FUNSEARCH ON TRAVELING SALESMAN PROBLEM

Le Chen, Xinzhu Tian, Yuan Chen
Student ID: 59039379, 58873864, 58482158
City University of Hong Kong
lchen546-c@my.cityu.edu.hk, xinzhtian4-c@my.cityu.edu.hk, ychen2752-c@my.cityu.edu.hk

1 INTRODUCTION

The Traveling Salesman Problem (TSP) is an NP-hard combinatorial optimization problem aiming to identify the shortest possible route visiting each city exactly once before returning to the origin. It has extensive applications in logistics, transportation, and manufacturing.

FunSearch, a novel AI-driven heuristic optimization technique leveraging Large Language Models (LLMs), iteratively generates and refines solution programs Romera-Paredes et al. (2024). Its adaptability makes it highly suited for solving complex optimization challenges like TSP, potentially uncovering innovative solutions beyond conventional heuristics.

Our project focuses on applying an enhanced FunSearch method, using GPT-3.5, to evolve efficient heuristics for solving TSP, benchmarking its performance against classical heuristics and modern learning-based approaches. Our code is provided in `https://github.com/Perkzi/TSP_funsearch` and the Jupyter notebook for running this project is provided in `https://colab.research.google.com/drive/1RyS66cFn_xuRa-nTjGUFfTF3qrFn79Kh?usp=sharing`.

2 MOTIVATION

We apply FunSearch to the Traveling Salesman Problem (TSP) because of several distinct advantages. First, FunSearch leverages a pre-trained LLM to automatically generate improved code—discovering novel heuristics and creative solutions that traditional, hand-designed algorithms might miss Liu et al. (2024). Second, it employs a distributed evolution strategy via an island model, where independent subpopulations periodically exchange information to overcome local optima and enhance exploration in the vast combinatorial space typical of TSP. Finally, the output of FunSearch is both executable and human-readable, ensuring code interpretability so that domain experts can easily understand and further refine the solution.

3 METHOD

3.1 FUNSEARCH

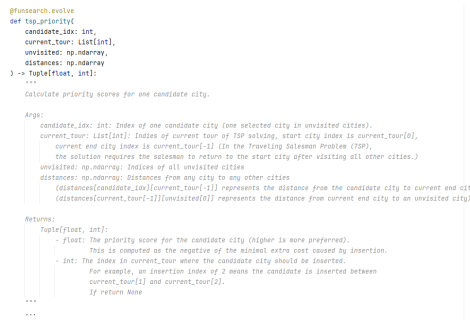
FunSearch (searching in the function space) is an innovative approach that combines pre-trained Large Language Models (LLM) with a system evaluator to tackle complex problems in mathematics and algorithm design. The key concept lies in harnessing the creative potential of LLMs to generate innovative solutions, while the evaluator ensures these solutions are refined, logical, and free from errors or inconsistencies.

How FunSearch Works The core mechanism of FunSearch is an evolutionary process. Starting with an initial program, the system iteratively refines and enhances the critical logic of the program. The goal is to continuously improve the program's output, evaluated using a defined "evaluation" function, to achieve higher performance scores.

The Initial Program The process begins with a specification, represented as a program string that serves as the initial foundation for solving a given problem (e.g., the Traveling Salesperson Problem). This program typically includes two main components:

- Evaluation Function: Decorated with funsearch.run, this function assesses the overall performance of a program.

- Priority Function: Decorated with funsearch.evolve, this function determines the next optimal step for the problem at hand. Specifically, it returns the score (priority) of one candidate in current step.

