CSCU9A3 – The Assignment 2024

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

# Introduction

This assignment focuses on applying object-oriented programming principles and optimisation algorithms to solve the **Traveling Salesperson Problem (TSP)**. You will extend a starter codebase in Java, implementing different algorithms and exploring the use of appropriate data structures to solve the problem efficiently. Throughout the assignment, you will develop modularised software, emphasising clean object-oriented design and effective use of abstraction through interfaces and inheritance.

The objectives of the assignment align with **all Intended Learning Outcomes** (**ILOs**). You will demonstrate your ability to build larger software components using objects and modularisation (**ILO1**), apply fundamental data structures and algorithms to real-world computational problems (**ILO2**), and understand how these structures and algorithms work, including their trade-offs (**ILO3**). Additionally, you will explore recursion and divide-and-conquer techniques (**ILO4**), contrast the complexity and efficiency of algorithms (**ILO5**), and demonstrate computational thinking by solving unseen variations of the TSP problem independently (**ILO6**).

This assignment is divided into **four parts**, with a total of **18 tasks**. Each task consists of either a programming task or an analysis task, along with a corresponding reflection task. For the programming and analysis tasks, you will be required to make additions or alterations to the baseline code provided, implementing and testing different algorithms and data structures. The reflection tasks are designed to encourage you to think critically about your design choices, complexity analysis, and problem-solving approach. Your reflections should be documented in the **provided report template**, which you will submit alongside your code.

Additionally, there is a word limit of **2,000 words** for the entire report. While this limit will not be strictly enforced, failure to be clear and concise may affect your grade. Overly generic text that does not directly address your specific programming or analysis solution will not benefit you and should be avoided. Focus on providing thoughtful, precise reflections on your work. Use examples from your code and analysis to make your points.

# Part 1: Interface Design and Object-Oriented Structure

## Introduction

In this part of the assignment, you will focus on designing a flexible and modular codebase using object-oriented principles. You will implement an interface that defines the structure for different TSP optimisation algorithms and refactor existing algorithms to conform to this structure. This will allow the system to easily switch between different algorithm implementations. Additionally, you will create a class to encapsulate the TSP problem, separating the data from the logic used to solve it. This part emphasises clean software design, abstraction, and reuse, helping you to modularise larger pieces of software effectively.

## Tasks and Instructions

1. **Create the OptimisationAlgorithm Interface**
   1. **Programming Task**: Create an interface called OptimisationAlgorithm that declares a single method: DeliveryRoute findBestRoute(CityList cities). This method will serve as a contract for all optimisation algorithms. Your method should take a CityList object and return a DeliveryRoute object representing the optimal route found by the algorithm.
   2. **Reflection Task**: Write a few sentences explaining how designing an interface like OptimisationAlgorithm contributes to flexibility and modularity in your code. What benefits do you anticipate this design will bring as you continue to develop more algorithms?
2. **Refactor Existing Classes to Implement the Interface**
   1. **Programming Task**: Refactor the DeliveryRouteOptimiser (brute-force) and NearestCityOptimiser (greedy nearest-city) classes to implement the OptimisationAlgorithm interface. Ensure that the findBestRoute method in both classes uses the existing logic, and test your refactoring by running the main program to verify correctness.
   2. **Reflection Task**: In a few sentences, describe any challenges you encountered when refactoring the existing code to implement the interface. Did the new structure make it easier or harder to integrate the algorithms? Why?
3. **Design the TSPProblem Class**
   1. **Programming Task**: Design and implement a TSPProblem class that loads cities from a file and calculates the total distance of a delivery route. Your class should include a constructor to load cities into a CityList, a method to calculate route distances, and a private method to handle file loading. Make sure to test your calculateTotalDistance method with a known small example.
   2. **Reflection Task**: In a few sentences, explain how you structured the TSPProblem class and why. How does separating the problem data from the algorithm logic improve the overall organisation of the code?
4. **Write Unit Tests with JUnit**
   1. Programming Task: Write unit tests using JUnit to verify the correctness of your code. Test the TSPProblem class to ensure cities load correctly and distances are accurate. Write additional tests for both the DeliveryRouteOptimiser and NearestCityOptimiser to ensure that the findBestRoute method works as expected for small input problems. Test edge cases (e.g., two cities or no cities).
   2. Reflection Task: Write a few sentences on your approach to testing. What kinds of bugs or edge cases did your tests help identify? Did unit testing change how you approached your code design?

