Balancing Exploration and Exploitation in GAs for the TSP

Oriol Miró and Joan Caballero

MAI, UPC
Computational Intelligence, Practice 1

November 10, 2024

Travelling Salesman Problem (TSP) is a well-known combinatorial optimization problem, notable for its NP-completeness. Genetic Algorithms (GAs) have become a popular heuristic for finding near-optimal solutions efficiently. A key challenge in GAs is balancing exploration, the search of diverse solution spaces, and exploitation, the refinement of promising solutions. This paper examines how GA components——selection, crossover, mutation, and diversity management——affect this trade-off in TSP. Using TSPLIB instances as benchmarks, we conduct a full factorial experiment, testing 108 configurations across various problem sizes. Our results show that balancing selection for exploitation and mutation for exploration improves performance, with other parameters having a lesser impact. Additionally, we found parameters should be tuned in groups rather than individually, as their synergy is crucial. Results also suggest that smaller, cheaper problem instances can serve as effective testbeds for parameter settings, with relative configuration performance in terms of exploration-exploitation consistent across problem size, while harder maps exacerbate shortcomings.

1 Introduction

The Travelling Salesman Problem (TSP) is one of the most well-studied combinatorial optimisation problems of all time. It involves finding the shortest—per a distance metric—possible tour that visits each vertex in a graph exactly once and returns to the origin vertex—this is equivalent to finding the shortest hamiltonian cycle in the graph. The TSP, classified as NP-complete, is inherently complex: given N vertices to visit, there are $N!$ possible permutations. To contextualise this number, planning an optimal route visiting Spain's major cities (with populations of at least 100,000 inhabitants) would result in *far* more possible combinations than the number of atoms in the observable universe.[1] [2, 26].

Given the computational challenges posed by the TSP, extensive research has developed a range of solution strategies over the past few decades. Some methods rely on exact algorithms, such as branch-and-bound [4], integer linear programming with constraints to prevent subtours [9, 18, 27], and cutting planes [20]. While these approaches may solve smaller instances optimally, they become computationally infeasible for larger problem instances; thus, heuristic and approximate methods have become popular alternatives. These include simulated annealing [15], tabu search [11], and local search techniques like 2-opt [17].

Among these heuristic approaches, Genetic Algorithms (GAs) [12] are particularly interesting due to their ability to efficiently explore vast solution spaces while maintaining diversity through evolutionary mechanisms. They operate through a process inspired by natural selection, evolving solutions over generations to minimise a fitness function——in this case, the total length of the route. Numerous studies have explored their applicability to the TSP [16, 21] and demonstrated their effectiveness in producing near-optimal solutions within a feasible time-frame.

However, a challenge remains: balancing exploration

[1] There are 63 cities with more than 100 thousand inhabitants, and the number of atoms in the observable universe is estimated to be around $10^8 2$; $63! \approx 10^{87} \gg 10^{82}$

and exploitation during the search process. **Exploration** refers to the algorithm's ability to investigate diverse regions of the solution space, thereby avoiding premature convergence to suboptimal solutions. **Exploitation**, on the other hand, focuses on intensifying the search around promising solutions to refine and improve them. Balancing the two is essential; too much exploration can lead to inefficiency, while excessive exploitation may cause the algorithm to become trapped in local minima [8]. This is specially important in the context of the TSP's complex, uneven solution space.

Recognising this, our study aims to investigate how different selection, mutation, crossover, and diversity maintenance mechanisms within GAs influence the exploration-exploitation trade-off for the TSP. We will experiment with different configurations and test our algorithms on the TSPLIB [22] library, which contains several instances of the problem coupled with optimal solutions.

2 Previous Work

The exploration-exploitation trade-off in Genetic Algorithms (GAs) has been thoroughly discussed, particularly in the context of combinatorial optimisation problems like the TSP. A survey by Črepinšek et al. [5] studied the importance of preserving diversity within the population while improving solutions; this research offers a theoretical basis for our work.

Specific to the TSP, several studies have examined the influence of GA components on this trade-off. Hussain et al. [14] points to the impact of selection mechanisms, even proposing an operator that, as the study claims, optimally balances exploration with exploitation, by preserving diversity without sacrificing the advantage of fitter individuals. Moving on to other components, Larrañaga et al. [16] investigates the impact of crossover methods and how they refine solutions while allowing broader search. It specifically discusses the two methods we employ in our study, Partially Mapped Crossover (PMX) and Order Crossover (OX).

Another key component are mutation operators, often of a more explorative nature; Deep and Mebrahtu [7] proposed a mutation strategy aimed at preventing premature convergence. Similarly, hybrid approaches like that of Wang et al. [25], which integrate local search methods, have been shown to further enhance exploitation without compromising global search capabilities. Vidal et al. [24] strengthened these findings, showing the significance of diversity management for maintaining the robustness of GAs on complex TSP instances.

From these, we can take that a thoughtful integration of GA components—selection, crossover, mutation, and diversity management—–can significantly influence the exploration-exploitation trade-off. We build on these insights, and study different components on a stand-alone manner, their interaction, and how they performed under different map sizes.

