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

Reflection 4 b.

The unit testing selected edge cases like zero or one city, ensuring robustness of the algorithms. Writing these 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 they were visited, 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 biggest challenge for me 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.

"N/A”: Algorithm couldn't run for these cases due to time limits or memory issues.

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 number 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 more dynamically by resizing, which can handle larger datasets. But adding or removing elements in the middle is slower being O(n) due to shifting, in comparison a doubly linked list would be O(1) for such operations. Even though this is a trade-off to consider, the operations tend to be less frequent.

## 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 a more efficient distance lookup when looking from city to city. It has an O(1) retrieval on average, making it faster and more memory efficient for sparse datasets compared to a 2D matrix. Hash Maps are also scalable as they only store relevant distances, avoiding redundant data, which all helps as the number of cities grows.

The major trade-off is the O(n^2) time to setup and populate the hash map, plus the extra use of memory due to hash table management. Although, this approach improves scalability and flexibility in handling sparse graphs, optimizing performance for more cities in the dataset.

## 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 showed 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.

Refactoring drastically improved performance. The NearestCityOptimiser decreased from 0.3266 ms to 0.0605 ms (10 cities) and scaled efficiently to 50 cities with just 0.2392 ms. The HeuristicOptimiser dropped from 0.405 ms to 0.02179 ms (10 cities) and reached 0.1539 ms for 50 cities. Both are still highly scalable to an increase in the number of cities.

The DeliveryRouteOptimiser improved from 89.16484 ms to 13.0266 ms (10 cities), and the BacktrackingOptimiser from 231.3494 ms to 12.1019 ms. Still showing signs of a lack of scalability.

"N/A”: Algorithm couldn't run for these cases due to time limits or memory issues.

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 | 0.0605 | 0.1502 | 0.1933 | 0.2114 | 0.2392 |
| HeuristicOptimiser | 0.02179 | 0.04758 | 0.07424 | 0.09836 | 0.1539 |
| BacktrackingOptimiser | 12.1019 | N/A | N/A | N/A | N/A |

The changes in CityList has helped as most operations are of O(1).

The main one that has changed in this case is the time complexity of NearestCityOptimiser from O(n^2 \* log n) to O(n^2).

# Part 4: Complexity Analysis and Algorithm Comparison

## Analysis Task 14 a.

DeliveryRouteOptimiser:

Core Operation: Generating all permutations of cities to evaluate every possible route.

Time Complexity:

* Generating permutations: O(n!)
* Calculating distances for each permutation: O(n) per permutation.
* Total = O(n! \* n)

Scaling with Number of Cities: Increasing from 10 to 20 cities is unfeasible due to n!.

NearestCityOptimiser:

Core Operation: Finding the nearest unvisited city for each city.

Time Complexity:

* Finding the nearest city: O(n) per city.
* For n cities: O(n^2)

Scaling with Number of Cities: Growth is polynomial, increasing cities from 10 to 50 is manageable compared to brute-force.

HeuristicOptimiser:

Core Operation:

* Nearest-City Heuristic: O(n^2)
* 2-Opt Refinement: Comparing and potentially reversing all pairs of edges, O(n^2) iterations of O(n) comparisons.

Time Complexity: O(n^2)

Scaling with Number of Cities: Growth is polynomial, but higher than NearestCityOptimiser because of 2-opt refinement.

BacktrackingOptimiser:

Core Operation: Exploring all possible routes recursively, pruning invalid routes.

Time Complexity:

* Recursion tree depth: n (number of cities).
* At each level, n−1, n−2..., 1 of the branches are explored.
* Total: O(n!)

Scaling with Number of Cities: Growth is exponential, but pruning can be faster than brute-force.

## Discussion Task 14 b.

DeliveryRouteOptimiser and BacktrackingOptimiser (O(n!)): They are both O(n!) as they are exploring all possible permutations, which leads to exponential growth in the computational effort. In both they remain O(n!) as every permutation must be evaluated to the worst case.

NearestCityOptimiser and HeuristicOptimiser (O(n^2)): Nearest-city iteratively selects the closest unvisited city for each city, which requires scanning through all unvisited cities at each step. This process is repeated for the cities it must traverse through, which leads to O(n^2) complexity. Meanwhile, 2-opt will refine paths based on adjacency or local improvements operating over city pairs and resulting in O(n^2) complexity.

The distance matrix in all algorithms but NearestCityOptimiser reduces distance lookups to a time complexity of O(1), meanwhile in NearestCityOptimiser it uses a hash map, which also has an O(1) time complexity while using memory more efficiently as it does not have to allocate space for every possible pair of cities. Meaning that if raw speed is what is wanted then a distance matrix is likely the best option if the number of cities does not exceed something such as 100 cities, past that point hash maps are likely the better choice.

## Analysis Task 15 a.

DeliveryRouteOptimiser:

Memory Usage:

* Distance matrix: O(n^2)
* Storing all permutations: O(n \* n!)

Space Complexity: O(n^2 + n \* n!)

