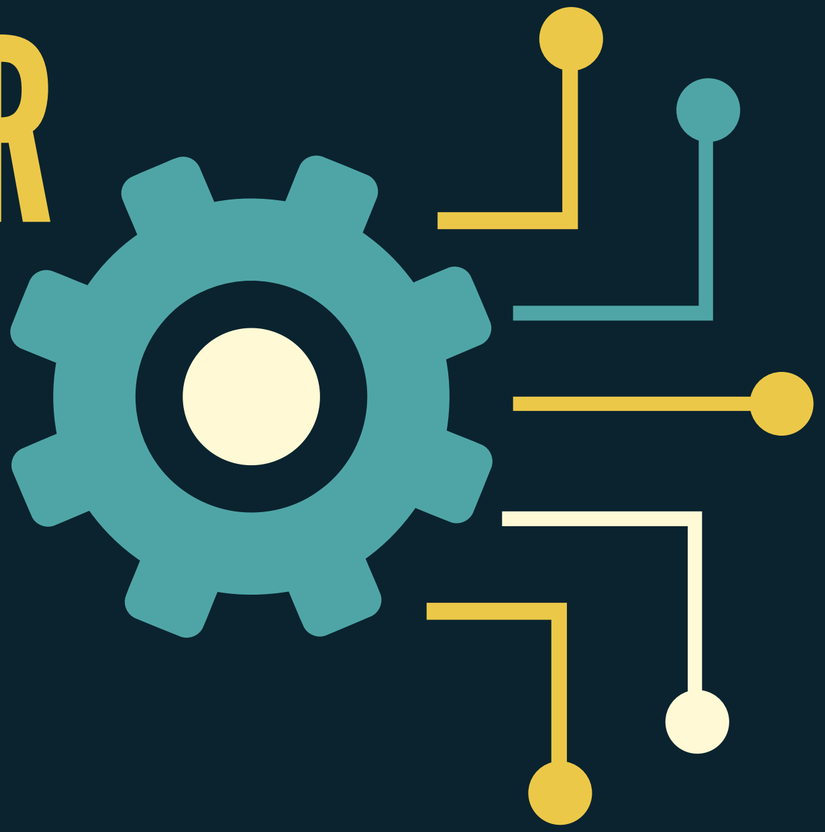
MAZE ROUTER



CSCE3304 – Spring 2025

Mohamed Sabry, Omar Bahgat, Ahmed El Dessouky

TECHNICAL BACKGROUND

TECHNICAL BACKGROUND

OBJECTIVES

- Implement a Scalable Maze-Routing Engine:
  - Build a routing core based on Lee's breadth-first search algorithm that guarantees shortest-path solutions under a user-defined cost model.
- Two-Layer Support with Directional Bias:
  - M1 (Horizontal Layer): Optimize for X-direction traces
  - M2 (Vertical Layer): Optimize for Y-direction traces
  - Seamlessly handle layer transitions via via penalties.
- Large-Scale Grid Handling:
  - Support grid dimensions up to 1000 × 1000 per layer (2 million total nodes)
  - Efficient memory management to track visited nodes and frontier queues
- Customizable Cost Metrics:
  - Via Penalty: Additional cost every time the path switches between M1 and M2
  - Non-Preferred Direction Penalty: Extra cost for moves against the layer's "ideal" orientation
  - Allow users to tune these parameters to influence path selection
- Modular, Extensible Architecture:
  - Parsing Module: Ingest netlists, pin locations, and obstacle definitions
  - Routing Module: Encapsulate BFS logic with a priority queue for cost-ordered expansion
  - Visualization Module: Python-based renderer for multi-layer path inspection

TECHNICAL BACKGROUND

OVERVIEW

- VLSI Routing Context:
  - Maps net pins to metal-layer interconnects on a discrete grid; critical for meeting timing, area, and manufacturability constraints.
- Grid & Obstacle Representation:
  - Models each metal layer as a 2D grid; blocked cells represent pre-routed wires, blockages, or forbidden regions.
- Lee's Algorithm:
  - A uniform-cost BFS that expands all reachable nodes in "rings" from the source until the sink is found; guarantees shortest-path under the chosen cost model but requires storing the full frontier.
- Complexity Considerations:
  - Worst-case time and memory of $O(W \times H \times L)$, where W and H are grid width/height and L is the number of layers (here $L = 2$). Heuristics and early-exit conditions help keep practical runtimes low.

