Developing and Optimizing Data Structures for Real-World Applications Using Python

Project Phase 2 Deliverable 2: Proof of Concept Implementation

Group 1: Haeri Kyoung, Oishani Ganguly, Akash Shrestha, Nasser Hasan Padilla, Kannekanti Nikhil Kanneganti, Venkata Tadigotla.

University of the Cumberlands

MSCS-532-M20 – Algorithms and Data Structures

Professor Brandon Bass

July 26, 2025

Pareto-Optimal Pathfinding: Designing Multi-Criteria Route Optimization for Real-World Navigation

GitHub Repository: https://github.com/Oishani/MSCS_532_Project

1. Partial Implementation Overview

In this second phase of the project, we implemented the foundational components of the data structures designed in Phase 1. The goal was to provide a modular, extensible proof of concept (PoC) that demonstrates the core operations central to multi-criteria pathfinding. The following subsections describe the portions of each data structure that were implemented and explain how they integrate into the overall system design.

A. Graph (Adjacency List)

- **Implemented Features**: A directed adjacency-list representation that supports dynamic node and edge operations.
    - `add_node(node)`: Registers a new vertex if it does not already exist.

    - `add_edge(u, v, cost)`: Creates a directed edge from u to v with a three-dimensional cost tuple `(travel_time, toll_cost, scenic_quality)`.
    - `remove_node(node)` and `remove_edge(u, v)`: Provides full mutability by allowing deletion of vertices and edges, complete with error handling for non-existent references.
    - `neighbors(node)`: Retrieves outgoing edges and their associated cost vectors for traversal and expansion.
    - `bfs(start)` and `dfs(start)`: Perform breadth-first and depth-first traversals, respectively, primarily for debugging, coverage validation, and sanity checks on connectivity.

- **Integration**: The adjacency list is the backbone of the routing graph. By storing multi-dimensional weights directly within neighbor tuples, the graph interface exposes all necessary cost data for label expansion without additional lookups. This design ensures that each traversal or path search can access cost information in O(1) per edge.

B. Label and LabelSet

- **Implemented Features**: Label structures represent Pareto-cost states at each node, and LabelSet manages non-dominated labels.
    - `Label(cost, node, predecessor)`: Encapsulates a cost tuple, the associated graph node, and a pointer to its predecessor label for route reconstruction.
    - `Label.dominates(other)`: Implements Pareto dominance, returning true if and only if the invoking label is equal or better in all dimensions and strictly better in at least one.
    - `LabelSet.add(label)`: Inserts a new label into the set if it is non-dominated, automatically pruning any existing labels that it dominates.

- **Integration**: At each graph node, the LabelSet maintains a local Pareto frontier of cost vectors. During path expansion, only non-dominated labels propagate forward. This mechanism ensures that the algorithm can generate a diverse set of optimal trade-off paths without redundant exploration.

C. LabelPriorityQueue (Min-Heap)

- **Implemented Features**: A simple wrapper over Python's `heapq` to prioritize labels for expansion.
    - `push(label)` and `pop()`: Add and remove labels from the priority queue, enforcing lexicographic ordering on the cost tuple `(travel_time, toll_cost, scenic_quality)`.

    - Error handling for invalid pushes and empty-queue pops.

- **Integration**: The priority queue drives the label-expansion loop by always selecting the next most promising label according to lexicographic priority. Although lexicographic ordering is a simplification of true multi-criteria ranking, it provides a deterministic policy for exploration and ensures algorithmic correctness when labels are processed in non-decreasing primary cost order.

Together, these components establish the core machinery of a Pareto-optimal multi-criteria pathfinding system:

1. The Graph stores the network structure and cost information.
2. Label and LabelSet manage Pareto-front costs at each node.
3. LabelPriorityQueue ensures systematic exploration of the label space.

In future phases, these modules will integrate into a unified pathfinding algorithm that iteratively expands labels from the priority queue, updates label sets, and reconstructs complete paths upon reaching destination nodes. This modular separation of concerns simplifies maintenance, testing, and further extension (e.g., custom dominance relations or alternative priority policies).

2. Demonstration and Testing

To ensure the correctness and robustness of the core data structures, we implemented comprehensive test cases along with a demonstration script. These validation efforts underscore the functionality of key operations and highlight how each component behaves under both typical and edge conditions.

Unit Test Case Summaries

Graph Operations

- **Node and Edge Manipulation**: Tests verified that adding and removing nodes and edges correctly updates the internal adjacency representation. For example, removing a non-existent node raised an expected `KeyError`, demonstrating proper error handling.
- **Traversal Methods**: Breadth-first and depth-first searches were tested on small graphs, confirming that the visit order matched expected sequences. Both `bfs('A')` and `dfs('A')` returned `['A', 'B', 'C']` in our test scenario.
- **Neighbor Retrieval**: Accessing neighbors for valid nodes returned the correct list of `(neighbor, cost)` tuples, and querying an invalid node raised a `KeyError`.

