REPORT

Alejandro Mejía A00399937
Santiago Angel Ordoñez A00379822
Julio Prado A00399637

INTRODUCTION

This project consists of developing an interactive escape room game modeled both in Python and Prolog. The system allows the player to actively participate in the game, and it also provides functionalities for the player to request hints or puzzle solutions. The game combines the power of logic programming with practical decision-making processes, enabling the player to make strategic choices as they navigate the escape room.

The project is built using Python, PySWIP, and Prolog, with three versions of the game. In the first two versions, Python handles both the interface and the game mechanics, including the puzzle logic and decision-making. In the third version, Python is used solely for displaying the game, while Prolog takes over the puzzle-solving and decision-making processes. These different versions offer varied gameplay experiences, with the option for players to explore the game at their own pace and request hints or solutions when needed.

METHODOLOGY

Prolog

For the Prolog implementation, the main objective was to integrate and present a solution for the Python-based game using the Prolog logic covered during class sessions. The core game was developed in Python, but with the addition of Prolog to assist the player in finding clues or alternative ways to solve the puzzles presented in each room.

The player must navigate through a matrix, choosing the best path to solve the puzzle, while also collecting items, solving challenges, and opening doors. If the player needs assistance, there are two support options available. The first provides a single hint suggesting the next optimal move in the room, while the second reveals the complete solution to the puzzle if needed. (**Logic Programming (Prolog)**)

Additionally, for players seeking a more challenging experience, there's a feature that allows puzzles to be solved by deciphering a secret code using a set of clues. (**Constraint Logic Programming (Prolog with Constraints)**)

A star

Problem Definition

For this implementation the primary objective is to find an optimal sequence of actions (path) for a player agent to navigate a grid-based escape room environment. The agent starts at a defined position and must achieve a specific goal state reaching a designated door location while possessing a key within a maximum allowed number of turns (actions). The environment includes interactable elements: an item required to solve a puzzle, and the puzzle itself which grants the key upon solving. Each action, including movement (up, down, left, right), picking up the item, and solving the puzzle, consumes one turn. Using the door to exit also consumes a final turn.

Approach Chosen

Given the need to find an optimal path (minimum number of turns) in a state space that includes not only position but also the player's inventory (has item, has key), an informed search algorithm is the most suitable approach. The A* search algorithm was selected for the following reasons:

- Optimality: A* guarantees finding the shortest path (in terms of accumulated cost, which is 'turns' in this case) if the heuristic function used is admissible (never overestimates the true cost to reach the goal) and consistent (optional, but helps efficiency).
- Efficiency: As an informed search, A* uses a heuristic function to prioritize exploring states that appear closer to the goal, generally exploring fewer states than uninformed search algorithms like Breadth-First Search (BFS) or Dijkstra's algorithm (especially in large state spaces). While Dijkstra also finds the shortest path, A*'s heuristic guidance typically makes it faster.
- State Space Handling: A* can effectively navigate complex state spaces where the cost depends not just on movement but also on the current state (e.g., needing the item before going to the puzzle).

A* algorithm core concepts

A* explores the state space by maintaining a priority queue of states to visit, ordered by an evaluation function $f(n)$:

$$f(n) = g(n) + h(n)$$

Where:

- n: Represents a specific state (State(x, y, has_key, has_item, solved_puzzle)).
- $g(n)$: The cost so far to reach state n from the start state. In this implementation, $g(n)$ is the number of turns taken to reach state n.
- $h(n)$: The heuristic estimate of the cost (turns) remaining to reach the goal state from state n. This is an informed guess.

Heuristic Function $h(n)$

A crucial component of A* is the heuristic. We employed the **Manhattan Distance** adapted to the game's state. The Manhattan Distance ($abs(x_1 - x_2) + abs(y_1 - y_2)$) is used as the base because movement is restricted to horizontal and vertical steps (cost of 1 per step). It's admissible as it represents the minimum number of moves required on a grid without obstacles.

The heuristic dynamically changes based on the player's state (has_key, has_item):

- If **has_key** is **True**: The player only needs to reach the door. $h(n)$ = Manhattan distance from the current position (n.x, n.y) to the door_pos.
- If **has_item** is **True** (but no key): The player needs to go to the puzzle, then the door. $h(n)$ = Manhattan distance (current -> puzzle_pos) + Manhattan distance (puzzle_pos -> door_pos).
- If **has_item** is **False** and **has_key** is **Fals**e: The player needs the full sequence: item -> puzzle -> door. $h(n)$ = Manhattan distance (current -> item_pos) + Manhattan distance (item_pos -> puzzle_pos) + Manhattan distance (puzzle_pos -> door_pos).