3 Methodology

3.1 Problem Representation

For the TSP, we employ a permutation-based representation where each chromosome is a sequence of integers representing the order of cities visited. While binary encoding is traditional in GAs [10], permutation representation is more natural and efficient for the TSP for several reasons. First, binary encodings would require complex repair mechanisms to maintain tour validity, as not all binary strings map to valid tours [16]. Second, permutation encoding allows for specialised operators that preserve tour feasibility and utilise problem-specific knowledge [19]. Third, empirical studies have shown that permutation-based GAs consistently outperform binary-encoded variants for sequencing problems [6].

3.2 Genetic Algorithm Framework

Our implementation follows the canonical GA structure [12], as detailed in Algorithm 1.

Our implementation follows the canonical Genetic Algorithm (GA) structure, as outlined in Algorithm 1; specifically, we adopt a (μ, λ)-generational strategy:

Algorithm 1 Genetic Algorithm for TSP

```
Initialise population with random permutations
for each generation do
    Evaluate fitness of all individuals
    Select parents based on selection method
    Apply crossover to parents to produce offspring
    Apply mutation to offspring
    if Elitism is enabled then
        Preserve top elites to next generation
    end if
    if Diversity Injection is enabled then
        Introduce new random individuals periodically
    end if
    Form new population
end for
Return best individual found
```

Based on preliminary experimentation, we fix the following parameters:

- **Population Size**: 75 individuals
- **Number of Generations**: 3000
- **Crossover Rate**: 0.8
- **Mutation Rate**: 0.1

We also introduce a ´´patience" parameter, where if the best fitness in the population has not improved

for t generations, we prematurely stop the execution. This helps avoid useless iterations when stuck on local minima. We fix $t = 200$.

3.3 Experimental Variables

We investigate three categories of genetic operators, each contributing differently to the exploration-exploitation balance; moreover, we apply different techniques to directly manage diversity [5, 14]:

3.3.1 Selection Mechanisms

Selection pressure directly influences the exploration-exploitation trade-off [16]. We examine three mechanisms, organised from most explorative to most exploitative:

- **Stochastic Universal Sampling (SUS)**: Selects individuals using μ equally spaced pointers across the cumulative probability distribution of fitness values.
- **Rank Selection**: Assigns selection probabilities based on the rank of individuals rather than their actual fitness values; provides more exploitation than SUS, nevertheless still allows choosing less-favoured individuals.
- **Tournament Selection ($k = 3$)**: Provides relatively high selection pressure by selecting the best individual from randomly chosen groups of three candidates; our most exploitative selection method. Higher tournament size favours exploitation over exploration. We fix k given computational constraints.

3.3.2 Crossover Operators

Crossover operators for TSP must preserve tour validity while effectively combining parent information [21]; since these are often of exploitative nature, their investigation adds limited novelty to our study. Nevertheless, we inspect two techniques widely used in the literature, the first being more explorative and the second more exploitative:

- **Order Crossover (OX)**: Selects a random segment from one parent and fills remaining positions taking cities from the other parent, in the order they appear; preserves the *relative* order of cities while introducing new combinations, somewhat promoting exploration.
- **Partially Mapped Crossover (PMX)**: Like OX, PMX begins by selecting a random segment from one parent and copying it into the offspring, but fills remaining positions differently. PMX creates a mapping between the cities in this segment and their corresponding positions in the other parent, used to resolve conflicts: if a city from Parent 2 outside the copied segment is already present in the offspring, the mapping provides the valid replacement to ensure all cities are unique. Preserves *absolute* positions of cities and is more exploitative, strongly retaining structural components from the parents.

We also implemented Edge Recombination Crossover, an extremely exploitative approach; however, due to its prohibitively high computational cost, we had to discard it.

3.3.3 Mutation Operators

Mutation provides the primary means of exploration, with different operators offering varying levels of disruption [7]. As we have been doing, they are presented from most explorative to most exploitative:

- **Scramble Mutation**: Randomly reorders a subset of cities in the tour; very explorative, as we do not preserve any order.
- **Inverse Mutation**: Reverses the order of cities between two random positions; somewhat balances exploration and exploration, as the sequence of cities remains the same, yet we visit them in the opposite direction.
- **Swap Mutation**: Exchanges the positions of two randomly selected cities; very exploitative, almost a *local* method, introducing very small changes to the offspring.

3.3.4 Diversity Management Mechanisms

These mechanisms provide explicit control over the exploration-exploitation balance [24]:

- **Elitism**:
  - **Enabled**: Preserves the top 2% of solutions between generations; more exploitative.
  - **Disabled**: All individuals are subject to replacement; more explorative.
- **Diversity Injection**:
  - **Enabled**: Introduces random immigrants every 50 generations, replacing 10% of the population (excluding elites, if applicable); more explorative.
  - **Disabled**: Population evolves without external intervention; more exploitative.

3.4 Experimental Design

Given our four categories of experimental variables (selection, crossover, mutation, and diversity management), we employ a full factorial design to systematically evaluate all possible combinations:

- 3 selection mechanisms
- 3 crossover operators
- 3 mutation operators

- 2 elitism options (enabled/disabled)
- 2 diversity injection options (enabled/disabled)

This results in $3 \times 3 \times 3 \times 2 \times 2 = 108$ unique configurations to be tested.