TECHNICAL BACKGROUND

ARCHITECTURE

- Parsing Module (parse.cpp / parse.hpp):
  - Reads and tokenizes each line of the input file, skipping comments and blank lines
  - Extracts grid dimensions, layer count, obstacle coordinates, and per-net pin lists
  - Constructs C++ data structures—Grid, ObstacleMap, and Net—stored in std::vector and std::unordered_map for fast lookup
  - Validates input consistency and reports format errors with line numbers

- Routing Module (route.cpp):
  - Defines a Node { x, y, layer, cost, parent } and uses std::priority_queue to perform a cost-ordered BFS
  - Generates up to six neighbors per step (N, S, E, W, via-up, via-down), applying user-tunable via and directional penalties
  - Marks visited nodes in a 3D visited[W][H][L] array to avoid re-exploration
  - On reaching the target pin, backtracks via parent pointers to emit the shortest-cost path

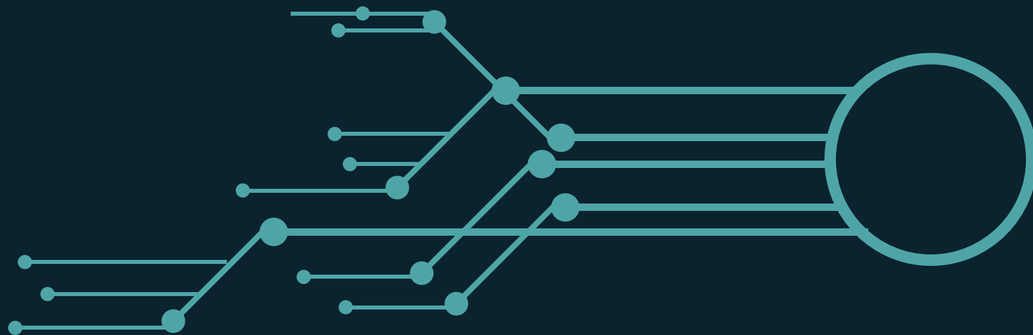
TECHNICAL BACKGROUND

ARCHITECTURE

- Heuristic Module:
  - Computes the Manhattan span of each net's pin pair as an initial cost heuristic
  - Sorts nets in descending order of estimated routing cost to tackle the hardest nets first
  - Dynamically adjusts ordering based on partial results to mitigate early congestion

- Visualization Module (visualize_2.py):
  - Parses the router's output file to load routed paths, via locations, and obstacle maps
  - Uses Matplotlib to draw each layer in a separate subplot, color-coding nets and vias, and overlaying obstacles
  - Annotates pin start/end points and provides a legend for clarity
  - Supports exporting to PNG or displaying interactive windows for detailed inspection

