CSCU9A3 – The Assignment 2024

Optimising the Traveling Salesperson Problem: Data Structures, Algorithms, and Performance Analysis in Java

# Part 1: Interface Design and Object-Oriented Structure

## Reflection 1 b.

The OptimisationAlgorithm interface makes the code more flexible by separating the way algorithms work from how they are used. The modular design allows the addition or swapping of optimization algorithms without changing the rest of the code. It also makes the program easier to expand by adding new algorithms in the future. Different algorithms can be tested or used in the program without altering its main workflow, and each algorithm can be tested on its own to ensure it works correctly.

## Reflection 2 b.

The refactoring process was straightforward because both algorithms were already structured to work with cities and distances. Implementing the interface made the contract clearer and allowed both classes to have consistent behaviour. It also made testing easier, as both classes could now be swapped without modifying the main program. The new structure will allow for simpler implementation of future extensions as it emphasizes modularity.

Reflection 3 b.

Separating TSPProblem has allowed the data handling (city input) and distance calculation to remain independent of the other algorithms. This in turn helps improve reusability and clarity, as classes focus purely on optimization logic while the problem class handles domain-specific details.

Reflection 4 b.

The unit testing highlighted edge cases like zero or one city, ensuring robustness. Writing tests has helped make cleaner and more modular code with better error handling, which in turn has improved the maintainability of the overall system.

# Part 2: Algorithm Development and Optimisation

## Reflection 5 b.

I used Simulated Annealing because it helps find good solutions quickly, especially for bigger problems. Unlike brute-force, which checks every possible path, Simulated Annealing doesn’t check everything but still gets good results by jumping out of local solutions that aren’t the best. It’s better than the greedy nearest-city algorithm because it looks at more options, but it’s not perfect and depends on setting the right parameters.

## Reflection 6 b.

The stack helped manage the cities in the route and made it easier to go back and try a different path when needed. By adding cities to the stack as we visit them, I could keep track of where we are. If I needed to backtrack, I could remove cities from the stack and try a new path. The challenge was making sure I correctly updated which cities had been visited and handling the stack properly when backtracking.

## Reflection 7 b.

I focused on making sure the algorithms gave valid routes that visited every city once and returned to the start. I tested the algorithms with small problems where I knew the correct answer, to make sure they were working right. I also tested with very few cities to make sure the algorithms could handle simple cases.

## Reflection 8 b.

Average of executing 10 times (in ms):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | 10 Cities | 20 Cities | 30 Cities | 40 Cities | 50 Cities |
| DeliveryRouteOptimiser | 89.16484 | N/A | N/A | N/A | N/A |
| NearestCityOptimiser | 0.3266 | 0.4103 | 0.502 | 0.538 | 0.782 |
| HeuristicOptimiser | 0.405 | 0.4552 | 0.6049 | 0.6279 | 0.8186 |
| BacktrackingOptimiser | 231.3494 | N/A | N/A | N/A | N/A |

I found that the Brute Force algorithm (DeliveryRouteOptimiser) performs well for small datasets but becomes impractical for larger ones due to its factorial time complexity, which causes execution times to grow extremely fast as the number of cities increases.

The Greedy algorithm (NearestCityOptimiser) performs significantly better, showing a quadratic increase in execution time, which is much more scalable for datasets with up to 50 cities.

The Heuristic algorithm (HeuristicOptimiser) performs similarly to the greedy algorithm, with slightly longer execution times, but still is efficient compared to brute force and backtracking.

The Backtracking algorithm performs well for a small numbers of cities but quickly becomes impractical for larger datasets due to its exponential time complexity.

# Part 3: Data Structures and Efficiency

## Reflection 9 b.

I chose an array-backed list because it provides significant performance improvements for random access and iteration. Its O(1) access time and efficient memory usage make it ideal for TSP, where accessing and modifying cities by index is quite common. Also, no extra memory is required for pointers like in a linked list and arrays can grow in a more dynamic manner by resizing, which in the end it can handle larger datasets effectively. But adding or removing elements in the middle is slower being O(n) due to shifting, in comparison to a doubly linked list which would be O(1) for such operations. Even though this is a trade-off to consider these operations are less frequent in the TSP.

