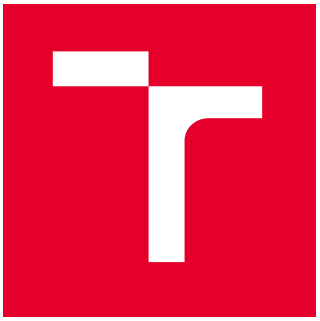


BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING

FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF MATHEMATICS

ÚSTAV MATEMATIKY

TRAVELING SALESMAN GAME

HRA OBCHODNÍHO CESTUJÍCÍHO

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR — **Andrej Lonek**

AUTOR PRÁCE

SUPERVISOR — **Ing. Ivan Eryganov, Ph.D.**

VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF MECHANICAL ENGINEERING

Assignment Bachelor's Thesis

Institut:	Institute of Mathematics
Student:	**Andrej Lonek**
Degree programm:	Mathematical Engineering
Branch:	no specialisation
Supervisor:	**Ing. Ivan Eryganov, Ph.D.**
Academic year:	2024/25

As provided for by the Act No. 111/98 Coll. on higher education institutions and the BUT Study and Examination Regulations, the director of the Institute hereby assigns the following topic of Bachelor's Thesis:

Traveling Salesman Game

Brief Description:

The Traveling Salesman Problem (TSP) asks for the shortest possible route that visits a given set of cities exactly once and returns to the original city. Cooperative game theory offers a strategic framework where entities can form coalitions to minimize the total travel cost. The problem is transformed into a traveling salesman game (TSG), where the worth of a coalition is defined by the minimum travel cost for visiting all members. Cooperative game theory helps in fair cost allocation among multiple entities. The Shapley value and other cooperative game–theoretic solutions (like the core and nucleolus) are used to distribute the cost savings fairly among all participants.

Bachelor's Thesis goals:

Studying the traveling salesman problem and its formulations.
Studying the fundamental concepts of game theory.
Analysis of different methods for fairly distributing the total travel cost among the participating agents.
Application of these methods to realistic scenarios using your own implementation or existing programming language libraries.

Recommended bibliography:

APPLEGATE, D. L., et al. The Traveling Salesman Problem: A Computational Study. Princeton University Press, 2006.

MYERSON, R. B. Game Theory: Analysis of Conflict. Harvard University Press, 1991.

GILLES, R. P. The Cooperative Game Theory of Networks and Hierarchies. Springer, 2010.

Deadline for submission Bachelor's Thesis is given by the Schedule of the Academic year2024/25

In Brno,

L. S.

doc. Mgr. Petr Vašík, Ph.D. Director of the Institute	doc. Ing. Jiří Hlinka, Ph.D. FME dean

Abstract

This thesis focuses on the study of the Traveling Salesman Problem (TSP), a classical combinatorial optimization problem concerned with finding the shortest possible route through a set of cities. The introductory part presents a formal definition of TSP and an overview of its computational complexity. The following chapters are dedicated to genetic algorithms, with a particular focus on permutation-based approaches tailored for solving TSP instances. Subsequently, the thesis explores the foundations of cooperative game theory and its solution concepts, with emphasis on their application to cost allocation problems. The practical part of the thesis consists of the implementation of a modular and parallelized TSP solver based on a genetic algorithm. This solver is then extended to approximate the Shapley value, which is used to fairly distribute travel costs among agents in the context of TSP-based cooperative games.

Abstrakt

Tato bakalářská práce se zabývá problémem obchodního cestujícího (TSP), klasickým kombinatorickým optimalizačním problémem, jehož cílem je nalezení nejkratší možné trasy spojující danou množinu měst. Úvodní část práce poskytuje formální definici TSP a přehled jeho výpočetní složitosti. Následující kapitoly jsou věnovány genetickým algoritmům, se zvláštním důrazem na permutační přístupy přizpůsobené řešení TSP. Dále se práce zaměřuje na základy kooperativní teorie her a její řešitelské koncepty, zejména z hlediska spravedlivého rozdělení nákladů. Praktická část práce spočívá v implementaci modulárního a paralelizovaného řešiče TSP založeného na genetickém algoritmu. Tento řešič je následně rozšířen o aproximaci Shapleyho hodnoty, která slouží ke spravedlivému rozdělení cestovních nákladů mezi agenty v kontextu kooperativních her založených na TSP.

Keywords

Traveling Salesman Problem, Genetic Algorithm, Metaheuristics, Cooperative Game Theory, Shapley Value, Cost Allocation

Klíčová slova

Problém obchodního cestujícího, Genetický algoritmus, Metaheuristiky, Kooperativní teorie her, Shapleyho hodnota, Rozdělení nákladů

Lonek, A.: *Traveling salesman game*, Brno: Brno University of Technology, Faculty of Mechanical Engineering, 2025. 69 pages. Bachelor's thesis supervisor: Ing. Ivan Eryganov, Ph.D.

Rozšířený abstrakt

Tato bakalářská práce se zabývá problémem *spravedlivého rozdělení nákladů* v úlohách trasování prostřednictvím kombinací konceptů z oblasti *kooperativní teorie her* a *kombinatorické optimalizace*. Práce je strukturována do dvou částí: první představuje nezbytný teoretický základ, zatímco druhá se zaměřuje na praktickou implementaci a benchmarking.

Teoretická část začíná *problémem obchodního cestujícího* (TSP), klasickým NP-těžkým problémem, jehož cílem je nalézt nejkratší možnou trasu, která navštíví každé město právě jednou a vrátí se do výchozího bodu. Následuje podrobný přehled *genetických algoritmů* (GA)—stochastických optimalizačních metod, které pracují s populací řešení a jsou zvláště efektivní pro úlohy s permutačním kódováním, jako je právě TSP. Poté jsou představeny klíčové pojmy z kooperativní teorie her, se zaměřením na *nákladové hry*, kde hráči spolupracují na snížení celkových nákladů namísto maximalizace zisku. Diskutovány jsou čtyři hlavní koncepty řešení: *jádro*, *relaxované jádro*, *nukleolus* a *Shapleyho hodnota*, přičemž hlavní důraz je kladen na Shapleyho hodnotu díky její škálovatelnosti a srozumitelnosti v kontextu trasování.

Druhá část práce představuje komplexní implementaci modulárního a rozšiřitelného řešiče TSP v jazyce `Python` využívající genetický algoritmus. Návrh klade důraz na flexibilitu a umožňuje uživatelům konfigurovat operátory výběru, křížení, mutace a strategie přežití. Kandidátní řešení jsou reprezentována jako permutace a podporovány jsou běžné operátory, jako například turnajový výběr, řádkové křížení nebo inverzní mutace. Implementace zahrnuje grafické uživatelské rozhraní (GUI) postavené na knihovně `customtkinter`, které umožňuje interaktivní nastavení parametrů a vizualizaci výsledků v reálném čase.

Pro ověření výkonnosti a správnosti algoritmu je genetický řešič porovnáván s optimálními řešeními získanými pomocí `Concorde TSP Solver`, a to na standardních instancích z databáze `TSPLIB`. Tyto externí benchmarky slouží jako referenční bod pro hodnocení kvality přibližných řešení a ladění parametrů algoritmu.

V rámci kooperativní teorie her je tento rámec rozšířen o výpočet Shapleyho hodnoty—alokace nákladů založené na průměrném příspěvku každého města ke koaličním nákladům. Vzhledem k faktoriálnímu růstu počtu uspořádání koalic je přesný výpočet reálně proveditelný pouze pro malé instance. K tomuto účelu je implementována aproximační metoda založená na náhodném výběru permutací, která odhaduje mezní příspěvky. Výpočet je paralelizován pomocí modulu `multiprocessing` jazyka Python, což umožňuje efektivní škálování na větší problémy.

Klasické statistické nástroje, jako jsou Pearsonův a Spearmanův korelační koeficient, byly použité pro zkoumání vztahu mezi geometrickými charakteristikami měst (např. vzdáleností od depa nebo průměrnou vzdáleností od ostatních) a jejich Shapleyho hodnotou. Tato analýza poskytuje vhled do vlivu prostorového umístění na mezní přínos v kooperativních trasovacích problémech.

Kombinací teorie, návrhu algoritmu a empirického ověření práce ukazuje, že heuristické optimalizační techniky lze efektivně integrovat s kooperativními koncepty spravedlivé alokace nákladů. Vyvinutý rámec představuje základ pro další výzkum v oblasti kolaborativní logistiky, sdílení zdrojů a optimalizace řízené umělou inteligencí.

I declare that I have written the bachelor's thesis titled *The Traveling Salesman Game* independently under the supervision of Ing. Ivan Eryganov, Ph.D., using the materials listed in the bibliography.

Andrej Lonek

I would like to express my sincere gratitude to my thesis supervisor, Ing. Ivan Eryganov, Ph.D., for his valuable guidance, insightful advice, and constructive feedback throughout the writing of this thesis. His willingness to provide direction, clarify complex ideas, and suggest improvements was essential to the completion of this work.

Andrej Lonek

Contents

1 Introduction

The aim of this thesis is to study cost allocation methods in cooperative settings arising from the TSP and to implement efficient computational tools for solving such problems. TSP is a classical combinatorial optimization problem with a wide range of practical applications, particularly in logistics and routing. Due to its complexity, exact solutions become computationally intractable for larger instances, making heuristic approaches, such as genetic algorithms, an essential part of modern solution techniques. Additionally, the cooperative aspect of TSP enables the study of fair cost distribution among multiple agents using concepts from cooperative game theory.

In Chapter 2, the Traveling Salesman Problem is introduced and various mathematical formulations are discussed, starting with integer linear programming models. The chapter then transitions to a graph-theoretical perspective, where the problem is reframed as a search for Hamiltonian cycles. Both exact and heuristic solution approaches are reviewed, emphasizing their applicability and limitations.

Chapter 3 is dedicated to genetic algorithms, focusing on permutation-based representations tailored for solving TSP. Specific genetic operators such as crossover and mutation are discussed in detail, along with their behavior on permutation-encoded solutions.

Chapter 4 explores cooperative game theory and its relevance to TSP as a multi-agent cost allocation problem. Fundamental solution concepts such as the core, the relaxed core, the nucleolus, and the Shapley value are introduced, and their theoretical properties are examined.

Chapter 5 presents the implementation of a genetic algorithm-based TSP solver. Its performance is benchmarked across various instances, and insights into parameter selection, operator effectiveness, and scalability are discussed. This chapter also includes proposals for future improvements.

Finally, Chapter 6 builds upon the previous implementation to construct a solver for the Shapley value. The genetic algorithm TSP solver is used to evaluate the characteristic function of coalitions, enabling the approximation of the Shapley value in otherwise intractable cooperative games.

2 Traveling salesman problem

Before presenting a formal definition, we begin with an informal formulation of the TSP:

> **Given a set of cities and pairwise distances between them, determine the shortest possible route that visits each city exactly once and returns to the original city.**

In this chapter, we primarily build upon the concepts and explanations provided in [1, 2].

2.1 Formal Mathematical Formulation of TSP

TSP can be expressed as an Integer Linear Program (ILP), with several formulations available in the literature. Among the most prominent are the Miller–Tucker–Zemlin (MTZ) formulation and the Dantzig–Fulkerson–Johnson (DFJ) formulation.

To formulate the TSP as an optimization problem, we begin with a set of n cities denoted by $V = \{v_1, v_2, \ldots, v_n\}$. Each city $v_i \in V$ is associated with a location in two-dimensional Euclidean space $\mathbb{E}^2$, represented by coordinates $(x_i, y_i) \in \mathbb{R}^2$. We also need to introduce a binary decision variable representing a tour between two cities

Definition 2.1. The *decision variable* is a binary variable x_{ij} defined for each pair of distinct cities $v_i, v_j \in V$ as follows:

$$x_{ij} = \begin{cases} 1, & \text{if the tour includes a direct path from city } v_i \text{ to city } v_j, \\ 0, & \text{otherwise.} \end{cases}$$

The travel cost between two cities v_i and v_j is given by their Euclidean distance:

$$c_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2},$$

which captures the straight-line distance between cities and ensures the symmetry of the cost matrix $C = [c_{ij}]$.

This cost structure forms the basis of the Euclidean TSP, where the objective is to select values of $x_{ij} \in \{0, 1\}$ that minimize the total travel cost subject to constraints ensuring a valid tour. We also consider the cooperative extension of this problem, known as the Traveling Salesman Game (TSG), which builds upon the same geometric cost structure and focuses on fair cost allocation among the participating cities.

2.1.1 Miller-Tucker-Zemlin formulation

In addition to the binary decision variables x_{ij} introduced earlier, the MTZ formulation incorporates a set of auxiliary variables u_i for all $v_i \in V$. These variables represent the relative position of each city in the tour and are used to eliminate subtours, a tour that visits only a proper subset of the cities, instead of all cities, by enforcing a consistent ordering of the visited cities. The interpretation is that $u_i < u_j$ implies that city i is visited before city j in the tour. Together, x_{ij} and u_i form the basis of the following integer programming model.

Objective function

$$\min_{x_{ij}, u_i} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} c_{ij} \tag{1}$$

Subject to:

$$\sum_{\substack{i=1 \\ i \neq j}}^{n} x_{ij} = 1 \qquad \forall v_j \in V \tag{2}$$

$$\sum_{\substack{j=1 \\ i \neq j}}^{n} x_{ij} = 1 \qquad \forall v_i \in V \tag{3}$$

$$u_i - u_j + 1 \leq (n-1)(1 - x_{ij}) \qquad 2 \leq i \neq j \leq n \tag{4}$$

$$2 \leq u_i \leq n \qquad 2 \leq i \leq n \tag{5}$$

$$x_{ij} \in \{0, 1\}, u_i \in \mathbb{N} \tag{6}$$

Constraints (2) and (3) ensure that each city is entered and departed exactly once through selected arcs, forming a collection of one or more cycles that together cover all cities.

Constraint (4) eliminates subtours by imposing an ordering on the cities using auxiliary variables u_i. If a arc (v_i, v_j) is selected in the tour (i.e., $x_{ij} = 1$), the constraint enforces $u_i + 1 \leq u_j$, meaning city v_j must appear later in the tour than city v_i. If directed arc (v_i, v_j) is not selected (i.e., $x_{ij} = 0$), the constraint becomes non-restrictive. This prevents the formation of smaller cycles that do not include all cities, ensuring that only a single tour can satisfy all constraints.

Finally, Constraint (5) defines the valid bounds for the MTZ variables u_i. These bounds are essential to ensure the correctness of the subtour elimination mechanism and to maintain feasibility in the ILP model.

2.1.2 Dantzig-Fulkerson-Johnson Formulation

An alternative to the MTZ formulation is the DFJ model. Unlike the MTZ model, it eliminates subtours by adding one constraint for each subset of cities, leading to an exponential number of constraints. Although this increases the model size, the linear relaxation of the DFJ formulation is tighter, meaning that its optimal value is closer to the true integer solution. This makes the DFJ model more effective in exact solution methods. It is formulated as follows:

Objective function:

$$\min_{x_{ij}} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} c_{ij} x_{ij} \tag{1}$$

Subject to:

$$\sum_{\substack{i=1 \\ i \neq j}}^{n} x_{ij} = 1 \qquad \forall v_j \in V \tag{2}$$

$$\sum_{\substack{j=1 \\ i \neq j}}^{n} x_{ij} = 1 \qquad \forall v_i \in V \tag{3}$$

$$\sum_{i \in S} \sum_{\substack{j \in S \\ j \neq i}} x_{ij} \leq |S| - 1 \qquad \forall S \subset V,\ 2 \leq |S| \leq n-1 \tag{7}$$

$$x_{ij} \in \{0, 1\} \tag{6}$$

Constraints (2) and (3) were already introduced in the MTZ formulation and serve the same purpose in the DFJ model.

However, these constraints alone are insufficient to prevent the formation of multiple disconnected subtours. To eliminate such subtours, constraints (6) are introduced. For every proper subset $S \subset V$ with $2 \leq |S| \leq n-1$, the total number of edges traversed within S is restricted to at most $|S|-1$. This condition ensures that no subset of cities can form an independent cycle, thereby enforcing that the solution constitutes a single connected tour over all cities.

2.1.3 The TSP and Hamiltonian Cycles

Another powerful perspective on the TSP comes from graph theory. While the previous section introduced the problem through the lens of optimization, we now shift focus to its combinatorial structure. In this view, cities are modeled as vertices in a graph, and potential direct connections between cities correspond to edges. This abstraction allows us to formulate the TSP as the problem of finding a Hamiltonian cycle of minimal total length.

At the foundation of this graph-theoretic approach lies the concept of a graph, which formally describes how entities (in our case, cities) are connected through pairwise relationships.

Definition 2.2. An *undirected graph* is a pair $G = (V, E)$, where V is a finite set of vertices, and $E \subseteq \{\{u, v\} \mid u, v \in V,\ u \neq v\}$ is a set of unordered pairs of distinct vertices, called *edges*.

Definition 2.3. A *complete graph* is an undirected graph in which every pair of distinct vertices is connected by a unique edge. Formally, a complete graph on n vertices is denoted by K_n and consists of:

- a vertex set $V = \{v_1, v_2, \ldots, v_n\}$, and
- an edge set $E = \{e_{ij} = \{v_i, v_j\} \mid 1 \leq i < j \leq n\}$,

such that the number of edges satisfies $|E| = \frac{n(n-1)}{2}$, reflecting the fact that each pair of distinct vertices is connected exactly once.

Definition 2.4. The *degree* of a vertex in an undirected graph is the number of edges connected to it.

With these basic notions in place, we now turn to the concept of Hamiltonian paths and cycles, which lie at the heart of the TSP's graph-theoretic formulation.

Definition 2.5. A *Hamiltonian path* in a graph $G = (V, E)$ is a sequence of distinct vertices in which each vertex of V is visited exactly once, and consecutive vertices are connected by edges in E. If, in addition, the first and last vertices are connected, the path forms a *Hamiltonian cycle*.

Definition 2.6 (Hamiltonian graph)**.** A graph $G = (V, E)$ is called *Hamiltonian* if it contains a Hamiltonian cycle.

A fundamental sufficient condition ensuring the existence of a Hamiltonian cycle is provided by Dirac's theorem.

Theorem 2.7 (Dirac)**.** *Every graph with $n \geq 3$ vertices and minimum degree at least $n/2$ has a Hamiltonian cycle [2].*

Applying Dirac's theorem to complete graphs, we immediately see that for a complete graph K_n, each vertex has degree $n-1$, which is greater than $n/2$ for all $n \geq 3$. Therefore, every complete graph with at least three vertices is Hamiltonian.