This state dependent heuristic guides the search more effectively towards the necessary sub-goals (item, puzzle) before heading to the final door.

A star Minimax

This algorithm is used to control the decision-making process in an environment for a competitive game in which two competing agents—a player and an enemy—are involved. The algorithm is perfectly suitable for use when there are different possible moves and a best path must be found through different states of the game.

Problem Definition

The goal is to develop an enemy that makes the game more challenging for the player. The position in which the enemy is cannot be the same as the user's position.

Approach Chosen

Minimax algorithm exhausts different game states recursively, looking ahead a fixed number of turns (depth). It alternates between the enemy's turn and the player's turn, looking at what moves will maximize the best possible outcome in light of the game's scoring mechanism. By looking at all possible future states, it ensures that both players are playing as well as possible.

The algorithm goes on to study two primary operations:
1. Minimizing Player (Enemy's Move): The enemy will try to minimize the score, i.e., it will pick moves that will give the least score.
2. Maximizing Player (Player's Move): The player aims to maximize the score, selecting moves that maximize the score to the highest extent.

In our case:
1. If the enemy's turn in the current iteration (turn == 0), it recursively calculates enemy moves and selects the best one with the lowest score.
2. If the player's turn in the current iteration (turn == 1), it calculates player moves and selects the best one with the highest score.

The recursive process goes through different future turns and selects the best move for each turn.Minimax ensures the best possible decision for the player and the opponent within the depth of the search space.

IMPLEMENTATION DETAILS

Prolog

The escape_solver.pl module encapsulates the game's navigation logic and state transitions. It defines the movement rules, game states, room structure, and pathfinding mechanics to reach the goal.

Room Configuration: Each room is defined using room_data/3, which specifies the positions of the door, puzzle, and item within a fixed 5x5 grid. The player starts in room 1 and progresses sequentially to room 3.

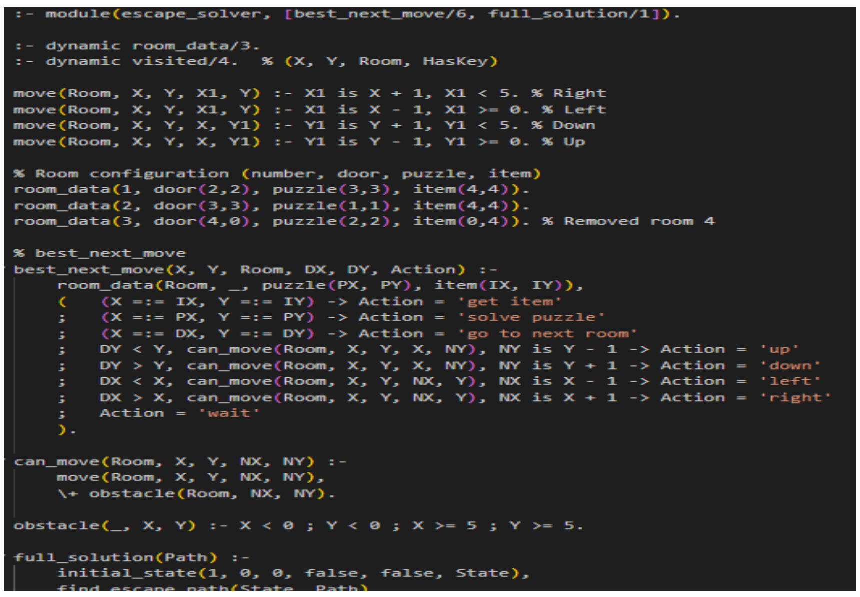
Game State Representation: States are represented as state(Room, X, Y, HasKey, HasItem), capturing the player's room number, position, and inventory status. This structure is used to evaluate legal actions and determine progression.

Pathfinding: The full_solution/1 predicate serves as the system's entry point for computing a complete solution. It initializes the starting state and recursively explores valid actions using find_escape_path/2.