## Reflection 10 b.

The array-backed list is faster due to its efficient resizing, while the linked list is slower as it must traverse to the end to add a new element. But it's much quicker with O(1) access, compared to the linked list’s O(n) access time while getting cities. And for removing cities it is slower as the elements must shift while the linked list adjusts the pointers.

## Reflection 11 b.()

I chose a HashMap for efficient city-to-city distance lookups. It provides O(1) average-time retrieval, making it faster and more memory-efficient for sparse datasets compared to a 2D matrix. As the number of cities grows, HashMaps scale well by only storing relevant distances, avoiding redundant data.

A trade-off is the O(n²) setup time to populate the HashMap and slight memory overhead due to hash table management. However, this approach improves scalability and flexibility in handling sparse graphs, optimizing performance for larger datasets.

## Reflection 12 b.

The unit tests helped me identify an issue I had with the refactored NearestCityOptimiser. testNearestCityOptimiserEdgeCaseEmptyList helped identify a bug in which the optimiser did not handle empty city lists correctly, and caused the test to fail. This revealed a gap in the algorithm's robustness for edge cases.

After which I updated the findBestRoute method to check for empty inputs explicitly and return a valid DeliveryRoute with an empty route and zero distance. This fix made sure the optimiser could handle all input sizes, including edge cases.

## Reflection 13 b.

Significant Reduction in Execution Time for NearestCityOptimiser and HeuristicOptimiser. After refactoring there was a deacrease from 0.3266 ms to 0.00605 ms and 0.405 ms to 0.00827, making both about 50 times faster.

DeliveryRouteOptimiser and BacktrackingOptimiser also had a big improvement in performance, here execution time dropped from 89.16484 ms to 13.0266 ms (DeliveryRouteOptimiser) and 231.3494 ms to 12.1019 ms (BacktrackingOptimiser). Even with the improvement they are still unable to run 20 cities and above, making both not scalable.

This shows that after refactoring the NearestCityOptimiser and HeuristicOptimiser were more scalable, being able to handle larger problem sizes with little increase in runtime.

Average of executing 10 times (in ms):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | 10 Cities | 20 Cities | 30 Cities | 40 Cities | 50 Cities |
| DeliveryRouteOptimiser | 13.0266 | N/A | N/A | N/A | N/A |
| NearestCityOptimiser | 0.00605 | 0.01502 | 0.01933 | 0.02614 | 0.03087 |
| HeuristicOptimiser | 0.00827 | 0.02017 | 0.0279 | 0.03592 | 0.04524 |
| BacktrackingOptimiser | 12.1019 | N/A | N/A | N/A | N/A |

# Part 4: Complexity Analysis and Algorithm Comparison

## Analysis Task 14 a.

Describe your analysis and results here

## Discussion Task 14 b.

Write your discussion here

## Analysis Task 15 a.

Describe your analysis and results here

## Discussion Task 15 b.

Write your discussion here

## Analysis Task 16 a.

Describe your analysis and results here

## Discussion Task 16 b.

Write your discussion here

## Analysis Task 17 a.

Describe your analysis and results here

## Discussion Task 17 b.

Write your discussion here

## Analysis Task 18 a.

Describe your analysis and results here and in the table below

## Discussion Task 18 b.

Write your discussion here

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | Time Complexity | Space Complexity | Runtime (10 cities) | Runtime (50 cities) | Total Distance (10 cities) | Total Distance (50 cities) |
| Brute-force | O(?) | O(?) | X ms | X ms | X km | X km |
| Nearest-city (refactored) | O(?) | O(?) | X ms | X ms | X km | X km |
| New Algorithm | O(?) | O(?) | X ms | X ms | X km | X km |