Maze Router: Optimal Net Routing with Multi-Layer Grid

Andrew Ishak - Michael Reda - Freddy Amgad - Kirolous Fouty

Objective: Automate net routing with constraints (bend/via penalties)

Features:

Multi-layer grid (M0, M1)
Dynamic handling of obstacles
Visual output for routed paths

Method 1: (__init__)

```
def __init__(self, grid_size, bend_penalty, via_penalty, move_cost=0):
    self.rows, self.cols = grid_size
    self.bend_penalty = bend_penalty
    self.via_penalty = via_penalty
    self.move_cost = move_cost
    self.grid_M0 = np.zeros((self.rows, self.cols))
    self.grid_M1 = np.zeros((self.rows, self.cols))
    self.obstacles = set()
    self.routes = []
```

Objective: Initializing the Maze Router object with grid parameters, penalties, and data structures

Parameters:

- grid_size: Size of the grid (rows, cols).
- bend_penalty: Cost for directional changes.
- via_penalty: Cost for switching layers.

Notes:

- Initializes two layers (M0, M1) as zero-filled grids.
- Stores obstacles and routes as separate attributes.

Method 2: (add_obstacle)

```
def add_obstacle(self, layer, x, y):
    if layer == 0:
        self.grid_M0[x, y] = -1
    elif layer == 1:
        self.grid_M1[x, y] = -1
    self.obstacles.add((layer, x, y))
```

Objective: Add obstacles to the specified layer at given coordinates

Parameters:

- layer: Grid layer (0 or 1).
- (x, y): Coordinates on the grid.

Notes:

- Updates the grid layer with -1 to mark obstacles.
- Adds obstacle to the obstacles set for easy tracking.

Method 3: (parse_input)

```
def parse_input(self, filename):
    with open(filename, 'r') as file:
        lines = file.readlines()
    ...
```

Objective: Parse grid size, penalties, obstacles, and nets from an input file

Notes:

- Extracts grid parameters (rows, cols, penalties).
- Identifies obstacles (OBS) and nets (net).
- Handles malformed input gracefully.
- Reads file line by line.
- Differentiates between obstacle and net lines.
- Populates internal data structures for routing.

Method 4: (route_net) *first_part*

```
def route_net(self, net_name, pins):
    visited = set()
    queue = []
    start_pin = pins[0]
    heapq.heappush(queue, (0, start_pin[0],
start_pin[1], start_pin[2], [], None))
    ...
```

Objective: Find an optimal route for a single net considering bend and via penalties

Notes:

- Implements A* algorithm.
- Uses a priority queue (heapq) for efficient pathfinding.
- Initializes the queue with the starting pin.
- Tracks visited nodes to avoid redundant computations.

Method 4: (route_net) *second_part*

```
while queue:
    cost, layer, x, y, path, prev_dir = heapq.heappop(queue)
    ...
    for d_layer, dx, dy, penalty in ...:
        if valid_move(nx, ny):
            heapq.heappush(queue, (cost + move_cost, nl, nx, ny, path, new_dir))
```

Notes:

- Processes each node in the queue.
- Considers penalties for bends and vias
- Explores neighbors and updates costs.
- Adds valid moves to the queue with updated penalties.

Method 5: (heuristic_order)

```
def heuristic_order(self):
    def net_priority(net):
        _, pins = net
        return sum(abs(p1[1] - p2[1]) + abs(p1[2] - p2[2]) for p1 in pins for p2 in pins)
    self.routes.sort(key=net_priority)
```

Objective: Prioritize nets based on distance between pins

Notes:

- Computes Manhattan distances for all pin pairs.
- Sorts nets by total distance.
- Ensures nets with closer pins are routed first

Method 6: (route_all)

```
def route_all(self):
    self.heuristic_order()
    output_routes = []
    for net_name, pins in self.routes:
        route = self.route_net(net_name, pins)
        if route:
            output_routes.append((net_name, route))
    return output_routes
```

Objective: Route all nets in the priority order

Notes:

- Combines heuristic_order and route_net to route all nets sequentially.

Method 7: (visualize)

```
def visualize(self, output_routes):
    fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(20, 10))
    ...
    for net_name, route in output_routes:
        for layer, x, y in route:
            ...
```

Objective: Generate visual representation of routed paths

Notes:

- Plots obstacles and routed paths for M0 and M1.
- Highlights start (S), end (E), and via points.