Since complete graphs are Hamiltonian, and in fact much richer in structure, we can observe an even stronger property: in a complete graph K_n, every permutation of the n vertices naturally defines a Hamiltonian path, and by adding the edge from the last vertex back to the first, a Hamiltonian cycle. Conversely, every Hamiltonian cycle in K_n corresponds to a unique ordering of the vertices, up to cyclic shifts.

Thus, there is a one-to-one correspondence between the set of permutations of the vertices and the set of Hamiltonian cycles in K_n, provided we fix a starting vertex. This observation allows us to reformulate the Traveling Salesman Problem as the search for a permutation that minimizes the total travel cost.

To formalize this idea, we fix a starting vertex, say v_1, and consider all permutations of the remaining $n-1$ vertices. Each such permutation uniquely determines a Hamiltonian cycle in K_n.

Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ denote a permutation of the vertices, where σ_1 is the fixed starting vertex, and $\sigma_2, \ldots, \sigma_n$ represent the order in which the remaining cities are visited. The corresponding tour follows the sequence:

$$v_{\sigma_1} \longrightarrow v_{\sigma_2} \longrightarrow \cdots \longrightarrow v_{\sigma_n} \longrightarrow v_{\sigma_1}.$$

Given a cost matrix $C = [c_{ij}]$, where c_{ij} denotes the cost of traveling from city v_i to city v_j, the total cost associated with a permutation σ is

$$\text{Cost}(\sigma) = \sum_{k=1}^{n-1} c_{\sigma_k \sigma_{k+1}} + c_{\sigma_n \sigma_1}.$$

Thus, the Traveling Salesman Problem can be reformulated as the problem of finding a permutation σ that minimizes $\text{Cost}(\sigma)$. The final term $c_{\sigma_n \sigma_1}$ ensures that the tour returns to the starting city, thereby completing the Hamiltonian cycle.

Formally, the objective can be expressed as

$$\min_{\sigma \in S_n} \text{Cost}(\sigma)$$

where S_n denotes the set of all permutations of the n vertices.

Although the permutation-based formulation of the TSP is compact and intuitively appealing, it is not well suited for exact solution methods when the number of cities becomes large. The number of possible permutations grows factorially with n, making exhaustive search or exact optimization approaches computationally infeasible for all but very small instances. However, this representation is particularly advantageous for heuristic and metaheuristic algorithms. In the later chapters, we will build on this formulation to develop a genetic algorithm tailored to solving the TSP.

2.2 Computational Complexity of the TSP

An algorithm is said to run in *polynomial time* if its worst-case running time is bounded above by a polynomial function of the input size. That is, for an input of size n, the algorithm completes in at most $O(n^k)$ steps for some constant $k \in \mathbb{N}$. In contrast, a *non-polynomial time* algorithm may require $O(2^n)$, $O(n!)$, or other super-polynomial bounds, which grow much faster and are generally considered infeasible for large input sizes.

Problems in class **P** are those that can be solved in polynomial time. The class **NP** contains problems for which a given solution can be verified in polynomial time, and it includes **P** as a subset. **NP-Complete** is the complexity class representing the set of all problems X in **NP** such that any other problem Y in **NP** can be reduced to X in polynomial time. Finally, **NP-hard** encompasses problems that are at least as hard as NP-complete problems, but which may not themselves belong to **NP**; they may not even have solutions verifiable in polynomial time.

The famous open question in computational complexity theory is whether **P** = **NP**, that is, whether every problem whose solution can be verified quickly can also be solved quickly.

The TSP is formally classified as NP-hard, whereas the decision version of the TSP, which asks whether there exists a tour with a total cost less than or equal to a given bound, is NP-complete.

The combinatorial complexity of the problem arises from the factorial growth of the solution space. For a set of n cities:

- in the symmetric TSP (where $c_{ij} = c_{ji}, \quad \forall v_i, v_j \in V, i \neq j$), the number of distinct tours is $(n-1)!/2$.
- in the asymmetric TSP (where $c_{ij} \neq c_{ji}, \quad \forall v_i, v_j \in V, i \neq j$), the number of distinct tours is $(n-1)!$.

Thus, the number of possible tours grows factorially with n, making exhaustive search infeasible even for moderate instance sizes.

Despite this theoretical intractability, a variety of exact and heuristic methods have been developed to solve practical instances of the TSP efficiently. In the following sections, we examine the principal approaches for solving the TSP, starting with exact algorithms capable of guaranteeing optimality and then exploring heuristic and metaheuristic methods that provide high-quality solutions for larger and more complex instances.

2.3 Exact Solution Methods

While the TSP is computationally difficult, exact algorithms play a crucial role by aiming to find provably optimal solutions. These methods are designed to guarantee optimality, at the cost of potentially high computational effort. In this section, we examine the two most prominent exact approaches for solving the TSP: Branch and Bound (B&B), which systematically explores the solution space while pruning suboptimal regions, and Branch and Cut (B&C), an advanced method that combines B&B with cutting-plane techniques to strengthen the search process.

2.3.1 Branch and Bound

Basic Principle. The B&B algorithm explores a search tree in which each node represents a partial tour, and each branch corresponds to a decision, such as selecting or excluding a particular edge. The full solution space is thus divided recursively into smaller subproblems. At each node, a lower bound on the total tour cost is computed. If this lower bound exceeds the cost of the best complete tour found so far (the current upper bound), the subproblem can be safely discarded (pruned) because it cannot lead to an optimal solution.

Bounding Strategy. Bounding is crucial for the efficiency of the algorithm. Common bounding techniques include:

- The cost of a minimum spanning tree on the remaining unvisited cities.
- Computation of an approximate dual solution that serves as a lower bound
- Heuristics that provide fast approximations of the remaining tour cost.

Branching Strategy. Branching decisions vary by implementation. One common approach is to branch on the next city to visit from the current city, generating child nodes corresponding to each possible continuation. Alternatively, one may branch based on the inclusion or exclusion of specific edges in the tour.

Termination. The algorithm terminates when all subproblems have been either solved or pruned. The best feasible tour found during the search is then guaranteed to be optimal.

Algorithm 1 Branch and Bound for the TSP

```
Initialize bestTourCost ← ∞, bestTour ← ∅
Initialize a priority queue with the root node (no cities visited, cost 0, lower bound 0)
while priority queue is not empty do
    Extract the node with the lowest lower bound from the queue
    if node represents a complete tour then
        if tourCost < bestTourCost then
            Update bestTour and bestTourCost with the current tour
        end if
    else if lowerBound(node) < bestTourCost then
        Generate child nodes by visiting an unvisited city from the current node
        for each child node do
            Compute the lower bound for the child node
            if lowerBound(child) < bestTourCost then
                Add the child node to the priority queue
            end if
        end for
    end if
end while
return bestTour and bestTourCost
```

To see a more detailed examination of the B&B method, see [1]. We now move on to the next method, called Branch and Cut (B&C), which builds upon B&B by incorporating cutting plane techniques.

2.3.2 Branch and Cut

The B&C algorithm is the best exact method currently known for solving the TSP. It combines the systematic exploration of the solution space, as in B&B, with the addition of dynamically generated linear inequalities, called *cutting planes*, that tighten the linear programming (LP) relaxation of the problem where relaxation is obtained by replacing $x \in \{0, 1\}$ with $x \in [0, 1]$. This method is particularly effective for large instances and forms the foundation of the state-of-the-art Concorde TSP Solver [9].

Basic Principle. At the core of the algorithm is a relaxation of the TSP as an LP, typically based on the DFJ formulation. Since this LP relaxation permits fractional solutions, additional constraints (cuts) are added dynamically to eliminate infeasible solutions and strengthen the relaxation. If the solution remains fractional after applying cuts, the algorithm proceeds by branching on a decision, such as fixing an edge to zero or one, thereby generating subproblems that are solved recursively.

Cutting Planes. The cutting planes are valid inequalities that are satisfied by all feasible solutions of the original integer TSP problem but violated by some LP-relaxed solutions. Common classes of cuts include:

- Subtour elimination constraints
- Comb inequalities
- Blossom inequalities
- Clique tree inequalities

The effectiveness of Branch and Cut depends critically on the ability to detect (or *separate*) violated cuts efficiently, which is typically done using specialized combinatorial algorithms.

Branching Strategy. When no further violated cuts can be found and the LP solution remains fractional, the algorithm branches, typically by selecting a fractional variable x_{ij} and creating two subproblems: one with $x_{ij} = 0$, and one with $x_{ij} = 1$. The resulting subproblems are solved recursively, each with their own cutting plane phase.

Termination. The algorithm continues recursively, branching and cutting, until all subproblems are either solved or pruned. The best integer solution found across all branches is then guaranteed to be optimal.

Algorithm 2 Branch and Cut for the TSP

```
Initialize bestTourCost ← ∞, bestTour ← ∅
Add root node (full relaxed problem) to queue
while queue is not empty do
    Extract node and solve LP relaxation
    while violated cuts are found do
        Add cuts to LP and re-solve
    end while
    if solution is integer then
        if tour cost < bestTourCost then
            Update bestTour and bestTourCost
        end if
    else if lower bound < bestTourCost then
        Branch on fractional variable
        Add resulting subproblems to queue
    end if
end while
return bestTour and bestTourCost
```

For a more detailed examination of the B&C method, see [1].

Above mentioned Concorde solver uses B&C as its core exact method for solving the TSP. However, it also incorporates heuristic approaches, mainly based on local search techniques, to quickly find high-quality tours and to enhance the performance of the exact solver. In the following chapter, we introduce some of the heuristic methods, as well as metaheuristic strategies, that are commonly employed in TSP solvers.

2.4 Heuristic and Metaheuristic solution and refinement methods

Heuristic: A heuristic is a problem-solving method designed to quickly find a good, though not necessarily optimal, solution by using rules of thumb, approximations, or local search strategies.

Metaheuristic: A metaheuristic is a higher-level framework that guides and controls a set of heuristics to explore the solution space more effectively, often to escape local optima and find better solutions over time.

Heuristic methods are particularly valuable for tackling large or complex instances where exact methods become computationally infeasible, offering a practical means to obtain high-quality approximate solutions within reasonable time frames.

For further examination of 2-opt, Lin-Kernighan, and Lin-Kernighan-Helsgaun local search heuristic see again [1].

2.4.1 Nearest Neighbor Heuristic

The *Nearest Neighbor* (NN) heuristic is one of the simplest and most intuitive methods for constructing an approximate solution to the TSP. Despite its simplicity and speed, it does not guarantee high-quality solutions and can perform poorly depending on the starting city and structure of the distance matrix.

Algorithm 3 Nearest Neighbor Heuristic for the TSP

Input: Distance matrix $C = [c_{ij}]$, set of cities/vertices V
Output: Tour T, total cost
Choose a starting city $s \in V$
Initialize $T \leftarrow [s]$, visited $\leftarrow \{s\}$
while not all cities are visited **do**
    Let v_i be the last city in T
    Find $v_j \in V \setminus$ visited that minimizes c_{ij}
    Append v_j to T, add v_j to visited
end while
Append starting city s to T to complete the tour
Compute total tour cost
return T, total cost

While construction heuristics like Nearest Neighbor focus on generating an initial feasible tour, local search heuristics such as *2-opt* and *k-opt* aim to refine and improve an existing tour by iteratively applying cost-reducing modifications.

2.4.2 2-opt and k-opt Local Search

The *2-opt* and, more generally, *k-opt* local search heuristics are powerful techniques for improving existing tours in the TSP.

2-opt Algorithm. The 2-opt algorithm removes two non-adjacent edges from the tour and reconnects the two resulting paths in a different way that still produces a valid tour. If the new tour is shorter, the change is accepted. This process is repeated until no improving move exists, yielding a locally optimal tour with respect to all 2-edge exchanges.

k-opt Generalization. The k-opt algorithm extends this idea by simultaneously replacing k edges in each move. While this allows for exploration of a larger neighborhood and potentially better solutions, the computational cost increases rapidly with k.

Algorithm 4 2-opt Local Search for the TSP

Input: Initial tour $T = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, distance matrix $C = [c_{ij}]$
Output: Locally optimized tour T
improvement $\leftarrow$ true
while improvement **do**
    improvement $\leftarrow$ false
    for $i = 1$ to $n - 2$ **do**
        for $j = i + 2$ to n **do**
            if $i = 1$ and $j = n$ **then**
                continue (skip swapping first and last city)
            end if
            Compute change in cost if segment $\sigma_{i+1} \ldots \sigma_j$ is reversed
            if cost decreases **then**
                Reverse segment $\sigma_{i+1} \ldots \sigma_j$
                improvement $\leftarrow$ true
            end if
        end for
    end for
end while
return T

While *2-opt* and *k-opt* algorithms provide effective frameworks for improving TSP tours by performing fixed-size exchanges of edges, their ability to escape local optima is limited by the fixed value of k chosen in advance. To address this limitation, the Lin–Kernighan heuristic extends the k-opt approach by dynamically selecting the number of edges to exchange during the search process. This flexibility enables deeper and more adaptive local improvements, significantly enhancing the ability to find high-quality tours even for large instances. We now describe the Lin–Kernighan method in more detail.

2.4.3 Lin–Kernighan Heuristic

The *Lin–Kernighan* heuristic is one of the most successful and widely used local search methods for improving TSP tours.

Basic Idea. The Lin–Kernighan algorithm iteratively searches for sequences of edge exchanges that reduce the total tour length. Starting from an initial tour, it removes and reconnects edges to create a new tour, where the number of exchanged edges can vary depending on the local structure of the solution space. By allowing flexible-depth moves, the heuristic escapes shallow local optima more effectively than standard 2-opt or 3-opt methods.

Advantages and Limitations. The flexibility of dynamically adjusting the move size makes Lin–Kernighan highly effective for producing high-quality tours. It typically finds near-optimal solutions quickly, even for large instances. However, the complexity of managing dynamic moves also increases the algorithm's implementation complexity compared to simpler fixed-k heuristics.

Algorithm 5 Lin–Kernighan Local Search for the TSP

Input: Initial tour $T = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, distance matrix $C = [c_{ij}]$
Output: Locally optimized tour T
improvement ← true
while improvement **do**
    improvement ← false
    for each city v_i in the tour **do**
        for each neighbor v_j of i **do**
            if removing edge (v_i, v_{next}) and adding edge (v_i, v_j) improves tour length **then**
                Perform the move
                improvement ← true
                Try additional exchanges recursively to deepen the move (variable-depth)
            end if
        end for
    end for
end while
return T

While the Lin–Kernighan heuristic remains one of the most effective local search methods for the Traveling Salesman Problem, it has been further refined in the state-of-the-art Lin–Kernighan–Helsgaun (LKH) heuristic. The LKH algorithm builds upon the core principles of LK but introduces more sophisticated mechanisms for selecting candidate edges. These enhancements allow LKH to guide the search more intelligently, significantly improving both efficiency and solution quality on large-scale TSP instances. Due to its complexity, the LKH algorithm is not discussed in detail here, but it is widely regarded as the most powerful heuristic for symmetric TSPs currently available.

2.4.4 Genetic algorithms

Genetic Algorithms (GAs) are a class of population-based metaheuristics inspired by the principles of natural selection and genetics. In the context of the TSP, solutions are typically represented as permutations of cities, and new solutions are evolved over generations through genetic operators such as selection, crossover, and mutation. GAs are particularly advantageous due to their ability to explore a large and complex solution space in a highly parallel and adaptive manner, making them well-suited for combinatorial optimization problems like the TSP. Given their flexibility, scalability, and effectiveness, GAs will be the method of choice going forward. The following chapter will examine their structure and application to the TSP in greater detail.

3 Genetic algorithms

In this chapter, we focus on genetic algorithms in which the genome is represented as a permutation. The operators introduced are specifically tailored for this type of representation. Before discussing these operators in detail, we briefly review essential concepts in genetic algorithms, such as the structure of the search space, the role of the objective function, and the definition of fitness. This overview is primarily based on [3], which served as the main reference during the preparation of this chapter, with additional inspiration drawn from [4].

To begin, let S denote the search space of all possible solutions, and let $f : S \to \mathbb{R}$ be the objective function, where lower values of f correspond to better solutions (e.g., shorter tour lengths in the case of the TSP). Genetic algorithms typically operate within a maximization framework, where higher fitness values indicate better candidates. To align minimization objectives with this framework, we apply a transformation function g to the objective values to define the fitness function:

$$Fit(s) = g(f(s)), \qquad s \in S$$

with property $f(s_1) < f(s_2)$, than $Fit(s_1) > Fit(s_2)$. A typical transformation g is then:

$$Fit(s) = \frac{1}{f(s)}$$

However, based on the operators used, we can also define the fitness function directly as:

$$Fit(s) = f(s)$$

Once the fitness function has been defined, genetic algorithms operate on populations $P^{(t)} \subset S$ of candidate solutions. In each generation, a new population is generated from the previous one, guided by the fitness values of the individuals, which influence selection, reproduction, and survival. This iterative process is designed to gradually steer the population towards a local optimum or, preferably, the global optimum.

To formalize this, we define $P^{(t)} \subset S$ as the population consisting of μ individuals at generation t, where the population size μ remains constant throughout the evolutionary process:

$$P^{(t)} = \{s_1^{(t)}, s_2^{(t)}, \ldots, s_\mu^{(t)}\}.$$

The evolution of the population across generations can then be modeled as a stochastic process:

$$P^{(t+1)} = \mathcal{E}(P^{(t)}),$$

where $\mathcal{E}$ encapsulates the genetic operators applied to the population. These operators include selection, which stochastically favors individuals with higher fitness; crossover, which combines parts of two parent solutions to create offspring; and mutation, which introduces random variations to maintain diversity. The stochastic nature of $\mathcal{E}$ arises from the probabilistic components inherent in these operations, such as the random selection of parents, crossover points, and mutation events. A detailed description of each operator will be provided later in this chapter.

Algorithm 6 Genetic Algorithm for the TSP

Input: Problem to solve, Parameter configuration
Output: Best tour found
Initialize population $P^{(0)}$ with μ random permutations of $\{1, 2, \ldots, n\}$
Evaluate fitness of the initial population
while not (Termination conditions) **do**
    Select parents from $P^{(t)}$ using a selection operator
    Generate offsprings through crossover operator
    Mutate offsprings with a given probability
    Evaluate fitness of offsprings
    Form next population $P^{(t+1)}$ using a replacement strategy
end while
return the best tour found in $P^{(T)}$

Having established the general structure of a genetic algorithm, we now focus on its application to the TSP. As a first step, it is necessary to specify how candidate solutions (genomes) are represented within this framework.