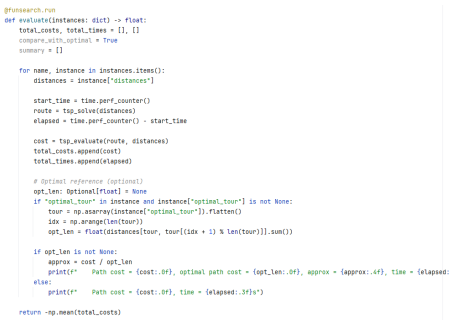


```python
@funsearch.evolve
def tsp_priority(
    candidate_idx: int,
    current_tour: List[int],
    unvisited: np.ndarray,
    distances: np.ndarray
) -> Tuple[float, int]:
    """
    Calculate priority scores for one candidate city.

    Args:
        candidate_idx: int: Index of one candidate city (one selected city in unvisited cities).
        current_tour: List[int]: Indies of current tour of TSP solving, start city index is current_tour[0],
            current end city index is current_tour[-1] (In the Traveling Salesman Problem (TSP),
            the solution requires the salesman to return to the start city after visiting all other cities.)
        unvisited: np.ndarray: Indices of all unvisited cities
        distances: np.ndarray: Distances from any city to any other cities
            (distances[candidate_idx][current_tour[-1]] represents the distance from the candidate city to current end city)
            (distances[current_tour[-1]][unvisited[0]] represents the distance from current end city to an unvisited city).

    Returns:
        Tuple[float, int]:
            - float: The priority score for the candidate city (higher is more preferred).
                     This is computed as the negative of the minimal extra cost caused by insertion.
            - int: The index in current_tour where the candidate city should be inserted.
                     For example, an insertion index of 2 means the candidate is inserted between
                     current_tour[1] and current_tour[2].
                     If return None
    """
    ...
```

Figure 1: Priority Function.



```python
@funsearch.run
def evaluate(instances: dict) -> float:
    total_costs, total_times = [], []
    compare_with_optimal = True
    summary = []

    for name, instance in instances.items():
        distances = instance["distances"]

        start_time = time.perf_counter()
        route = tsp_solve(distances)
        elapsed = time.perf_counter() - start_time

        cost = tsp_evaluate(route, distances)
        total_costs.append(cost)
        total_times.append(elapsed)

        # Optimal reference (optional)
        opt_len: Optional[float] = None
        if "optimal_tour" in instance and instance["optimal_tour"] is not None:
            tour = np.asarray(instance["optimal_tour"]).flatten()
            idx = np.arange(len(tour))
            opt_len = float(distances[tour, tour[(idx + 1) % len(tour)]].sum())

        if opt_len is not None:
            approx = cost / opt_len
            print(f"    Path cost = {cost:.0f}, optimal path cost = {opt_len:.0f}, approx = {approx:.4f}, time = {elapsed:.3f}s")
        else:
            print(f"    Path cost = {cost:.0f}, time = {elapsed:.3f}s")

    return -np.mean(total_costs)
```

Figure 2: Evaluation Function.

For instance, Fig. 1 showcases a priority function that evaluates the current state of a TSP problem and determines the next city to visit. Fig. 2 illustrates an evaluation function that iterates over datasets, generates feasible TSP paths using the priority function, computes the path cost for each instance, and calculates the average loss. This provides a straightforward metric for evaluating TSP solutions.

Evolutionary Process FunSearch employs an ”island” model to store and evolve different versions of priority functions. Initially, only the simplest version of the priority function is available. During each iteration:

1. Selection: Multiple versions (e.g., k versions) of priority functions are randomly sampled from the island and ranked based on their performance.
2. Generation: These selected functions are then fed into the LLM with an additional prompt (as illustrated in Fig. 3) to generate a new and potentially improved version of the priority function.
3. Update: The newly generated version is added to the island, enriching the diversity of solutions.

```python
base_prompt = (
            "Complete a different and more complex Python function. "
            "Be creative and you can insert multiple if-else and for-loop in the code logic. "
            "Only output the Python code, no descriptions."
        )
```

Figure 3: Additional Prompt.

To prevent stagnation in local optima, FunSearch incorporates a program database with multiple islands. Each island independently evolves its own set of priority functions. This ensures a broad exploration of the solution space. The complete evolutionary process is visually summarized in Fig. 4.

3.2 FUNSEARCH ON TSP

We adjust our prompting strategy and adopt three traditional TSP algorithms for evolution. As shown in Fig. 3, the base prompt is previously used for the bin packing problem. However, we find that the TSP priority function is more complex, and the LLM often changes its input/output logic, which causes errors. To address this, we add detailed parameter descriptions to the priority function and develop a TSP specific prompt (see Fig. 6). This prompt clearly describes the TSP problem and instructs the LLM not to modify the input/output logic or parameter descriptions.