# Part 2: Algorithm Development and Optimisation

Here, you will expand the range of optimisation algorithms available in the system by implementing two additional algorithms, with at least one involving recursion or divide-and-conquer strategies. You will build on your understanding of algorithm design, exploring how different methods can tackle the same problem with varying efficiency and accuracy. In addition, you will integrate unit testing to validate the correctness of your algorithms, ensuring that they work correctly for small TSP instances before tackling larger ones.

## Tasks and Instructions:

1. **Implement a New Algorithm (Algorithm 1)**
   1. **Programming Task**: Implement a new optimisation algorithm to solve the TSP problem. This algorithm can be a heuristic or metaheuristic approach, such as Simulated Annealing, Genetic Algorithm, or another well-known approach. Your implementation must adhere to the OptimisationAlgorithm interface, ensuring that it can be swapped out seamlessly with the previous algorithms.
      1. Requirements:
         1. The new algorithm should provide a more efficient or approximate solution for larger problem sizes (e.g., 30 or 50 cities).
         2. Make sure to test your algorithm with small problems to verify that it returns a valid route (visits all cities exactly once) and that the total distance is reasonably optimized.
   2. **Reflection Task**: Write a few sentences reflecting on the design of this algorithm. Why did you choose this particular approach? What advantages and disadvantages does it have compared to the brute-force or greedy nearest-city algorithms?
2. **Implement a Backtracking Algorithm Using a Stack**
   1. **Programming Task**: Implement a backtracking algorithm for the TSP using a stack to manage the cities visited. The stack will help keep track of the current route as the algorithm explores different paths. When the algorithm finds that a certain route is invalid or suboptimal, it will backtrack by popping cities from the stack and trying alternative routes.
      1. Requirements:
         1. Implement the stack from scratch, using an array or linked list.
         2. Use the stack to simulate the recursive backtracking process: pushing cities onto the stack as they are visited, and popping them off when backtracking occurs.
         3. Ensure the backtracking algorithm adheres to the OptimisationAlgorithm interface, allowing it to be swapped with other algorithms.
         4. Test the algorithm with the provided .txt files for small to medium-sized TSP problems (10, 20, 30 cities).
   2. **Reflection Task**: In a few sentences, reflect on the role of the stack in your backtracking algorithm. How did the stack make it easier to manage partial routes and backtrack when necessary? What challenges did you encounter in ensuring the stack was used correctly to track visited cities?
3. **Write Unit Tests with JUnit**
   1. **Programming Task**: Develop unit tests to verify the correctness of the two new algorithms using JUnit. Your tests should ensure that:
      1. Both algorithms return a valid route (i.e., they visit all cities exactly once).
      2. The results for small problem sets (e.g., the 10-city problem) are compared with known optimal solutions or expected outcomes.
      3. Edge cases (e.g., problems with very few cities) are handled correctly.
   2. **Reflection Task**: In a few sentences, describe your testing strategy for the new algorithms. What types of test cases did you prioritise, and how did you ensure that your recursive algorithm is functioning as expected?
4. **Performance Comparison**
   1. **Programming Task**: After implementing both new algorithms, compare their performance against the brute-force and greedy nearest-city algorithms. Run each algorithm on problem sets of increasing size (e.g., 10, 20, 30, 50 cities) and measure the time taken to compute a solution.
   2. **Reflection Task**: In a few sentences, summarise your findings. How do the two new algorithms perform relative to the brute-force and greedy algorithms? Were there any surprises in terms of execution time or solution quality? Reflect on how algorithmic complexity influenced the performance of each approach.

# Part 3: Data Structures and Efficiency

In this part, you will focus on using different data structures to affect the efficiency of your algorithms. You will explore different ways to represent the list of cities and refactor your code to use a more efficient data structure. By analysing and testing the performance of each structure, you will learn the trade-offs between different implementations. Unit tests will also play a key role in ensuring that your data structures function correctly as the basis for your algorithms. This part is crucial for understanding how the right data structure can impact algorithm performance.