3.5 Test Instances

We evaluate our configurations on benchmark instances from TSPLIB [22], specifically selecting:

- **Small instances** (under 100 cities): ulysses22, gr48, berlin52, brazil58, st70
- **Medium instances** (100-200 cities): eil101, lin105, gr137, kroA150, si175
- **Large instances** (over 200 cities): gr202, kroA200, ts225, pr226, a280

We would ideally like to select larger instances (+300 cities), however our computational limitations bound us to smaller cases.

3.6 Evaluation Metrics

To assess the performance of each genetic algorithm configuration, we capture different metrics both during the execution of an experiment and at the end of it. For the former, we store every 50 generations:

- **Best Fitness Error**: Current best solution.
- **Average Fitness Error**: Mean fitness across the population.
- **Diversity**: Measured using Matrix-Based Entropy, which has been shown to be more suitable for TSP-focused genetic algorithms than traditional measures like Hamming distance [13]. We measure diversity every 50 generations, just before a diversity injection (where applicable), to provide an uninfluenced, valid pulse on our algorithm

Meanwhile, we store the following metrics *per-run* (in contrast to *per-X-generations* within a run):

- **Best Fitness Error**: Final best solution.
- **Convergence Rate**: The number of generations required to reach a plateau in the best fitness, expressed as a percentage w.r.t. the maximum number of iterations (e.g. if we converge on 1000 generations out of 3000 possible ones, the convergence rate is $\frac{1000}{3000} = 0.33$.
- **Computational Time**: The total runtime for each configuration.

All metrics related to fitness are computed as the percentage of error w.r.t the optimal fitness, as done in the literature [23, 1]. More concretely, such error is computed as:

$$\text{Error}(\%) = \left| \frac{\text{Fitness}_{\text{obtained}} - \text{Fitness}_{\text{optimal}}}{\text{Fitness}_{\text{optimal}}} \right| \times 100 \tag{1}$$

3.7 Implementation and Reproducibility

The genetic algorithms were implemented from scratch in *Python* 3.9. Given the *embarrassingly parallel* nature of the experiments, they were executed in parallel, on Ubuntu 22.04 using a 6-core Ryzen 5600X processor.

To ensure reproducibility, we provide all code used. Moreover, due to the stochastic nature of Genetic Algorithms, each experiment (GA configuration $\times$ TSP instance) was repeated 3 times, averaging results. While literature emphasises the importance of multiple independent runs [3], computational constraints limited our capacity; our experimental setup, with 108 configurations across 15 problem instances with 3 repetitions ($108 \times 15 \times 3 = 4,860$ total runs) exhausts our resources.

4 Results

We present the most relevant results of our work, focusing on three studies: the impact of four experimental variables (selection, crossover, mutation, and diversity management) on performance and diversity; their interaction in balancing exploration and exploitation; and the effect of problem size on GA performance and convergence. Statistical analysis includes a Shapiro-Wilk test for normality and a Mann-Whitney U test to assess the impact of each parameter, as all distributions are non-normal.

4.1 Impact of Genetic Algorithm Parameters on Performance and Diversity

In this section, we compare the different methods of selection, crossover, mutation, elitism, and diversity injection, and analyse their impact on the performance of genetic algorithms in terms of average population fitness error and matrix-based diversity. Our objective is to determine if more explorative operators dominate over more exploitative operators, both overall and within each method.

Our experiments show that exploitative selection methods perform better. Stochastic Universal Sampling (SUS) maintains high diversity across generations (see Figure 2) due to its explorative nature. However, its lack of exploitation leads to consistently high average fitness errors (see Figure 1), failing to effectively enhance fitness and resulting in significantly higher errors compared to other methods. In contrast, Rank Selection offers a moderate trade-off between exploration and exploitation, while Tournament Selection is more exploitative. Their limited exploration enables genetic algorithms to focus on fitter individuals, rapidly reducing diversity, as seen in Figure 2, since they tend to select the best individuals. Statistical tests suggest the three methods follow a non-normal distribution ($p \approx 10^{-60}$), and the Mann-Whitney U test shows Tournament and Rank Selection differ from SUS ($p \approx 0$)

but not from each other ($p = 0.717$).

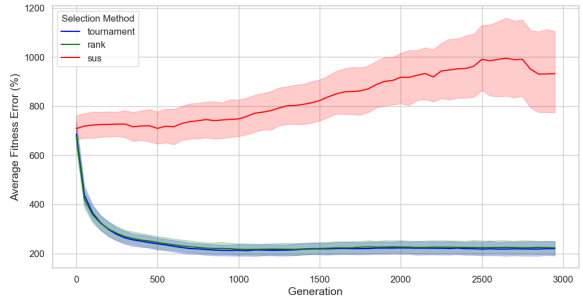

Figure 1: *Average Population Fitness Error over Generations by Selection Method. Each line represents a different selection method. Shaded areas indicate the 95% confidence intervals.*

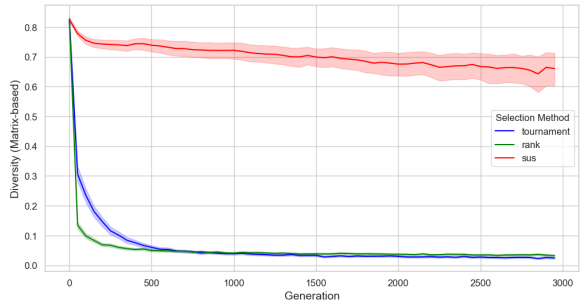

Figure 2: *Average Diversity (Matrix-based) over Generations by Selection Method. Each line represents a different selection method. Shaded areas indicate the 95% confidence intervals.*