3.1 Representation of the Genome for the TSP

For the TSP, a candidate solution is a tour that visits each city exactly once before returning to the starting point. The natural and most effective representation for this problem is the permutation representation, where each individual is encoded as a permutation of the set $\{1, 2, \ldots, n\}$. This encoding has several advantages:

- **Feasibility by Design:** Every permutation inherently represents a valid tour without duplicate or missing cities.
- **Direct Mapping:** The order of the permutation corresponds directly to the order in which cities are visited, making the interpretation and evaluation straightforward.
- **Operator Compatibility:** Specialized crossover and mutation operators can be designed to maintain the permutation structure, ensuring offspring remain feasible.

Thus, for the TSP, the permutation-based representation is the most natural and efficient encoding strategy.

3.2 Parent Selection Operators

Selection operators are responsible for choosing individuals from the current population to serve as parents for the next generation. The goal of selection is to favor individuals with higher fitness, thereby guiding the evolutionary process towards better solutions, while also maintaining sufficient diversity to avoid premature convergence. Various selection mechanisms have been developed to balance these objectives, each introducing different stochastic or deterministic strategies for choosing parents. The main selection operators used in genetic algorithms will be described in the following subsections.

3.2.1 Fitness-Proportional Selection

Given a population $P = \{s_1, s_2, \ldots, s_\mu\}$, the selection probability of individual s_i is defined as:

$$p_{\text{FPS}}(s_i) = \frac{Fit(s_i)}{\sum_{j=1}^{\mu} Fit(s_j)}$$

This ensures that individuals with higher fitness are more likely to be selected for reproduction.

The advantages of FPS are its simplicity of implementation and its natural bias toward selecting fitter individuals.

The disadvantages: *Premature convergence:* Outstanding individuals may quickly dominate the population, narrowing the search and reducing diversity, which can prevent the discovery of better solutions. *Loss of selection pressure:* When fitness values are very similar, selection becomes almost random, and the average fitness improves very slowly over time. *Sensitivity to fitness function scaling:* Changes to the fitness function, such as rescaling or transformations, can alter selection pressure and negatively impact the algorithm's behavior.

To address weak selection pressure in FPS, two common techniques are employed:

- **Windowing**: A baseline β^t is subtracted from all fitness values in the current generation to increase relative differences. Typically,

  $$\beta^t = \min_{s \in P} \text{Fit}(s),$$

  where the adjusted fitness is then

  $$\text{Fit}'(s) = \text{Fit}(s) - \beta^t \quad \forall s \in P$$

  ensuring that all adjusted fitness values remain non-negative.
- **Sigma Scaling**: Fitness values are adjusted dynamically based on sample statistics of the current generation. Specifically,

  $$\text{Fit}'(s_i) = \max_{s \in P} \left(\text{Fit}(s) - \left(\bar{\text{Fit}}^t - c \cdot \sigma_{\text{Fit}}^t\right),\ 0\right),$$

  where c is a constant (typically $c = 2$), and $\bar{\text{Fit}}^t$ and σ_{Fit}^t are the sample mean and standard deviation of the fitness values, computed respectively as:

  $$\bar{\text{Fit}}^t = \frac{1}{\mu} \sum_{s \in P} \text{Fit}(s),$$

  $$\sigma_{\text{Fit}}^t = \sqrt{\frac{1}{\mu} \sum_{s \in P} \left(\text{Fit}(s) - \bar{\text{Fit}}^t\right)^2}.$$

  Both the sample mean and standard deviation are recomputed independently at each generation, ensuring that the scaling adapts dynamically to the current state of the population.

These modifications enhance selection pressure when fitness values are closely clustered, promoting a more effective evolutionary search.

Since fitness proportionate selection only defines selection probabilities without specifying a sampling method. However, before discussing these methods, we first examine ranking selection as a robust alternative to fitness proportionate selection.

3.2.2 Ranking Selection

Ranking Selection addresses the limitations of FPS by assigning selection probabilities based on the rank of individuals rather than their absolute fitness values. The population is first sorted according to fitness, and each individual is assigned a rank, with the worst individual receiving rank 0 and the best receiving rank $\mu - 1$, where μ is the population size.

In the case of linear ranking, the selection probability for an individual of rank i is given by:

$$p_{\text{lin-rank}}(i) = \frac{2-\alpha}{\mu} + \frac{2i(\alpha-1)}{\mu(\mu-1)},$$

where $\alpha \in (1, 2]$ is a parameter controlling the selection pressure. This parameter ensures that the best individual can receive at most α times the average selection probability. Higher values of α correspond to stronger selection pressure, while lower values promote greater diversity.

The advantages of Ranking Selection include its ability to maintain a constant selection pressure throughout the evolutionary process. Since selection probabilities are determined solely by the rank ordering of individuals rather than their absolute fitness values, Ranking Selection is less sensitive to changes or distortions in the fitness scale. As a result, it avoids the problems associated with fitness scaling that can affect Fitness-Proportional Selection, providing a more consistent and predictable selection behavior across generations.

The disadvantages of Ranking Selection stem primarily from its disregard for the actual magnitude of fitness differences between individuals. Significant differences in fitness do not translate into proportionally higher selection probabilities, as only the relative ordering is considered. This limitation can slow down convergence in problems where recognizing and favoring exceptional individuals early is critical. Furthermore, in linear ranking, the maximum achievable selection pressure is constrained by the requirement that selection probabilities remain non-negative, limiting the selection pressure parameter α to values within the interval $(1, 2]$.

To increase selection pressure when necessary, particularly when a strong emphasis on selecting the highest-ranked individuals is desired, an alternative exponential ranking scheme can be used. In exponential ranking, the selection probability for an individual of rank i is defined as:

$$p_{\text{exp-rank}}(i) = \frac{1 - e^{-i}}{c},$$

where c is a normalization constant ensuring that the total probability sums to 1 and depends on the population size. This scheme emphasizes top-ranked individuals more aggressively compared to linear ranking, allowing for stronger selection pressure when needed.

As with fitness-proportionate selection, ranking selection defines selection probabilities but does not specify a sampling method. In the next sections, we introduce roulette wheel selection and stochastic universal sampling (SUS), which can implement both approaches.

3.2.3 Roulette Wheel Selection

Roulette Wheel Selection simulates spinning a wheel where each slice is proportional to an individual's selection probability. Given a selection probability distribution $p_{\text{sel}}(i)$, the cumulative probability array $a = [a_1, \ldots, a_\mu]$ is computed as:

$$a_i = \sum_{j=1}^{i} p_{\text{sel}}(j).$$

The underlying selection probabilities $p_{\text{sel}}(i)$ can be based on either fitness-proportionate or ranking-based selection schemes.

The selection process than proceeds by drawing a random number r uniformly from the interval $[0, 1]$ and choosing the first individual i satisfying $r \leq a_i$.

Algorithm 7 Roulette Wheel Selection

```
1: Input: Cumulative probability array a = [a_1, ..., a_μ], number of selections λ
2: Output: Mating pool of λ individuals
3: for k = 1 to λ do
4:     Draw a random number r uniformly from [0, 1]
5:     Initialize i ← 1
6:     while r > a_i do
7:         i ← i + 1
8:     end while
9:     Add individual s_i to the mating pool
10: end for
11: return Mating pool
```

In Roulette Wheel Selection, each individual selection is made independently by drawing a separate random number. This independence introduces sampling variance, where the number of times individuals are selected can deviate significantly from their expected values based on their selection probabilities. As a result, individuals with high fitness may occasionally be underrepresented, while weaker individuals may be selected more often than intended. High sampling variance can distort the intended selective pressure, slow convergence, and lead to less stable evolutionary dynamics.

To reduce the sampling variance inherent in Roulette Wheel Selection, Stochastic Universal Sampling introduces multiple evenly spaced selection points

3.2.4 Stochastic Universal Sampling

Conceptually, Stochastic Universal Sampling (SUS) can be viewed as making a single spin of a wheel with λ evenly spaced pointers, rather than performing λ independent spins as in Roulette Wheel Selection.

The selection is then performed by drawing a random number r uniformly from the interval $[0, 1/\lambda]$, and incrementing r by $1/\lambda$ after each selection. Because of this even spacing, SUS guarantees that the number of times an individual i is selected is at least $\lfloor \lambda \cdot p_{\text{sel}}(i) \rfloor$ and at most one greater. This occurs because each increment of r corresponds to an equally spaced pointer, and an individual may either receive a number of selections exactly matching the floor of its expected count, or slightly more if a pointer falls just across its cumulative probability boundary. As a result, the realized number of selections closely matches the expected probabilities, greatly reducing sampling variance.

It is also worth noting that when $\lambda = 1$, SUS reduces to standard Roulette Wheel Selection.

Algorithm 8 Stochastic Universal Sampling (SUS)

Input: Cumulative probability array $a = [a_1, \ldots, a_\mu]$, number of selections λ
Output: Mating pool of λ individuals
current_member $\leftarrow 1$
$i \leftarrow 1$
Draw a random number r uniformly from $[0, 1/\lambda]$
while current_member $\leq \lambda$ **do**
    while $r \leq a_i$ **do**
        Add individual s_i to the mating pool
        $r \leftarrow r + 1/\lambda$
        current_member $\leftarrow$ current_member + 1
    end while
    $i \leftarrow i + 1$
end while
return Mating pool

While SUS improves selection reliability by reducing sampling variance, it still relies on predefined selection probabilities. Tournament Selection, introduced next, adopts a different strategy based on direct comparisons between individuals.

3.2.5 Tournament Selection

Tournament Selection is a widely used parent selection method that does not require global knowledge of the population or even an explicit numerical fitness function. Instead, it relies solely on pairwise comparisons between individuals, making it especially useful in situations where evaluating absolute fitness values is impractical. Examples include evolving game-playing strategies, where direct comparisons (e.g., matches) determine relative strength, or in evolutionary art and design, where subjective evaluations are made by a user.

Unlike fitness-proportionate and ranking-based methods, Tournament Selection selects individuals based on local relative comparisons. To select λ individuals from a population of μ, λ tournaments are conducted according to the following procedure:

Algorithm 9 Tournament Selection

Input: Population of μ individuals, number of selections λ, tournament size k, winning probability p
Output: Mating pool of λ individuals
for $k = 1$ to λ **do**
    Randomly select k individuals from the population (with or without replacement)
    Compare the k individuals and determine the best individual
    With probability p, select the best individual; otherwise select a random tournament participant
    Add the selected individual to the mating pool
end for
return Mating pool

The probability that an individual wins a tournament depends on four key factors:

- **Rank in the population:** Higher-ranked individuals are more likely to win tournaments without needing a full population sort.
- **Tournament size k:** Larger tournaments increase the likelihood of selecting high-fitness individuals, thus increasing selection pressure.
- **Winning probability p:** If $p = 1$, the tournament is deterministic, and the best individual is always selected. If $p < 1$, the tournament becomes stochastic, allowing a less-fit individual to be selected with nonzero probability, thereby reducing the selection pressure.
- **Replacement policy:** Selecting tournament participants with or without replacement affects the possibility of low-fitness individuals being selected. Without replacement, lower-ranked individuals may be systematically excluded.

Because Tournament Selection depends only on relative rankings, it shares the same invariance properties as ranking-based selection: translation or rescaling of fitness values does not affect the outcome. Moreover, by adjusting the tournament size k and probability p, Tournament Selection provides an intuitive and simple way to control selection pressure.

Despite its simplicity, Tournament Selection suffers from a drawback similar to that of Roulette Wheel Selection: high sampling variance between runs can cause fluctuations relative to the theoretical selection probabilities. Nonetheless, due to its extreme simplicity, efficiency, and controllable pressure, Tournament Selection is one of the most commonly used parent selection operators.

Having completed the selection stage, we will now focus on crossover operators, which generate offspring by combining genetic material from selected parents.

3.3 Crossover (Recombination) Operators

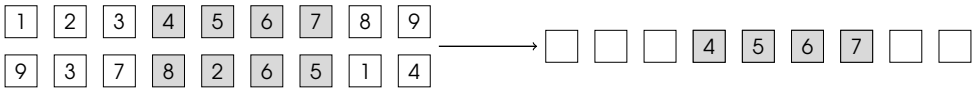
Crossover operators are responsible for combining the genetic information of selected parents to create new offspring, enabling the exploration of new regions of the search space. In standard genetic algorithms, crossover operates on simple string-based representations; however, when working with permutation-based representations, such as in the Traveling Salesman Problem, additional constraints must be respected. Specifically, the crossover must preserve the validity of offspring as permutations, ensuring that each element appears exactly once. In the following sections, we introduce various crossover operators designed specifically for permutation-based genomes.

3.3.1 Partially Mapped Crossover

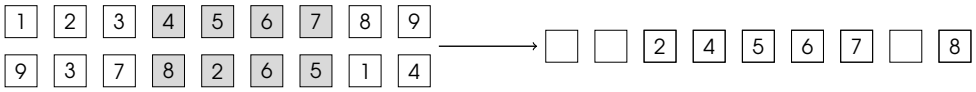
Partially Mapped Crossover (PMX) is a widely used recombination operator for permutation-based problems, especially TSP. PMX preserves relative order and position by defining a mapping between two parent permutations. The operator works as follows:

1. Select two crossover points at random and copy the segment between them from the first parent (P1) into the offspring.
2. For each element, let say i, in the same segment of the second parent (P2) that is not already in the offspring:
   - Identify the position of that element i in P2.
   - Determine the element j at same position in P1 (which was copied into the offspring).
   - Place the element i from P2 into the position where the element j from P1 occurs in P2.
   - If that position is already occupied, repeat this mapping recursively until an empty position is found.

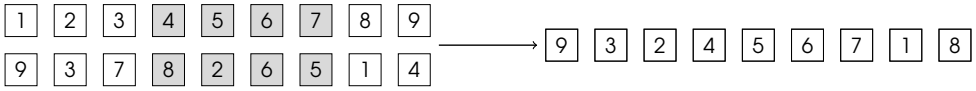
3. Fill the remaining positions in the offspring using elements from P2 that are not yet present, maintaining their original order.
4. Create the second offspring analogously by reversing the roles of the parents.



Step 1



Step 2



Step 3

Step 4

While PMX preserves a significant portion of the parental structure, it does not guarantee the respect property, meaning that any information carried in both parents should also be present in the offspring. To address this limitation, alternative crossover operators have been developed, as discussed in the following sections.

3.3.2 Edge Crossover

Edge crossover is based on preserving adjacency relationships between elements in the parents, rather than their absolute positions or orderings. To facilitate this, an edge table (also called an adjacency list) is constructed, where each element is associated with its neighbors from both parents. Edges common to both parents are specially marked to prioritize their preservation.

In this work, we use the Edge-3 crossover variant proposed by Whitley [**?**], which is specifically designed to maximize the preservation of common edges. The offspring construction proceeds as follows:

1. Construct the edge table based on both parents' adjacency information.
2. Randomly select an initial element and place it as the first entry of the offspring.
3. Set the current element to this initial entry.
4. Remove all references to the current element from the edge table.
5. Examine the adjacency list for the current element:
   - If a common edge (present in both parents) exists, select it as the next element.
   - Otherwise, select the element in the list that has the shortest adjacency list (smallest number of neighbors).
   - If there is a tie, select randomly among tied elements.
6. If an empty adjacency list is encountered, attempt to continue construction from the opposite end of the partial offspring; otherwise, select a new element at random.

An example of the construction process is shown in Tables 1 and 2, where two parent permutations [1 2 3 4 5 6 7 8 9] and [9 3 7 8 2 6 5 1 4] are combined using Edge-3 crossover. Notably, this operator produces only a single offspring per recombination.

Table 1: Edge crossover: example edge table

Element	Edges	Element	Edges
1	2,5,4,9	6	2,5+,7
2	1,3,6,8	7	3,6,8+
3	2,4,7,9	8	2,7+,9
4	1,3,5,9	9	1,3,4,8
5	1,4,6+		

Table 2: Edge crossover: example of permutation construction

Choices	Element Selected	Reason	Partial Result
All	1	Random	[1]
2,5,4,9	5	Shortest list	[1 5]
4,6	6	Common edge	[1 5 6]
2,7	2	Random choice (both lists size 2)	[1 5 6 2]
3,8	8	Shortest list	[1 5 6 2 8]
7,9	7	Common edge	[1 5 6 2 8 7]
3	3	Only item in list	[1 5 6 2 8 7 3]
4,9	9	Random choice	[1 5 6 2 8 7 3 9]
4	4	Last element	[1 5 6 2 8 7 3 9 4]

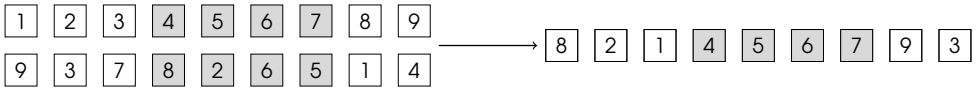
Although Edge-3 crossover effectively preserves common adjacency information between parents, it may sometimes produce offspring that are minor variations, such as cyclic shifts, of one of the parents. This conservative behavior can limit the genetic diversity of the population and may reduce the algorithm's ability to explore new regions of the search space.

3.3.3 Order Crossover

Order Crossover (OX) is a recombination operator specifically designed for order-based permutation problems. It aims to preserve the relative ordering of elements from both parents. The procedure resembles PMX in its first step, but diverges in how it fills the remaining positions to maintain relative order.

1. Select two crossover points at random and copy the segment between them from the first parent P1 into the offspring.
2. remaining unused elements from P1 are inserted into the offspring in the order they appear in P2 starting at the second cut-off point
3. Create the second offspring analogously by reversing the roles of the parents.

The key strength of OX lies in balancing structural inheritance with controlled randomness, making it one of the most widely adopted crossover operators for permutation problems such as the Traveling Salesman Problem.



