CT421 – Artificial Intelligence

Technical Report for Genetic Algorithm TSP Solver

Name: Ayodeji Ali
Student ID: 20343733

Contents

Implementation Details and Design Choices

Overview

The Generic Algorithm is designed to solve TSP instances represented by a TSBLIB format. The code is split into 2 separate files, one file for the core GA logic (tsp_ga.py) and the other file for running the grid-search style experiments (tsp_experiments.py).

Key Components

TSPLIB Parser and Distance Matrix

I parsed the TSPLIB file. The parser reads the file and extracts the (x, y) coordinates for each city. These coordinates are then used to compute a distance matrix, where the distance between two cities is calculated using Euclidean distance. For large datasets—such as pr1002.tsp, which has 1002 cities— NumPy was used to vectorise operations to significantly speed up the computation.

```
def read_tsplib(filename):  2 usages
    """
    Reads a TSPLIB file and returns a list of (x, y) coordinates.
    """
    coords = []
    with open(filename) as f:
        # Skip lines until we find the NODE_COORD_SECTION
        for line in f:
            if line.strip().upper().startswith("NODE_COORD_SECTION"):
                break
        # Read coordinate data until "EOF" or an empty line is encountered
        for line in f:
            line = line.strip()
            if line == "EOF" or not line:
                break
            parts = line.split()
            if len(parts) >= 3:
                x = float(parts[1])
                y = float(parts[2])
                coords.append((x, y))
    return coords


def create_distance_matrix(coords):  2 usages
    """
    Creates a distance matrix using NumPy vectorized operations.
    """
    arr = np.array(coords)  # shape: (n, 2)
    # Compute pairwise differences using broadcasting
    diff = arr[:, np.newaxis, :] - arr[np.newaxis, :, :]
    # Compute Euclidean distances
    dist_matrix = np.hypot(diff[:, :, 0], diff[:, :, 1])
    return dist_matrix
```

Figure 1: A diagram showing the TSPLIB file structure, highlighting the extraction of city coordinates and the subsequent formation of a coordinate array.

Representation and Initialisation

Each TSP solution is represented as a permutation of city indices. This encoding directly reflects the order in which cities are visited. The initial population was created by randomly shuffling the list of cities, this ensured that the GA starts with a diverse set of solutions.

Selection

For selecting parents, tournament selection was used. This method randomly selects a subset of individuals from the population and then chooses the one with the highest fitness. Fitness is defined as the inverse of the tour length; shorter tours have higher fitness. Tournament selection effectively balances exploration and exploitation by favouring good individuals while still giving a chance to others.

Genetic Operators

Two types of genetic operators were used in the implementation:

- **Crossover:**
  Two crossover operators we implemented:
    - **Order Crossover (OX):** This preserves the relative order of cities from one parent.
    - **Partially Mapped Crossover (PMX):** This exchanges segments between parents while preserving absolute positions.

These operators are essential for recombining good sub-solutions from parent solutions.

- **Mutation:**
  Mutation introduces random changes to individuals, which helps avoid local optima. The implementation includes:
    - **Swap Mutation:** Two cities swap positions.
    - **Inversion Mutation:** A segment of the tour is reversed.

Together, the crossover and mutation maintain diversity and guide the GA towards high-quality solutions.

The Main GA Loop and Termination

The GA loop continuously evaluates the fitness of the population, applies tournament selection, and uses crossover and mutation to generate new offspring. This process repeats for a fixed number of generations. The framework is flexible enough to support alternative termination criteria like convergence.

```python
def run_ga(dist_matrix, pop_size=100, generations=500, crossover_rate=0.8, mutation_rate=0.2):  2 usages
    """
    Runs the Genetic Algorithm for the TSP.
    Returns the best solution, its tour length, the fitness history, and computational time.
    """
    num_cities = dist_matrix.shape[0]
    population = initialize_population(pop_size, num_cities)
    best_solution, best_length = None, float('inf')
    fitness_history = []
    start_time = time.time()

    for _ in range(generations):
        fitnesses = evaluate_population(population, dist_matrix)
        for ind in population:
            L = tour_length(ind, dist_matrix)
            if L < best_length:
                best_length, best_solution = L, ind[:]
        fitness_history.append(1.0 / best_length)

        selected = tournament_selection(population, fitnesses, tournament_size=5)
        new_population = []
        while len(new_population) < pop_size:
            p1, p2 = random.choice(selected), random.choice(selected)
            if random.random() < crossover_rate:
                # Randomly choose between Order Crossover and PMX
                child = order_crossover(p1, p2) if random.random() < 0.5 else pmx_crossover(p1, p2)
            else:
                child = p1[:]
            if random.random() < mutation_rate:
                # Randomly choose between swap and inversion mutation
                swap_mutation(child) if random.random() < 0.5 else inversion_mutation(child)
            new_population.append(child)
        population = new_population

    comp_time = time.time() - start_time
    return best_solution, best_length, fitness_history, comp_time
```