Regarding crossover operators, statistical tests revealed once again non-normality (for both, $p < 0.001$), and the Mann-Whitney U test showed statistically significant difference between the crossover operators (p $= 3 \times 10^{-59}$), strongly pointing that OX performs better than PMX (regarding obtaining a lower average fitness error). This result is consistent with Figure 3, where OX shows a lower average fitness error compared to PMX. OX is a more explorative method, while PMX is more exploitative, as seen in Figure 4, where OX maintains higher diversity than PMX. PMX reduces population diversity more rapidly in the early generations due to its more exploitative nature, as evidenced by the steeper initial slope in the line plot.

Our results show that balanced mutation methods perform better. Inverse Mutation achieves the lowest average fitness error and highest diversity across generations (see Figures 5 and 6). Its balance between exploration and exploitation enables it to explore different solution spaces and improve fitness. Swap Mutation, being more exploitative, ranks second in fitness error but suffers from reduced exploration, as seen in its low diversity in Figure 6. Scramble Mutation, while highly explorative, lacks exploitation, resulting in the highest average fitness error. Interestingly, Inverse Mutation maintains higher diversity than Scramble despite

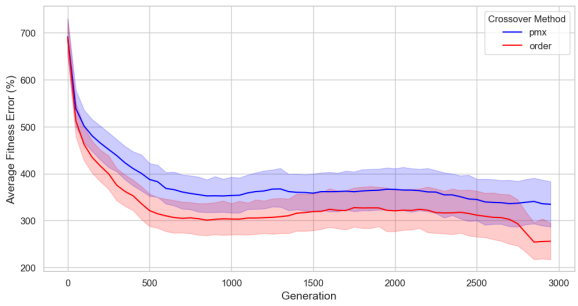

Figure 3: *Average Population Fitness Error over Generations by Crossover Method. Each line represents a different crossover method. Shaded areas indicate the 95% confidence intervals.*

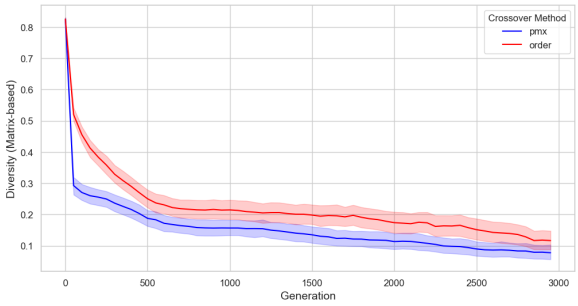

Figure 4: *Average Diversity (Matrix-based) over Generations by Crossover Method. Each line represents a different crossover method. Shaded areas indicate the 95% confidence intervals.*

Scramble's explorative nature. This is because Scramble often produces low-fitness individuals unlikely to be selected, limiting its ability to explore additional solution areas. Statistical tests confirm non-normality ($p \ll 0.001$) and significant differences between all methods (pairwise $p \ll 0.001$).

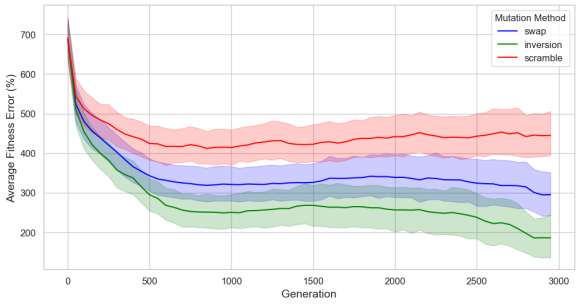

Figure 5: *Average Population Fitness Error over Generations by Mutation Method. Each line represents a different mutation method. Shaded areas indicate the 95% confidence intervals.*

Disabling elitism leads to faster convergence and individuals with lower average fitness errors. Surprisingly, maintaining elitism increases diversity, despite its exploitative nature. Initially, the population is scattered randomly across the solution space. Without elitism, low-fitness individuals in less promising ar-

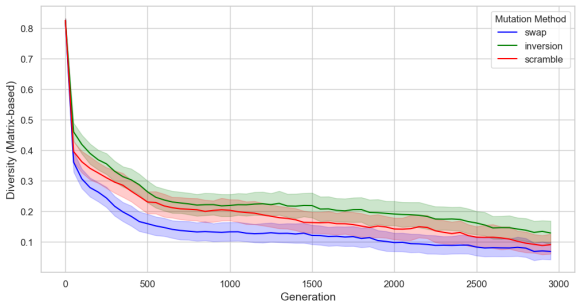

Figure 6: *Average Diversity (Matrix-based) over Generations by Mutation Method. Each line represents a different mutation method. Shaded areas indicate the 95% confidence intervals.*

eas are quickly discarded, focusing the algorithm on high-fitness areas. This results in faster convergence and lower diversity, as shown in Figure 15 and the steep slope in the first 500 generations in Figure 7, where average fitness error rapidly decreases to a low, constant value. Figure 8 also shows low, constant diversity. In contrast, elitism increases diversity by retaining significantly different elites, each occupying distinct solution areas. This allows the algorithm to explore diverse regions. Statistical tests confirm non-normality ($p \approx 10^{-100}$) and the Mann-Whitney U test shows a significant performance difference in average fitness ($p \approx 10^{-200}$).