NearestCityOptimiser:

Memory Usage:

* Distance Map: O(n^2)
* Visited array and route: O(n)

Space Complexity: O(n^2)

HeuristicOptimiser:

Memory Usage:

* Distance matrix: O(n^2)
* Visited array and route: O(n)
* Temporary storage during 2-opt refinement: O(n)

Space Complexity: O(n^2)

BacktrackingOptimiser:

Memory Usage:

* Distance matrix: O(n^2)
* Current route and visited array: O(n)
* Recursive call stack: O(n)

Space Complexity: O(n^2)

## Discussion Task 15 b.

DeliveryRouteOptimiser is the algorithm which requires the largest amount of memory as it has to caculate and store the pairwise distance in a matrix. It would also have to generate and store all the possible permutations of the cities, making it O(n!) and very memory intensive as the number of cities n grows.

Meanwhile, NearestCityOptimiser avoids this by using memory for storing distances and routes. These scale in a more moderate manner, O(n^2) for distance storage and O(n) for the routes.

The hash map used in NearestCityOptimiser helped further reduce the constraint on memory as it dynamically allocates space and stores only the distances that have been calculated, making it more memory efficient when not all of the pairwise distances are needed.

## Analysis Task 16 a.

Following times are from an average of executing 10 times:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | 10 Cities ms | 30 Cities ms | 50 Cities ms | 10 Cities  Route Distance | 30 Cities Route Distance | 50 Cities Route Distance |
| DeliveryRouteOptimiser | 13.1216 | N/A | N/A | 30.1976 | N/A | N/A |
| NearestCityOptimiser | 0.09545 | 0.24712 | 0.34095 | 36.1814 | 126.5922 | 208.1905 |
| HeuristicOptimiser | 0.04167 | 0.13561 | 0.25823 | 36.1814 | 126.5922 | 208.1905 |
| BacktrackingOptimiser | 11.8649 | N/A | N/A | 30.3231 | N/A | N/A |

## Discussion Task 16 b.

The heuristic algorithms prioritize speed and memory efficiency, making them better for a larger number of cities. But in doing so sacrifice the optimal solution. An example is how both NearestCityOptimiser and HeuristicOptimiser consistently give back results quickly, but the results are not guaranteed to be the shortest.

While brute force algorithms such as BacktrackingOptimiser guarantee an optimal solution, it does lack the scalability to be able to calculate even past the 10 cities mark. This is as the computational time and memory usage increases at an exponential rate with the number of cities, making these algorithms only viable on small samples.

Overall, the time complexity of the algorithms and the results that were outputted match with the theoretical expectations.

## Analysis Task 17 a.

DeliveryRouteOptimiser:

* Finds the optimal route for 10 cities (30.1976) but fails for larger number of cities.

NearestCityOptimiser:

* 10 cities: 36.1814, 30 cities: 126.5922, 50 cities: 208.1905.
* Fast but possibly suboptimal, with increasing route distances as problem size grows.

HeuristicOptimiser:

* Matches NearestCityOptimiser, 10 cities: 36.1814, 30 cities: 126.5922, 50 cities: 208.1905.
* Similar performance, offering quick results but with possibly suboptimal routes.

BacktrackingOptimiser:

* 10 cities: 30.3231, close to the optimal solution, but fails for larger number of cities.

## Discussion Task 17 b.

The trade-off between the solution quality and runtime is clear, faster algorithms like NearestCityOptimiser and HeuristicOptimiser sacrifice solution quality to achieve scalability and quick results. While they could be providing suboptimal routes, they were completed in even less than a millisecond, even for the largest number of cities of 50. Which in contrast with the brute force algorithms couldn't even go beyond 10 cities, making them computationally unfeasible at scale.

## Analysis Task 18 a.

"N/A" : Algorithm couldn't run for these cases due to time limits or memory issues.

Run time in ms:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | Time Complexity | Space Complexity | Runtime (10 cities) | Runtime (50 cities) | Total Distance (10 cities) | Total Distance (50 cities) |
| DeliveryRouteOptimiser | O(n!) | O(n^2 + n \* n!) | 13.1216 | N/A | 30.1976 | N/A |
| NearestCityOptimiser | O(n^2) | O(n^2) | 0.09545 | 0.34095 | 36.1814 | 208.1905 |
| HeuristicOptimiser | O(n^2) | O(n^2) | 0.04167 | 0.25823 | 36.1814 | 208.1905 |
| BacktrackingOptimiser | O(n!) | O(n^2) | 11.8649 | N/A | 30.3231 | N/A |

## Discussion Task 18 b.

NearestCityOptimiser and HeuristicOptimiser strike the best balance between performance and solution quality, being fast and practical for a larger number of cities but are suboptimal in terms of solution quality.

DeliveryRouteOptimiser is optimal for smaller instances but becomes impractical as the number of cities grows due to its exponential time complexity.

BacktrackingOptimiser offers near-optimal results for small problems but struggles to scale with larger amounts of cities.

Overall, if I were to go beyond to 100+ cities then NearestCityOptimiser is likely the better choice for speed, but if a higher-quality solution is needed and the time and computer are not part of the equation, then I would probably consider HeuristicOptimiser.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 |