. *Annals of Operations Research*, 41:421–451. https://doi.org/10.1007/BF02023004. 13

Prins, C. (2004). Prins, c.: A simple and effective evolutionary algorithm for the vehicle routing problem. computer operations research 31(12), 1985-2002. *Computers Operations Research*, 31:1985–2002. https://doi.org/10.1016/S0305-0548(03)00158-8. 14

Schrimpf, G., Schneider, J., Stamm-Wilbrandt, H., and Dueck, G. (2000). Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159:139–171. https://doi.org/10.1006/jcph.1999.6413. 38

Talbi and Talbi, E.-G. (2009). *Metaheuristics: From Design to Implementation*, volume 74. https://doi.org/10.1002/9780470496916. 12

van Laarhoven, P. J. M., Aarts, E. H. L., and Lenstra, J. K. (1992). Job shop scheduling by simulated annealing. *Operations Research*, 40(1):113–125. http://www.jstor.org/stable/171189. 13

Van Wassenhove, L. (2006). Humanitarian aid logistics: Supply chain management in high gear. *Journal of The Operational Research Society - J OPER RES SOC*, 57:475–489. https://doi.org/10.1057/palgrave.jors.2602125. 8

Vansteenwegen, P. and Souffriau, W. (2011). The orienteering problem: A survey. *European Journal of Operational Research*, 209:1–10. https://doi.org/10.1016/j.ejor.2010.03.045. 24

Wang, C., Ma, S., Zha, B., and Ma, B. (2025). Surrogate-accelerated evolutionary algorithm for solving electric vehicle routing problems. *IEEE Transactions on Transportation Electrification*, 11:3524–3537. https://doi.org/10.1109/TTE.2024.3443515. 15

Yanıkoğlu, and Yavuz, T. (2021). Branch-and-price approach for robust parallel machine scheduling with sequence-dependent setup times. *European Journal of Operational Research*. https://doi.org/10.1016/j.ejor.2021.11.023. 17

Yi, W. and Ozdamar, L. (2007). A dynamic logistics coordination model for evacuation and support in disaster response activities. *European Journal of Operational Research*, 179:1177–1193. https://doi.org/10.1016/j.ejor.2005.03.077. 24

Yu, Q., Cheng, C., and Zhu, N. (2022). Robust team orienteering problem with decreasing profits. *Informs Journal on Computing*, 34. https://doi.org/10.1287/ijoc.2022.1240. 17, 19, 24, 27, 28

EXECUTIVE SUMMARY

Disaster response operations require fast and reliable planning under uncertainty and time pressure. This thesis addresses this challenge by studying and developing a robust time-sensitive capacitated orienteering problem (RT-CTOP) tailored to humanitarian logistics. The objective is to maximize collected rewards representing disaster victims, while accounting for uncertain travel times, limited resources, and time-dependent service values.

A robust optimization framework based on a budgeted uncertainty set is proposed to explicitly model travel-time variability. As is well known, exact methods such as Mixed-Integer Linear Programming (MILP) are NP-hard for this class of problems, meaning that they quickly become computationally intractable as instance size increases. To overcome this limitation, two metaheuristic approaches, Genetic Algorithm (GA) and Variable Neighborhood Search (VNS), are designed and adapted to the RT-CTOP formulation.

The results show that GA and VNS consistently deliver high-quality and stable solutions while significantly outperforming the MILP in terms of scalability and computational behavior. In instances where the MILP is solvable, heuristic solutions closely approximate optimal values. Under tight runtime limits or increased uncertainty, heuristic methods maintain solution quality, whereas MILP performance degrades sharply.

In conclusion, this work demonstrates the suitability of metaheuristic methods for addressing large-scale and time-critical disaster response routing problems under uncertainty and provides a strong basis for future research on adaptive and data-driven robust optimization frameworks.

KEYWORDS: Disaster response, Humanitarian logistics, Team Orienteering, Robust optimization, Metaheuristics methods, Genetic algorithm, Variable neighborhood search

WORD COUNT: 15658

Ecole de Gestion de l'Université de Liège