

The Maze Runners present...



You are a clever thief navigating a maze filled with coins. Your goal is to collect as many coins as you can while avoiding Silly, the determined (and slightly goofy) cop chasing you down. Use your wits to place locks and block Silly's path, giving yourself a chance to escape. Be strategic — Silly won't give up easily!

Can you outsmart the cop and grab all the loot?

A brief introduction in which you explain your problem/topic

We aimed to create a Lock 'n' Chase-inspired maze game that evolves in complexity across three levels of difficulty. Each level leverages a different AI strategy:

1. **Easy Level:** The police officer chases the thief using the Simulated Annealing algorithm.
2. **Medium Level:** The police officer uses the A* algorithm integrated with Constraint Satisfaction Problems (CSP) to navigate around obstacles.
3. **Hard Level:** The police officer combines pathfinding with adversarial strategies (Mini-Max + Alpha-Beta Pruning) to improve chasing ability by finding the best possible move.

This game directly applies the AI concepts we're exploring in our course. It begins with Simulated Annealing, a straightforward approach, then advances to integrating A* with CSP, further reducing the search space. Finally, at the highest difficulty, adversarial search techniques are introduced.

Through this game, we will see how different AI strategies enhance the police officer's search ability in a dynamic environment where the user can lock certain locations and block the police officer's path.

Your proposed solution

Game mechanics: Theory

Characters:

1. Lupin (Player-controlled Thief):
 - Lupin's goal is to navigate the maze and collect as many coins as possible while avoiding capture by Silly.
 - **Controls:** Use the **WASD keys** to move Lupin up, left, down, and right.

- **Special Ability:** Press **Q** to strategically place locks in the maze, blocking paths to hinder Silly's pursuit.

2. Silly (AI-controlled Cop):

- Silly is a determined cop who relentlessly pursues Lupin using different AI strategies based on the difficulty level.
- Easy Level: Uses the **Simulated Annealing** algorithm.
- Medium Level: Uses the **A* algorithm** with **Constraint Satisfaction Problems (CSP)**.
- Hard Level: Uses a combination of **Minimax** and **Alpha-Beta Pruning** with depth limited search.

Game mechanics: Code

We have three .py files: main.py (the entry point that initializes and runs the game), menu.py (manages menus), and game.py (handles game mechanics and AI). This structure ensures modularity, making the code scalable and easy to maintain or extend with new features (thank you, SDA).

1. Game Class (in game.py):

- Manages the overall game loop, menus, and player inputs.
- Key Functions:
 - `game_loop()`: The main loop that handles game updates, drawing, and AI logic.
 - `check_events()`: Checks for player inputs.
 - `reset_keys()`: Resets key flags after each loop iteration.
 - `draw_text()`: Helper function to draw text on the screen.

2. MazeGame Class (in game.py):

- Implements the core game mechanics, including the maze layout, player movement, AI behavior, and collisions.
- Key Mechanics:
 - **Maze Structure:** Defined by a 2D list where different numbers represent walls, paths, and coins.

- Player movement: Handled in `handle_input_player()` and `move_player()`, with controls mapped to WASD keys.
- Lock Placement: Players can place a lock with the Q key using `place_lock()` after every 5 seconds using `update_lock()`.
- Silly's AI: Level 1 uses Simulated Annealing (`handle_input_silly_level1_simulated_annealing()`), Level 2 implements A* with CSP to navigate obstacles (`a_star_search()`), and Level 3 uses Minimax with Alpha-Beta Pruning for adversarial pathfinding (`best_move()`).
- Collision Detection: `check_collisions()` checks if the player is caught by Silly and updates lives accordingly.

Game mechanics: Search Algorithms

Level 1: Simulated Annealing:

This is the easiest level, where Silly tries to catch Lupin using the Simulated Annealing algorithm. We chose this approach because it is the simplest algorithm to start with.

Simulated Annealing is a local search algorithm that looks for nearby better moves and accepts them. In this case, Silly's local moves are up, down, left, and right. If no better move is found, it discards the current move, similar to hill climbing. The key difference is that Simulated Annealing introduces an initial temperature, which decreases over time. When the temperature is high at the start, Silly is more likely to accept suboptimal moves, showing a greedy behavior. This means that early in the game, Silly may make poor decisions in its eagerness to catch Lupin. A probability factor is applied, allowing even bad moves to be accepted at the beginning. As the temperature decreases, Silly becomes more selective and starts making smarter choices to better catch Lupin.