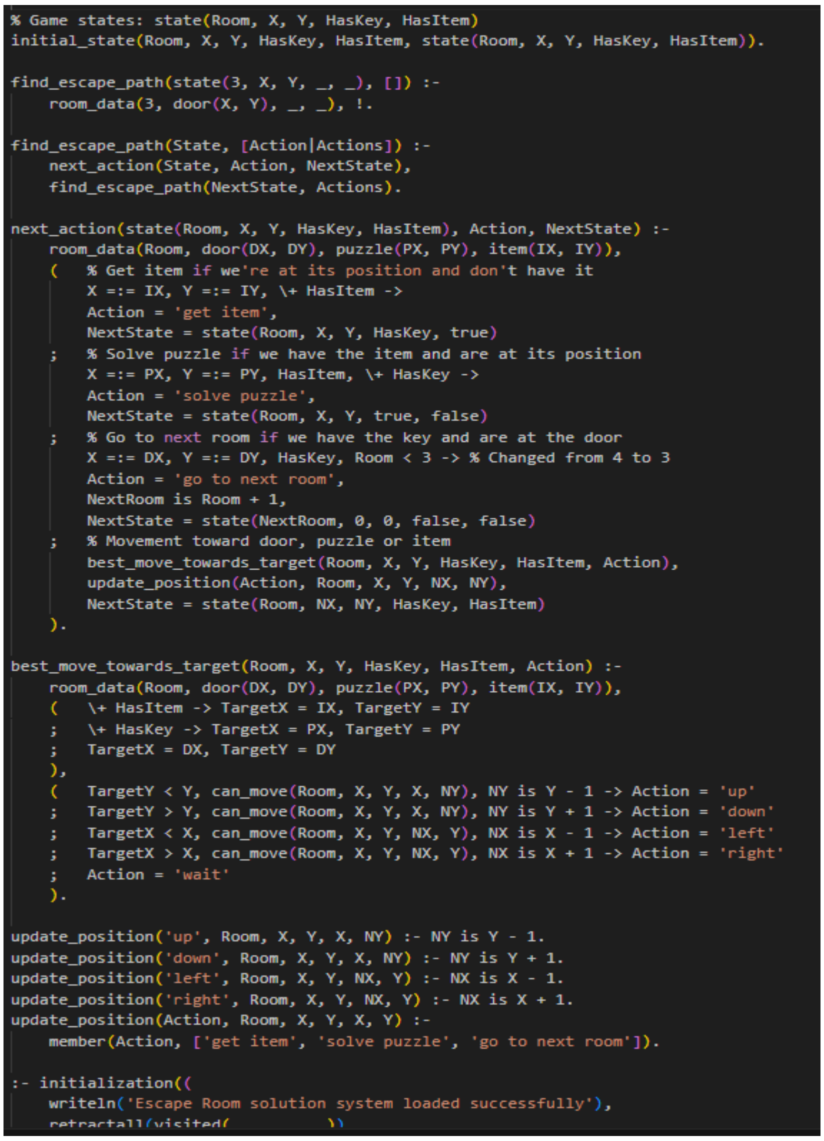
Action Generation: The predicate next_action/3 defines context-sensitive decisions. Depending on the player's location and inventory, it may trigger actions such as 'get item', 'solve puzzle', or 'go to next room'. If none of these conditions apply, it calculates the best direction to move using best_move_towards_target/6.

Movement Mechanics: Directional actions (up, down, left, right) are implemented through predicates such as move/5, while update_position/6 handles state transitions after a movement or interaction.

