ACIT4420 Problem-solving with scripting

Candidate: 503

Contents

1. Introduction

This report covers the development and implementation details of a two-part Python project. The project consists of two main components, a courier routing system for delivery optimization, and a cellular automaton simulator, based on Conway's Game of Life.

The first component, the `courier_optimizer` package, addresses the challenge of planning efficient delivery routes. It takes into account multiple factors, including delivery urgency, package weight, and the transport mode (car, bicycle, or walking), and allows routes to be optimized based on different objectives such as minimum travel time, lowest cost, or reduced CO_2 emissions. The package demonstrates a practical application of Python scripting for solving combinatorial optimization problems, incorporating validation, logging, and user-defined parameters.

The second component, the `conway` package, is a flexible system for simulating Conway's Game of Life and other Life-like cellular automata, such as 'HighLife'. This package highlights modularity and flexibility, employing metaprogramming techniques to dynamically incorporate other rulesets without modifying the core simulation system.

This report documents the structure, functionality, and design decisions of both packages. It provides a detailed overview of the systems and implementation process, highlighting key architectural choices, the rationale behind design decisions, and details how Python's features were leveraged to create efficient, maintainable, and interesting solutions.

2. Implementation

This section covers the implementation details of both tasks, going in-depth into the systems and modules that build up the packages.

2.1. Courier optimizer

Modern courier services must plan delivery routes that account for several constraints simultaneously, such as delivery urgency, package weight, and the specific cost or emission profile of the chosen transport vehicle.

Mathematically, this is a variation of the Traveling Salesman Problem (TSP). Given a list of cities (or delivery stops) and the distances between each pair of cities, the goal is to find the shortest possible route that visits each city exactly once and returns to the origin. The TSP is classified as an NP-hard problem in combinatorial optimization. This means that as the number of delivery stops (n) increases, the computational time required to find the perfect solution grows factorially ($O(n!)$). For a small dataset, a brute-force approach is acceptable, but for a dataset with dozens or hundreds of stops, an exact solution becomes computationally infeasible to calculate in a reasonable timeframe.

Therefore, the implementation relies on heuristic approximation algorithms. While these algorithms do not guarantee the mathematically "perfect" shortest path, they provide a "good enough" solution within a highly efficient execution time ($O(n^2)$), which is suitable for the scope of this project.

2.1.1. Design overview

The system accepts user-defined input via command-line arguments. This design ensures that the routing system is decoupled from the user interface, allowing the core logic to be potentially reused in a web API or GUI application in the future.

2.1.2. Algorithm Implementation

To solve the routing problem, a Weighted Nearest Neighbor heuristic was implemented. The standard Nearest Neighbor algorithm operates by starting at a depot, finding the closest unvisited node, moving there, and repeating the process.

However, this project introduces a "Priority" constraint. Deliveries marked as "High Priority" must be serviced earlier in the route than "Low Priority" ones, even if the low-priority stop is geographically closer. To achieve this without complex backtracking, the algorithm modifies the "effective distance" calculation.

The distance between two points on the Earth's surface is calculated using the Haversine formula, which accounts for the spherical shape of the planet:

$$a = \sin^2\left(\frac{\Delta\varphi}{2}\right) + \cos(\varphi_1) \cdot \cos(\varphi_2) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)$$

$$c = 2 \cdot \operatorname{arctan2}\left(\sqrt{a}, \sqrt{1-a}\right)$$

$$d = R \cdot c$$

Where φ is latitude, λ is longitude, and R is the Earth's radius (approx. 6371 km).

The optimizer applies a weight multiplier to this distance based on urgency. For example, a "High" priority stop has its distance multiplied by 0.6. This makes the stop appear mathematically "closer" to the algorithm than it is in reality, increasing the likelihood that it will be selected as the next stop in the greedy search. Conversely, "Low" priority stops are penalized with a multiplier > 1.0.

2.1.3. Validation and Error Handling

Input data is rarely perfect. The system implements a strict validation layer using Python's regular expression module (`re`). Before optimization begins, every row in the source CSV is checked.

