Race – EP4 Racing Maze Navigator

Challenge Preamble

In this challenge, your task is to implement the decision-making logic of the AI that controls the car navigating a vertical scrolling maze. The grid is 100 rows by 10 columns, and obstacles (#) appear throughout the path. At each time step, the AI can move left (-1), stay (0), or move right (+1). Your objective is to create a lookahead-based algorithm that evaluates possible future moves and chooses the safest and least costly path forward.

Part 1 – Concepts and Theory

This part introduces the concepts needed to implement a predictive AI search. At every tick, the AI receives a list of visible rows ahead (the lookahead window). The goal is to evaluate possible move sequences up to a limited depth using a recursive depth-first exploration. Each branch of this decision tree represents a sequence of left, stay, or right movements. By exploring these possibilities, the AI chooses the first move of the lowest-cost safe path.

This recursive depth-limited exploration resembles simplified minimax logic without an opponent. The goal is purely to minimize horizontal movement cost while avoiding collisions. Exploring 3 moves over a depth of 5 rows results in a manageable **3^5** = **243** possible paths per turn, which is computationally feasible.

You will receive in the package several files (**`main.py, Engine.py, Loader.py`** and 1 track file) that are already implemented. You are only responsible for the **`AI.py`** file

Part 2 – Implementing AI.py

Part 2.1: Introduction

In this section, we walk through the implementation of the **`AI.py`** file. The file contains an AI class with a **`choose_move()`** method that is called every tick by the Engine. The AI examines the next few rows, recursively simulates possible moves, evaluates costs, and selects the best immediate action.

You are expected to modify or expand this logic to create an effective pathfinding strategy within the constraints of the challenge.

One potential implementation of this is to use recursive to create a sort of tree of all possible path, assigning a score based of its movement.

Basically the logic is at each recursion layer:

1. Try moving left, stay, or right.
2. Reject moves that are:
    - Out of bounds
    - Collisions
3. Add movement cost
4. Recursively explore deeper rows
5. Return the **smallest total cost** found

Imagine you use those 3 methods in your `AI.py` :
- `choose_move()` : Entry point called by the engine
- `_search_branch()` : Recursively explores future move sequences
- `_is_better_move()` : Tie-breaking logic to choose between equally-good moves

Part 2.2 : Line-by-Line Explanation : `search_branch`

Function signature

```
def _search_branch(self, visible_rows, depth, cur_x, cost_so_far):
```

- `visible_rows`: the next N rows ahead of the car.
- `depth`: how deep we are in the recursion (which future row we're considering).
- `cur_x`: the car's simulated column at this recursion level.
- `cost_so_far`: movement cost accumulated along this hypothetical path.

This function answers the question:
"If I am at (depth, cur_x) with cost X, and I look further ahead, what is the best total cost I can achieve?"

Base condition — stop the recursion

```
if depth >= self.max_depth:
    return cost_so_far
```

- If we have simulated enough rows (= reached lookahead horizon), we stop.
- There are no more predicted moves to evaluate.
- We return the total cost accumulated so far.
- This is similar to **leaf node evaluation** in minimax.

This ensures we do not explore the infinite future.

Extract the row we're entering at this recursion step

```
row = visible_rows[depth]
```

- At recursion depth `depth`,
- We examine **`visible_rows[depth]`** because:
  - **`depth = 0`** → AI already checked this outside
  - **`depth = 1`** → row immediately after
  - etc.

Initialize best cost for this branch

```
best: Optional[int] = None
```

- `best` will hold the **minimum total cost** among all sub-moves.
- Starts as `None` to indicate no valid move has been found yet.

Iterate through all possible moves

```
for move in (-1, 0, 1):
```

At each depth, the car can:

- `-1` = go left
- `0` = stay
- `+1` = go right

This is the branching factor of 3.

Compute next simulated position

```
nx = cur_x + move
if nx < 0 or nx >= self.width:
    continue
```

- Applying the move gives column `nx`.
- If `nx` is outside grid bounds:
    - This path is impossible → skip it.

Check for collision

```
if row[nx] == "#":
    continue
```

- If the next cell is a wall:
    - This move is invalid → prune this branch.
- This drastically reduces search size.

Accumulate cost

```
new_cost = cost_so_far + (1 if move != 0 else 0)
```

Movement cost rules:

- Staying still (`0`) is free
- Moving left or right (±1) costs **1**

So this calculates the updated cost after taking this move.

Recursive call

```
sub = self._search_branch(visible_rows, depth + 1, nx, new_cost)
```

This means:

- "Now that I've moved to column **`nx`** and increased cost, simulate the next row."

If the recursive call returns:

- An integer → the minimal total cost from this point to the horizon
- `None` → no valid continuation (dead end)

Skip dead-end branches

```
if sub is None:
    continue
```

If the next steps inevitably crash:

- This path is invalid → ignore it.

Update best cost

```
if best is None or sub < best:
    best = sub
```

- First valid total cost becomes the best.
- If future cost is smaller than current best → replace it.

This implements **minimization**.

Return minimal cost

```
return best
```

- Returns the minimal possible total movement cost that can be achieved from this depth onward.
- If all branches were invalid → `best` stays `None` → meaning "no safe path".

Note: All required files (**`AI.py`**, **`Engine.py`**, **`Loader.py`**, **`main.py`**, and the tracks folder) must be placed in the same directory for the project to run properly.

Part 3 – Extended Challenge

For the extended challenge, you may create a CLI visualization tool that displays the car navigating the maze in real time. This involves printing the grid to the terminal each tick and updating the car's position as it progresses. You may also experiment with adding color, animation timing, or more advanced visual effects. This extension is open-ended and encourages creativity.