Action Prioritization: The solver favors purposeful movement toward required elements. If the item has not been picked up, it guides the player to the item's coordinates. If the item is acquired but the puzzle remains unsolved, it targets the puzzle. Finally, with the key in hand, it directs the player toward the door.



```
:- module(escape_solver, [best_next_move/6, full_solution/1]).

:- dynamic room_data/3.
:- dynamic visited/4.  % (X, Y, Room, HasKey)

move(Room, X, Y, X1, Y) :- X1 is X + 1, X1 < 5. % Right
move(Room, X, Y, X1, Y) :- X1 is X - 1, X1 >= 0. % Left
move(Room, X, Y, X, Y1) :- Y1 is Y + 1, Y1 < 5. % Down
move(Room, X, Y, X, Y1) :- Y1 is Y - 1, Y1 >= 0. % Up

% Room configuration (number, door, puzzle, item)
room_data(1, door(2,2), puzzle(3,3), item(4,4)).
room_data(2, door(3,3), puzzle(1,1), item(4,4)).
room_data(3, door(4,0), puzzle(2,2), item(0,4)). % Removed room 4

% best_next_move
best_next_move(X, Y, Room, DX, DY, Action) :-
    room_data(Room, _, puzzle(PX, PY), item(IX, IY)),
    (   (X =:= IX, Y =:= IY) -> Action = 'get item'
    ;   (X =:= PX, Y =:= PY) -> Action = 'solve puzzle'
    ;   (X =:= DX, Y =:= DY) -> Action = 'go to next room'
    ;   DY < Y, can_move(Room, X, Y, X, NY), NY is Y - 1 -> Action = 'up'
    ;   DY > Y, can_move(Room, X, Y, X, NY), NY is Y + 1 -> Action = 'down'
    ;   DX < X, can_move(Room, X, Y, NX, Y), NX is X - 1 -> Action = 'left'
    ;   DX > X, can_move(Room, X, Y, NX, Y), NX is X + 1 -> Action = 'right'
    ;   Action = 'wait'
    ).

can_move(Room, X, Y, NX, NY) :-
    move(Room, X, Y, NX, NY),
    \+ obstacle(Room, NX, NY).

obstacle(_, X, Y) :- X < 0 ; Y < 0 ; X >= 5 ; Y >= 5.

full_solution(Path) :-
    initial_state(1, 0, 0, false, false, State),
    find_escape_path(State, Path)
```



```prolog
% Game states: state(Room, X, Y, HasKey, HasItem)
initial_state(Room, X, Y, HasKey, HasItem, state(Room, X, Y, HasKey, HasItem)).

find_escape_path(state(3, X, Y, _, _), []) :-
    room_data(3, door(X, Y), _, _), !.

find_escape_path(State, [Action|Actions]) :-
    next_action(State, Action, NextState),
    find_escape_path(NextState, Actions).

next_action(state(Room, X, Y, HasKey, HasItem), Action, NextState) :-
    room_data(Room, door(DX, DY), puzzle(PX, PY), item(IX, IY)),
    (   % Get item if we're at its position and don't have it
        X =:= IX, Y =:= IY, \+ HasItem ->
        Action = 'get item',
        NextState = state(Room, X, Y, HasKey, true)
    ;   % Solve puzzle if we have the item and are at its position
        X =:= PX, Y =:= PY, HasItem, \+ HasKey ->
        Action = 'solve puzzle',
        NextState = state(Room, X, Y, true, false)
    ;   % Go to next room if we have the key and are at the door
        X =:= DX, Y =:= DY, HasKey, Room < 3 -> % Changed from 4 to 3
        Action = 'go to next room',
        NextRoom is Room + 1,
        NextState = state(NextRoom, 0, 0, false, false)
    ;   % Movement toward door, puzzle or item
        best_move_towards_target(Room, X, Y, HasKey, HasItem, Action),
        update_position(Action, Room, X, Y, NX, NY),
        NextState = state(Room, NX, NY, HasKey, HasItem)
    ).

best_move_towards_target(Room, X, Y, HasKey, HasItem, Action) :-
    room_data(Room, door(DX, DY), puzzle(PX, PY), item(IX, IY)),
    (   \+ HasItem -> TargetX = IX, TargetY = IY
    ;   \+ HasKey -> TargetX = PX, TargetY = PY
    ;   TargetX = DX, TargetY = DY
    ),
    (   TargetY < Y, can_move(Room, X, Y, X, NY), NY is Y - 1 -> Action = 'up'
    ;   TargetY > Y, can_move(Room, X, Y, X, NY), NY is Y + 1 -> Action = 'down'
    ;   TargetX < X, can_move(Room, X, Y, NX, Y), NX is X - 1 -> Action = 'left'
    ;   TargetX > X, can_move(Room, X, Y, NX, Y), NX is X + 1 -> Action = 'right'
    ;   Action = 'wait'
    ).

update_position('up', Room, X, Y, X, NY) :- NY is Y - 1.
update_position('down', Room, X, Y, X, NY) :- NY is Y + 1.
update_position('left', Room, X, Y, NX, Y) :- NX is X - 1.
update_position('right', Room, X, Y, NX, Y) :- NX is X + 1.
update_position(Action, Room, X, Y, X, Y) :-
    member(Action, ['get item', 'solve puzzle', 'go to next room']).

:- initialization((
    writeln('Escape Room solution system loaded successfully'),
    retractall(visited(           ))
```

The clp_hints.pl module encapsulates the logic of each puzzle using Constraint Logic Programming over Finite Domains (CLP(FD)). Each puzzle defines a set of variables, domain constraints, and relationships which must be satisfied to obtain a numeric solution or code.