Label and LabelSet Behavior

- **Label Initialization and Dominance**: Tests ensured that labels could only be created with 3-tuple costs, and that invalid inputs triggered a `ValueError`. Dominance checks

confirmed that a label with strictly better costs in at least one dimension correctly dominated another.
- **Pareto Frontier Maintenance**: The `LabelSet` insertion logic was validated by adding both dominated and non-dominated labels. In our test, adding `(2,2,2)` after `(1,1,1)` was rejected, while adding `(1.5,0.5,1.0)` was accepted. Subsequent pruning left exactly two non-dominated labels in the set.

Priority Queue Mechanics

- **Ordering and Errors**: The `LabelPriorityQueue` was tested to guarantee that `pop()` returned the lexicographically smallest label. In a sample push of labels `(5,1,1)`, `(3,2,2)`, and `(3,1,3)`, the queue delivered `(3,1,3)` first. Attempting to push a non-`Label` object or popping from an empty queue raised the appropriate exceptions.

Demonstration Script Outcomes

A standalone script executed core functionality in sequence and captured its output to `demo_results.txt`. Representative excerpts include:

Graph Demo:

- Adjacency List: {'A': [('B', (1, 1, 1)), ('C', (2, 0, 1))], 'B': [('D', (0, 3, 2))], ...}
- BFS from A: ['A', 'B', 'C', 'D']
- Expected error removing non-existent edge: ValueError('Edge A -> D not found')

LabelSet Demo:

- Adding Label(node=X, cost=(5, 5, 5)): Added; Current set = [Label(node=X, cost=(5, 5, 5))]
- Adding Label(node=X, cost=(3, 7, 5)): Added; Current set = [Label(node=X, cost=(5, 5, 5)), Label(node=X, cost=(3, 7, 5))]
- Adding Label(node=X, cost=(5, 4, 6)): Dominated; Current set unchanged

Priority Queue Demo:

- Queue content before pops: [Label(node=B, cost=(3, 2, 2)), Label(node=A, cost=(5, 1, 1)), ...]
- Popped: Label(node=B, cost=(3, 2, 2))
- Expected error popping empty queue: IndexError('Pop from empty priority queue')

Combined Demo:

- Expanding Label(node=S, cost=(0, 0, 0))
    - -> New non-dominated label at A: Label(node=A, cost=(1, 2, 3))

- Expanding Label(node=A, cost=(1, 2, 3))
    - -> New non-dominated label at T: Label(node=T, cost=(2, 3, 4))
- Final label sets:
    - S: [Label(node=S, cost=(0, 0, 0))]
    - A: [Label(node=A, cost=(1, 2, 3))]
    - B: [Label(node=B, cost=(2, 1, 3))]
    - T: [Label(node=T, cost=(2, 3, 4))]

These test cases and demonstration results confirm that the implemented data structures perform as expected, correctly managing dynamic graph updates, maintaining Pareto-front labels, and prioritizing label expansion in a predictable order. This foundation sets the stage for integrating these modules into a full pathfinding algorithm in future phases.

3. Documentation of Implementation Process

During Phase 2 development, translating the Phase 1 designs into working Python modules surfaced several implementation challenges. Below are documented some of the key issues encountered, the solutions applied, and critical code excerpts.

Challenges and Solutions

1. **Enforcing Three-Dimensional Cost Tuples**
    - **Issue:** Silent errors arose when cost vectors of incorrect length slipped through, causing downstream indexing failures.
    - **Solution:** Added upfront validation in constructors and mutators to reject any cost that isn't a 3-tuple.

```
    def add_edge(self, u, v, cost):

        if not (isinstance(cost, tuple) and len(cost) == 3):

            raise ValueError("Cost must be a tuple of three
floats")

        self.add_node(u)

        self.add_node(v)

        self._adj[u].append((v, cost))
```

2. **Correct Pareto Dominance and Pruning**
    - **Issue:** Ensuring that each new label is compared correctly against all existing labels, and that dominated labels are removed without multiple list passes.
    - **Solution:** Implemented a single-pass algorithm in `LabelSet.add` that both rejects dominated insertions and removes labels dominated by the new one.

```
class LabelSet:

    def add(self, label):

        new_labels = []

        for existing in self.labels:

            if existing.dominates(label):

                return False   # New label is dominated →
reject

            if not label.dominates(existing):

                new_labels.append(existing)   # Retain
non-dominated existing label

        new_labels.append(label)

        self.labels = new_labels

        return True
```

3. **Deterministic Multi-Criteria Ordering**
    - **Issue:** Multi-criteria label costs have no natural total order. A deterministic policy was needed for the priority queue.
    - **Solution:** Used Python's built-in tuple comparison to define lexicographic ordering on `(travel_time, toll_cost, scenic_quality)`.

```
class Label:

    def __lt__(self, other):

        # Compare cost tuples lexicographically

        return self.cost < other.cost
```