Step 2

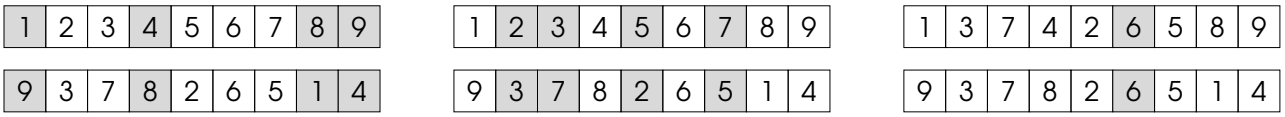
Typically, two offspring are created by applying the same procedure with parents' roles reversed. Its simplicity and structural preservation make it a standard operator.

3.3.4 Cycle Crossover

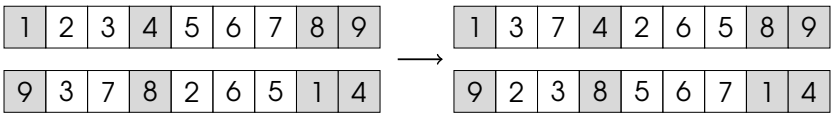
Cycle Crossover (CX) is designed to preserve information about the absolute positions of elements in permutation-based representations. The key idea is to identify cycles of values that occupy the same positions across both parents and use these cycles to construct offspring. Each offspring inherits alternating cycles from each parent, ensuring that each value appears exactly once and that positional consistency is partially preserved.

The process of identifying a cycle works as follows:

1. Start with the first unused position in the first parent (P1) and record the allele at that position.
2. Look at the allele in the same position of the second parent (P2).
3. Find the position in P1 where this allele occurs.
4. Add the allele from that position to the cycle.
5. Repeat steps 2–4 until the cycle returns to the starting allele from P1.



Cycle Identification



Resulting offsprings

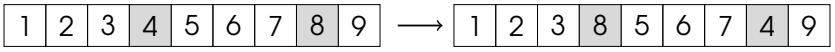
Similarly to OX, CX also produces two offspring simultaneously by alternating cycles between parents, maintaining positional consistency while introducing diversity.

3.4 Mutation Operators

Mutation plays a crucial role in evolutionary algorithms by introducing diversity into the population. It is specifically designed to widen the search space and help escape local optima by making random alterations to candidate solutions. In permutation-based representations, mutation operators rearrange elements in a solution, enabling the algorithm to explore new regions of the solution space that may not be reachable through crossover alone. This prevents premature convergence and increases the likelihood of finding globally optimal or near-optimal solutions.

3.4.1 Swap Mutation

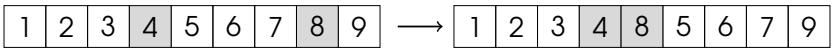
Two positions in the permutation are selected at random, and their values are swapped. This operator makes minimal changes and is often used for its simplicity.



Swap Mutation

3.4.2 Insert Mutation

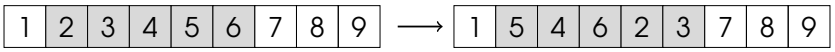
Two elements are selected at random, and one is inserted next to the other, shifting the intermediate elements accordingly. This operator slightly adjusts the ordering and is useful for fine-tuning solutions.



Insert Mutation

3.4.3 Scramble Mutation

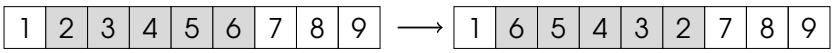
A randomly chosen subset of the permutation is selected and its elements are randomly shuffled. This introduces more randomness than swap or insert, while still preserving feasibility.



Scramble Mutation

3.4.4 Inversion Mutation

A substring between two randomly selected positions is reversed. This is particularly suited for adjacency-based problems, as it minimizes the number of broken links and preserves internal structure.



Inversion Mutation

Among these, inversion mutation is especially effective for problems like the Traveling Salesman Problem.

3.5 Survivor Selection

Survivor selection, also referred to as replacement, determines which individuals from the combined pool of parents and offspring survive to form the next generation. While similar mechanisms as used in parent selection can be applied, a number of specialized strategies have been developed in evolutionary computation to manage survivor selection efficiently. These strategies are generally categorized based on whether selection is based on fitness, age, or a combination of both.

3.5.1 Age-Based Replacement

In age-based replacement, individuals are removed from the population based solely on their age, regardless of fitness. This strategy ensures that all individuals have an equal lifespan in terms of evolutionary iterations. In generational models, where the number of offspring λ equals the number of parents μ, the entire parent population is replaced at each generation. This approach maintains diversity but may result in temporary decreases in average fitness.

3.5.2 Fitness-Based Replacement

Fitness-based replacement selects survivors based on their performance. A range of strategies exist under this category:

- **Replace Worst (GENITOR)**: The worst λ individuals (by fitness) are removed. This can lead to fast fitness improvement but risks premature convergence and diversity loss.
- **Elitism**: The best individual(s) are always preserved in the population.
- **Round-Robin Tournament**: Common in Evolutionary Programming, each individual competes against q randomly selected peers, earning a "win" if it outperforms them. The μ individuals with the most wins are selected. This introduces stochasticity and helps preserve diversity.
- $(\mu + \lambda)$ **Selection**: The union of parents and offspring is ranked by fitness, and the top μ individuals form the next generation. This strategy preserves strong individuals across generations and is widely used in Evolution Strategies.

Survivor selection plays a crucial role in balancing exploitation and exploration in evolutionary algorithms. The appropriate strategy depends on the specific problem, representation, and goals of the search process.

4 Cooperative Game Theory

This chapter introduces key concepts from cooperative game theory to address the problem of cost allocation in the Traveling Salesman Game (TSG). In this setting, cities cooperate to minimize the total travel cost of a tour, and the challenge lies in fairly distributing this cost among them. Since we are concerned with minimizing and allocating costs rather than distributing payoffs, all concepts will be defined in the context of *cost games*. We explore four fundamental solution concepts—the Core, Least Core, Nucleolus, and Shapley Value—each offering a different perspective on fairness and stability. The main sources used in preparing this chapter were [5, 6, 7, 8].

Remark. In cooperative game theory, a fundamental distinction is made between *payoff games* and *cost games*.

- In a **payoff game**, the characteristic function $v(S)$ represents the *maximum total payoff* that coalition S can jointly achieve. The goal is to fairly distribute this *value* among players.
- In a **cost game**, which is the focus of this thesis, $v(S)$ denotes the *minimum total cost* that coalition S must incur. The objective is to allocate this cost among the players in a way that reflects fairness and stability.

As a result, many standard inequalities and definitions must be *reversed* in cost games.

4.1 Basic Notation and Transferable Utility Games

To study cooperative behavior in the context of cost-sharing, we first introduce the mathematical structure used throughout this chapter: *transferable utility games*. These provide a flexible framework for modeling how groups of players (e.g., cities in the TSG) can cooperate to reduce their collective cost.

Definition 4.1. A *Transferable Utility Game (TU-game)* is a fundamental model in cooperative game theory. It is defined as a pair (N, v), where:

- $N = \{1, 2, \ldots, n\}$ is a finite set of players, and
- $v : 2^N \to \mathbb{R}$ is the *characteristic function*, which assigns to each coalition $S \subseteq N$ a real number $v(S)$, representing the total payoff (or cost reduction, in cost games) that the members of S can collectively achieve. By convention, $v(\emptyset) = 0$.

A game is called a *transferable utility* game because the total payoff or cost is assumed to be perfectly divisible and can be freely redistributed among the players without any loss. This assumption allows us to work with numerical allocations that can be split arbitrarily between coalition members.

We will denote by $\mathcal{G}_N$ the set of all such TU cost games on the player set N.

To express the total allocation assigned to a coalition, we frequently use the notation:

$$x(S) := \sum_{i \in S} x_i$$

for a cost (or payoff) distribution $x = (x_1, \ldots, x_n) \in \mathbb{R}^N$ and any coalition $S \subseteq N$. This notation allows us to compactly express how much a group of players receives (or pays) under the allocation x.

Definition 4.2. An *imputation*, is a vector $x \in \mathbb{R}^N$ that satisfies two key conditions:

- **Efficiency:** The total cost is fully allocated among the players:

$$x(N) = v(N).$$

- **Individual Rationality:** No player pays more than their standalone cost:

$$x(\{i\}) \leq v(\{i\}) \quad \text{for all } i \in N.$$

The set of all imputations of the game v is denoted by $I(v) \subset \mathbb{R}^N$. Efficiency ensures that no more is allocated than what the coalition can achieve collectively, while individual rationality guarantees that each player is not allocated more cost than they would incur on their own.

Definition 4.3. A game v is *essential* if

$$v(N) \leq \sum_{i \in N} v(\{i\}).$$

In addition to the properties of individual rationality and efficiency, several structural properties of the characteristic function are frequently studied:

- **Additivity:** A game is additive if

$$v(S \cup T) = v(S) + v(T) \quad \text{for all disjoint } S, T \subseteq N.$$

  Additive games model situations where the value of a union of coalitions is simply the sum of their separate values.
- **Superadditivity:** A game is superadditive if

$$v(S \cup T) \geq v(S) + v(T) \quad \text{for all disjoint } S, T \subseteq N.$$

  Superadditivity models situations where cooperation between disjoint coalitions creates additional value.
- **Subadditivity:** A game is subadditive if

$$v(S \cup T) \leq v(S) + v(T) \quad \text{for all disjoint } S, T \subseteq N.$$

  Subadditivity reflects economies of scale in cost games; merging coalitions can reduce overall costs.
- **Convexity:** A game is convex if for all $S, T \subseteq N$,

$$v(S \cup T) + v(S \cap T) \leq v(S) + v(T).$$

  Convexity implies that incentives to cooperate increase as coalitions grow larger.

Beyond structural properties, it is often valuable to understand how a game can be broken down into simpler building blocks. One powerful approach is to represent any game as a combination of standardized elementary games. This leads us to the concept of the *unanimity basis*, which forms a foundational tool for analyzing games in a framework of linear algebra.

4.2 The Unanimity Basis Representation of games

The set of all TU-games $\mathcal{G}_N$ forms a real vector space in the sense that for all games $v, w \in \mathcal{G}_N$ and scalars $\lambda, \mu \in \mathbb{R}$, we can define the linear combination $\lambda v + \mu w \in \mathcal{G}_N$ by

$$(\lambda v + \mu w)(S) = \lambda v(S) + \mu w(S), \quad \text{for all } S \subseteq N.$$

The zero element of this space is the *zero game* $0 \in \mathcal{G}_N$, defined by $0(S) = 0$ for all $S \subseteq N$.

This vector space structure allows any game $v \in \mathcal{G}_N$ to be expressed as a linear combination of basis elements. To construct such representations explicitly, we begin by defining the building blocks of this basis the unanimity games.

Definition 4.4 (Unanimity Game)**.** For every nonempty subset $S \subseteq N$, the *unanimity game* $u_S \in \mathcal{G}_N$ is defined by:

$$u_S(T) := \begin{cases} 1 & \text{if } S \subseteq T, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 4.5 (Unanimity Basis)**.** The set of all unanimity games

$$\mathcal{U} := \{u_S \mid \emptyset \neq S \subseteq N\}$$

forms a basis of the vector space $\mathcal{G}_N$. This *unanimity basis* is particularly useful for decomposing games and analyzing solution concepts like the Shapley value.

To express a game $v \in \mathcal{G}_N$ in terms of the unanimity basis $\mathcal{U}$, we need to determine the coefficients associated with each basis element. These coefficients are known as the *Harsanyi dividends* and capture the unique contribution of each coalition.

Definition 4.6 (Harsanyi Dividend)**.** Let $v \in \mathcal{G}_N$ be a TU-game. The *Harsanyi dividend* $\Delta_v(S)$ of a coalition $S \subseteq N$ represents the unique contribution of S that is not attributable to any of its strict subcoalitions.

The Harsanyi dividends can be computed in two equivalent ways:

- **Closed form:**
$$\Delta_v(S) = \sum_{T \subseteq S} (-1)^{|S|-|T|} \cdot v(T)$$
- **Recursive form:**
$$\Delta_v(\emptyset) = 0, \qquad \Delta_v(S) = v(S) - \sum_{R \subsetneq S} \Delta_v(R) \quad \text{for all } S \neq \emptyset$$

The Harsanyi dividends provide a natural way to decompose any game as a linear combination of unanimity games, as formalized in the following theorem.

Theorem 4.7 (Unanimity Basis Representation)**.** *Every TU-game $v \in \mathcal{G}_N$ can be written uniquely as a linear combination of unanimity games:*

$$v(T) = \sum_{\substack{S \subseteq N \\ S \neq \emptyset}} \Delta_v(S) \cdot u_S(T),$$

where $\Delta_v(S)$ denotes the Harsanyi dividend of coalition S.

A complete proof of this theorem can be found in [5].

4.3 The Core

The *core* is one of the central solution concepts in cooperative game theory. It captures the idea of stability: an allocation belongs to the core if no coalition has an incentive to deviate and act on its own. In cost games, this means that the total cost allocated to any coalition should be no more than what the coalition would incur by organizing its own tour or solution independently.

Formally, given a transferable utility game (N, v), an allocation $x \in \mathbb{R}^N$ is said to be in the *core* of the game if it satisfies:

- **Efficiency:** The total cost is fully allocated among the players:
$$x(N) = v(N),$$
- **Coalitional Rationality:**
$$x(S) \leq v(S) \quad \text{for all } S \subseteq N,$$
meaning that no coalition S would benefit by breaking away and bearing its own cost $v(S)$.

Definition 4.8. The *core* of the game is then defined as the set:
$$C(v) := \left\{ x \in \mathbb{R}^N \;\middle|\; x(N) = v(N),\ x(S) \leq v(S) \text{ for all } S \subseteq N \right\}.$$

In contrast to individual rationality, which only considers the incentives of individual players, the core strengthens this requirement by imposing rationality for all coalitions. It ensures that no group of players has a financial incentive to break away from the grand coalition, as each coalition is guaranteed an allocation no worse than its standalone cost. Because the core also satisfies the conditions of efficiency and individual rationality, it is naturally contained within the imputation set: $C(v) \subseteq I(v)$. However, the core of a game can be empty; there may exist no allocation that satisfies all coalitional constraints simultaneously, especially in cost games. A notable exception occurs in *convex games*, where the core is always guaranteed to be non-empty due to the increasing incentives for larger coalitions to form.

4.3.1 Existence of Core Imputations

The existence of core allocations in cooperative games is intimately tied to the concept of *balancedness*, which provides a necessary and sufficient condition for the non-emptiness of the core.

Definition 4.9. Let $N = \{1, 2, \ldots, n\}$ be the set of players. A map $\lambda : 2^N \setminus \{\emptyset\} \to \mathbb{R}_+ := \{t \in \mathbb{R} \mid t \geq 0\}$ is called a *balanced map* if
$$\sum_{S \in 2^N \setminus \{\emptyset\}} \lambda(S)\, e^S = e^N,$$
where $e^S \in \mathbb{R}^N$ is the characteristic vector of coalition S, with components defined by
$$e_i^S = \begin{cases} 1 & \text{if } i \in S, \\ 0 & \text{if } i \notin S. \end{cases}$$

Intuitively, a balanced map assigns weights to coalitions so that each player is covered with a total weight of exactly 1. This condition is expressed as:
$$\sum_{\substack{S \subseteq N \\ i \in S}} \lambda(S) = 1 \quad \text{for all } i \in N.$$

Definition 4.10. A collection $\mathcal{B} \subseteq 2^N \setminus \{\emptyset\}$ of nonempty coalitions is called a *balanced collection* if there exists a balanced map λ such that

$$\mathcal{B} = \{S \in 2^N \mid \lambda(S) > 0\}.$$

Definition 4.11. A game (N, v) is called a *balanced game* if for every balanced map

$$\lambda : 2^N \setminus \{\emptyset\} \to \mathbb{R}_+,$$

we have

$$\sum_{S \subseteq N} \lambda(S) v(S) \leq v(N).$$

This naturally leads to one of the central results in cooperative game theory:

Theorem 4.12 (Bondareva–Shapley)**.** *Let (N, v) be a TU-game. Then the following two assertions are equivalent:*

1. *$C(v) \neq \emptyset$,*
2. *(N, v) is a balanced game.*

A complete proof of this theorem can be found in [6] or [5].

This theorem provides both a necessary and sufficient condition for the existence of core allocations. In practice, verifying balancedness for arbitrary games can be computationally intensive due to the exponential number of coalitions. However, the result remains central to cooperative game theory and forms the theoretical basis for understanding why the core may be empty.

4.4 Relaxing the Core: The Least Core

In many cooperative games, especially cost games such as the Traveling Salesman Game, the core may be empty. According to the Bondareva–Shapley Theorem, this occurs when the game is not balanced, meaning that no allocation can simultaneously satisfy both efficiency and coalitional rationality for all coalitions. To address this issue, the concept of the *least core* provides a systematic relaxation of the core constraints. Specifically, we allow each coalition to be charged up to a small uniform excess $\varepsilon \geq 0$, replacing the strict condition $x(S) \leq v(S)$ with the relaxed version:

$$x(S) \leq v(S) + \varepsilon \quad \text{for all } S \subset N.$$

This defines a family of feasibility regions, one for each ε, and the goal is to find the smallest such ε for which a feasible allocation $x \in \mathbb{R}^N$ still satisfies efficiency. The resulting set of solutions defines the *least core*.

Definition 4.13. Let (N, v) be a TU-game. The *least core* of the game is the set of allocations

$$LC(v) := \left\{ x \in \mathbb{R}^N \;\middle|\; x(N) = v(N),\ x(S) \leq v(S) + \varepsilon^* \quad \forall S \subset N \right\},$$

where ε^* is the optimal value of the linear program:

$$\varepsilon^* = \min \left\{ \varepsilon \;\middle|\; \exists x \in \mathbb{R}^N :\ x(N) = v(N),\ x(S) \leq v(S) + \varepsilon \quad \forall S \subset N \right\}.$$

By construction, the least core always exists, since increasing ε sufficiently renders the system of inequalities feasible. The parameter ε^* quantifies the minimum level of dissatisfaction that must be tolerated to achieve an approximately stable allocation. When $\varepsilon^* = 0$, the least core coincides with the core, meaning the game is balanced.

The least core also plays a central role in the computation of the *nucleolus*. The nucleolus is defined as the unique allocation that lexicographically minimizes the vector of excesses over all coalitions. Its computation proceeds by solving a sequence of linear programs, the first of which is precisely the least core problem.

4.5 The Nucleolus

The nucleolus is a central solution concept in cooperative game theory that aims to identify the most equitable allocation among players. It does so by minimizing the dissatisfaction of the most dissatisfied coalitions in a lexicographic sense. Unlike the core, which may be large or even empty, the nucleolus always exists, is unique, and, if the core is non-empty, it lies within the core. To define the nucleolus, we first introduce the concept of a coalition's *excess*.