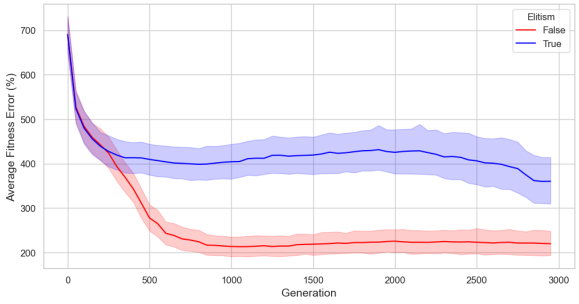

Figure 7: *Average Population Fitness Error over Generations by Elitism. Each line indicates whether elitism is enabled. Shaded areas indicate the 95% confidence intervals.*

Statistical results suggest that not using diversity injection is superior. The Shapiro-Wilk test confirmed non-normality ($p \ll 0.001$), and the Mann-Whitney U test showed a significant difference between using and not using diversity injection ($p \approx 10^{-110}$). Figure 9 shows similar average fitness errors in both cases. Introducing random individuals helps the algorithm escape local minima and explore new regions, but these individuals often come from higher-error areas, increasing the population's average fitness error. Figure 10 indicates that diversity remains largely unchanged, as diversity injection occurs infrequently, and less fit individuals are quickly eliminated, returning diversity to prior levels.

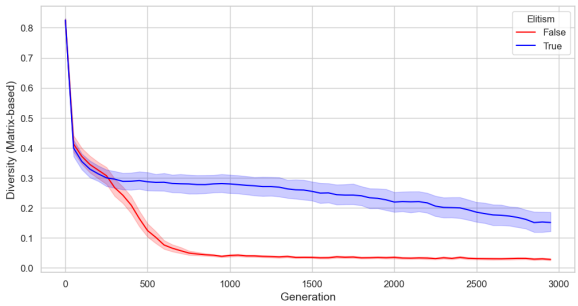

Figure 8: *Average Diversity (Matrix-based) over Generations by Elitism. Each line indicates whether elitism is enabled. Shaded areas indicate the 95% confidence intervals.*

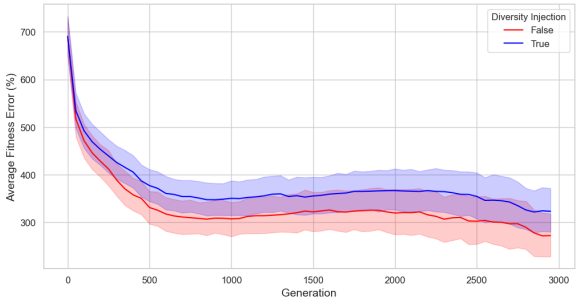

Figure 9: *Average Population Fitness Error over Generations by Diversity Injection. Each line indicates whether diversity injection is enabled. Shaded areas indicate the 95% confidence intervals.*

4.2 Interaction Effects of Genetic Algorithm Operators

In this section, we analyse the interaction between different methods on best fitness error. By examining various combinations, we aim to identify synergistic effects that lead to improved GA performance—in terms of balancing exploration and exploitation. We will show the most relevant results.

Figure 11 shows the interaction between mutation and selection methods on the final best fitness error. SUS consistently produces the highest errors, with SUS and Scramble being the worst combination. This aligns with their poor performance in Figures 1 and 5. Rank and Tournament methods have minimal impact, sometimes slightly improving or worsening fitness error. The optimal combination is Inversion Mutation and Rank Selection, as Inversion is the most effective mutation method, and Rank rapidly reduces diversity (see Figure 2). Tournament and Rank Selection are statistically equivalent in average fitness error (Mann-Whitney U Test: $U = 64,765.5$, $p = 0.990$).

An interesting result is the impact of mutation and crossover combinations on the final best fitness error. Mutation methods influence fitness error more than crossover methods, as shown in Figure 12, with larger variations across mutation methods. For Inversion, the crossover method has little effect on fitness

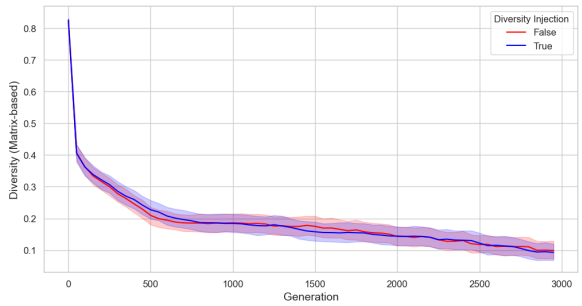

Figure 10: *Average Diversity (Matrix-based) over Generations by Diversity Injection. Each line indicates whether diversity injection is enabled. Shaded areas indicate the 95% confidence intervals.*