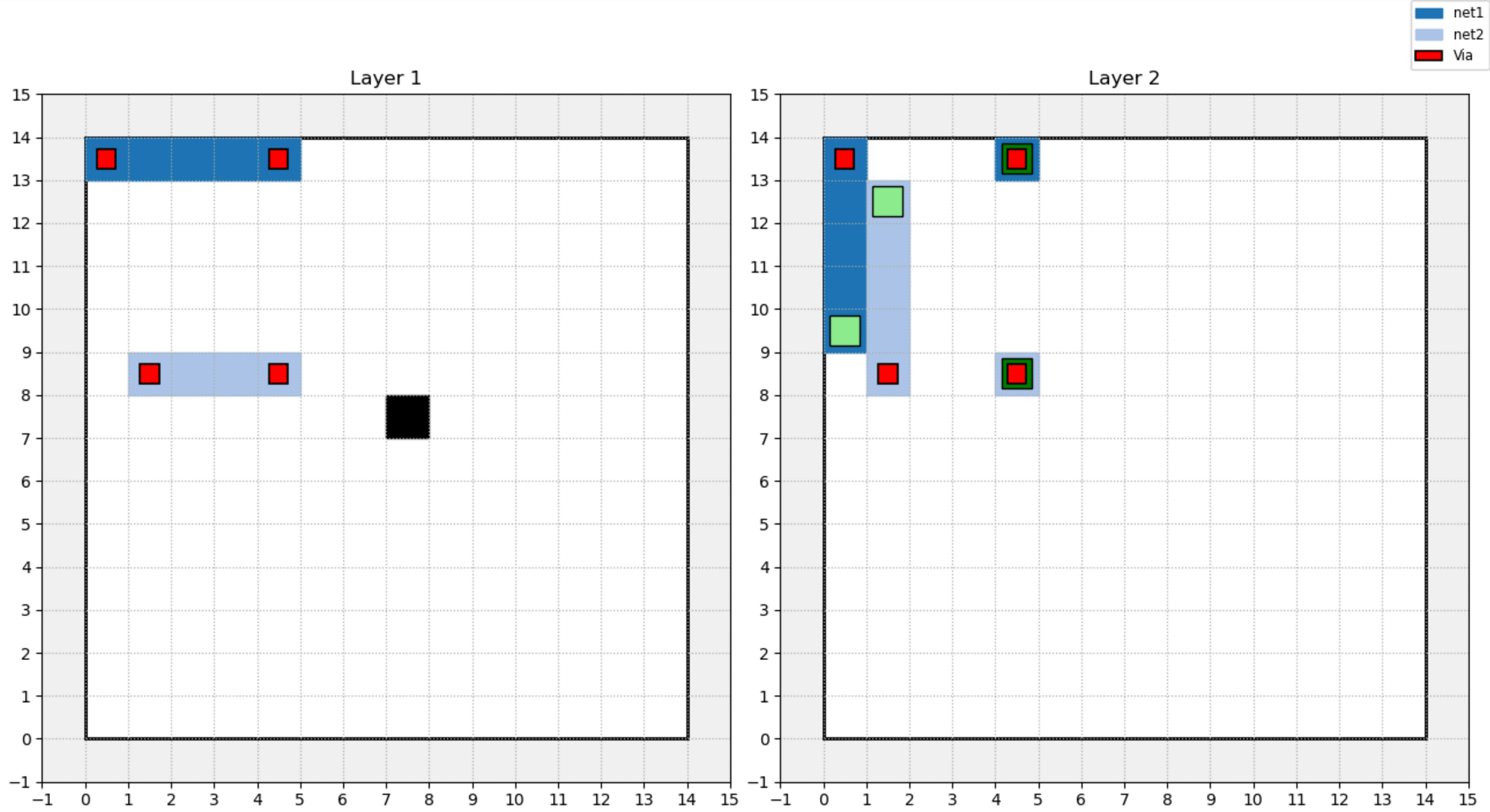
EXAMPLES



EXAMPLES

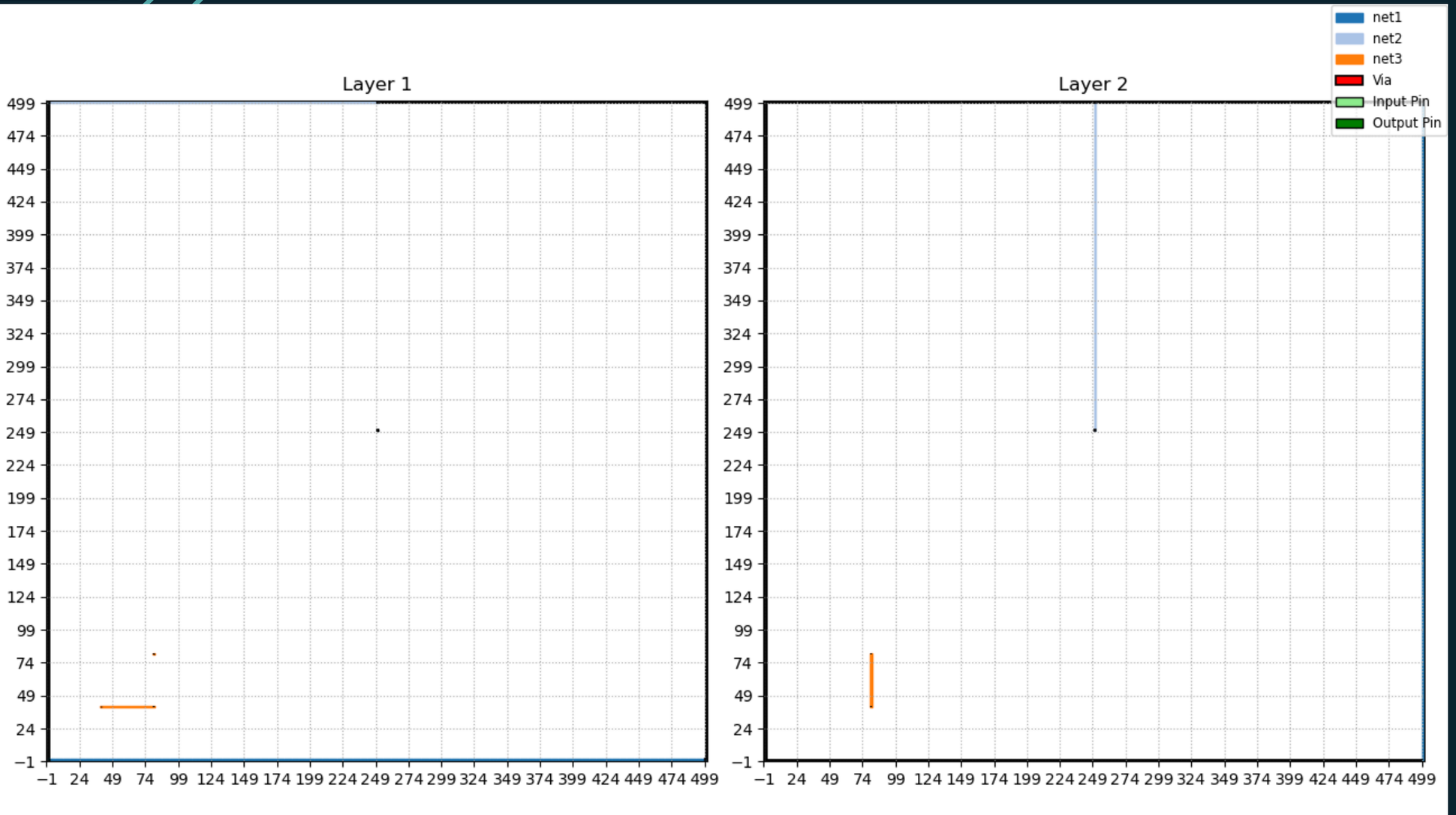
TESTCASE 1

14, 14, 10, 1000
OBS (1, 7, 7)
net1 (2, 0, 9) (2, 4, 13)
net2 (2, 1, 12) (2, 4, 8)