When playing at Level 1, you'll notice that Silly initially makes uncertain movements but gradually improves, making smarter decisions as the temperature decreases.

Among all the AI algorithms we implemented, Silly using Simulated Annealing is the slowest and least efficient at catching Lupin.

Key Functions:

1. *calculate_distance(self, pos1, pos2):*

We are using manhattan distance in our game cz its grid based navigation meaning u could move up down right left (no diagonal moves(straight line like euclidean distance) allowed) hence a manhattan distance heuristic used in levels

2. *get_random_neighbor(self, current_pos):*

For given current position generates a random neighbor in up,down,left,right direction and makes sure that the neighbor is valid such as not a wall and avoids locks placed by lupin

3. *is_one_step_away(self, pos, lock_pos):*

Returns if a given position is exactly one step away from the lock's position meaning lock places one step ahead of current position .

4. *handle_input_silly_level1_simulated_annealing(self):*

Implements the simulated annealing algorithm for "Silly" by probabilistically choosing better moves due to heuristic i.e. Manhattan distance to "Lupin" and temperature as we discussed above.

Time Complexity:

$O(T \cdot N)$, where:

- T = number of temperature steps (iterations), and
- N = number of neighbors evaluated at each temperature step.

Level 2: A* algorithm with Constraint Satisfaction Problem (CSP):

In level 2, as we are trying to make our police officer, Silly, smarter, we implemented a better and more efficient pathfinding algorithm. We can now confidently say that Silly is smarter because A* is a global pathfinding algorithm that finds the best

shortest path from the source to the goal which is a definite step up from the simulated annealing (a local search algorithm) used in Level 1.

For implementing A*, we used the same heuristic as in Level 1, the Manhattan distance between Silly and Lupin. The coolest aspect of our A* implementation is its ability to dynamically counter the player. When the player places a lock in Silly's path, Silly immediately reconstructs the path to catch the player at that moment. This makes the gameplay more challenging and ensures Silly always finds the shortest path from its current position to the player instantly.

One significant challenge we faced, which took some to debug, was handling the old path. If a lock was placed in the previously calculated best path, the old path needed to be completely deleted (set to None or null). A new path would then be reconstructed dynamically based on the updated positions of the player and locks. To address this, we added boolean attributes, such as obstacle and player_moved, which are set to True whenever the player (Lupin) moves or places a lock in the maze, obstructing the previous path.

In addition to A*, we incorporated Constraint Satisfaction Problems (CSP) concepts, such as backtracking, into the algorithm. For instance, if the next move leads to Silly making an invalid move (e.g., running into a lock placed by the player), the A* algorithm backtracks and avoids that move. This further optimizes the search space and prevents Silly from getting into loops, stuck positions, or blocked areas.

In this context, each cell of the maze is treated as a variable, and its domain consists of all possible adjacent cells.

Key Functions:

1. *a_star_search(self, start, goal):*

Implements the A* search algorithm to find the shortest path from the start position to the goal, using the Manhattan distance heuristic for prioritizing paths.

2. *heuristic(self, a, b):*

Returns the Manhattan distance between two points a and b, serving as a heuristic to guide A* towards the goal.

3. *get_neighbors(self, pos):*

Returns a list of valid neighboring positions (adjacent positions) for a given position by considering movement directions, walls and blocking obstacles like locks.

4. *is_blocked(self, pos):*

Checks whether a position is blocked by a wall, lock, or other obstacle, and returns True if the position is inaccessible.

5. *reconstruct_path(self, came_from, start, goal):*

Constructs the shortest path from the start position to the goal by backtracking through the came_from dictionary. The path is reversed to create a proper sequence for Silly to follow and is returned as the final result.

6. *move_silly_locks(self, next_pos):*

Attempts to move Silly to the specified next_pos if it is not blocked, and updates Silly's position if the move is successful and returns True. If the position is blocked, the move is unsuccessful, and the function returns False.

Time Complexity:

$O(b^d)$, where:

- b = branching factor, and
- d = depth.

Level 2: Minimax with Alpha-Beta Pruning:

For the final level of the game, Level 3, we implemented an advanced AI for Silly where pathfinding is achieved by evaluating all possible moves Silly could make and choosing the best-performing one to ensure its victory (and Lupin's defeat 🐈). Evaluating the entire game tree to its full depth, including all terminal nodes, would have required significant processing power and resulted in a very laggy game. To address this, we limited the depth of the game tree to three levels. Despite this constraint, the algorithm performs well within the game's requirements.