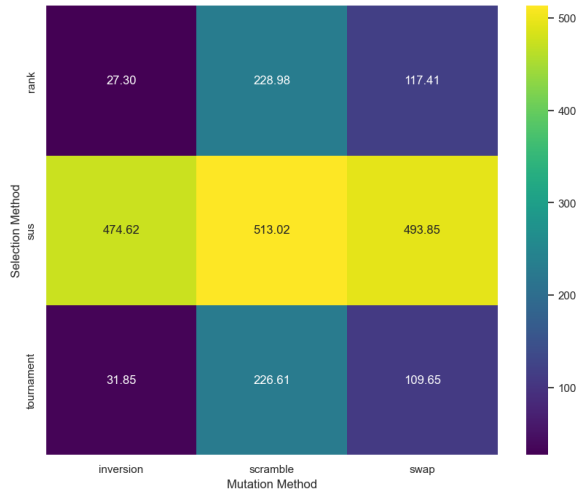

Figure 11: *Heatmap of the final best fitness error across combinations of selection and mutation methods. Lower fitness error values indicate better performance. The value and colour intensity in each cell indicate the average best fitness error for the combination.*

error, achieving the lowest errors overall. For Swap, Order results in lower errors than PMX, as Swap's exploitative nature pairs well with Order's explorative balance. Conversely, combining Swap with another exploitative method like PMX lacks this balance. For Scramble, the crossover choice significantly affects performance: pairing Scramble with Order reduces fitness error by 27% compared to PMX. Scramble, a highly explorative method, conflicts with PMX, which preserves absolute city order and produces offspring similar to parents. This mismatch increases error. In contrast, Order, which preserves relative order, aligns better with Scramble's flexibility, producing solutions closer to the population and avoiding distant regions of the search space.

The heatmap in Figure 13 shows a variety of results. Best fitness errors with elitism are consistently lower than without it. Although this may seem contradictory to Figure 7, the latter depicts average population fitness error, while here we focus on the best individual. Elitism increases average fitness error but

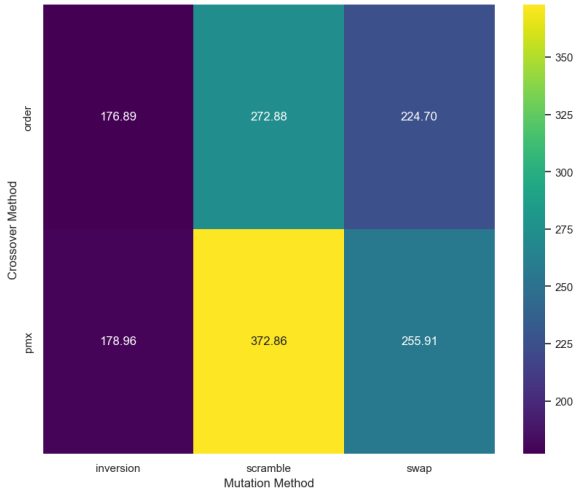

Figure 12: *Heatmap of the final best fitness error across combinations of crossover and mutation methods. Lower fitness error values indicate better performance. The value and colour intensity in each cell indicate the average best fitness error for the combination.*

preserves superior individuals with lower fitness. Mutation methods follow the same ranking as in Figure 5: Inversion achieves the lowest errors, followed by Swap and Scramble. Elitism enhances results most with Inversion (38%), then Swap (26%), and finally Scramble (13%). Scramble, being highly explorative, benefits less from elitism due to fewer high-fitness individuals. In contrast, Inversion and Swap exploit solution spaces effectively, preserving promising individuals and improving best fitness errors in the heatmap.

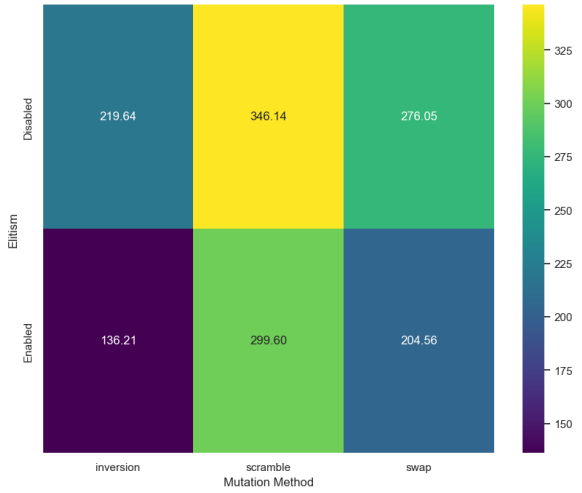

Figure 13: *Heatmap of the final best fitness error across combinations of elitism and mutation methods. Lower fitness error values indicate better performance. The value and colour intensity in each cell indicate the average best fitness error for the combination.*

4.3 Impact of Map Size on Genetic Algorithm Performance and Convergence

In this section, we provide a more detailed analysis of some of the previously discussed results, considering problem size, and we will also present other relevant findings.

In the graph shown in Figure 14, we observe the same trends as in Figure 1, but now segmented by map size and providing more detailed information. For instance, the average fitness error of SUS increases drastically with problem size. This is expected: for smaller maps, the fitness error is more constrained by the number of cities. However, as the map size increases, the inherently highly explorative and minimally exploitative nature of SUS causes the average fitness error to also rise. Additionally, we observe that for Rank and Tournament methods, these two methods achieve practically identical average fitness errors regardless of the map size.

Figure 15 presents an unexpected result discussed earlier: maintaining elitism slows convergence, regardless of problem size. This happens because elitism preserves diverse high-fitness individuals, preventing premature convergence to a single solution space. For smaller problems, the difference is minor, but it becomes more pronounced as the number of cities increases.