4. **Robust Error Handling for Traversals and Lookups**
    - **Issue:** Invalid node lookups or queue operations sometimes failed silently or with unclear errors.
    - **Solution:** Added explicit guards and clear exception messages.

```
def neighbors(self, node):

    if node not in self._adj:

        raise KeyError(f"Node {node} not found")
```

```
        return list(self._adj[node])

    def pop(self):

        if not self._heap:

            raise IndexError("Pop from empty priority queue")

        return heapq.heappop(self._heap)
```

Next Steps to Complete Full Implementation

1. **Unified Expansion Loop**
    - Develop a driver function that repeatedly pops the next label, expands it across all outgoing edges, and inserts new labels into their respective `LabelSet` and priority queue.
2. **Path Reconstruction**
    - Trace back from destination labels using the `predecessor` pointers to build complete routes for user presentation.
3. **Customizable Criteria and Ordering**
    - Abstract dominance logic and priority ordering behind interfaces to allow alternative multi-criteria policies (e.g., weighted sums or user-defined comparators).
4. **Performance and Scalability**
    - Profile on larger, realistic road networks. Consider optimized heaps, spatial indexing, or parallel expansions to handle millions of nodes and edges.
5. **User Interface and Integration**
    - Build a command-line or graphical interface to accept start/end inputs, display multiple Pareto-optimal paths, and integrate with external mapping data sources.

These insights will guide the next phases toward a complete, user-ready Pareto-optimal pathfinding application.

4. Code Snippets and Documentation

Below are the three most critical excerpts from our Phase 2 implementation, each illustrating a foundational mechanism in the multi-criteria pathfinding PoC. Full code available at https://github.com/Oishani/MSCS_532_Project.

A. Graph Edge Addition with Cost Validation

Link: https://github.com/Oishani/MSCS_532_Project/blob/5aa0d28972ec949f511d3ee38cc2ec8e439a3899/pathfinder/graph.py#L14

```
# graph.py

def add_edge(self, u: Any, v: Any, cost: Tuple[float, float, float])
-> None:
    """
    Add directed edge u → v with a 3-dimensional cost tuple.
    Raises ValueError if cost is not exactly three elements.
    """
    if not (isinstance(cost, tuple) and len(cost) == 3):
        raise ValueError("Cost must be a tuple of three floats")
    self.add_node(u)
    self.add_node(v)
    self._adj[u].append((v, cost))
```

Why it's critical: Validates that all edges carry exactly three cost components - travel time, toll cost, and scenic quality - preventing subtle downstream bugs and ensuring consistent data for label expansion.

B. Pareto-Front Maintenance in LabelSet

Link: https://github.com/Oishani/MSCS_532_Project/blob/5aa0d28972ec949f511d3ee38cc2ec8e439a3899/pathfinder/label_set.py#L3

```
# label_set.py

class LabelSet:
    def add(self, label: Label) -> bool:
        """
        Insert `label` into the Pareto frontier if it is
non-dominated.
        Prunes any existing labels that the newcomer dominates.
        Returns True if added, False if dominated.
        """
        new_list: List[Label] = []
        for existing in self.labels:
            if existing.dominates(label):
                return False           # New label is dominated
            if not label.dominates(existing):
                new_list.append(existing)
        new_list.append(label)
        self.labels = new_list
        return True
```

Why it's critical: Ensures the label set at each node remains the true Pareto frontier, rejecting dominated insertions and removing outdated labels in a single, linear-time pass.

C. Core Expansion Loop Integrating All Components

Link:
https://github.com/Oishani/MSCS_532_Project/blob/5aa0d28972ec949f511d3ee38cc2ec8e439a3899/demo.py#L89

```
# demo.py (excerpt)

while pq:
    current = pq.pop()
    for neighbor, edge_cost in graph.neighbors(current.node):
        total_cost = tuple(a + b for a, b in zip(current.cost,
edge_cost))
        new_label = Label(total_cost, neighbor, predecessor=current)
        if label_sets[neighbor].add(new_label):
            pq.push(new_label)
```

Why it's critical: Demonstrates the interaction between Graph, LabelSet, and LabelPriorityQueue: popping the next best label, computing cumulative costs, inserting into the neighbor's Pareto frontier, and re-queuing non-dominated labels for further exploration. This loop is the heart of the pathfinding algorithm.

References

Ehrgott, M. (2005). *Multicriteria Optimization*. Springer. https://ideas.repec.org/b/spr/sprbok/978-3-540-27659-3.html

Mandow, L., & De la Cruz, J. L. (2005). A new approach to multiobjective A* search. In *IJCAI*. https://www.ijcai.org/Proceedings/05/Papers/0867.pdf

The Decision Lab. (n.d.). *The Pareto Principle*. Retrieved July 25, 2025, from https://thedecisionlab.com/reference-guide/economics/the-pareto-principle

Zitzler, E., Laumanns, M., & Thiele, L. (2001). *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*. ETH Zurich.

https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/145755/eth-24689-01.pdf