- **Priority validation:** A regex pattern `^(High|Medium|Low)$` ensures only valid urgency levels are processed.
- **Coordinate validation:** Latitudes are checked to be within $[-90, 90]$ and longitudes within $[-180, 180]$.

Invalid rows are not discarded silently. Instead, they are logged to a specific `rejected.csv` file, preserving data integrity and allowing the user to correct the source data.

2.2. Conway's Game of Life simulator

2.2.1. Design overview

Similarly to `courier_optimizer`, this package can be ran with command-line arguments, allowing for user-defined input. In this case, the user can supply the program a ruleset for use in cellular automata evolution (more on this later).

2.2.1.1. Grid module

The `grid` module provides an abstraction for managing the two-dimensional cellular automaton. Each cell in the grid can either be filled or empty, and the module exposes methods to set and query cells, as well as evolve the grid based on a provided rule set.

An interesting and deliberate implementation choice in this module is the use of a flat, one-dimensional list to represent the two-dimensional grid, rather than the more traditional approach of using nested lists (i.e., a list of rows, each containing a list of columns, or the other way around). This implementation detail offers some advantages like memory efficiency and performance, as only one list object is created, and accessing a contiguous block of memory can improve cache performance, though this has not been fully tested in this specific use-case.

A formula can be used to improve the ease-of-use of this solution, where the list can be indexed using simple arithmetic to find a specified `x,y` coordinate:

$$\text{index} = \text{row} \times \text{width} + \text{col}$$

This eliminates the need for nested iteration and makes operations like neighbor lookups straightforward to implement programatically.

2.2.1.2. Rule sets

The Eppstein paper (2009) provides an explanation of Life-like cellular automata rules using a `Bxxx/Syyy` notation, where `B` stands for 'Birth' and indicates the neighbor counts that cause a dead cell to be born, while `S` stands for 'Survives' and indicates the neighbor counts that allow a living cell to survive an evolution step. Each digit in the `x` and `y` positions ranges from 0 to 8, corresponding to the number of neighboring live cells considered in the Moore neighborhood. This concise notation enables the specification of a wide variety of Life-like rule sets, allowing exploration of many different cellular automata behaviors.

In the project, the `RuleSet` class is responsible for evaluating cells based on their neighbor count. The class leverages this Life-like automata notation style to dynamically generate the evolution logic for the grid. The parser reads a user-provided string such as `B3/S23` (The notation for Conway's Game of Life), extracts the birth and survival neighbor counts, and constructs a function definition to apply the rules to the grid at each evolution step dynamically by leveraging metaprogramming techniques. This approach decouples the rule definition from the simulation, allowing the user to specify the rule set upon running the system.

2.2.1.3. Renderer module

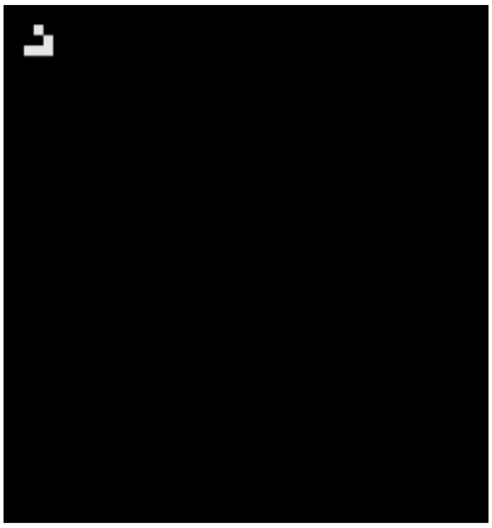

Figure 1: A glider in the terminal view using Conway's ruleset (`B3/S23`)

Select Graphic Rendition (SGR) control codes were used in the `renderer` module to color a special ASCII half-block character and the background behind it in the terminal in order to achieve uniformly sized cells on a grid, as rendering the cells using ASCII characters only would cause them to be taller. These escape codes were defined in ANSI X3.64, which gave them the ‘ANSI escape code’ name, but they were later adapted by the ECMA-48 (Ecma International, 1991) and ISO/IEC 6429 (International Organization for Standardization, 1992) standards that most terminals comply with today.