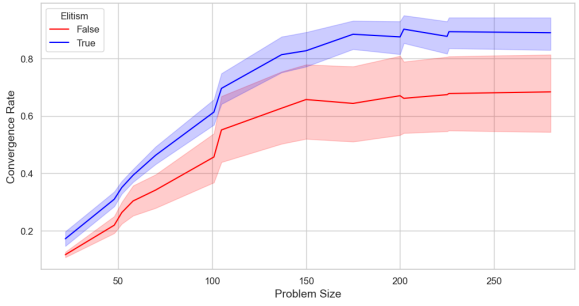

Figure 15: *Convergence Rate over Problem Size by Elitism. Each line indicates whether elitism is enabled. Shaded areas indicate the 95% confidence intervals.*

An interesting case in Figure 16 is SUS, which converges faster than more exploitative methods like Rank and Tournament for problem sizes of 40 cities or more. This is surprising, as SUS, being the most explorative method, would typically be expected to converge the slowest. However, SUS's low selection pressure and minimal exploitation result in minimal fitness improvement across generations. This triggers the early stopping criteria, causing the algorithm to terminate earlier.

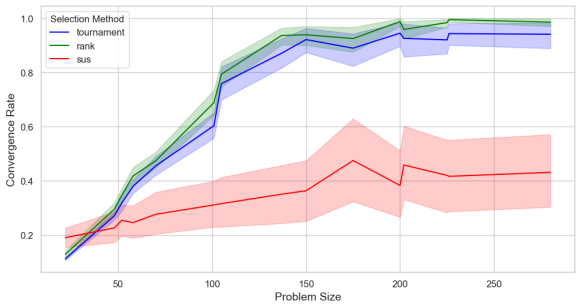

Figure 16: *Convergence Rate over Problem Size by Selection Method. Each line represents a different selection method. Shaded areas indicate the 95% confidence intervals.*

5 Discussion

We found that various genetic algorithm components have a strong impact on the exploration-exploitation balance when solving the TSP, and that the choice of selection, crossover, mutation, and diversity management techniques's success can be viewed through a lens of how they balance the overall trade-off.

Selection methods are the primarily exploitative force, and we found that those prioritizing high-fitness individuals, such as Tournament and Rank Selection, generally surpass the more diversity-focused Stochastic Universal Sampling (SUS); we believe this is so because no other component is as capable of exerting exploitation as selection, therefore this parameter needs to tip the scale correctly, suggesting that we should focus on this when our algorithm lacks exploitation.

On the other hand, mutation methods are our primary explorative force, and proved key to maintaining the balance. Inversion was the most effective, as it allowed exploration without producing offspring too far from the current focus on the search space, while the highly explorative Scramble Mutation and the strongly exploitative Swap Mutation underperformed, demonstrating the adverse effects of either extreme.

Crossover proved less important, with Order Crossover (OX) outperforming Partially Mapped Crossover (PMX) although not as strongly as in the case of other parametes's best choices. Their impact on diversity and convergence was comparable.

Interestingly, elitism increased diversity by maintaining multiple fronts in the search space. Although it worsened the average fitness error, it resulted in the best final fitness error by preserving high-quality solutions. Meanwhile, diversity injection had a minor impact overall, possibly due to its infrequent application or the rapid elimination of newly introduced individuals.

Interactions between algorithm components also showed to be key, as we have briefly discussed earlier; we need components to balance each other. Inversion Mutation combined with Rank Selection yielded the best fitness errors, demonstrating that a balanced

Figure 14: *Average Population Fitness Error over Generations by Selection Method and Problem Size. Each line represents a different selection method. Shaded areas indicate the 95% confidence intervals.*

mutation method paired with an exploitative selection technique enhances outcomes. Furthermore, coupling the explorative Scramble Mutation with Order Crossover surpassed combinations involving the more exploitative crossovers, an unexpected synergy. Our study on interaction reinforced the lesser importance of the crossover choice, as for Inversion Mutation we saw little change when varying crossover strategy.

Problem size significantly influenced performance and convergence, with larger instances amplifying configuration differences, but not the comparative performance between parameters: as the city count increased, the roles of selection, elitism, and other components became more pronounced. The shortcomings of unbalanced configurations were more evident in larger cases, leading to slower convergence and suboptimal solutions. Interestingly, the consistent impact of configurations across problem sizes suggests that smaller instances can effectively guide algorithm refinement. This enables configuration optimization on smaller, computationally simpler instances before addressing larger TSP problems.

This study has some limitations. The raw implementation, using both custom and NumPy functions, made measuring time inexact; perhaps a component is much faster despite a slightly worse performance, yet we can not measure that. Computational constraints limited experiments on larger instances, prevented extensive parameter tuning, and restricted the number of TSP instances tested. Additionally, the fixed number of generations may not be optimal for all problem sizes, and the convergence metric used does not account for the computational cost of individual components. Furthermore, the study focused solely on symmetric TSP instances, which may not fully represent real-world scenarios.

Future research could enhance exploitation by integrating local search techniques. Other parameters could be varied, such as population size, generation limits, mutation rates, or crossover rates. The study of *adaptative* techniques, where we modify parameters mid-execution where needed could also prove very interesting.