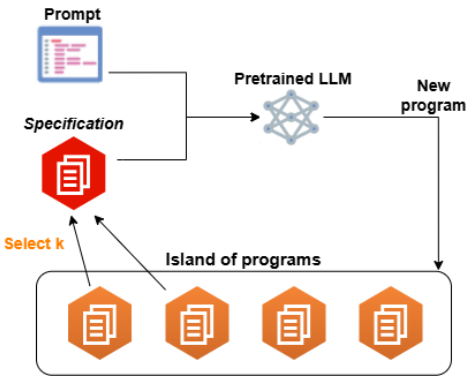


Figure 4: Evolutionary Process.

```
Start with two cities: [C0, C1]
visited = {C0, C1}

While not all cities are visited:
    For each unvisited city C:
        For each edge (i, j) in current Tour:
            Calculate cost of inserting C between i and j
    Choose city C and position (i, j) that gives minimal increase
    Insert C between i and j
    Add C to Visited

Return Tour
```

Figure 5: Priority Function of Cheapest Insertion.

```
tsp_specific_prompt = (
            "Complete a different and more complex Python function to solve TSP problem. "
            "You have to travel like a circle to visit all cities and finally go back to the start city."
            "For example, if you always choose the nearest, you will have to pay a lot for going back to the start city"
            "Be creative and you can insert multiple if-else and for-loop in the code logic. "
            "Only output the Python code without any additional explanations. "
            "Do not alter the input/output logic of the tsp_priority function (ensure it only returns a single float). "
            "Do not modify the parameter description in the tsp_priority function."
        )
```

Figure 6: TSP Specific Prompt.

3.2.1 TRADITIONAL TSP ALGORITHM

We start Funsearch Iteration on three traditional TSP algorithms Rosenkrantz et al. (1977) Lin & Kernighan (1973) Potvin (1996).

Cheapest Insertion: This method evaluates each potential gap in the current tour—considering every pair of consecutive cities (and wrapping around between the last and first cities)—and calculates the extra cost incurred if the candidate city were inserted into that gap. The extra cost is computed as: $cost = distance(prev, candidate) + distance(candidate, next) - distance(prev, next)$. The candidate's priority is then defined as the negative of the smallest extra cost found, so that candidates causing the least increase in the overall tour trajectory are prioritized. Essentially, this approach seeks to minimize the additional distance added to the tour by finding the optimal insertion position. The pseudo code of cheapest insertion is shown in Fig. 5.

Farthest Insertion: This strategy picks the candidate city that is most isolated from the currently visited cities, thereby helping to secure coverage of far-flung areas early in the tour construction. It begins by determining the candidate's "farness" by computing the minimum distance from the candidate city to any city in the existing tour—the larger this minimal distance, the more remote the candidate is. In addition to this score, the function also calculates the best insertion point (using the extra cost method) to integrate the candidate while keeping the tour efficient. The overall logic is to embrace candidates that are far away, giving them a high priority so as to prevent difficult insertions later in the solution process.

Nearest Insertion: This approach, in contrast, favors candidates that are near the current tour. The primary score is computed as the negative of the candidate's minimum distance to any visited city, meaning that those with smaller distances (thus closer proximity) yield a higher priority (since the negative of a lower number is higher). Similar to the other methods, it then evaluates all possible gaps in the tour using the extra cost calculation to decide the best insertion position. The rationale behind this strategy is that by adding the nearest candidate, the overall tour length will likely remain short, as the new addition connects easily to the existing clusters of visited cities.

3.2.2 PROMPTING METHOD

To achieve more flexible prompting, we implement a two-step prompting process. An optional pre-prompt is added before the main prompt, allowing us to direct the LLM to propose a specific evolution method based on the pre-prompt's guidance. After that, we combine main prompt and the output of pre-prompt to ask LLM to get new version of function. A sample of this two-step prompt is shown in Fig. 7.

```
self_analysis_prompts = [
    # Step 1: Require the model to provide a detailed explanation of the current heuristic TSP code's strategy, limitations, and failed instances
    (
        "To solve TSP problem, you have to travel like a circle to visit all cities and finally go back to the start city."
        "Please analyze the given heuristic TSP code by outlining its implementation strategy, limitations, and potential failure cases (including a specific example). "
        "Discuss how the algorithm might be improved. Only output your analysis text, no code."
    ),
    # Step 2: Require the model to generate improved code based on the above analysis
    (
        "Complete a different and more complex Python function for the TSP problem. "
        "Based on the following analysis: <insert self-analysis here>, "
        "rewrite and improve the code to address the identified issues. "
        "Only output the Python code without any additional explanations. "
        "Do not alter the input/output logic of the tsp_priority function (ensure it only returns a single float). "
        "Do not modify the parameter description in the tsp_priority function."
    )
]
```

Figure 7: Self Analysis Prompts.