## Tasks and Instructions

1. **Modify CityList to Use a More Efficient Data Structure**
   1. **Programming Task**: Refactor the CityList class to improve its efficiency. Currently, CityList is a custom implementation of a singly linked list. You will modify this class to use a more efficient data structure, such as an array-backed list (e.g., similar to ArrayList) or a doubly linked list. The goal is to reduce the time complexity for common operations (e.g., access, insertion, deletion).
      1. Requirements:
         1. Maintain the original functionality of CityList while improving its internal structure.
         2. Ensure the class still provides methods for adding cities, retrieving cities, and iterating through the list.
   2. **Reflection Task**: In a few sentences, explain why you chose the specific data structure (e.g., array-backed or doubly linked list) to replace the original singly linked list. How does this new data structure improve the efficiency of the TSP algorithms? What are the trade-offs, if any?
2. **Analyse and Compare the Performance of the Refactored Data Structure**
   1. **Programming Task**: Compare the performance of the refactored CityList with the original singly linked list. Measure and record the time it takes to perform key operations (e.g., adding a city, retrieving a city) on both the original and modified versions for different list sizes (e.g., 10, 20, 50 cities).
      1. Requirements:
         1. Write a method that measures the execution time of key operations in CityList.
         2. Compare how the time scales as the number of cities increases.
   2. **Reflection Task**: In a few sentences, summarise your findings. How does the performance of your modified CityList compare with the original version? Did the new data structure perform as expected when scaling to larger problem sizes?
3. **Refactor NearestCityOptimiser to Use a More Efficient Evaluation Structure**
   1. **Programming Task**: Refactor the NearestCityOptimiser class to use a more efficient data structure for evaluating and selecting the nearest city. Consider using a structure such as a hash map (or HashMap) for quick access to city distances or an adjacency matrix to store and retrieve the distances between cities. The goal is to improve the efficiency of selecting the nearest city, especially as the number of cities grows.
      1. Requirements:
         1. Refactor the algorithm to use your chosen structure, ensuring that the nearest city can still be selected in a time-efficient manner.
         2. Consider how the new structure can reduce the time complexity of finding the nearest city compared to iterating through all remaining cities for each step.
         3. The new implementation should still adhere to the OptimisationAlgorithm interface.
         4. Test the refactored NearestCityOptimiser on different problem sizes (e.g., 10, 20, 50 cities) using the provided .txt files to ensure correctness and performance.
   2. **Reflection Task**: In a few sentences, explain why you chose this specific evaluation structure and how it improved the performance of the NearestCityOptimiser. Reflect on the trade-offs between your chosen structure and other possible alternatives (e.g., array-based or iterative solutions). Did the new structure make a significant difference in the performance of the algorithm?
4. **Write Unit Tests with JUnit**
   1. **Programming Task**: Write unit tests to validate the correctness of the modified CityList and the refactored NearestCityOptimiser using JUnit. Your tests should:
      1. Verify that the new CityList implementation correctly adds, retrieves, and removes cities.
      2. Test the NearestCityOptimiser to ensure that it still returns valid routes and that the performance improvements hold up under larger city sets.
      3. Include tests for edge cases such as empty city lists or lists with only one or two cities.
   2. **Reflection Task**: In a few sentences, explain how your unit tests helped you identify potential issues in the refactored data structures and algorithms. Were there any unexpected bugs that surfaced during testing, and how did you address them?
5. **Analyse the Impact on Overall Algorithm Efficiency**
   1. **Programming Task**: After modifying the CityList and refactoring the NearestCityOptimiser, analyse how these changes have impacted the overall efficiency of the algorithms. Run the algorithms on problem sets of increasing size (10, 20, 30, 50 cities) and compare the time taken before and after the refactor.
   2. **Reflection Task**: In a few sentences, discuss your observations regarding the impact of the refactored data structures on the overall algorithm performance. Did the improvements make a significant difference, particularly as problem size increased? How did the complexity of the algorithms change as a result of the modifications?

# Part 4: Complexity Analysis and Algorithm Comparison

This part is about evaluating the performance of your algorithms from both a theoretical and practical standpoint. You will perform complexity analysis to understand the time and space efficiency of each algorithm, using Big-O notation. Then, you will compare the real-world performance of your algorithms by running them on different-sized TSP problems, recording and interpreting the results. This analysis will help your understanding of algorithm efficiency and its impact on computational problems, preparing you to reason about algorithmic complexity in future work.

