Presentation Script: Pareto-Optimal Pathfinding for Real-World Navigation

Slide 1: Title Slide

- Pareto-Optimal Pathfinding: Designing Multi-Criteria Route Optimization for Real-World Navigation
- Group 1: Haeri Kyoung, Oishani Ganguly, Akash Shrestha, Nasser Hasan Padilla, Kannekanti Nikhil, Venkata Tadigotla
- University of the Cumberlands
- MSCS-532-M20 - Algorithms and Data Structures
- Professor Brandon Bass
- July 27th, 2025

Slide 2: The Problem with Current Navigation - It's Not Just About "Fastest"

- **The User Problem:** Think about your own experience with navigation apps like Google Maps, Apple Maps, or Waze. Do you always **want** to pick the "fastest" route?
- **Real-World Choices:** Sometimes you might want the cheapest route (avoiding tolls), or the most scenic option, or maybe even a route that balances a few factors.
- **The Limitation:** Traditional navigation algorithms, like Dijkstra's, focus on minimizing a single scalar value, such as time or distance, which simplifies routing and can overlook critical tradeoffs in real-world scenarios. This project addresses this limitation by implementing a Pareto-optimal multi-criteria pathfinding system.

Slide 3: Why Do We Need Multiple Criteria? Understanding User Trade-offs

- **Conflicting Objectives:** Imagine a route that's fastest but has tolls, versus a slightly slower route with no tolls. Which one is "better"? It depends on *your* priorities.
- **User Preferences are Dynamic:** Your preferences can change! Fastest for a weekday commute, scenic for a weekend drive.
- **The Optimization:** Our project addresses this by implementing a "Pareto-optimal multi-criteria pathfinding system." Instead of one "best" path, we find a *set* of "best possible compromise" paths.

Slide 4: Understanding Pareto-Optimal Pathfinding: Finding the "Best Compromises"

- **Beyond Single-Objective:** We're moving from "what's the best route?" to "what are all the *smartest ways to balance* travel time, toll cost, and scenic quality?"
- **What is "Pareto-Optimal"?**
    - The concept originated from Vilfredo Pareto, an Italian economist, who observed that in many situations, there isn't one single "best" solution, but rather a set of solutions where improving one aspect means sacrificing another.
    - A path is Pareto-optimal (or "non-dominated") if you can't make it better in *one* aspect without making it worse in *at least one other* aspect.
    - Here, it means finding a set of paths where you can't make one path better in terms of time, toll, or scenery without making it worse in at least one of the other aspects.
    - Think of it as offering you a diverse set of truly valuable options, where each option has a unique strength and no option is clearly inferior to another in all aspects.
- **Empowering Users:** Our solution allows users to make richer decisions based on their complex, dynamic preferences.

Slide 5: The Core Data Structures: Building Our Smart Navigator

- **1. The Graph (Your Road Network):**
    - We represent roads and intersections as a network, similar to how navigation apps see the world.
    - The improvement: each road (edge) has not just one cost (like time) but a *vector* of costs: `(travel_time, toll_cost, scenic_quality)`.
    - We use an "adjacency list" for efficiency, which is great for representing sparse networks like roads.
- **2. Labels and Label Sets:**
    - Instead of tracking just one best path to each location, we keep a *set* of "labels".
    - Each "label" represents a unique Pareto-optimal path found so far to that location, with its accumulated cost vector.
    - The "Label Set" ensures that we only keep the non-dominated paths, automatically removing any inferior options.
- **3. Label Priority Queue:**
    - We use a min-heap to efficiently decide which path segment to explore next
    - We prioritize based on weights given to the criteria (time > toll > scenic value).

Slide 9: How It Works: A Simplified Algorithm Flow

- **Inspired by Dijkstra's, But Evolved:** Our algorithm extends the familiar concept of exploring paths but with multi-criteria support.
- **Priority Queue:** We use a "Priority Queue" (a min-heap) to efficiently decide which path segment (label) to explore next. We prioritize paths that are "better" based on a predefined order (e.g., primarily by time, then by toll, etc.).
- **The Process:**
    1. Start at your origin.
    2. Explore neighboring roads, calculating the new multi-dimensional costs.
    3. For each new path to a destination, compare it to existing paths for that destination.
    4. If it's a "better compromise" (non-dominated), keep it and discard any paths it now "dominates".
    5. Repeat until all promising paths are explored.

Slide 10: Why Optimize? The Challenge of Scaling to Real-World Maps

- **From Concept to Reality:** Building a proof-of-concept is one thing, but making it work for an entire city or country, with millions of roads and intersections, is another.
- **Key Challenges (Bottlenecks):**
    - **Slow Road Removal:** Removing or updating roads was taking too long in dynamic networks.

- **Repetitive Checks:** We were repeatedly checking if one path dominated another, even for the same cost combinations.
- **"Frontier Explosion":** The number of "compromise" paths (labels) at each intersection could grow extremely large, consuming a lot of memory and slowing things down.

Slide 11: Smart Solutions for Speed and Memory Efficiency

- **1. Faster Road Network Updates:**
  - We changed how the "Graph" stores connections from a list to a dictionary.
  - This made adding or removing roads almost instant (`O(1)`), which is crucial for dynamic maps.
- **2. Remembering Dominance Checks (Memoization):**
  - For repetitive comparisons between path costs, we added a cache.
  - Now, if we've compared two cost vectors before, we just look up the answer instantly, avoiding recalculation.
- **3. Lightweight Path Objects (`__slots__`):**
  - In Python, objects can be memory-heavy. We made our "Label" objects more efficient by using `__slots__`.
  - Python's `__slots__` saves memory by replacing the default `__dict__` for storing object attributes with a fixed-size array, thereby reducing the memory footprint, especially for classes with many instances like the `Label` objects. This optimization directly allocates space for specified attributes, eliminating the overhead of a dictionary per object and allowing the application to handle larger datasets more efficiently.
  - This cut down memory usage significantly (tens of megabytes saved), allowing us to store millions of path options.

Slide 12: Handling Massive Data & Future Growth

- **1. Epsilon-Pruning: Taming the Frontier:**
  - To prevent the "frontier" (the set of non-dominated paths at a node) from exploding, we introduced "epsilon-pruning".
  - This means we'll merge paths that are "almost" identical in cost, trading a tiny bit of precision for huge gains in speed and memory. It keeps the number of options manageable.
- **2. Streaming Graph Data (Lazy Loading):**
  - Instead of loading an entire city's road network into memory at once (which could crash your computer!), we developed a way to stream data from disk only when needed.
  - This allows us to handle arbitrarily large maps, scaling beyond available RAM.
- **Future Work:** Customizable ranking for users, advanced spatial indexing for faster retrieval of geographic data, and parallel processing to explore paths even faster.

Slide 13: Impact and Future Vision

- **A Smarter Navigation Experience:** Our project moves navigation systems towards a more user-centric approach, offering truly intelligent choices that reflect diverse real-world needs, not just a single "best" path.
- **Beyond Roads:** The principles of multi-criteria pathfinding and the data structures developed can be applied to many other complex decision-making problems:
  - Logistics and supply chain optimization
  - Public transit planning
  - Even multi-objective decision systems in finance and healthcare.
- **Thank You!**