Puzzle Definition: Each room has a corresponding room_puzzle/4 entry, which includes:

A list of Prolog variables.

A constraint expression that defines the logical conditions.

A solution pattern used to extract the puzzle's final code.

Puzzle Solving: The main interface, get_puzzle_info/3, performs the following:

Applies the constraints via call(Constraints).

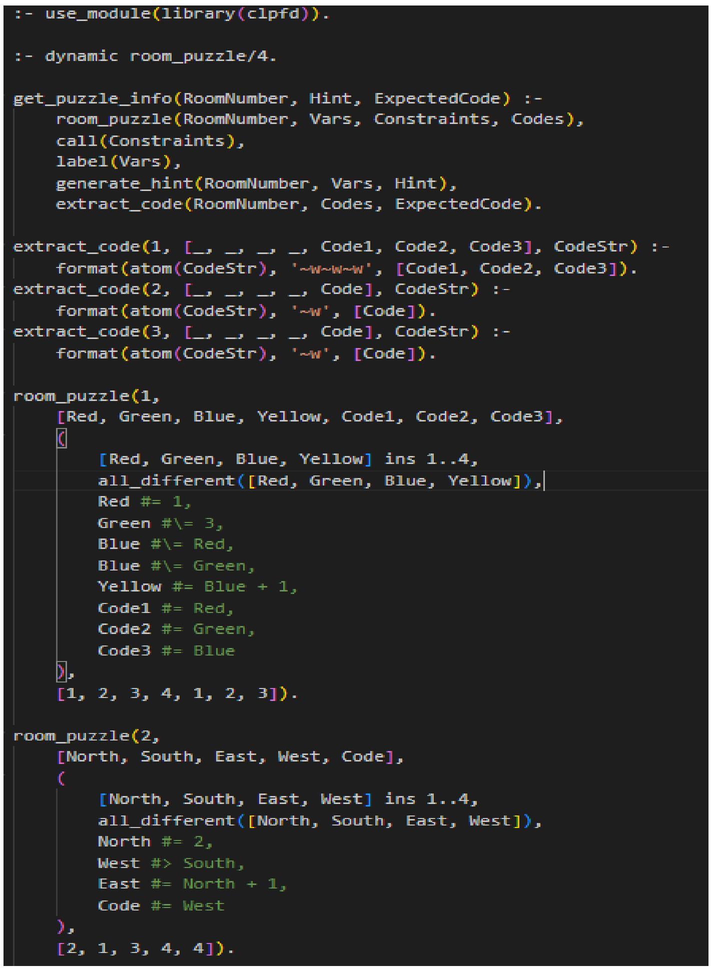
Uses label/1 to assign concrete values to the variables.

Generates a hint using generate_hint/3.

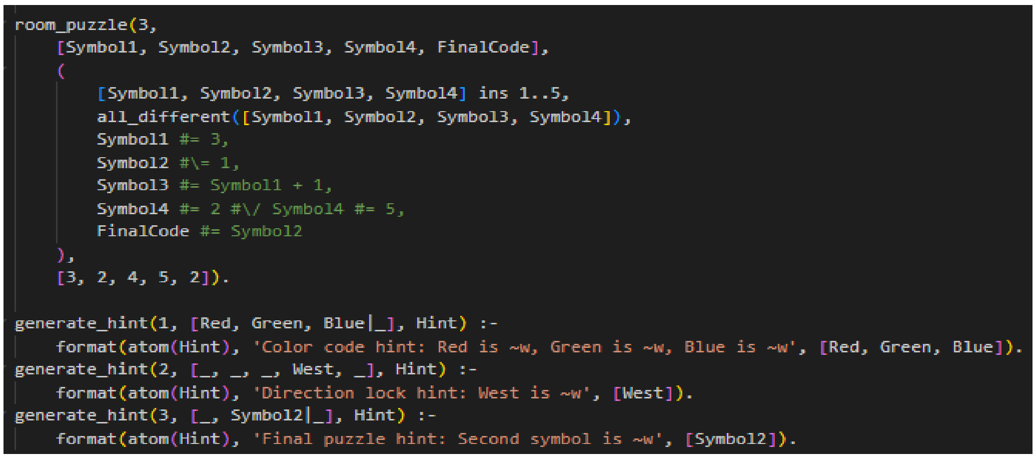
Extracts the expected code using extract_code/3.