Figure 3: Code Snippet show how Genetic algorithm loop is implemented.

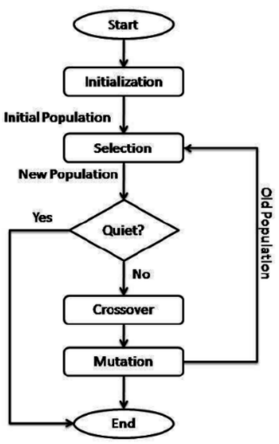

Figure 4: The main GA loop: initialization → fitness evaluation → selection → crossover/mutation → new generation → termination.

The function crossover_and_mutate encapsulates the logic of choosing between the two crossover operators and applying one of the two mutations, keeping the main loop concise.

Modularity

The separation of functionality into **tsp_ga.py** and **tsp_experiments.py** enhances reusability and maintainability. The core GA functions can be independently tested and reused, while the experimental runner manages parameter sweeps and data collection without modifying the underlying algorithm.

Experimental Results and Analysis

Experimental Setup

I evaluated the GA on three standard TSP datasets:

- **berlin52.tsp (52 cities)**
- **kroA100.tsp (100 cities)**
- **pr1002.tsp (1002 cities)**

I conducted experiments using a grid search over a range of parameter values:

- **Population sizes:** 50, 100, 200
- **Crossover rates:** 0.6, 0.8, 1.0
- **Mutation rates:** 0.1, 0.2, 0.3

Each parameter combination was run multiple times (e.g., 10 runs) to account for the stochastic nature of the algorithm. For each configuration, the best tour length and computation time were recorded, with averages and standard deviations computed to assess performance variability.

Results Collection

Results for each dataset are saved into separate CSV files (e.g., *berlin52_results.csv, kroA100_results.csv, pr1002_results.csv*). This separation allows for targeted analysis of each dataset's performance across different configurations.

```python
def save_results_to_csv_by_dataset(results):  1 usage
    """
    Saves separate CSV files for each dataset.
    """
    grouped = defaultdict(list)
    for r in results:
        grouped[r["dataset"]].append(r)
    for dataset, res_list in grouped.items():
        filename = f"{os.path.splitext(dataset)[0]}_results.csv"
        fieldnames = ["dataset", "cities", "pop_size", "crossover_rate", "mutation_rate",
                      "avg_length", "std_length", "avg_time"]
        with open(filename, "w", newline="") as csvfile:
            writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
            writer.writeheader()
            writer.writerows(res_list)
        print(f"Results for {dataset} saved to {filename}")
```

Figure 5: Sample code snippet of one of the CSV files of dataset, population size, crossover rate, mutation rate, average tour length, and standard deviation is created.