Besides TSP Specific Prompt, we also incorporate some more complex prompts in experiment:

Step by Step Prompts: This prompt asks the model to break down the solution into clear, ordered steps. It encourages the model to explain the process in phases (like starting, improving, and finishing) so that the logic is easy to follow and debug.

Multi Path Analysis Prompts: This prompt encourages the model to explore and compare multiple algorithmic strategies, such as combining divide-and-conquer approaches with heuristic search methods. It directs the model to discuss the benefits, pitfalls, and even potential failure cases of each pathway.

Heuristic Meta Prompts: This prompt aims to meld local heuristic methods with global metaheuristic optimization techniques, like simulated annealing or genetic algorithms Burke et al. (2013).

Multi Phase Prompts: This prompt asks the model to plan the solution in several parts or stages. It wants the model to explain the benefits of splitting the work into phases (such as early, middle, and final stages) to make the code easier to read and maintain.

Self Analysis Prompts: This prompt asks the model to look at the current algorithm and explain how it works, point out where it might fail, and suggest improvements. After this self-review, the model then writes improved code based on the analysis Ye et al. (2024).

3.2.3 2-OPT OPTIMIZATION

We applied 2-opt during one of our FunSearch evolution procedures.

Optimization Logic: 2-opt iteratively removes two edges and reconnects the route to eliminate crossings and shorten the tour. It applies edge reversals whenever they yield a shorter path, repeating until no further improvements are possible.

Pros and Cons with FunSearch: Integrating 2-opt provides local refinement without altering the global structure of candidate solutions. It helps clean noisy or inefficient routes produced by language model-guided mutations, enabling more accurate evaluations. However, excessive reliance on 2-opt can bias evolution toward small local improvements, suppress exploration, and reduce diversity, leading to premature convergence. Thus, while 2-opt enhances evaluation stability, it must be applied judiciously to preserve the creative potential of function evolution.

4 EXPERIMENTS

4.1 EXPERIMENT SETTING

We selected seven benchmark instances from the TSPLIB dataset: *berlin52*, *eil76*, *kroA100*, *pr124*, *d198*, *lin318*, and *pcb442*. These symmetric TSP instances range from 52 to 442 cities and offer diverse spatial patterns, providing a robust testbed for evaluating heuristic generalization. FunSearch

was applied to *d198*, using six prompt templates and three traditional algorithms as seeds. For each prompt, four candidate functions were sampled per iteration, yielding 100 samples in total across 10 parallel islands. Table 2 describes the properties of the datasets used.

To assess the impact of local optimization, we conducted two FunSearch runs—one with 2-opt enabled during solving, and one without. The best-evolved heuristic from each setting, together with three baseline algorithms and a neural network solver, was evaluated across all instances under both conditions: with and without 2-opt refinement. This setup enabled a fair comparison between intrinsic heuristic quality and improvements brought by local search.

We take the neural network solver in Kool et al. (2019) for comparison. The model is based on attention mechanisms for solving the traveling salesman problem (TSP), vehicle routing problem (VRP), and other combinatorial optimization problems. The core idea is to use a transformer encoder to process input data, where each node (city) is represented as a vector, and multi-head attention is applied to capture relationships between nodes, allowing the model to learn complex routing structures. The optimization process is conducted through reinforcement learning (RL), avoiding the need for supervised labels by directly optimizing the model's policy during the search process.

Algorithm	Time	Tour Length
cheapest	1.590	17632
farthest	2.996	16309
nearest	2.446	18053
neural solver	-	29892

Table 1: Baseline performance on d198

Instance	Number of Cities	Optimal Tour Length
berlin52	52	7542
eil76	76	538
kroA100	100	21282
pr124	124	59030
d198	198	15780
lin318	318	42029
pcb442	442	50778

Table 2: Dataset Property

4.2 Experiment Result

Both best-performing evolved heuristics were derived from the Farthest Insertion baseline. Compared to the original algorithm, which prioritized cities solely based on maximum minimum distance, the first evolved version (without 2-opt) introduced a slight penalty for selecting overly close cities, encouraging exploration and mitigating greedy behavior. The second evolved version (with 2-opt) further refined priority computation by incorporating normalized insertion costs, favoring cities that were both near and minimally disruptive to the tour structure. These modifications significantly improved global route quality and enhanced generalization across different TSP instances. Additionally, for each algorithm, we computed the ratio between the generated tour length and the known optimal length across all TSP instances, and calculated the average gap before and after applying 2-opt optimization. Table 1 reports baseline and neural solver results on the d198 instance. The summarized results are presented in Figure 8 and Table 3. Table 4 summarizes the performance across different prompt templates.