Hints and Codes: The hint system is designed to assist players without giving away full answers. Hints are derived from partially revealed variable values. Codes, meanwhile, represent the numeric solution needed to progress after solving a puzzle.

This CLP-driven puzzle engine adds depth to the escape room by creating logical challenges that are solvable through declarative constraints rather than hardcoded solutions.



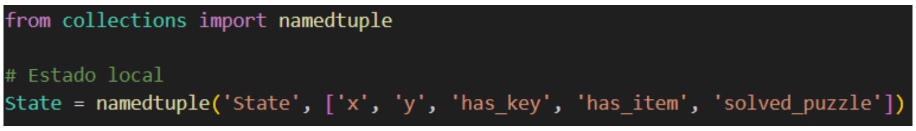
```prolog
:- use_module(library(clpfd)).

:- dynamic room_puzzle/4.

get_puzzle_info(RoomNumber, Hint, ExpectedCode) :-
    room_puzzle(RoomNumber, Vars, Constraints, Codes),
    call(Constraints),
    label(Vars),
    generate_hint(RoomNumber, Vars, Hint),
    extract_code(RoomNumber, Codes, ExpectedCode).

extract_code(1, [_, _, _, _, Code1, Code2, Code3], CodeStr) :-
    format(atom(CodeStr), '~w~w~w', [Code1, Code2, Code3]).
extract_code(2, [_, _, _, _, Code], CodeStr) :-
    format(atom(CodeStr), '~w', [Code]).
extract_code(3, [_, _, _, _, Code], CodeStr) :-
    format(atom(CodeStr), '~w', [Code]).

room_puzzle(1,
    [Red, Green, Blue, Yellow, Code1, Code2, Code3],
    (
        [Red, Green, Blue, Yellow] ins 1..4,
        all_different([Red, Green, Blue, Yellow]),
        Red #= 1,
        Green #\= 3,
        Blue #\= Red,
        Blue #\= Green,
        Yellow #= Blue + 1,
        Code1 #= Red,
        Code2 #= Green,
        Code3 #= Blue
    ),
    [1, 2, 3, 4, 1, 2, 3]).

room_puzzle(2,
    [North, South, East, West, Code],
    (
        [North, South, East, West] ins 1..4,
        all_different([North, South, East, West]),
        North #= 2,
        West #> South,
        East #= North + 1,
        Code #= West
    ),
    [2, 1, 3, 4, 4]).
```



```prolog
room_puzzle(3,
    [Symbol1, Symbol2, Symbol3, Symbol4, FinalCode],
    (
        [Symbol1, Symbol2, Symbol3, Symbol4] ins 1..5,
        all_different([Symbol1, Symbol2, Symbol3, Symbol4]),
        Symbol1 #= 3,
        Symbol2 #\= 1,
        Symbol3 #= Symbol1 + 1,
        Symbol4 #= 2 #\/ Symbol4 #= 5,
        FinalCode #= Symbol2
    ),
    [3, 2, 4, 5, 2]).

generate_hint(1, [Red, Green, Blue|_], Hint) :-
    format(atom(Hint), 'Color code hint: Red is ~w, Green is ~w, Blue is ~w', [Red, Green, Blue]).
generate_hint(2, [_, _, _, West, _], Hint) :-
    format(atom(Hint), 'Direction lock hint: West is ~w', [West]).
generate_hint(3, [_, Symbol2|_], Hint) :-
    format(atom(Hint), 'Final puzzle hint: Second symbol is ~w', [Symbol2]).
```