Definition 4.14. Let (N, v) be a cooperative game, and let $x \in \mathbb{R}^N$ be an allocation. The *excess* of coalition $S \subseteq N$ with respect to x is defined as:

$$e(S, x) = v(S) - x(S),$$

In cost games, the excess measures how much more the coalition would have to pay if it acted independently. A coalition is dissatisfied when the excess is negative, meaning it is being charged more than its standalone cost. The larger the excess, the more satisfied the coalition is with the allocation.

Definition 4.15 (Excess Vector)**.** Let (N, v) be a cooperative game and let $x \in \mathbb{R}^N$ be an allocation. The *excess vector* of x is the vector

$$\theta(x) = (e(S_1, x), e(S_2, x), \ldots, e(S_{2^n-2}, x)),$$

where each $e(S, x)$ is the *excess* of a non-empty proper coalition $S \subset N$, and the vector is sorted in *non-decreasing* order:

$$e(S_1, x) \leq e(S_2, x) \leq \cdots \leq e(S_{2^n-2}, x).$$

In order to define the nucleolus, we first need to introduce a specific ordering on excess vectors that allows us to compare allocations in terms of dissatisfaction.

Definition 4.16 (Lexicographic Ordering)**.** Let $a = (a_1, a_2, \ldots, a_m) \in \mathbb{R}^m$ and $b = (b_1, b_2, \ldots, b_m) \in \mathbb{R}^m$. We say that a is *lexicographically greater* than b, written $a \succ_{\text{lex}} b$, if there exists an index $k \in \{1, \ldots, m\}$ such that:

- $a_i = b_i$ for all $i < k$, and
- $a_k > b_k$.

Given the previous definitions of excess and lexicographic ordering, we can now define the *nucleolus* as the unique allocation that lexicographically minimizes the excess vector.

Definition 4.17 (Nucleolus)**.** Let (N, v) be a cooperative cost game, and let $\theta(x)$ denote the excess vector. The *nucleolus* of the game is the unique imputation $x^* \in \mathbb{R}^N$ that satisfies:

$$\theta(x^*) \succeq_{\text{lex}} \theta(y) \quad \text{for all imputations } y \in \mathbb{R}^N$$

In other words, the nucleolus maximizes the smallest excess and, subject to that, the second smallest, and so on, resulting in the most stable and least dissatisfying allocation. If the core is non-empty, the nucleolus lies within the core.

Computation via Successive Linear Programs. The nucleolus can be computed by solving a sequence of linear programs that iteratively refine the set of tight constraints. The process begins by solving the least core problem to maximize the minimum excess w_1. Coalitions for which the excess equals this maximum (identified via positive dual variables $\Pi_k^*(S) > 0$) are added to the active set Γ_k as equality constraints in the next iteration. This procedure is repeated until the solution becomes unique, yielding the nucleolus.

Algorithm 10 Computing the Nucleolus via Successive Linear Programs

1: Initialize $k := 1$, active constraint set $\Gamma_0 := \emptyset$
2: **repeat**
3: Solve the LP:

$$\begin{aligned} \max \quad & w_k \\ \text{s.t.} \quad & x(N) = v(N) \\ & x_i \le v(\{i\}) \quad \text{for all } i \in N \\ & x(S) + w_k \le v(S) \quad \text{for all } S \subset N \setminus \Gamma_{k-1} \\ & x(S) + w_k = v(S) \quad \text{for all } S \in \Gamma_{k-1} \end{aligned}$$

4: Let x_k be the solution and $\Pi_k^*(S)$ the dual variables
5: Define $\Gamma_k := \Gamma_{k-1} \cup \left\{ S \subset N \mid \Pi_k^*(S) > 0 \right\}$
6: $k := k + 1$
7: **until** The solution x_k is unique
8: **return** x_k

In this algorithm for computing the nucleolus, the *dual variables* $\Pi_k^*(S)$ correspond to the constraints $x(S) + w_k \le v(S)$ and measure how strongly each coalition S influences the current solution. A dual value $\Pi_k^*(S) > 0$ indicates that the constraint for S is *binding*, meaning its excess is equal to the current worst-case excess w_k. These coalitions are added to the active set Γ_k to ensure their excess remains fixed in future iterations. This process systematically identifies the most critical coalitions and refines the allocation until a unique solution is found, which will be the nucleolus.

This process is guaranteed to terminate in at most $2^n - 1$ steps.

4.6 The Shapley Value

Another prominent solution concept in cooperative game theory is the *Shapley value*. It provides a unique and fair allocation of the total value generated by a coalition of players based on their marginal contributions. Unlike the core or the nucleolus, the Shapley value is always well-defined and exists for every transferable utility (TU) game.

The key idea behind the Shapley value is that each player's payoff should reflect their average marginal contribution to all possible coalitions they can join. This makes the concept particularly appealing in settings where fairness and symmetry are important considerations.

Definition 4.18 (Shapley Value)**.** The *Shapley value* is the value function $\varphi : \mathcal{G}_N \to \mathbb{R}^N$, which

for every player $i \in N$ is given by

$$\varphi_i(v) = \sum_{\substack{S \subseteq N \\ i \in S}} \frac{\Delta_v(S)}{|S|},$$

where $\Delta_v(S)$ denotes the Harsanyi dividend of coalition S.

In contrast to the dividend-based formulation, the classical combinatorial form computes the Shapley value directly from the characteristic function by averaging marginal contributions. This representation is particularly useful for algorithmic and numerical computation.

Definition 4.19 (Shapley Value – Combinatorial Form)**.** Let (N, v) be a TU-game with $|N| = n$. The *Shapley value* of player $i \in N$ is defined as:

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(n - |S| - 1)!}{n!} \left(v(S \cup \{i\}) - v(S)\right),$$

Equivalently, the Shapley value can be expressed using binomial coefficients as:

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{1}{n} \cdot \binom{n-1}{|S|}^{-1} \left(v(S \cup \{i\}) - v(S)\right),$$

The term $v(S \cup \{i\}) - v(S)$ represents the *marginal contribution* of player i to coalition S, and the coefficient expresses the probability that S precedes i. The Shapley value can also be defined as a function that uses only the marginal contributions of player i as the arguments.

Definition 4.20 (Shapley Value – Marginalist Form)**.** Let (N, v) be a TU-game with $|N| = n$, and let S_n denote the set of all $n!$ permutations of the player set N. For a given permutation $\sigma \in S_n$, let P_i^σ be the set of players who precede player i in σ. Then the *Shapley value* of player $i \in N$ can be defined as:

$$\phi_i(v) = \frac{1}{n!} \sum_{\sigma \in S_n} \left[v(P_i^\sigma \cup \{i\}) - v(P_i^\sigma)\right].$$

This definition reflects the average *marginal contribution* of player i when players join the coalition in a random order.

4.6.1 Axiomatization of the Shapley Value

The Shapley value is uniquely characterized by four intuitive and independently necessary axioms. These axioms capture desirable fairness principles and provide a strong foundation for justifying the Shapley value as a solution concept for cooperative games. The formal axiomatization can be stated as follows:

- **Efficiency:** The total value generated by the grand coalition is fully distributed among the players:
$$\sum_{i \in N} \phi_i(v) = v(N).$$
- **Null-Player Property:** If player i contributes nothing to any coalition, then their allocation is zero:
$$v(S \cup \{i\}) = v(S) \text{ for all } S \subseteq N \setminus \{i\} \quad \Rightarrow \quad \phi_i(v) = 0.$$

- **Symmetry:** If two players contribute equally to all coalitions, then they receive equal payoffs:

$$v(S \cup \{i\}) = v(S \cup \{j\}) \text{ for all } S \subseteq N \setminus \{i, j\} \quad \Rightarrow \quad \phi_i(v) = \phi_j(v).$$

- **Additivity:** The Shapley value is linear in the characteristic function. For any two games $v, w \in \mathcal{G}_N$, and any player $i \in N$, it holds that:

$$\phi_i(v + w) = \phi_i(v) + \phi_i(w).$$

  Here, $\mathcal{G}_N$ denotes the set of all transferable utility (TU) games on the player set N, i.e., all set functions $v : 2^N \to \mathbb{R}$ such that $v(\emptyset) = 0$. The sum $(v + w)$ is defined: $(v + w)(S) = v(S) + w(S)$ for all $S \subseteq N$.

The fundamental result establishing the Shapley value is the following:

Theorem 4.21. *The Shapley value is the unique value function $\phi : \mathcal{G}_N \to \mathbb{R}^N$ that satisfies efficiency, the null-player property, symmetry, and additivity.*

A complete proof of this theorem can be found in [5] or [6].

In convex games, the Shapley value enjoys the additional property of always lying within the core, which is, in convex games, always non-empty, making it a robust solution concept when both fairness and stability are desired. While it is computationally feasible for small to medium-sized games, the factorial number of permutations involved renders it intractable for large player sets. In such cases, approximation algorithms or randomized sampling methods are often employed to estimate the Shapley value efficiently.

Similar to the TSP, many solution concepts in cooperative game theory are known to be NP-hard, NP-complete, or otherwise computationally intractable. For example, verifying whether a given payoff vector belongs to the core can be done via linear programming, but constructing the full set of core constraints involves an exponential number of coalitions, rendering general core computation NP-hard. The nucleolus, which selects the most equitable allocation by lexicographically minimizing the maximum excess among coalitions, is computationally even more demanding—it requires solving a sequence of linear programs, each with exponentially many constraints, and is classified as NP-hard in general cases. The Shapley value, while defined in closed form as an average of marginal contributions over all player permutations, also suffers from NP-Hard due to the factorial number of permutations or the exponential number of coalitions involved in its computation. However, unlike the core and nucleolus, the Shapley value lends itself well to Monte Carlo approximation techniques, which offer practical scalability and make it suitable for use in larger games where exact methods are infeasible.

5 Implementation of the Genetic Algorithm for TSP

This chapter presents a modular GA implementation in Python for solving the TSP. Designed for flexibility and experimentation, the solver supports configurable operators, dynamic parameters, and both serial and parallel execution via the *multiprocessing* module. A graphical interface built with *customtkinter* enables user-friendly configuration and real-time visualization.

The algorithm serves as the computational backbone for Shapley value approximation in cooperative games, discussed in the next chapter. Here, we detail its architecture, problem generation, operator logic, benchmarking setup, and key observations. For benchmarking purposes, we compared the GA-generated solutions to optimal tour lengths computed using the Concorde TSP Solver [9], with standard instances obtained from the TSPLIB library [10].

5.1 Software Architecture

The solver's architecture emphasizes the separation of concerns. Its components handle problem setup, configuration, evolutionary logic, and operator management independently, simplifying testing and future integration.

Core modules include: *TSPProblem* for instance definition, *GAConfig* for algorithm parameters, *tsp_solver* for execution logic (serial and parallel), and *pop_manager* and *operators* for genetic operations. Operators are registered dynamically, and parallelism is applied to fitness and crossover stages for scalability. The modular design facilitates reuse, including its role in cooperative game solution algorithms.

The following subsections describe each module in detail.

5.1.1 TSPProblem module

The *TSPProblem* class encapsulates the definition of a TSP instance. It supports two modes of problem specification: random generation of city coordinates within a bounded 2D grid, and loading of predefined coordinates from external sources.

This class also provides utility methods for exporting problem instances to standard formats, such as CSV and TSPLIB-compatible `.tsp` files. These features make it suitable for both internal benchmarking and external validation using exact solvers like Concorde. The set of cities is represented by integer indices, and the distance matrix is computed once and reused, ensuring efficiency.

By centralizing problem handling, *TSPProblem* simplifies downstream components that rely on consistent access to city coordinates and distances.

5.1.2 GAConfig module

The *GAConfig* class manages all parameters related to the GA run. It encapsulates configuration for population sizing, genetic operators, and termination criteria.

Population size can be determined dynamically based on problem size using one of several scaling modes: constant, linear, power, or factorial. Operator configuration includes the selection method (e.g., tournament, SUS), crossover operator (e.g., OX, PMX), mutation strategy, and survivor selection parameters. Termination conditions are set via maximum generation count and stagnation tolerance.

GAConfig centralizes all GA-related parameters and provides methods for updating and inspecting configurations. This design ensures consistency across modules and facilitates integration with both the GUI and automated benchmarking routines.

5.1.3 operators module

The *operators* module implements the core evolutionary operators used by the GA: selection, crossover, and mutation. Each category includes multiple strategies suitable for permutation-based representations of the TSP. The operators are implemented as standalone functions and registered in dictionaries for dynamic selection based on the configuration.

Supported selection methods include tournament selection, SUS, and roulette wheel selection. Crossover operators include OX, PMX, ERX, and CX. Mutation strategies include inversion, insertion, swap, and scramble mutation.

This structure simplifies experimentation with different operator combinations and allows for modular extension. Notably, an unintentionally modified version of OX used during early testing was observed to outperform the standard version by introducing greater diversity, highlighting the impact of operator behavior on search space exploration. Therefore all benchmarks will use this modified version of OX, which is described in detail at the end of this chapter.

5.1.4 pop_manager module

The *pop_manager* module implements all core population-level operations in the GA, including initialization, fitness evaluation, selection, crossover, mutation, and survivor filtering. It acts as an interface layer between the configuration and operator logic, handling population flow across generations.

Both serial and parallel versions of the main operations are provided. Parallelism is achieved using Python's `multiprocessing` module and a dynamic chunking strategy to distribute work evenly across processes. Fitness evaluation and crossover are the primary targets for parallel speedup, while other steps like selection remain serial due to their lower computational cost.

This separation of evolutionary flow into a dedicated module improves modularity and allows high-level solver routines to remain focused on orchestration rather than low-level mechanics.

5.1.5 tsp_solver module

The *tsp_solver* module manages the execution of the GA. It provides two main functions: `solve_tsp_ga_serial` and `solve_tsp_ga_parallel`, corresponding to sequential and parallel execution modes. Both follow the same evolutionary loop structure—initialization, evaluation, selection, crossover, mutation, survivor selection—guided by the configuration in *GAConfig*.

The parallel version uses a process pool and dynamic chunking to evaluate fitness and perform crossover concurrently. In contrast, the serial version avoids overhead for smaller problems. For very small instances, a brute-force exact solver based on permutations is used automatically.

This module serves as the central driver of the algorithm, coordinating the components defined in other modules to execute a full TSP-solving run.

5.1.6 User interface, tspGUI module

The *tspGUI* module provides a graphical interface for configuring and running the TSP GA. Built with *customtkinter*, it exposes all key parameters of the solver, including population size mode, selection, crossover, mutation type, and their respective rates. These inputs are located in the left configuration panel.

At the top, users can trigger key actions such as generating a random problem, loading a problem from a file, starting the solver in serial or parallel mode, and exporting data. The central area displays two real-time plots: the best tour found and the fitness progression across generations. The bottom pane serves as a terminal for logs, feedback, and final results.

This interface directly integrates with *TSPProblem*, *GAConfig*, and the solver routines. It improves usability, enforces parameter bounds, and facilitates interactive experimentation.

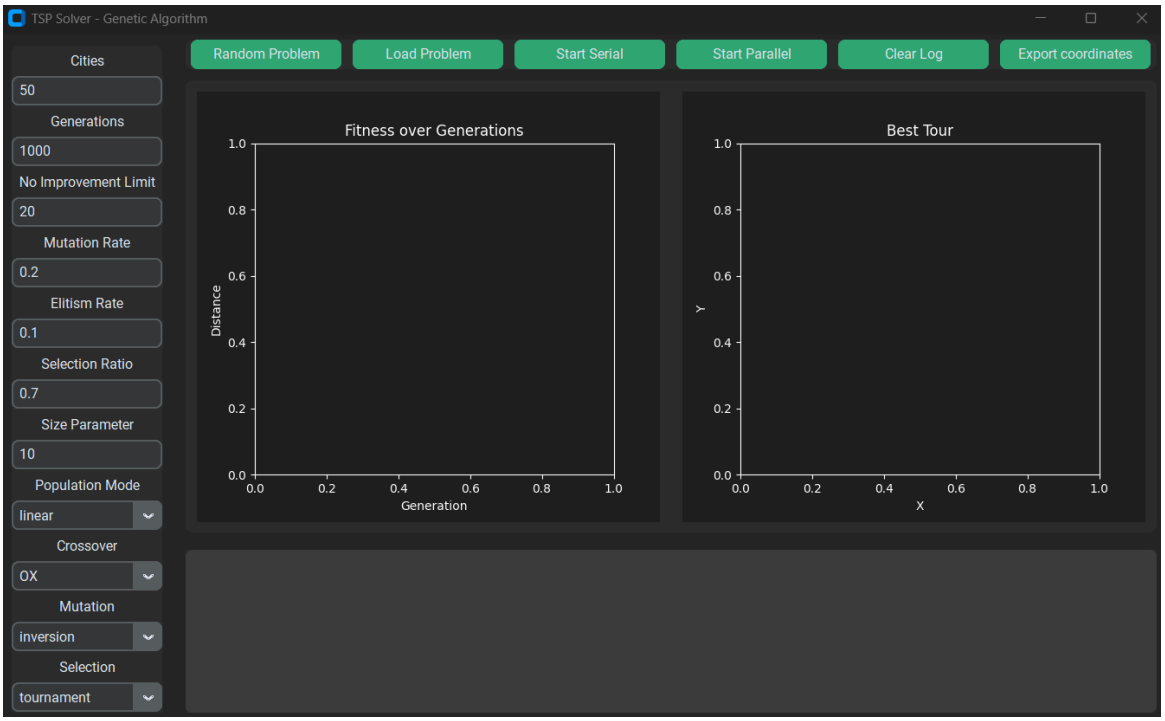


Figure 1: Graphical interface of the TSP Genetic Algorithm solver.

With the software architecture fully described, the next chapter evaluates its performance across diverse TSP instances and parameter settings, highlighting effective configurations and key observations.

5.2 Results and Benchmarking

This chapter presents a series of experiments designed to evaluate the performance and configurability of the implemented GA. The primary goals are to assess solution quality across diverse problem instances, identify effective combinations of operators and parameters, and analyze general trends in operator behavior.

Configurations are evaluated based on tour length, runtime, and deviation from known optima. Results are reported for individual configurations to reveal the overall impact of operators and tuning choices.