Test Case 1:

Input:

20, 20, 10, 50

OBS(1, 2, 1)

OBS(1, 3, 2)

OBS(1, 2, 4)

OBS(0, 2, 3)

OBS(0, 1, 3)

OBS(0, 4, 2)

OBS(1, 10, 7)

net1 (0, 1, 1) (1, 3, 3)

net2 (0, 1, 2) (1, 3, 4)

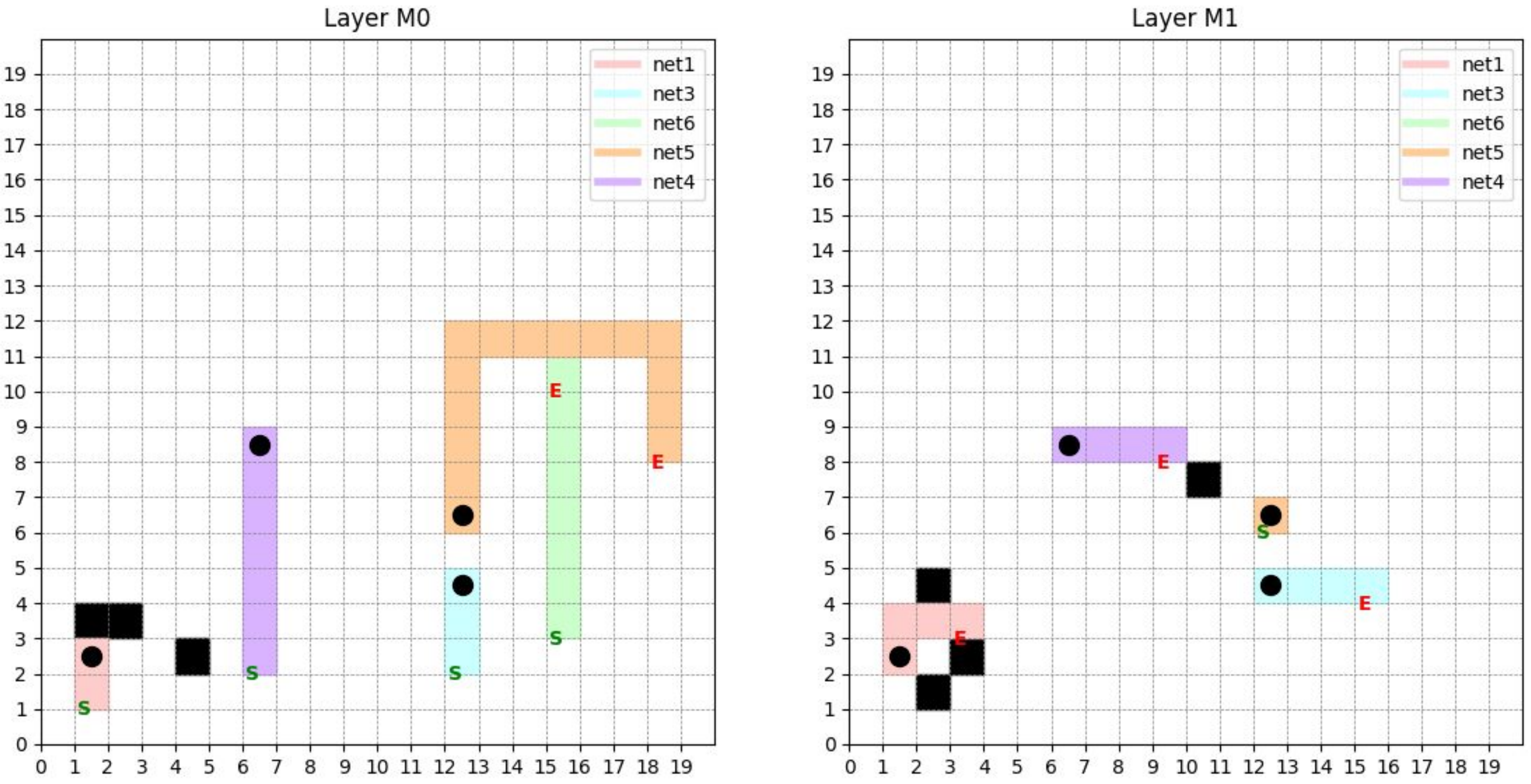
net2 (0, 1, 2) (1, 3, 4)

net3 (0, 12, 2) (1, 15, 4)

net4 (0, 6, 2) (1, 9, 8)

net5 (1, 12, 6) (0, 18, 8)

net6 (0, 15, 3) (0, 15, 10)


Layer M0
Layer M1
net1
net3
net6
net5
net4
S
E

Test Case 2:

Input:

5, 5, 10, 50

OBS(1, 2, 1)

OBS(1, 3, 3)

OBS(1, 3, 2)

OBS(1, 2, 4)

OBS(0, 2, 3)

OBS(0, 1, 3)

OBS(0, 4, 2)

net1 (0, 1, 1) (1, 4, 4)