dataset	cities	pop_size	crossover	mutation_	avg_length	std_length	avg_time
berlin52.tsp	52	50	0.6	0.1	8957.753419	322.8066482	0.594162631
berlin52.tsp	52	50	0.6	0.2	8667.129617	314.0481814	0.600011754
berlin52.tsp	52	50	0.6	0.3	8603.052377	184.581966	0.598947811
berlin52.tsp	52	50	0.8	0.1	9171.382444	375.976771	0.644009328
berlin52.tsp	52	50	0.8	0.2	8538.191218	272.5808719	0.647417617
berlin52.tsp	52	50	0.8	0.3	8516.474091	296.3262718	0.652595329
berlin52.tsp	52	50	1	0.1	8965.075915	396.4584373	0.701071811
berlin52.tsp	52	50	1	0.2	8558.04859	373.326531	0.7146065
berlin52.tsp	52	50	1	0.3	8286.753556	279.3004239	0.717255831
berlin52.tsp	52	100	0.6	0.1	8625.64384	378.8616688	1.516784239
berlin52.tsp	52	100	0.6	0.2	8229.439005	209.9291054	1.760068893
berlin52.tsp	52	100	0.6	0.3	8296.74691	149.80911	1.361485767
berlin52.tsp	52	100	0.8	0.1	8536.026833	451.8500258	1.279868746
berlin52.tsp	52	100	0.8	0.2	8393.576646	304.794098	1.291267228
berlin52.tsp	52	100	0.8	0.3	8370.70209	273.9931486	1.299075651
berlin52.tsp	52	100	1	0.1	8642.176827	358.0563886	1.387543964
berlin52.tsp	52	100	1	0.2	8445.377959	244.5722871	1.39560647
berlin52.tsp	52	100	1	0.3	8349.96212	366.9813207	1.404215717
berlin52.tsp	52	200	0.6	0.1	8338.553179	130.9492183	2.359936881
berlin52.tsp	52	200	0.6	0.2	8453.268304	201.3947977	2.376322842
berlin52.tsp	52	200	0.6	0.3	8328.728762	240.3178907	2.392144561
berlin52.tsp	52	200	0.8	0.1	8236.259497	248.0731672	2.563583183
berlin52.tsp	52	200	0.8	0.2	8219.216631	139.6595589	2.578868175
berlin52.tsp	52	200	0.8	0.3	8267.802179	213.9251829	2.599653316
berlin52.tsp	52	200	1	0.1	8344.698367	135.0204171	2.789499807
berlin52.tsp	52	200	1	0.2	8162.834389	183.5321593	2.805869603
berlin52.tsp	52	200	1	0.3	8207.101501	137.9977007	2.84094274

Figure 6: Berlin52 csv File

dataset	cities	pop_size	crossover_	mutation_	avg_length	std_length	avg_time
kroA100.tsp	100	50	0.6	0.1	44586.21837	2467.605805	0.946455026
kroA100.tsp	100	50	0.6	0.2	36262.35933	1890.815087	0.952533889
kroA100.tsp	100	50	0.6	0.3	34276.53976	1914.100268	0.957562661
kroA100.tsp	100	50	0.8	0.1	43723.35291	1693.582991	1.086248088
kroA100.tsp	100	50	0.8	0.2	37103.07897	1843.685129	1.094040895
kroA100.tsp	100	50	0.8	0.3	33960.48423	1854.290056	1.101656365
kroA100.tsp	100	50	1	0.1	40576.61245	2296.843894	1.232312441
kroA100.tsp	100	50	1	0.2	36796.49385	1534.787528	1.240919709
kroA100.tsp	100	50	1	0.3	34860.01999	1575.194158	1.249913549
kroA100.tsp	100	100	0.6	0.1	34198.63966	1798.53845	1.896806979
kroA100.tsp	100	100	0.6	0.2	31046.80408	920.7971536	1.908838463
kroA100.tsp	100	100	0.6	0.3	28617.84194	1437.852782	1.919901466
kroA100.tsp	100	100	0.8	0.1	33629.07824	1105.821159	2.184002972
kroA100.tsp	100	100	0.8	0.2	30535.29188	1170.123719	2.195074034
kroA100.tsp	100	100	0.8	0.3	29224.99189	1807.305365	2.206642556
kroA100.tsp	100	100	1	0.1	34540.99626	1191.709339	2.478895211
kroA100.tsp	100	100	1	0.2	31850.95059	1528.22586	2.500995588
kroA100.tsp	100	100	1	0.3	31207.45749	1952.256059	2.539178252
kroA100.tsp	100	200	0.6	0.1	28755.8262	1228.076283	3.790078115
kroA100.tsp	100	200	0.6	0.2	26202.50888	746.4370057	3.815021729
kroA100.tsp	100	200	0.6	0.3	25209.41112	945.289083	3.843195391
kroA100.tsp	100	200	0.8	0.1	28303.2931	1460.318704	4.380080509
kroA100.tsp	100	200	0.8	0.2	27042.43573	825.9778039	4.413206029
kroA100.tsp	100	200	0.8	0.3	26023.97899	857.499288	4.436610985
kroA100.tsp	100	200	1	0.1	30541.23629	1505.396063	5.063867426
kroA100.tsp	100	200	1	0.2	31275.75022	6009.176509	5.168530965
kroA100.tsp	100	200	1	0.3	51902.79534	14530.74352	5.441168284