Method	Gap (2-opt)	Gap
2-opt Evolved	2516	2646
Cheapest Insertion	2828	4322
Evolved	2906	2989
Farthest Insertion	2919	3006
Nearest Insertion	3058	5401

Table 3: Average Tour Gaps to Optimal with and without 2-OPT

From the ratio curves and average gap tables, several key observations can be made: First, applying 2-opt significantly improves solution quality across all methods, reducing both the tour length ratio and the average gap relative to the optimal tour. The impact is especially pronounced for traditional heuristics like Nearest Insertion and Cheapest Insertion, which show large improvements after

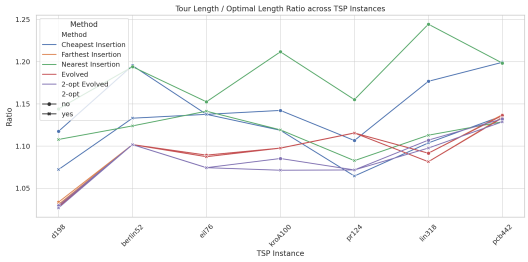


Figure 8: Tour Length / Optimal Length Ratio Across TSP Instances

applying 2-opt, narrowing the gap with the evolved heuristics. Second, the evolved heuristics, even without 2-opt, generally outperform traditional baselines, indicating that FunSearch is able to discover more effective construction strategies. When combined with 2-opt, evolved heuristics achieve the lowest ratios and gaps, demonstrating strong synergy between learned global structure and local refinement. Third, across different instances, the benefit of 2-opt tends to be larger for more complex or larger TSPs (such as lin318 and pcb442), where local suboptimalities accumulate more heavily. In smaller instances like berlin52 or eil76, the relative improvement is still present but less dramatic.

In addition, we compared the performance of heuristics evolved with 2-opt enabled during the evolution process versus those evolved without it. The results show that enabling 2-opt during evolution leads to slightly better final heuristics, especially when evaluated without additional 2-opt post-processing. This suggests that applying local refinement during evolution not only improves immediate fitness evaluation but also guides the search toward generating inherently stronger heuristics. However, when final solutions are further refined by 2-opt at test time, the performance gap between the two evolution settings becomes narrower.

Prompt	Basic Algorithm	Best Time	Best Tour Length
TSP Specific Prompt	cheapest	2.849	17173
	farthest	2.594	16231
	nearest	23.02	16675
Step by Step Prompts	cheapest	3.098	16309
	farthest	2.548	16309
	nearest	7.092	16850
Multi Path Analysis Prompts	cheapest	3.183	16699
	farthest	2.548	16309
	nearest	2.800	16520
Heuristic Meta Prompts	cheapest	-	-
	farthest	-	-
	nearest	2.658	16827
Multi Phase Prompts	cheapest	2.890	17269
	farthest	2.644	16309
	nearest	2.761	16827
Self Analysis Prompts	cheapest	2.603	17616
	farthest	2.632	16309
	nearest	2.724	17518

Table 4: Performance of Different Prompts on TSP

5 Conclusion

This study highlights the advantages of evolved heuristics in constructing high-quality travel routes and the crucial role of local search mechanisms like 2-opt in optimization. Evolutionary algorithms effectively identify superior strategies, while 2-opt significantly enhances path quality. Their combination not only improves solution accuracy but also strengthens generalization across different TSP instances. Future research can explore more effective ways to integrate local optimization into the evolutionary process to further enhance algorithm performance.

REFERENCES

E. K. Burke, M. Gendreau, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013. doi: 10.1057/jors.2013.71.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ByxBFsRqYm.

S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973. doi: 10.1287/opre.21.2.498.

F. Liu, X. Tong, M. Yuan, X. Lin, F. Luo, Z. Wang, Z. Lu, and Q. Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235, pp. 32201–32223, Vienna, Austria, Jul 2024.

J.-Y. Potvin. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research*, 63:337–370, 1996. doi: 10.1007/BF02125403.

B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. Kumar, E. Dupont, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6.

D. J. Rosenkrantz, R. E. Stearns, and P. M. II Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, 1977. doi: 10.1137/0206041.

H. Ye, J. Wang, Z. Cao, F. Berto, C. Hua, H. Kim, J. Park, and G. Song. Reevo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37, 2024. Available online at the NeurIPS Proceedings.