5.2.1 Experimental Setup

The solver was tested on two types of instances: randomly generated TSPs with 15 and 20 cities, and benchmark problems from TSPLIB (`eil51`, `eil76`, and `eil101`). The 15- and 20-city instances were created using uniformly distributed integer coordinates, generated using a fixed random seed (42); the code used for coordinate generation is shown below:

```
rng = Random(self.seed)
self.coordinates = [
    (rng.randint(0, self.size * 2), rng.randint(0, self.size * 2))
```

```
    for _ in range(self.size)
]
```

Listing 1: Uniform random coordinate generation used for the 15- and 20-city instances.

Only five core parameters were systematically varied across all experiments: selection method, selection ratio, crossover operator, mutation rate, and elitism rate. Other parameters, such as the mutation operator, population size mode, and tournament group size, were held constant, as preliminary testing showed their effect to be either negligible or detrimental to performance on the tested problems. The following snippet shows the base configuration template used to generate the full configuration space, with placeholders for parameters that were subject to testing:

```
BASE_CONFIG = {
    "mode": "linear",
    "size_parameter": 10.0,
    "max_pop": 10000,

    "selection_op": "---will be tested---",
    "selection_ratio": "---will be tested---",
    "tournament_k": 3,
    "crossover_op": "---will be tested---",
    "mutation_op": "inversion",
    "mutation_rate": "---will be tested---",

    "surv_strat": "elitism",
    "elitism_rate": "---will be tested---",

    "max_gen": 1000,
    "no_improvement_limit": 20
}
```

Listing 2: Base configuration template with placeholders.

For the 15-city, 20-city, and `eil51` instances, a total of 432 unique configurations were tested. These were generated as all combinations of the following parameter values:

```
CROSSOVERS = ["OX", "PMX", "CX"]
SELECTIONS = ["tournament", "sus", "roulette wheel"]
MUTATION_RATES = [0.1, 0.15, 0.2]
SELECTION_RATES = [0.5, 0.6, 0.7, 0.8]
ELITISM_RATES = [0.05, 0.1, 0.15, 0.2]
```

Listing 3: Parameter grid used for 15-city, 20-city, and `eil51` instances.

Due to the increased runtime complexity of larger instances, only a subset of configurations was evaluated on `eil76` and `eil101`. These were selected based on the best-performing combinations observed on `eil51`, and included all 32 configurations of:

```
CROSSOVERS = ["OX"]
SELECTIONS = ["tournament"]
MUTATION_RATES = [0.15, 0.2]
SELECTION_RATES = [0.5, 0.6, 0.7, 0.8]
ELITISM_RATES = [0.05, 0.1, 0.15, 0.2]
```

Listing 4: Reduced configuration grid used for `eil76` and `eil101` instances.

Each configuration was executed 100 times to account for stochastic variability. Metrics collected were final tour length, and elapsed time. The following sections present and analyze the results of these experiments.

5.2.2 Top Configuration Performance results

This section presents the best-performing configurations identified across the three smaller problem instances: the 15-city and 20-city randomly generated problems, and the TSPLIB benchmark `eil51`. For each instance, all 432 configurations were evaluated, and the top 10 are shown based on average tour length over multiple runs. These results highlight consistent trends in effective operator combinations and parameter settings. Both the standard deviation and the percentage above the optimum are calculated with respect to the mean tour length over 100 runs.

Table 3: Top 10 configurations for the 15-city problem. Optimal length = 99.6

Crossover	Selection	Mut. Rate	Sel. Rate	Elit. Rate	Mean Length	Std Length	% Above Opt.	Time (s)
OX	tournament	0.20	0.5	0.05	99.62	0.10	0.022	0.106
OX	tournament	0.20	0.6	0.15	99.62	0.10	0.022	0.098
OX	tournament	0.20	0.7	0.10	99.65	0.15	0.045	0.104
OX	tournament	0.15	0.8	0.05	99.65	0.22	0.047	0.112
OX	tournament	0.20	0.7	0.05	99.65	0.16	0.051	0.113
OX	tournament	0.15	0.5	0.15	99.66	0.21	0.056	0.092
OX	tournament	0.20	0.7	0.15	99.66	0.23	0.059	0.099
PMX	tournament	0.20	0.7	0.10	99.66	0.17	0.062	0.089
PMX	tournament	0.20	0.5	0.20	99.66	0.17	0.062	0.110
OX	tournament	0.20	0.8	0.05	99.66	0.24	0.065	0.109

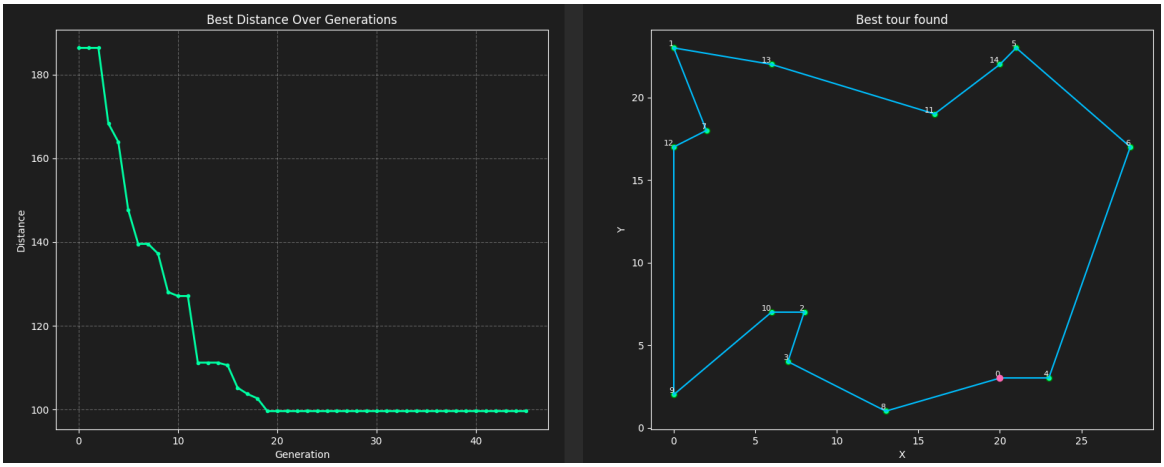


Figure 2: Optimal tour for custom 15-city problem.

Table 4: Top 10 configurations for the 20-city problem. Optimal length = 148.63

Crossover	Selection	Mut. Rate	Sel. Rate	Elit. Rate	Mean Length	Std Length	% Above Opt.	Time (s)
OX	tournament	0.20	0.7	0.15	149.22	1.53	0.82	0.210
OX	tournament	0.20	0.8	0.15	149.34	2.13	0.90	0.209
OX	tournament	0.20	0.7	0.10	149.35	1.59	0.91	0.219
OX	tournament	0.20	0.8	0.10	149.41	1.64	0.95	0.230
OX	tournament	0.20	0.6	0.10	149.47	2.29	1.00	0.182
OX	tournament	0.20	0.6	0.15	149.50	1.96	1.02	0.201
OX	tournament	0.20	0.8	0.05	149.60	2.05	1.08	0.230
OX	tournament	0.15	0.7	0.05	149.61	2.19	1.09	0.230
PMX	tournament	0.20	0.8	0.10	149.63	2.43	1.10	0.256
OX	tournament	0.15	0.7	0.10	149.64	2.49	1.11	0.226

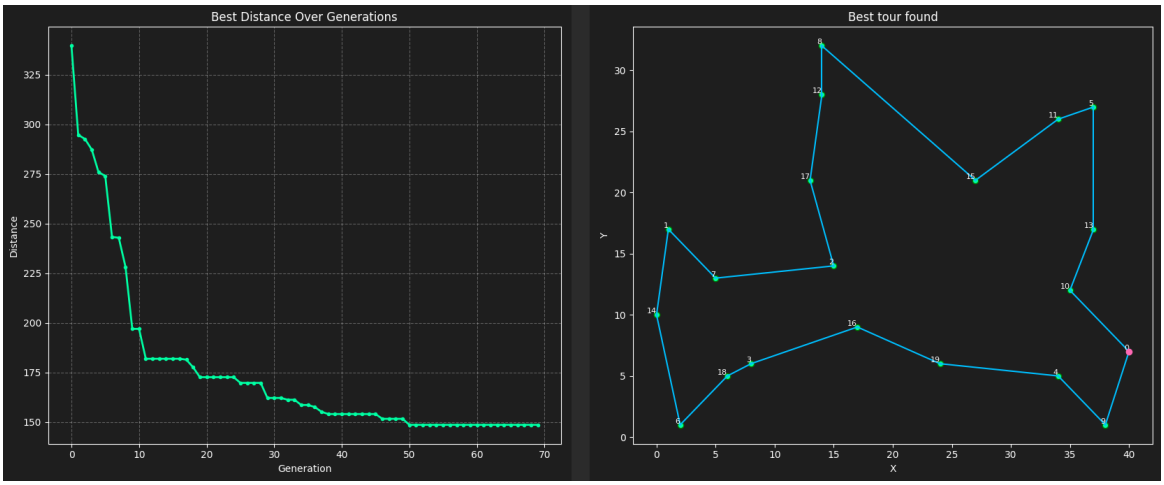


Figure 3: Optimal tour for custom 20-city problem.

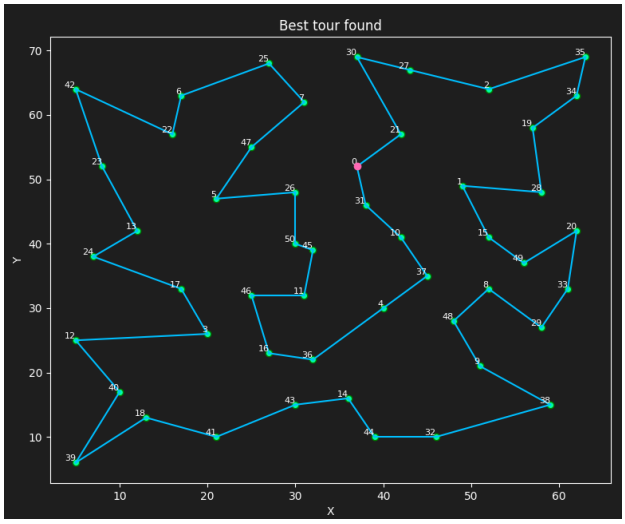
While the 15- and 20-city problems are small enough that most top configurations reliably reach or nearly reach the optimal solution, rerunning them a few times is often sufficient to obtain the exact optimum. The algorithm's behavior in these instances is, therefore, quite stable and predictable.

The `eil51` instance presents a significantly larger search space, where even strong configurations show increased variability and average tour lengths that are around 6.5–7.5% above the known optimum. The table below shows the best configurations found for this benchmark.

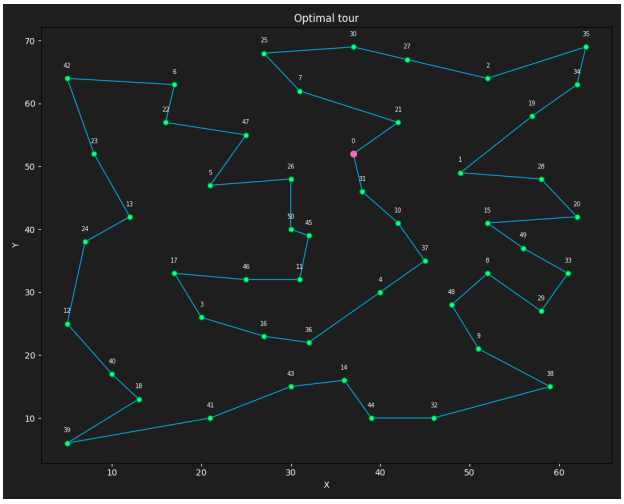
Table 5: Top 10 configurations for `eil51`. Optimal length = 426.0

Crossover	Selection	Mut. Rate	Sel. Rate	Elit. Rate	Mean Length	Std Length	% Above Opt.	Time (s)
OX	tournament	0.20	0.7	0.10	454.44	9.96	6.68	2.48
OX	tournament	0.20	0.8	0.15	455.76	9.98	6.99	2.31
OX	tournament	0.15	0.7	0.05	455.95	10.33	7.03	2.68
OX	tournament	0.20	0.6	0.20	455.97	9.41	7.03	2.26
OX	tournament	0.20	0.5	0.05	456.02	9.94	7.05	2.62
OX	tournament	0.20	0.6	0.05	456.24	9.80	7.10	2.59
OX	tournament	0.20	0.7	0.15	456.29	10.15	7.11	2.37
OX	tournament	0.20	0.7	0.05	456.42	9.66	7.14	2.60
OX	tournament	0.20	0.6	0.10	456.52	10.18	7.16	2.23
OX	tournament	0.20	0.7	0.20	456.53	10.71	7.17	2.29

Although the GA consistently finds high-quality tours for `eil51`, it does not guarantee convergence to the global optimum. Across all tested configurations and repeated runs, the optimal tour of length 426.0 was never reached. While structurally similar in some regions, the GA solutions tend to include minor detours or suboptimal crossings that accumulate into measurable deviations from the best-known path.



Best tour found by GA for `eil51`
Tour length = 438.52



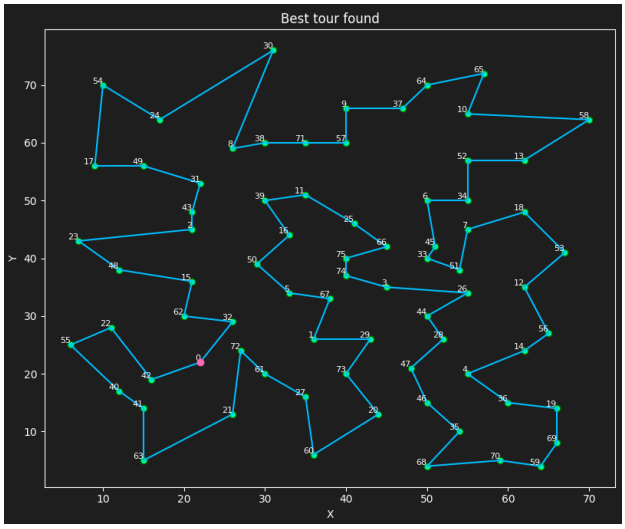
Optimal tour for `eil51`
Tour length = 426.0

Figure 4: Comparison of the best GA solution and the optimal tour for `eil51`.

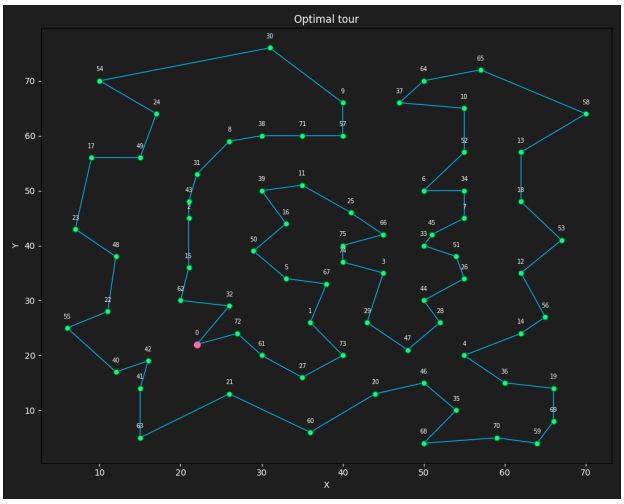
To evaluate the scalability of the GA, we tested it on two larger TSPLIB instances: `eil76` and `eil101`. These configurations were selected based on the best-performing parameter combinations from the earlier experiments on `eil51`. The following tables show the top 10 configurations per instance, all using modified OX crossover and tournament selection:

Table 6: Top 10 configurations for `eil76`. Optimal length = 538.0

Mut. Rate	Sel. Rate	Elit. Rate	Mean Length	Std Length	% Above Opt.	Time (s)
0.20	0.8	0.10	589.61	14.36	9.59	8.61
0.20	0.7	0.05	590.10	16.85	9.68	9.68
0.20	0.7	0.20	590.12	13.10	9.69	8.54
0.20	0.6	0.10	590.31	12.84	9.72	9.23
0.20	0.6	0.05	590.39	14.98	9.74	9.97
0.20	0.5	0.05	591.18	15.38	9.88	9.85
0.20	0.5	0.15	591.21	14.31	9.89	8.82
0.20	0.7	0.15	591.24	15.34	9.90	8.61
0.20	0.6	0.20	591.43	15.56	9.93	8.57
0.20	0.7	0.10	591.50	13.69	9.95	9.22



Best tour found by GA for `eil76`
Tour length = 563.71

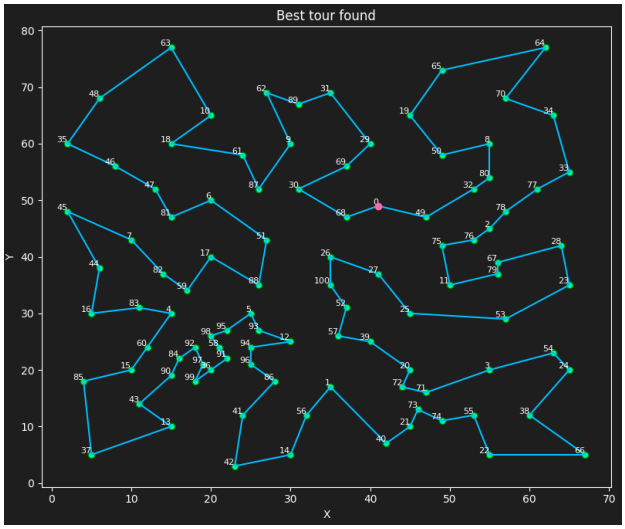


Optimal tour for `eil76`
Tour length = 538.0

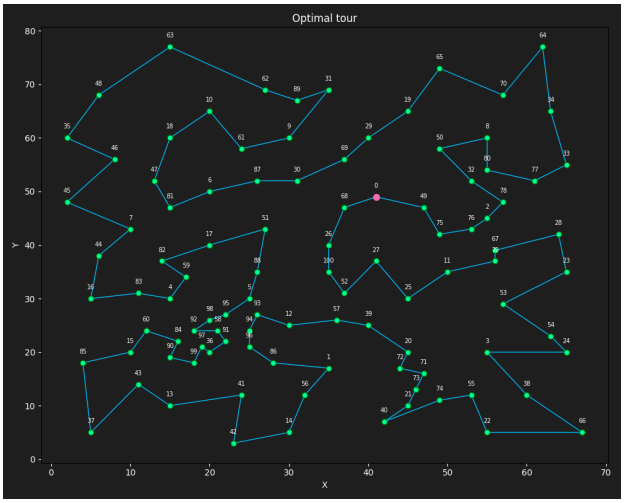
Figure 5: Comparison of the best GA solution and the optimal tour for `eil76`.