Figure 7: kroA100 csv File

dataset	cities	pop_size	crossover_	mutation_	avg_length	std_length	avg_time
pr1002.tsp	1002	50	0.6	0.1	3787045.868	40341.49762	37.78007848
pr1002.tsp	1002	50	0.6	0.2	3628811.802	35810.63595	38.52681525
pr1002.tsp	1002	50	0.6	0.3	3567423.452	40512.09768	38.87441788
pr1002.tsp	1002	50	0.8	0.1	3508914.13	50107.32892	137.7978158
pr1002.tsp	1002	50	0.8	0.2	3511837.507	41950.46167	53.72626519
pr1002.tsp	1002	50	0.8	0.3	3496618.256	36850.36447	56.10423481
pr1002.tsp	1002	50	1	0.1	3447130.006	26471.01046	70.28639395
pr1002.tsp	1002	50	1	0.2	3423676.104	38696.41172	71.90149341
pr1002.tsp	1002	50	1	0.3	3496665.924	50959.2236	68.98032689
pr1002.tsp	1002	100	0.6	0.1	3346452.667	28609.07094	81.55220156
pr1002.tsp	1002	100	0.6	0.2	3332911.712	27238.39665	87.26834157
pr1002.tsp	1002	100	0.6	0.3	3315097.323	42384.38874	85.58423481
pr1002.tsp	1002	100	0.8	0.1	3231072.015	35489.28582	115.7729718
pr1002.tsp	1002	100	0.8	0.2	3294774.458	49208.53014	120.9966584
pr1002.tsp	1002	100	0.8	0.3	3302846.62	62724.6886	119.7931096
pr1002.tsp	1002	100	1	0.1	3329505.464	65134.77637	142.9936216
pr1002.tsp	1002	100	1	0.2	3443568.631	66609.44529	143.4312106
pr1002.tsp	1002	100	1	0.3	3638516.48	81005.59307	144.2205396
pr1002.tsp	1002	200	0.6	0.1	3111916.23	43466.80129	184.3460354
pr1002.tsp	1002	200	0.6	0.2	3128136.048	32541.42793	275.4218927
pr1002.tsp	1002	200	0.6	0.3	3155168.143	49701.06456	266.4252516
pr1002.tsp	1002	200	0.8	0.1	3067080.652	40852.68503	353.2192778
pr1002.tsp	1002	200	0.8	0.2	3115313.513	38930.28969	443.3669498
pr1002.tsp	1002	200	0.8	0.3	3224523.963	43280.26932	233.729355
pr1002.tsp	1002	200	1	0.1	3600147.437	65075.25768	353.4574908
pr1002.tsp	1002	200	1	0.2	3738515.507	67954.66267	461.0836834
pr1002.tsp	1002	200	1	0.3	3843634.718	52821.19048	348.4117729

Figure 8: pr1002 csv File

These files highlight how different parameter combinations affect performance, with larger population sizes typically reducing **avg_length** but sometimes increasing **avg_time**.

Graphical Representation

To better visualize the experimental outcomes, I generated line graphs showing the relationship between population size and the best average tour length. Each dataset is plotted separately, and these graphs help identify the parameter points that yield the best performance.