## Tasks and Instructions

1. **Perform Theoretical Time Complexity Analysis**
   1. **Analysis Task**: For each of the algorithms you’ve implemented so far (e.g., brute-force, nearest-city, genetic algorithm, recursive approach), estimate the theoretical time complexity using Big-O notation. Consider the following aspects:
      1. The time complexity of core operations, such as finding the nearest city or generating all permutations (for brute-force).
      2. How the complexity scales with the number of cities (e.g., how does the algorithm behave as the number of cities increases from 10 to 50?).
   2. **Discussion Task**: In a few sentences, provide a short summary of your theoretical complexity findings. Which algorithm has the highest time complexity? Which algorithm has the lowest, and why? Reflect on how the choice of data structures affected these results. Justify your estimates.
2. **Perform Theoretical Space Complexity Analysis**
   1. **Analysis Task**: Estimate the space complexity of each algorithm. Consider how much memory is required for storing cities, distances, intermediate results (e.g., population in the genetic algorithm), or recursive calls. Write down the Big-O space complexity for each algorithm.
   2. **Discussion Task**: In a few sentences, describe how space complexity differs between the algorithms. Which algorithm requires the most memory, and why? Did any specific data structures significantly influence space usage? Justify your estimates.
3. **Empirical Performance Testing**
   1. **Analysis Task**: Conduct empirical performance tests on each algorithm using the provided .txt files with different problem sizes (e.g., 10, 20, 30, 50 cities). Measure the time taken for each algorithm to find a solution and record the results.
      1. Use System.currentTimeMillis() or System.nanoTime() to time the execution of each algorithm.
      2. Compare the runtimes across the different problem sizes to observe how each algorithm scales.
      3. Capture the solution quality (e.g., total distance of the route) for each algorithm, as well as the runtime.
   2. **Discussion Task**: In a few sentences, compare the empirical runtime results with your theoretical analysis. Were there any discrepancies between the expected time complexity and the real-world performance? Reflect on how factors like problem size and implementation choices influenced the results.
4. **Algorithm Comparison Based on Solution Quality**
   1. **Analysis Task**: Compare the solution quality (i.e., the total distance of the route) produced by each algorithm for the same problem sizes. Record the total distance of the route found by each algorithm for each problem size (e.g., 10, 20, 50 cities) and observe how the quality changes.
      1. Consider which algorithm consistently finds the shortest route.
      2. For heuristic algorithms like the genetic algorithm, test multiple runs to account for variability in the solution.
   2. **Discussion Task**: In a few sentences, reflect on the trade-off between solution quality and runtime. Did the faster algorithms (like nearest-city) find worse solutions than slower ones (like brute-force)? Were approximate algorithms able to achieve near-optimal results within reasonable time limits?
5. **Summarise and Reflect on Complexity and Performance**
   1. **Analysis Task**: Prepare a table like the example given in Table 1, or chart that summarises the following for each algorithm across all problem sizes:
      1. Theoretical time complexity (Big-O).
      2. Theoretical space complexity (Big-O).
      3. Empirical runtimes (in milliseconds or seconds).
      4. Solution quality (total distance of the route).
   2. **Discussion Task**: In a few sentences, summarise your findings. Which algorithm strikes the best balance between performance and solution quality? How does problem size affect the scalability of each algorithm? Reflect on what you learned from both the theoretical analysis and the empirical testing.

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

# Submission

To submit your solution, you need to upload the following to as two separate files Canvas:

1. The report as a DOCX file
   1. Do not include your name anywhere in this file, only your student number
   2. Make sure you have clearly indicated which answer applies to which question and/or task in the assignment.
2. Your code as a ZIP file
   1. Only include the .java files
   2. Do not include any data files, .class files, or the report

### Use of generative AI

|  |  |  |
| --- | --- | --- |
| 2 | AI-Assisted idea generation and structuring | AI can be used in the assessment for brainstorming, creating structures, and generating ideas to improving work.  No AI content is allowed in the final submission |

### Late submission

If you cannot meet the assignment deadline and have good cause, please use the extension request link on the left-hand navigation to explain your situation and ask for an extension. Coursework will be accepted up to seven days after the hand-in deadline (or expiry of any agreed extension), but the mark will be lowered by three marks per day or part thereof. After seven days, the work will be deemed a non-submission.

### Plagiarism

Work that is submitted for assessment must be your own work. Plagiarism means presenting the work of others as though it were your own. The University takes a very serious view of plagiarism, and the penalties can be severe (ranging from a reduced mark in the assessment, through a failure mark for the module, to expulsion from the University for more serious or repeated offences). We check submissions carefully for evidence of plagiarism and pursue those cases we find. Further details can be found here:

<https://www.stir.ac.uk/media/stirling/services/academic-registry/documents/Policy-and-Procedure-Academic-Integrity_v4_FINAL.docx>