Table 7: Top 10 configurations for `eil101`. Optimal length = 629.0

Mut. Rate	Sel. Rate	Elit. Rate	Mean Length	Std Length	% Above Opt.	Time (s)
0.20	0.7	0.15	699.34	17.02	11.18	19.71
0.20	0.6	0.10	700.57	15.40	11.38	21.19
0.20	0.5	0.10	701.30	13.43	11.49	20.92
0.20	0.5	0.05	701.41	16.16	11.51	22.50
0.20	0.8	0.05	702.25	14.23	11.65	21.67
0.20	0.7	0.05	702.35	16.52	11.66	22.00
0.20	0.8	0.20	702.37	17.23	11.67	18.37
0.20	0.8	0.10	702.40	15.05	11.67	19.16
0.20	0.6	0.05	702.76	16.38	11.73	22.38
0.20	0.5	0.15	704.09	17.38	11.94	19.72



Best tour found by GA for `eil101`
Tour length = 661.3



Optimal tour for `eil101`
Tour length = 629.0

Figure 6: Comparison of the best GA solution and the optimal tour for `eil101`.

Beyond identifying strong parameter combinations, several key observations emerged during the development and testing of the GA. Most notably, increasing population size beyond a linear scaling showed no improvement in solution quality. A linear growth mode with a size parameter of 10 (e.g., population size 510 for `eil51`) was found to be both efficient and sufficient for producing competitive results. Among the mutation operators, inversion mutation consistently outperformed alternatives, reinforcing its suitability for permutation-based problems like the TSP. Additionally, a termination condition based on 20 generations without improvement proved to be an effective stopping criterion, and setting the tournament group size parameter $k = 3$ reliably produced strong results across all instances.

The following subsection shifts focus from individual configurations to a broader examination of operator and parameter performance. Using the same experimental data, we now analyze how each genetic operator and key parameter influenced outcomes across all tested configurations.

5.2.3 Operator and Parameter Performance Analysis

While the previous sections focused on identifying the best-performing configurations, this part analyzes the performance of individual operators and parameter values in aggregate. The goal is to assess which operators consistently contributed to strong solutions regardless of their specific configuration context.

For each selection method, crossover operator, mutation rate, selection ratio, and elitism rate, we compute the average tour length and relative performance across all configurations in which they appeared. This provides a more global view of operator effectiveness.

Table 8: Operator and parameter performance ranking for the 15-city problem. Optimal length = 99.6

Rank	Operator	Mean Tour Length	Std Dev	% Above Opt.	Time (s)
		Crossover			
1	OX	100.34	0.78	0.75	0.110
2	CX	100.64	0.73	1.04	0.101
3	PMX	100.64	0.85	1.05	0.134
		Selection			
1	tournament	100.06	0.38	0.46	0.102
2	sus	100.55	0.61	0.95	0.118
3	roulette wheel	101.02	0.97	1.43	0.125
		Mutation Rate			
1	0.2	100.22	0.63	0.62	0.116
2	0.15	100.43	0.70	0.83	0.115
3	0.1	100.98	0.85	1.38	0.114
		Selection Rate			
1	0.8	100.47	0.74	0.88	0.113
2	0.7	100.52	0.75	0.92	0.114
3	0.6	100.55	0.83	0.95	0.116
4	0.5	100.62	0.87	1.03	0.117
		Elitism Rate			
1	0.2	100.33	0.52	0.74	0.105
2	0.15	100.35	0.49	0.75	0.110
3	0.1	100.43	0.60	0.83	0.117
4	0.05	101.06	1.16	1.46	0.128

Table 9: Operator and parameter performance ranking for the 20-city problem. Optimal length = 148.63

Rank	Operator	Mean Tour Length	Std Dev	% Above Opt.	Time (s)
		Crossover			
1	OX	152.75	3.04	2.78	0.254
2	PMX	153.20	2.19	3.08	0.302
3	CX	153.24	2.11	3.11	0.223
		Selection			
1	tournament	151.03	1.10	1.62	0.218
2	sus	153.24	1.55	3.11	0.270
3	roulette wheel	154.91	2.73	4.24	0.291
		Mutation Rate			
1	0.2	152.21	2.47	2.42	0.264
2	0.15	152.82	2.25	2.82	0.268
3	0.1	154.16	2.35	3.73	0.247
		Selection Rate			
1	0.8	152.92	2.48	2.89	0.261
2	0.7	152.94	2.37	2.91	0.257
3	0.6	153.11	2.43	3.02	0.258
4	0.5	153.28	2.69	3.14	0.262
		Elitism Rate			
1	0.2	152.33	1.43	2.50	0.237
2	0.15	152.40	1.59	2.54	0.249
3	0.1	152.86	1.96	2.85	0.263
4	0.05	154.66	3.60	4.07	0.290

Table 10: Operator and parameter performance ranking for `eil51`. Optimal length = 426.0

Rank	Operator	Mean Tour Length	Std Dev	% Above Opt.	Time (s)
		Crossover			
1	CX	473.78	11.24	11.22	3.10
2	PMX	485.04	35.35	13.86	4.65
3	OX	496.41	65.11	16.53	3.37
		Selection			
1	tournament	461.42	3.73	8.31	2.69
2	sus	474.73	8.49	11.44	4.06
3	roulette wheel	519.09	62.86	21.85	4.37
		Mutation Rate			
1	0.15	483.20	42.95	13.43	3.74
2	0.2	485.24	52.11	13.91	3.64
3	0.1	486.79	36.16	14.27	3.74
		Selection Rate			
1	0.7	483.46	41.78	13.49	3.69
2	0.8	484.36	44.26	13.70	3.65
3	0.6	485.77	45.67	14.03	3.73
4	0.5	486.73	45.31	14.25	3.76
		Elitism Rate			
1	0.2	473.41	13.52	11.13	3.39
2	0.15	475.26	16.81	11.56	3.58
3	0.1	479.84	23.63	12.64	3.83
4	0.05	511.81	76.44	20.14	4.02

While Order Crossover (OX) performed best on smaller instances, its ranking dropped significantly in the `eil51` operator analysis. This lower ranking is not necessarily indicative of poor overall performance, but rather a consequence of its broad variance: OX was used in a large number of both top- and bottom-performing configurations. As a result, its average performance was diluted by many poorly tuned combinations. In contrast, Cycle Crossover (CX), though generally weaker, appeared more consistently in mid-range configurations, avoiding extreme outliers. This centralized clustering led to a better average tour length, placing it at the top of the operator rankings for `eil51` despite its inferior best-case performance.

Table 11: Operator performance ranking for `eil76`. Optimal length = 538.0

Rank	Operator	Mean Tour Length	Std Dev	% Above Opt.	Time (s)
		Mutation Rate			
1	0.2	591.29	1.01	9.91	9.00
2	0.15	595.06	1.69	10.61	8.89
		Selection Rate			
1	0.8	592.76	1.72	10.18	8.98
2	0.6	592.81	2.54	10.19	8.98
3	0.7	593.39	2.96	10.30	9.01
4	0.5	593.75	2.37	10.36	8.81
		Elitism Rate			
1	0.05	592.40	1.99	10.11	9.72
2	0.1	592.76	2.02	10.18	9.04
3	0.15	593.46	2.35	10.31	8.66
4	0.2	594.08	3.01	10.42	8.36

Table 12: Operator performance ranking for `eil101`. Optimal length = 629.0

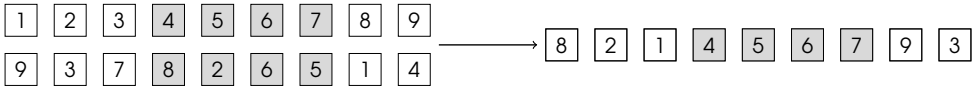
Rank	Operator	Mean Tour Length	Std Dev	% Above Opt.	Time (s)
		Mutation Rate			
1	0.2	703.05	1.87	11.77	20.23
2	0.15	706.81	1.59	12.37	20.20
		Selection Rate			
1	0.6	704.47	1.96	12.00	20.34
2	0.5	704.80	2.59	12.05	20.37
3	0.8	705.10	2.72	12.10	19.93
4	0.7	705.33	3.25	12.13	20.24
		Elitism Rate			
1	0.1	703.84	2.14	11.90	20.45
2	0.05	704.47	2.59	12.00	21.93
3	0.2	705.52	1.94	12.17	18.98
4	0.15	705.87	3.31	12.22	19.51

Based on the results of both ranking-based and aggregated operator performance tables, several consistent patterns emerged. Across all tested instances, the modified OX operator, tournament selection, and a mutation rate of 0.2 repeatedly demonstrated superior performance, making them the most robust choices among the configurations tested. For selection rate, higher values generally correlated with better outcomes. As for the elitism rate, a value of 0.2 appeared to be optimal for smaller instances (up to `eil51`). However, on larger problems like `eil76` and `eil101`, lower elitism rates began to outperform this setting, possibly highlighting a broader trend.

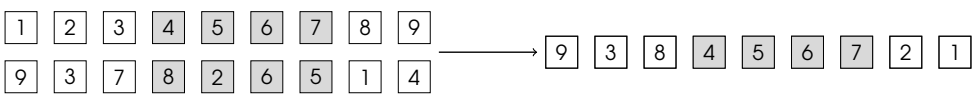
5.3 Additional observations

Although the ERX operator exhibited interesting structural preservation properties, it was excluded from the benchmark due to its significantly slower performance and unnoticeable advantage in solution quality. This was primarily caused by the overhead of maintaining an edge table during execution and its design, which limits it to producing only a single offspring per application, making it inefficient for even small populations compared to other crossover operators.

Interestingly, a performance improvement was also observed in a modified version of the OX operator that emerged unintentionally. Instead of filling the remaining positions in the offspring starting from the second cut-off point, as specified in the standard OX procedure, the implementation mistakenly began inserting from the first available position in the child. This deviation increased the structural distance between offspring and parents, likely enhancing the exploration of the search space and resulting in consistently better performance across multiple instances.



Standard OX procedure



A modified OX procedure

6 Implementation of the Shapley Value Solver for TSP

This chapter presents the implementation of a Shapley value solver as an extension of the previously developed GA framework for the TSP. The Shapley value, a central concept in cooperative game theory, offers a principled method for allocating the total tour cost among cities based on their marginal contributions to different coalitions [11, 12].

Due to the combinatorial explosion associated with enumerating all player orderings, the solver employs a Monte Carlo sampling approach, which estimates expected marginal contributions using randomly generated permutations and marginal formula 4.20. This estimator possesses key statistical properties: it is unbiased, consistent, and—by the Central Limit Theorem—asymptotically normal, given that marginal contributions are finite and independently sampled. These properties were verified through empirical testing, confirming the statistical validity of the estimation method used.

The approximation is seamlessly integrated into the GA infrastructure, reusing the same evaluation and configuration components. A GUI structurally similar to that of the TSP solver enables interactive configuration and execution of the Shapley computation. The remainder of the chapter details the architectural structure of the module, presents performance benchmarks, and concludes with a discussion of observed limitations and directions for further improvement.

6.1 Software Architecture

The Shapley value solver is implemented as a compact extension module built atop the TSP solver. Its core logic builds directly upon the existing problem configuration and evaluation mechanisms, focusing primarily on sampling and coalition evaluation.

This section outlines the structure of the module. We begin with the sampling strategy implemented via the `generate_samples()` and `generate_all_permutations()` functions. We then describe the serial Shapley computation, including the `solve_Shapley()` method and the auxiliary caching function `get_or_compute_tsp()`. Finally, we examine the `solve_Shapley_prl()` function, which provides a parallelized wrapper around the core computation, significantly accelerating large-scale approximations.

6.1.1 Sampling Methods

Due to the factorial growth of possible orderings, the implementation supports two sampling approaches: a Monte Carlo sampling method for approximation and an exhaustive enumeration method for small instances where exact computation is feasible.

The primary approach used for approximating the Shapley value is random sampling of permutations. This method fixes the first city (typically city 0) and randomly permutes the remaining cities. The number of samples is limited both by the user's specification and a hard cap to avoid excessive memory usage.

```
def generate_samples(size: int, num_samples: int, fixed_first: int = 0):
    base_elements = tuple(i for i in range(size) if i != fixed_first)
    num_samples = min(num_samples, min(factorial(size - 1), 10000))
    samples = set()
    while len(samples) < num_samples:
        perm = (fixed_first,) + tuple(sample(base_elements, len(
            base_elements)))
        samples.add(perm)
    return samples
```

Listing 5: Monte Carlo sampling of city permutations

Since the set of permutations is randomly generated, results are subject to stochastic variability, which decreases with increasing sample size.

For small problem instances or benchmarking, it is possible to compute the exact Shapley value by enumerating all permutations of the cities. The method below uses Python's `itertools.permutations` to generate all possible orderings with the first city fixed.

```
from itertools import permutations

def generate_all_permutations(size: int, fixed_first: int = 0):
    base_elements = [i for i in range(size) if i != fixed_first]
    all_perms = [(fixed_first,) + p for p in permutations(base_elements)]
    return all_perms
```

Listing 6: Full permutation generation for exact computation

This approach is computationally feasible only for small values of n, as the number of permutations grows as $(n-1)!$. It is primarily used for benchmarking and verifying the accuracy of the Monte Carlo method.

6.1.2 Shapley Solver

Initialization Before the Core Loop: The function `solve_Shapley` begins by preparing the data structures required for marginal contribution tracking, permutation sampling, and coalition cost caching. It also computes the tour length of the grand coalition:

```
marginalContributions = {i: [] for i in range(1, problem.size)}
samples = generate_samples(problem.size, num_samples)
stored_tours = {}
grand_set = [i for i in range(problem.size)]
key = frozenset(grand_set)
tspLen, tspTour, _ = solve_tsp_ga_serial(problem, config, grand_set, False)
stored_tours[key] = tspLen
```

Listing 7: Initialization phase of `solve_Shapley`

Core Computation Loop: The core logic of Shapley value estimation is shown below. Each permutation is interpreted as a sequence of coalition building, and the cost increase when each city joins is recorded as its marginal contribution.

```
for sample in samples:
    cost_without = 0.0
    for i in range(1, problem.size):
        cost_with = get_or_compute_tsp(problem, config, list(sample[:i+1]),
            stored_tours, store_limit)
        marginal_contribution = cost_with - cost_without
        cost_without = cost_with
        marginalContributions[sample[i]].append(marginal_contribution)

shapley_values = {
    key: round(sum(marginalContributions[key]) / len(samples), 3)
    for key in marginalContributions}
```

Listing 8: Main computation loop in `solve_Shapley`

To avoid redundant computations of tour costs, the implementation relies on a caching mechanism managed by the auxiliary function `get_or_compute_tsp()`, which either retrieves a stored tour length for a coalition or computes it if it has not been previously evaluated.

Caching Coalition Costs: The following auxiliary function manages caching of previously computed TSP tour lengths for specific coalitions, reducing the overall number of TSP evaluations and improving performance.

```
def get_or_compute_tsp(problem, config, subset_with, stored_tours,
    store_limit):
    key = frozenset(subset_with)
    if key in stored_tours:
        return stored_tours[key]
    tspLen, _, _ = solve_tsp_ga_serial(problem, config, subset_with, False)
    if len(subset_with) >= 9 and len(stored_tours) < store_limit:
        stored_tours[key] = tspLen
    return tspLen
```

Listing 9: Caching previously evaluated coalition costs

The cache is indexed using `frozenset` representations of the coalitions and is size-limited to manage memory usage. Coalitions of size nine or greater are stored, as these tend to be more computationally expensive to evaluate.

6.1.3 Parallel Computation Wrapper

To accelerate the approximation of Shapley values, especially for larger TSP instances or higher sample counts, a parallel wrapper `solve_Shapley_prl()` is provided. This function distributes the sampling workload across multiple CPU cores using Python's `multiprocessing` module. Each process runs an independent instance of the serial solver and returns partial Shapley estimates, which are then aggregated.

```
num_processes = max(1, cpu_count() - 1)
with Pool(processes=num_processes) as pool:
    args = [(problem, config, num_samples, store_limit, i==num_processes-1)
            for i in range(num_processes)]
    results = pool.starmap(solve_Shapley, args)
```

Listing 10: Structure of `solve_Shapley_prl`

Each process computes Shapley values independently using a disjoint set of random samples. When all processes are complete, their outputs are combined and averaged:

```
combined_shapley_values = {
    key: round(sum(values) / num_processes, 3)
    for key, values in extract_by_key.items()
}
```

Listing 11: Combining results from all threads

The averaged Shapley values are scaled to match the length of the shortest grand tour found among all threads. This ensures that their sum corresponds exactly to the best solution discovered during parallel computation, preserving proportional cost allocation relative to the most accurate tour available.

```
adjusted_shapley_values = {
    key: round((combined_shapley_values[key] / average_grand_tour_length)
                * shortest_grand_tour_length, 3)
    for key in combined_shapley_values
}
```

Listing 12: Final adjustment of averaged Shapley values

6.2 Results and Benchmarking

This chapter benchmarks the implemented Shapley value solver on two TSP-based test cases. The first uses a fully exact setup on a 9-city problem to evaluate two estimation methods against a ground truth. The second leverages a near-exact combinatorial Shapley computation on a 15-city instance—made possible by the high accuracy of our GA-based TSP solver—to evaluate the performance of the permutation-based Shapley approximation.

6.2.1 Experimental Setup

To evaluate the performance and accuracy of the Shapley value solver, two benchmark problems were used: a fully exact 9-city instance and a larger 15-city instance evaluated via a near-exact combinatorial method. Both problems were generated using:

```
rng = Random(self.seed)
self.coordinates = [
    (rng.randint(0, self.size * 2), rng.randint(0, self.size * 2))
    for _ in range(self.size)
]
```

Listing 13: Uniform random coordinate generation used for the 9- and 15-city instances.

9-City Benchmark (Exact Computation)
In the first benchmark, a custom instance with nine cities was created. One city was fixed as a depot, and the remaining eight cities were used to compute the Shapley value, reducing the permutation space from 9! to 8!.