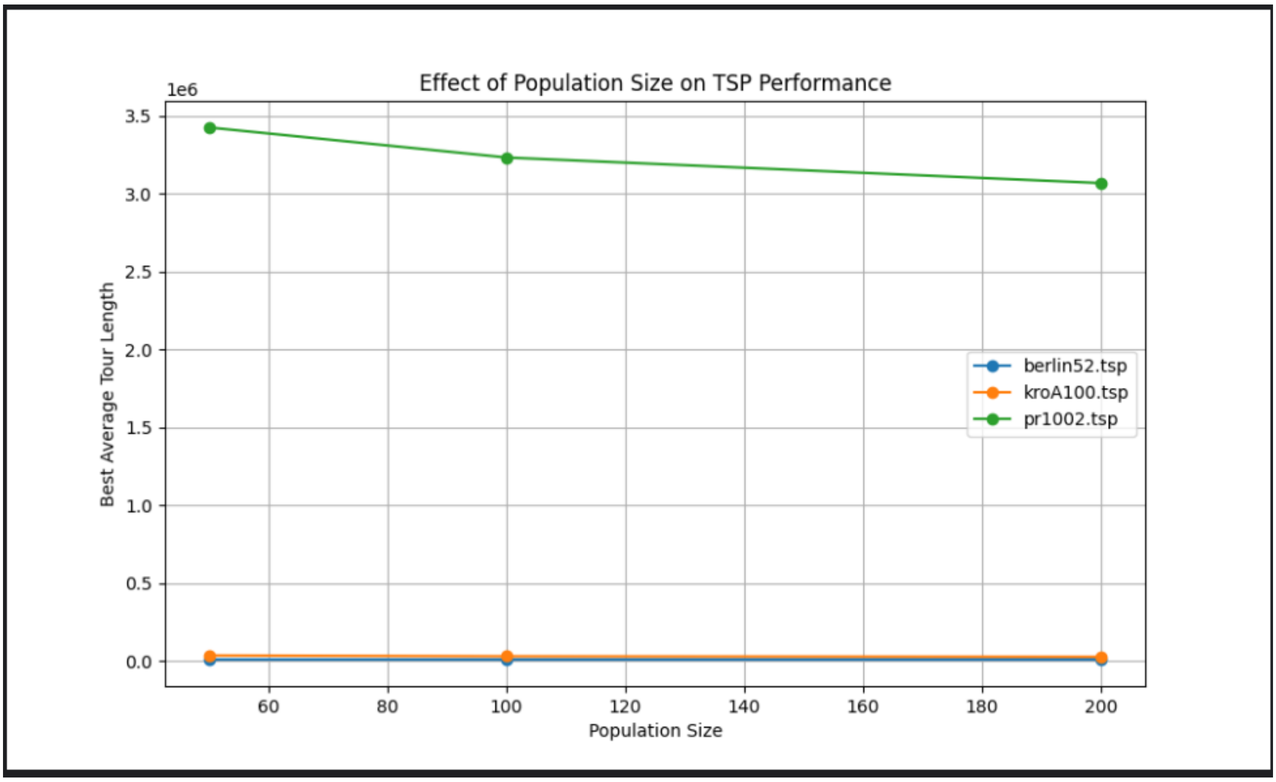

Figure 8: A line chart titled "Effect of Population Size on TSP Performance," plotting population size on the x-axis and best average tour length on the y-axis for berlin52, kroA100, and pr1002

In the figure, you can see that **berlin52** and **kroA100** remain near the lower end of the y-axis, reflecting relatively smaller tour lengths, while **pr1002**—being much larger—shows higher tour lengths overall. This chart helps identify parameter "sweet spots" for each dataset.

```python
def plot_results(results):  1 usage
    """
    Creates a line graph for each dataset showing the best (lowest) average tour length vs. population size.
    For each dataset, for every population size, the best average length (across all crossover and mutation settings) is plotted.
    """
    # Group results by dataset and then by population size
    dataset_data = defaultdict(dict)
    for r in results:
        dataset = r["dataset"]
        pop = r["pop_size"]
        # Choose the best (lowest) avg_length for this population size if multiple entries exist
        if pop in dataset_data[dataset]:
            dataset_data[dataset][pop] = min(dataset_data[dataset][pop], r["avg_length"])
        else:
            dataset_data[dataset][pop] = r["avg_length"]

    plt.figure(figsize=(10, 6))
    for dataset, pop_dict in dataset_data.items():
        pop_sizes = sorted(pop_dict.keys())
        avg_lengths = [pop_dict[pop] for pop in pop_sizes]
        plt.plot( *args: pop_sizes, avg_lengths, marker='o', label=dataset)
    plt.xlabel("Population Size")
    plt.ylabel("Best Average Tour Length")
    plt.title("Effect of Population Size on TSP Performance")
    plt.legend()
    plt.grid(True)
    plt.show()
```

Figure 9: Code Snippet for line graph for relationship between population size and the best average tour length.