3. Testing

Manual testing and debugging was carried out throughout development to make sure the packages were robust and worked consistently with a variety of different user inputs and that any edge-cases could be ironed out. While the scope of the project did not require a comprehensive pipeline, the manual debugging and implemented unit tests were sufficient enough to validate the logic and stability of each solution.

A total of twenty unit tests were written across the two packages, though only limited to the `grid`, `ruleset`, and `delivery` modules.

```
============================= test session starts ====================================
platform linux -- Python 3.13.7, pytest-9.0.1, pluggy-1.6.0
rootdir: /home/user/projects/acit4420/exam
collected 20 items

tests/test_delivery.py ...........                                              [ 55%]
tests/test_grid.py .....                                                        [ 80%]
tests/test_ruleset.py ....                                                      [100%]

============================= 20 passed in 0.03s =====================================
```

3.1. Courier optimizer

The `courier_optimizer` package was tested using small datasets to verify route optimizations and the correctness of cost and emission calculations. Mock delivery sets were manually constructed to ensure that the scoring and prioritization logic behaved as expected.

The output of the `run.log`:

```
2025-11-26 08:56:25,177 - INFO - --- Optimization Start: 2025-11-26 08:56:25 ---
2025-11-26 08:56:25,177 - INFO - Calling function: main
2025-11-26 08:56:25,177 - INFO - Input File: input.csv
2025-11-26 08:56:25,177 - INFO - Depot Location: 59.91,10.738
2025-11-26 08:56:25,177 - INFO - Mode: Walk, Criterion: cost
2025-11-26 08:56:25,178 - INFO - Function main finished. Total Duration: 0.00s
2025-11-26 08:56:25,178 - INFO - --- Optimization End ---
```

3.2. Conway’s Game of Life simulator

The `grid` module was tested by comparing known Life patterns against their expected evolution steps. Oscillators such as blinkers, as well as moving patterns like gliders, were used to validate that neighbor counting, grid wrapping behavior, and state transitions worked as intended. Tests included toroidal and non-toroidal grid configurations (wrapping) to confirm that boundary conditions were consistently applied. Additionally, manually calculated neighbor counts were compared against the module’s output to confirm that the flat-list indexing scheme produced correct behaviors.

The `RuleSet` class underwent some targeted testing to ensure correct parsing of other Life-like rule definitions. Several rule strings including `B3/S23`, `B36/S23`, and intentionally malformed inputs, were evaluated. This verified that the metaprogrammed evaluation function returned predictable results for known neighbor configurations. Invalid rule strings correctly triggered exceptions, confirming that the DSL-like parsing logic could validate the strings consistently.

The `renderer` module was tested manually, as the output depends on terminal rendering behavior. Tests focused on making sure that the grid was rendered correctly, and that the control codes that control the caret behavior to refresh the grid render, rather than print a new one each evolution, did not cause any problems.

4. Conclusion

The development of the `courier_optimizer` and `conway` packages demonstrated the versatility of Python as a tool for both operational optimization and scientific simulation.

The courier optimizer successfully met the requirements of solving a multi-objective routing problem. By implementing a weighted heuristic on top of the Haversine distance formula, the system balances the competing needs of geographical efficiency and delivery urgency. The inclusion of comprehensive logging and data validation ensures the tool is robust enough for real-world data scenarios where input quality cannot be guaranteed.

The Conway's Game of Life simulator highlighted the power of Python's dynamic nature. The use of a flat-list data structure for the grid proved to be an effective optimization for memory management, while the metaprogramming approach for rule parsing allowed for a highly extensible system. The separation of the simulation logic from the rendering layer ensures that the code remains modular and maintainable.

Ultimately, this project reinforced the importance of algorithmic analysis. Understanding the trade-offs between exact solutions and heuristics provided practical experience in structuring larger Python applications using Object-Oriented principles.

Bibliography

Ecma International. (1991, June). *Control Functions for Coded Character Sets.*

Eppstein, D. (2009). Growth and Decay in Life-Like Cellular Automata. *Arxiv.org*. https://doi.org/10.48550/arxiv.0911.2890

International Organization for Standardization. (1992,). *ISO/IEC 6429:1992-Information Technology-Control Functions for Coded Character Sets.*

5. Appendix

5.1. Complete code

GitHub repository link