Like any minimax algorithm, we began by identifying the max player (Silly) and the min player (Lupin). Lupin, as the min player, tries to block Silly from catching it by placing obstacles, while Silly, as the max player, focuses on two objectives: countering Lupin's moves and maximizing its own chances of winning.

The efficiency of the minimax algorithm heavily depends on the evaluation function. In our case, the function is tailored to favor Silly's victory. The `evaluate_positions` function assigns a score to each move, giving the highest preference (a score of infinity) to a move where Silly catches Lupin.

This adversarial environment creates a competitive dynamic, where one player strives to win while the other attempts to prevent that outcome. The coolest aspect of incorporating alpha-beta pruning alongside the minimax algorithm is that it guarantees optimality while significantly reducing the search space—sometimes by nearly half.

One challenge we encountered was Silly getting stuck in an oscillating loop, moving back and forth between two positions due to identical scores being assigned. To resolve this, we introduced a mechanism to penalize repeated moves. If a move appeared in the move history (updated iteratively), its score was reduced by -1. This ensured that the same move wouldn't be repeated, effectively breaking the oscillation.

Key Functions:

1. *evaluate_position(self)*:

This is the evaluation function, which returns a score representing how good a move is. The function also tracks the coins collected, contributing to the player's score for evaluation. A negative value is returned because Silly aims to minimize the distance to Lupin while maximizing the overall evaluation score. The closer Silly is to Lupin, the higher the score returned.

2. *get_all_possible_moves(self)*:

Returns a list of all valid adjacent moves for Silly, avoiding obstacles like walls and locks.

3. *minimax(self, depth, alpha, beta, is_maximizing)*:

Implements the recursive Minimax algorithm with alpha-beta pruning. The function evaluates moves turn by turn, depending on whether it's the min or max player's turn.

4. *best_move(self, depth, is_maximizing)*:

Initiates the Minimax algorithm starting from the root node. The default search depth is set to 3.

5. *game_over(self)*:

Checks for winning conditions for Silly and returns True if the game is over. This includes conditions like Silly catching Lupin or Lupin running out of lives.

6. *clone_state(self)*:

Creates a clone of the current game state to simulate future moves without affecting the original game state. This is necessary because moves cannot be simulated on the original state until the best move is decided. Only essential attributes are cloned, and the game state can be restored later.

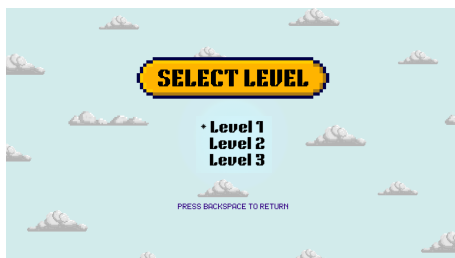
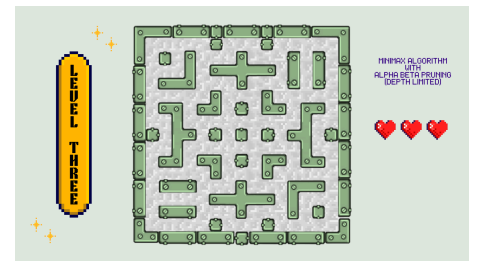
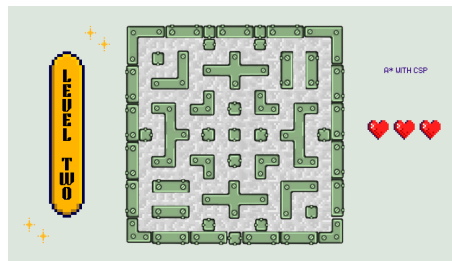
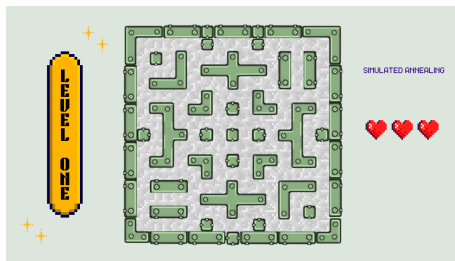
Time Complexity:

$O(b^d)$, where:


- b = branching factor, and
- d = depth (constant 3 in our game).

Results of your Approach

Screens



Game video

 i222336_i222348_demo.mp4