Analysis

Overall, the experiments reveal that:

- Increasing population size tends to improve solution quality (lower tour lengths), though returns diminish at larger sizes.
- Higher crossover rates often yield quicker improvements in early generations, while a moderate mutation rate helps maintain diversity.
- The consistent (relatively low) standard deviations across multiple runs indicate that the GA is robust and not overly sensitive to random variation.

Performance Comparison with Known Optimal Solutions

Benchmarking against known optimal tour lengths was important for evaluating the effectiveness of the GA. For example, berlin52.tsp has a known optimal tour length of approximately 7542.

In the experiments:

- **berlin52.tsp:**
  The GA often finds solutions close to this optimum when properly tuned.
- **kroA100.tsp:**
  Our GA produces competitive tour lengths, with some run-to-run variation due to stochastic elements.
- **pr1002.tsp:**
  While the GA may not always reach the absolute best-known solution, vectorized operations and careful parameter tuning allow us to approach high-quality solutions within an acceptable margin.

These findings underscore that, with appropriate parameters, the GA reliably converges to near-optimal results, even for larger, more complex instances.

Discussion of Potential Improvements

While the current GA implementation is effective, several enhancements can further improve its performance:

Algorithmic Enhancements

One promising direction is to combine GA with local search techniques, such as hill climbing or simulated annealing, as a post-processing step. This hybrid approach can fine-tune solutions by exploring the neighbourhood of high-quality individuals.

Adaptive parameter tuning—where the population size, crossover rate, and mutation rate adjust dynamically based on convergence behaviour—could accelerate the search process and yield better solutions.

Performance Optimization

The implementation already benefits from NumPy vectorization, but further speed improvements can be achieved through parallelization. For instance, parallelizing fitness evaluation using multi-threading or multiprocessing would significantly reduce computational time for large instances like pr1002. Advanced techniques such as just-in-time compilation with Numba or optimized data structures could also be explored.

Alternative Genetic Operators

Further experiments with alternative crossover and mutation operators could reveal methods that preserve useful building blocks better. Enhancing elitism—ensuring the best individuals persist between generations—could also contribute to improved performance.

Experimentation and Analysis

Extending the grid search to explore additional parameters and integrating more sophisticated visualization and statistical analysis tools will provide deeper insights into the convergence behaviour and parameter effects.

References

Mathworks.com. (2019). *VisibleBreadcrumbs*. [online] Available at: https://uk.mathworks.com/help/gads/what-is-the-genetic-algorithm.html.

GeeksforGeeks (2017). *Genetic Algorithms*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/genetic-algorithms/.

Otman Abdoun, Chakir Tajani and Jaafar Abouchabka (2012). Hybridizing PSM and RSM Operator for Solving NP-Complete Problems: Application to Travelling Salesman Problem. *International Journal of Computer Science Issues*, [online] 9(1). Available at: https://www.researchgate.net/publication/221901574_Hybridizing_PSM_and_RSM_Operator_for_Solving_NP-Complete_ProblemsApplication_to_Travelling_Salesman_Problem.

comopt.ifi.uni-heidelberg.de. (n.d.). *TSPLIB*. [online] Available at: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

www.tutorialspoint.com. (n.d.). *Genetic Algorithms - Parent Selection - Tutorialspoint*. [online] Available at: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm.

Instructure.com. (2025). *Sign in to your account*. [online] Available at: https://universityofgalway.instructure.com/courses/31961/files/2350557?module_item_id=752749.

Gharsalli, L. (2022). Hybrid Genetic Algorithms. *IntechOpen eBooks*. [online] doi:https://doi.org/10.5772/intechopen.104735.

Tsai, C.-W. and Chiang, M.-C. (2023). Hybrid metaheuristic and hyperheuristic algorithms. *Handbook of Metaheuristic Algorithms*, [online] pp.321–350. doi:https://doi.org/10.1016/b978-0-44-319108-4.00029-0.

GeeksForGeeks (2023). *Hyperparameter tuning*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/hyperparameter-tuning/.