The following three estimation methods were evaluated:

- **Exact solution** — full enumeration of the Shapley formula using 8! permutations, with coalition costs computed using exact TSP solutions.
- **Subset-based approximation** — 100 random permutations, with exact TSP costs computed for each sampled coalition.
- **GA-based approximation** — 100 permutations, with coalition costs approximated using a genetic algorithm.

Each approximate method was repeated 100 times to evaluate consistency and compute statistical error metrics.

15-City Benchmark (Near-Exact Computation)
The second benchmark used a 15-city TSP problem to assess the scalability of Shapley value computation based on the combinatorial formula from definition 4.19.

As in the previous case, one city was fixed as the depot, reducing the effective set of players from 15 to 14. The characteristic function was evaluated over all 2^{14} subsets, with TSP costs for

each coalition computed using a genetic algorithm. This provides a near-exact Shapley vector, serving as a practical reference solution given the solver's observed high accuracy on problems with 20 or fewer cities, as demonstrated in the previous chapter.

However, this approach does not scale well to larger instances due to exponential memory and disk requirements associated with storing coalition tour costs. In contrast, the approximation algorithm based on permutation sampling enables efficient control over memory usage by limiting the number of cached tours per thread. In the extreme case, no intermediate results need to be stored at all, allowing for scalable, low-footprint execution.

6.2.2 Evaluation Metrics

To evaluate the quality of Shapley value approximations, each estimated vector was compared to the ground truth using standard vector-based error metrics, as inspired by [11]. Let $\phi^* = (\phi_1^*, \phi_2^*, \ldots, \phi_n^*)$ denote the exact Shapley value vector, and let $\hat{\phi}^{(r)} = (\hat{\phi}_1^{(r)}, \hat{\phi}_2^{(r)}, \ldots, \hat{\phi}_n^{(r)})$ denote the approximation obtained in the r-th run. The following error metrics were computed for each run:

- **Mean Absolute Error (MAE):**

$$\mathrm{MAE}^{(r)} = \frac{1}{n} \sum_{i=1}^{n} \left| \hat{\phi}_i^{(r)} - \phi_i^* \right|$$

- **Mean Squared Error (MSE):**

$$\mathrm{MSE}^{(r)} = \frac{1}{n} \sum_{i=1}^{n} \left(\hat{\phi}_i^{(r)} - \phi_i^* \right)^2$$

- **Root Mean Squared Error (RMSE):**

$$\mathrm{RMSE}^{(r)} = \sqrt{\mathrm{MSE}^{(r)}}$$

- **Maximum Absolute Error (Max Error):**

$$\mathrm{MaxError}^{(r)} = \max_{i=1,\ldots,n} \left| \hat{\phi}_i^{(r)} - \phi_i^* \right|$$

- **Mean Relative Error (Percent Error):**

$$\mathrm{Percent}^{(r)} = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{\hat{\phi}_i^{(r)} - \phi_i^*}{\phi_i^*} \right| \cdot 100$$

For each method, its mean and standard deviation across all runs were computed to assess both accuracy and consistency.

6.2.3 Experimental Results

The following section presents the results of the Shapley value solver benchmarks, based on the two test cases introduced earlier, where a *parallel wrapper* was used for testing. This wrapper executed 11 independent single-threaded solver instances in parallel—corresponding to the number of available CPU cores—and merged their outputs into a single Shapley value estimate using the aggregation method described in the previous section. This process was repeated 30 times for

each estimation method and problem instance, resulting in a total of 30 parallel estimates and 330 single-threaded solutions per method. Each individual thread used a sample size of 100 permutations. Therefore, each parallel estimate was based on a combined total of 1100 sampled permutations. This setup allowed us to evaluate both the consistency of single-threaded runs and the quality of the final aggregated result while maintaining tractable runtime.

For the 9-city instance, the total tour length was 59.62 and the resulting Shapley values are shown under Figure 7. For the 15-city instance, the tour length was 99.60 and the resulting Shapley values are shown under Figure 8.

Both Tables 13 and 14 show that the parallel version significantly improves accuracy and reduces variability compared to the single-threaded approach. The elapsed time in the Single-thread column is not reported, as parallel computation does not provide timing for individual threads. While exact TSP evaluation yields the lowest error overall, the genetic algorithm achieves comparable accuracy.

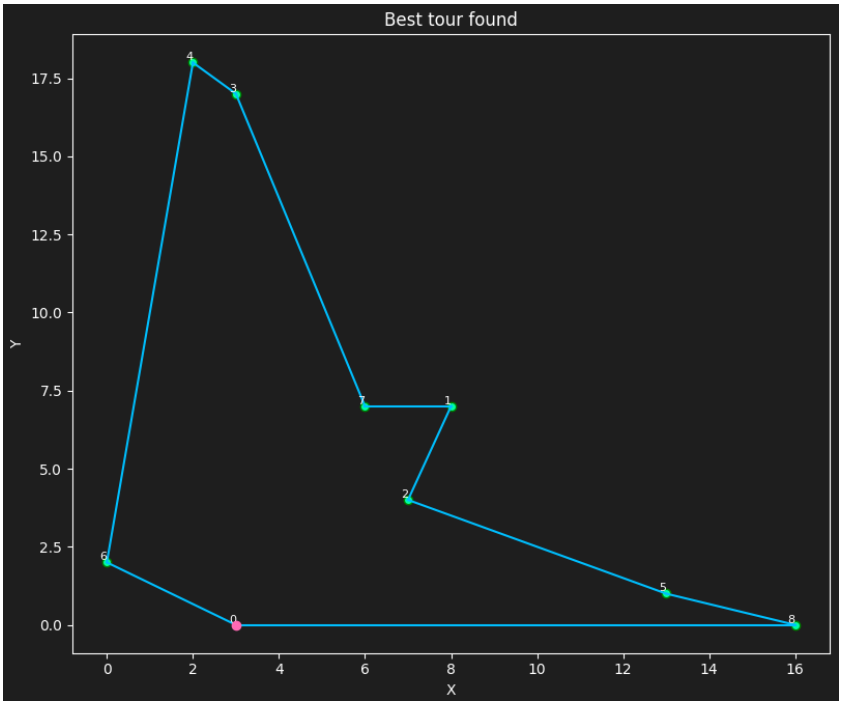


Figure 7: 9-city problem instance
Shapley values: 1: 4.058, 2: 2.937, 3: 12.129, 4: 14.332, 5: 6.779, 6: 3.303, 7: 3.378, 8: 12.701

Table 13: Results for 9-city Shapley approximation using the exhaustive TSP evaluation.

Metric	**Single-thread**		**Parallel**	
	Mean	Std Dev	Mean	Std Dev
MAE	0.5580	0.1905	0.1525	0.0494
MSE	0.6006	0.4580	0.0430	0.0333
RMSE	0.7253	0.2730	0.1949	0.0708
Max Error	1.4095	0.5974	0.3755	0.1512
Percent Error	8.2%	2.3%	2.4%	0.6%
Elapsed Time (s)	—	—	5.077	0.314

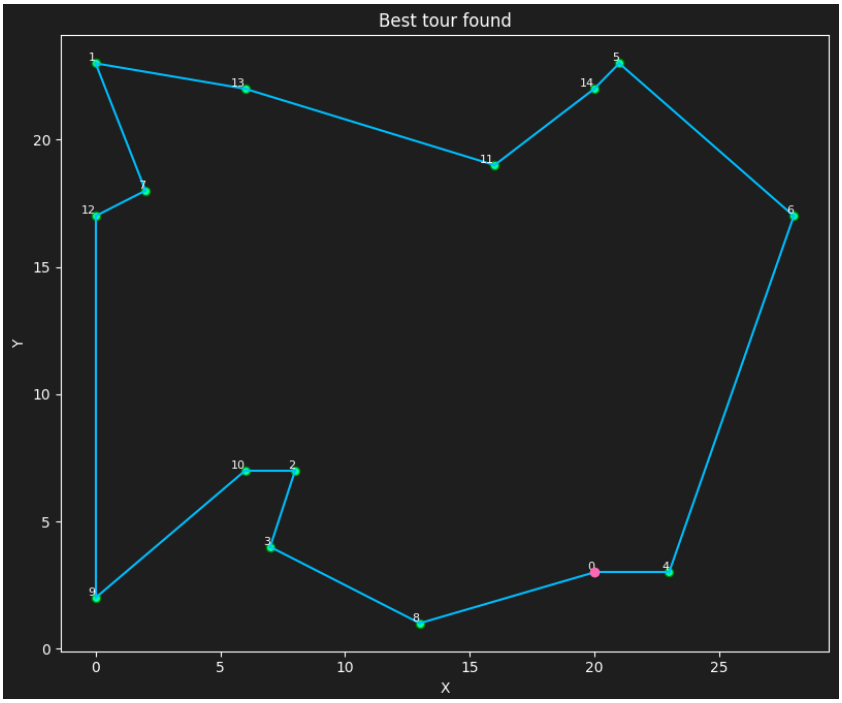


Figure 8: 15-city problem instance
Shapley values: 1: 13.518, 2: 3.986, 3: 4.477, 4: 2.987, 5: 8.272, 6: 9.818, 7: 7.404, 8: 3.057, 9: 14.319, 10: 4.451, 11: 5.130, 12: 8.257, 13: 7.270, 14: 6.659

Table 14: Results for the 9-city Shapley approximation using genetic algorithm to estimate TSP costs.

Metric	**Single-thread**		**Parallel**	
	Mean	Std Dev	Mean	Std Dev
MAE	0.5449	0.2033	0.1595	0.0543
MSE	0.5808	0.5199	0.0446	0.0306
RMSE	0.7048	0.2900	0.1994	0.0694
Max Error	1.3555	0.6306	0.3725	0.1379
Percent Error	8.1%	2.5%	2.5%	0.7%
Elapsed Time (s)	—	—	9.352	0.890

Table 15 confirms that even for the more complex 15-city instance, the parallel implementation maintains reasonable accuracy (3.3% error) with low variance, making it suitable for scalable Shapley value estimation.

Table 15: Results for the 15-city Shapley approximation using genetic algorithm to estimate TSP costs

Metric	**Single-thread**		**Parallel**	
	Mean	Std Dev	Mean	Std Dev
MAE	0.7257	0.1650	0.2210	0.0496
MSE	0.9422	0.4317	0.0868	0.0386
RMSE	0.9458	0.2180	0.2878	0.0632
Max Error	2.1298	0.6488	0.6552	0.1819
Percent	10.9%	2.4%	3.3%	0.7%
Elapsed Time (s)	—	—	32.438	1.380

It is important to note, that the Shapley values shown in Figures 7 and 8 represent each city's average marginal contribution to the total tour length across all sampled permutations. These values are expressed in raw distance units (e.g., kilometers), rather than monetary costs. In practice, the cost attributable to each city can be directly obtained by multiplying its Shapley value by a given unit cost per kilometer. This allows the distance-based allocation to be easily converted into a monetary cost-sharing scheme, assuming uniform travel costs per distance.

Computational remarks. When executed on a single thread, the permutation-based Shapley estimation for a 20-city problem using 1000 sampled permutations completes in approximately 7 to 8 minutes. For a 30-city problem under the same configuration, the runtime increases to around 40 minutes due to the growing complexity of evaluating longer TSP tours. In contrast, the parallel version—designed to run multiple independent threads with the same per-thread sample size—enables substantial scalability. The final estimate is aggregated across all threads, allowing users to improve solution quality proportionally with the number of processor cores. Although parallelization introduces some overhead, total runtime typically remains about twice that of a single-threaded run, regardless of the number of threads. This modest trade-off provides a predictable and flexible framework where accuracy scales with hardware resources. This behavior highlights the practicality of the parallel approach for large-scale problems, offering fine-grained control over the balance between runtime, memory usage, and approximation quality.

6.2.4 Correlation Analysis between Shapley Values and Cities Geometry

In addition to evaluating numerical accuracy, we investigated whether the computed Shapley values reflect geometric patterns of randomly generated instances. Since the Shapley value accounts for each city's marginal contribution to coalition costs, we hypothesized that cities located farther from the depot or more geographically isolated would tend to receive higher values.

To explore this hypothesis, we performed a correlation analysis between the Shapley value of each city and two geometric characteristics:

- Euclidean distance from the depot (`City_0`),
- Average distance to all other cities, including the depot.

To reduce variance and ensure robustness, the values were averaged across 30 independent runs of the parallel solver. Since the Shapley values obtained through Monte Carlo estimation were shown to be approximately normally distributed, the use of both Spearman's rank correlation (for monotonic trends) and Pearson's correlation coefficient (for linear relationships) is statistically justified.

As shown in Tables 16 and 17, we observe statistically significant correlations between the geometric properties of cities and their assigned Shapley values. In both the 9-city and 15-city instances, cities located farther from the depot tend to receive higher allocations, which supports the intuitive notion that more remote cities contribute more to the total cost of coalition tours.

Table 16: Correlation between Shapley values and city distances in the 9-city instance (GA computation).

Metric	**Spearman** ρ	p**-value**	**Pearson** r	p**-value**
Distance from Depot	0.9524	0.00026	0.9378	0.00057
Average Distance to Others	0.8571	0.00653	0.9255	0.00098

Table 17: Correlation between Shapley values and geometric city properties in the 15-city instance.

Metric	**Spearman** ρ	p**-value**	**Pearson** r	p**-value**
Distance from Depot	0.8009	0.0006	0.7305	0.0030
Average Distance to Others	0.5473	0.0428	0.6022	0.0227

A similar trend is observed for the average distance to all other cities: cities that are more isolated from the rest of the network also tend to receive higher Shapley values. These correlations suggest that geometric remoteness—whether from the depot or the network as a whole—is a meaningful indicator of marginal contribution in TSP-based cooperative games.

The effect is especially strong in the 9-city instance, where both Shapley values and TSP costs are computed exactly. In contrast, the 15-city instance shows weaker correlations. This difference is likely due to two factors: the increased complexity of the coalition interaction in larger problems, and the use of approximated TSP costs via a genetic algorithm, which introduces estimation noise. Despite this, the correlations in the 15-city case remain statistically significant, indicating that geometric features retain explanatory value, especially in smaller or more structured instances where coalition cost evaluations are more stable.

7 Conclusion

This thesis explored the integration of genetic algorithms and cooperative game theory to address fair cost allocation in routing problems, with a specific focus on the Traveling Salesman Problem. A flexible and modular genetic algorithm solver was implemented in Python, supporting configurable operators, GUI interaction, and parallel processing. The solver was further extended to estimate Shapley values using a sampling-based approach, enabling scalable approximation of marginal contributions in cost-sharing scenarios. Moreover, the correlation analysis demonstrated meaningful relationships between spatial structure and Shapley values.

In terms of practical extensibility, several improvements can be made to enhance the usability and robustness of the solver. One such improvement is the ability to load problem instances directly from a distance matrix, allowing support for non-Euclidean TSP variants or real-world datasets. Additionally, the current export functionality could be expanded to include not only city coordinates but also derived data such as tour sequences, final tour lengths, and convergence statistics. Further code refactorization would also help simplify integration with other tools and ensure a more modular and seamless implementation. To improve performance in larger problem instances, performance-critical components or the entire solver could be reimplemented in `C` or `C++`, allowing for significant speedups compared to Python.

On the algorithmic side, several promising directions remain open. The use of advanced fitness-scaling techniques, such as windowing or sigma scaling, could be implemented and dynamically activated based on population diversity or convergence stagnation. To further improve solution quality, especially in the later stages of evolution, hybridization with local search methods such as 2-opt or 3-opt could be introduced to refine solutions beyond what the genetic operators alone can achieve. Finally, incorporating more advanced crossover techniques, such as the EAX, which is considered state-of-the-art for TSP problems, could significantly enhance the algorithm's ability to preserve and recombine high-quality structures during reproduction.

Future improvements to the Shapley value solver could focus on modularizing the codebase—separating core logic, sampling methods, and cost evaluation—to simplify experimentation with alternative techniques. Unifying the solver's GUI with the TSP interface would streamline configuration and use. Adding built-in tools for error analysis, summary reporting, and correlation diagnostics would further enhance usability, reproducibility, and analytical depth.

Bibliography

[1] APPLEGATE, David L., BIXBY, Robert E., CHVÁTAL, V., COOK, William J. *The Traveling Salesman Problem: A Computational Study*. Princeton: Princeton University Press, 2006. ISBN: 978-0-691-12993-8.

[2] DIESTEL, Reinhard. *Graph Theory*. Electronic edition. Berlin: Springer-Verlag, 2005. ISBN: 3-540-26182-6.

[3] EIBEN, A. E., SMITH, J. E. *Introduction to Evolutionary Computing*. 2nd ed. Berlin: Springer, 2015. ISBN: 978-3-662-44874-8.

[4] YU, Xinjie, GEN, Mitsuo. *Introduction to Evolutionary Algorithms*. London: Springer, 2010. e-ISBN: 978-1-84996-129-5.

[5] GILLES, Robert P. *The Cooperative Game Theory of Networks and Hierarchies*. Berlin: Springer, 2010. e-ISBN: 978-3-642-05282-8.

[6] PETERS, Hans. *Game Theory: A Multi-Leveled Approach*. 2nd ed. Cham: Springer, 2015. e-ISBN: 978-3-662-46950-7.

[7] SHAPLEY, Lloyd S. *A Value for N-Person Games*. Santa Monica, CA: RAND Corporation, 1953. Research Memorandum RM-670. [Online]. Available: `https://www.rand.org/pubs/research_memoranda/RM0670.html`

[8] ENGEVÅLL, Stefan, GÖTHE-LUNDGREN, Maud a VÄRBRAND, Peter. The traveling salesman game: An application of cost allocation in a gas and oil company. *Annals of Operations Research*, 1998, vol. 82, no. 0, pp. 203–218. ISSN: 0254-5330.

[9] APPLEGATE, David L., BIXBY, Robert E., CHVÁTAL, V., COOK, William J. *Concorde TSP Solver*. [Online]. Available: `https://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm`.

[10] TSPLIB. *Library of Sample Instances for the TSP and Related Problems*. [Online]. Available: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/`.

[11] AZARIA, Amos, HAZON, Noam, and LEVINGER, Chaya. Efficient computation and estimation of the Shapley value for traveling salesman games. *IEEE Access*, 2021, vol. 9, pp. 129119–129129. DOI: 10.1109/ACCESS.2021.3113282.

[12] KENNEDY, Tom. *Monte Carlo Methods – A Special Topics Course* [online]. 2004. Available at: `https://math.arizona.edu/~tgk/mc/book.pdf`. Accessed May 2025.