EXAMPLES

TESTCASE 2

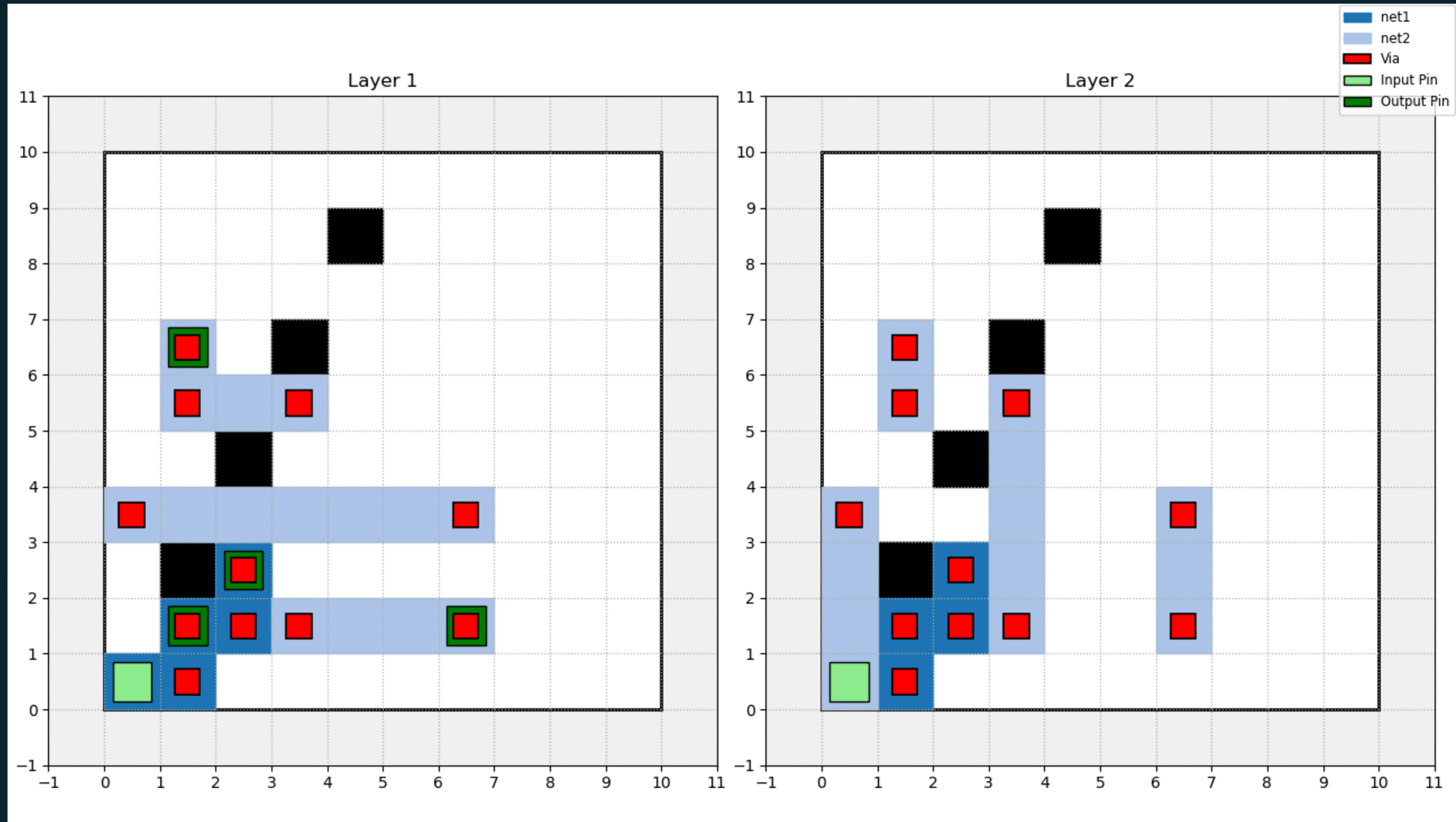


500, 500, 5, 9999
net1 (1, 0, 0) (1, 499, 499)
net2 (1, 0, 499) (1, 250, 250)
net3 (1, 40, 40) (1, 80, 80)

EXAMPLES

TESTCASE 3

10, 10, 5, 200
OBS (1, 1, 2)
OBS (1, 2, 4)
OBS (1, 3, 6)
OBS (1, 4, 8)
OBS (2, 1, 2)
OBS (2, 2, 4)
OBS (2, 3, 6)
OBS (2, 4, 8)
net1 (1, 0, 0) (1, 1, 1) (1, 2, 2)
net2 (2, 0, 0) (1, 6, 1) (1, 1, 6)



PROBLEM & LIMITATIONS

PROBLEM & LIMITATIONS

- Scalability Constraints: Routing and visualization runtimes degrade significantly on very large grids (e.g., 1000 × 1000), impacting usability and throughput.
- Layer Support: The router is hard-coded for only two metal layers (M1 horizontal, M2 vertical), limiting applicability to more complex multi-layer designs.
- Tracing & Debugging: There is no built-in mechanism to trace individual net segments or inspect intermediate routing decisions, making post-mortem analysis difficult.
- Routing Failure Handling: If no feasible path exists for a net, the system fails silently without reporting errors or suggestions for alternative strategies.
- Manual I/O Configuration: Visualization scripts require manual edits to specify input/output file names and paths, hindering automated workflow integration.
- Visualization Robustness: The renderer lacks input validation—misparsed grid dimensions or out-of-bounds net coordinates can produce incorrect or truncated plots without warning.

CONCLUSIONS

CONCLUSIONS

What We Built:

- A two-layer (M1 horizontal, M2 vertical) maze router using Lee's BFS with tunable via and direction penalties.

How We Did It:

- Parsing Module: C++ code to ingest grid, obstacles, and nets
- Routing Module: Priority-queue BFS over six moves (N/S/E/W, via up/down) with cost metrics
- Heuristic: Manhattan-span sorting to tackle hardest nets first
- Visualization: Python/Matplotlib renderer showing layer-separated, color-coded paths

Key Limitations:

- Layers: Fixed to two layers only
- Error Handling: Silent failures for unroutable nets; no path tracing
- Usability: Manual I/O setup and no input validation in visualization


THANK YOU