6 Conclusions

In this paper, we investigated the impact of various genetic algorithm operators on the trade-off between exploration and exploitation in solving the Travelling Salesman Problem (TSP). We implemented and tested different selection, crossover, and mutation operators, along with settings for elitism and diversity injection. Our findings suggest that some parameters, such as selection methods, are more important for exploitation, while others, like mutation methods, have a stronger effect on exploration. Moreover we found that one must consider the interaction between parameters, as it could result in either balance of strenghts or exacerbation of weaknesses. Results also show that the choice of parameters is independent of map size, allowing the tuning of the trade-off on smaller, cheaper TSP instances.

Bibliography

[1] Mohammad Alomari et al. “Solving Travelling Salesman Problem (TSP) by Hybrid Genetic Algorithm (HGA)”. In: *Journal of Optimization* 15.4 (2022), pp. 200–210.

[2] Harry Baker. *How many atoms are in the observable universe?* Accessed: 2024-10-25. July 2021. URL: https://www.livescience.com/how-many-atoms-in-universe.html.

[3] Erick Cantú-Paz and David E. Goldberg. “Are Multiple Runs of Genetic Algorithms Better than One?” In: *Annual Conference on Genetic and Evolutionary Computation*. 2003. URL: https://api.semanticscholar.org/CorpusID:14561836.

[4] G. Carpaneto and P. Toth. “Some new branching and bounding criteria for the asymmetric traveling salesman problem”. In: *Management Science* 26 (1980), pp. 736–743.

[5] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. “Exploration and exploitation in evolutionary algorithms: A survey”. In: *ACM Computing Surveys (CSUR)* 45.3 (2013), pp. 1–33.

[6] Lawrence Davis. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold, 1991. ISBN: 0-442-00173-8.

[7] Kusum Deep and Hadush Mebrahtu. "New variation of order crossover for travelling salesman problem". In: *International Journal of Combinatorial Optimization Problems and Informatics* 2.1 (2011), pp. 2–13.

[8] A.E. Eiben and C.A. Schippers. "On Evolutionary Exploration and Exploitation". In: *Fundam. Inf.* 35.1–4 (Jan. 1998), pp. 35–50. ISSN: 0169-2968.

[9] M. Fischetti and P. Toth. "An additive bounding procedure for combinatorial optimization problems". In: *Operations Research* 37 (1989), pp. 319–328.

[10] M. Gen and R. Cheng. *Genetic Algorithms and Engineering Design*. New York: Wiley, 1997.

[11] F. Glover. "Artificial intelligence, heuristic frameworks and tabu search". In: *Managerial & Decision Economics* 11 (1990), pp. 365–378.

[12] John H Holland. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975.

[13] Yuan Li-hua. "Study on Population Diversity of TSP in Genetic Algorithms". In: *Journal of Chinese Computer Systems* (2008). URL: https://api.semanticscholar.org/CorpusID:63247595.

[14] Abid Hussain and Younas Saber Muhammad. "Trade-off between exploration and exploitation with genetic algorithm using a novel selection operator". In: *Complex & Intelligent Systems* 6 (2019), pp. 1–14.

[15] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. "Configuration space analysis of travelling salesman problem". In: *Journal Physique* 46 (1985), pp. 1277–1292.

[16] P. Larrañaga et al. "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators". In: *Artificial Intelligence Review* 13.2 (1999), pp. 129–170. DOI: 10.1023/A:1006529012972.

[17] S. Lin and B. W. Kernighan. "An effective heuristic algorithm for the traveling salesman problem". In: *Operations Research* 21.2 (1973), pp. 498–516.

[18] J. Lysgaard. "Cluster based branching for the asymmetric traveling salesman problem". In: *European Journal of Operational Research* 119 (1999), pp. 314–325.

[19] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin, Heidelberg: Springer-Verlag, 1992.

[20] P. Miliotis. "Using cutting planes to solve the symmetric travelling salesman problem". In: *Mathematical Programming* 15 (1978), pp. 177–188.

[21] J.Y. Potvin. "Genetic algorithms for the travelling salesman problem". In: *Annals of Operations Research* 63 (1996), pp. 339–370.

[22] Gerhard Reinelt. "TSPLIB—A Traveling Salesman Problem Library". In: *ORSA Journal on Computing* 3.4 (1991), pp. 376–384.

[23] João P. Sousa and Other Authors. "Studying the Effect of Eliminating Repeated Individuals from the Population in a Genetic Algorithm: Solution Perspectives for the Travelling Salesman Problem". In: *International Journal of Computational Intelligence* 10.2 (2021). Accessed: 2024-10-25, pp. 123–135.

[24] Thibaut Vidal et al. "A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time-windows". In: *Computers & Operations Research* 39.3 (2012), pp. 475–489.

[25] Yong Wang. "Solving travelling salesman problem with an improved hybrid genetic algorithm". In: *Journal of Physics: Conference Series* 1324.1 (2019), p. 012006.

[26] Wikipedia contributors. *Ranked lists of Spanish municipalities*. Accessed: 2024-10-25. 2024. URL: https://en.wikipedia.org/wiki/Ranked_lists_of_Spanish_municipalities.

[27] R. Wong. "Integer programming formulations of the travelling salesman problem". In: *Proceedings of the IEEE International Conference of Circuits and Computers*. 1980, pp. 149–152.