A star

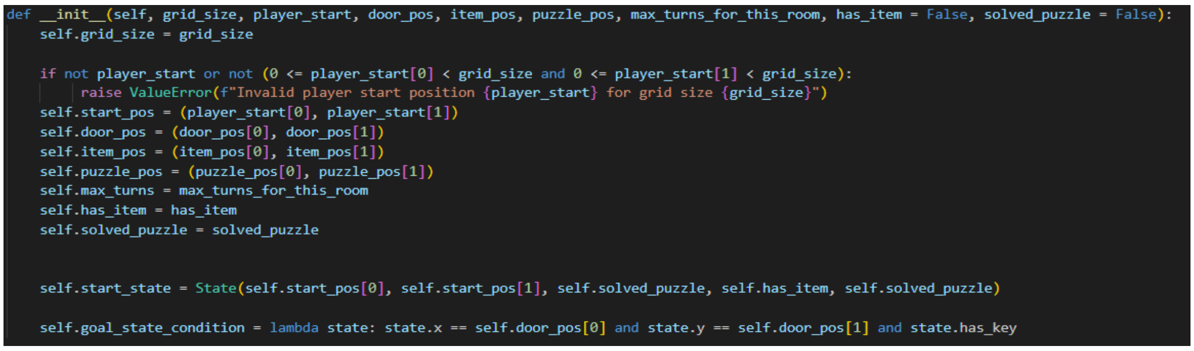
State Representation



```python
from collections import namedtuple

# Estado local
State = namedtuple('State', ['x', 'y', 'has_key', 'has_item', 'solved_puzzle'])
```

Solver Initialization

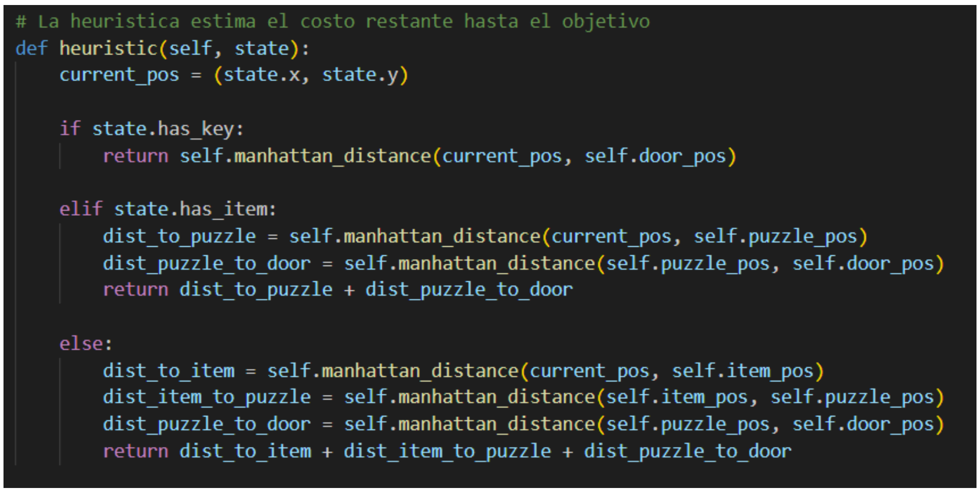
The EscapeRoomSolver class is initialized with all room parameters. The definition of the initial start_state is crucial, especially the has_key field, which depends on whether the puzzle is considered solved from the beginning (self.solved_puzzle). The goal condition (goal_state_condition) is defined as a **lambda** for quick checking.



```python
def __init__(self, grid_size, player_start, door_pos, item_pos, puzzle_pos, max_turns_for_this_room, has_item = False, solved_puzzle = False):
    self.grid_size = grid_size

    if not player_start or not (0 <= player_start[0] < grid_size and 0 <= player_start[1] < grid_size):
        raise ValueError(f"Invalid player start position {player_start} for grid size {grid_size}")
    self.start_pos = (player_start[0], player_start[1])
    self.door_pos = (door_pos[0], door_pos[1])
    self.item_pos = (item_pos[0], item_pos[1])
    self.puzzle_pos = (puzzle_pos[0], puzzle_pos[1])
    self.max_turns = max_turns_for_this_room
    self.has_item = has_item
    self.solved_puzzle = solved_puzzle


    self.start_state = State(self.start_pos[0], self.start_pos[1], self.solved_puzzle, self.has_item, self.solved_puzzle)

    self.goal_state_condition = lambda state: state.x == self.door_pos[0] and state.y == self.door_pos[1] and state.has_key
```

Heuristic Function

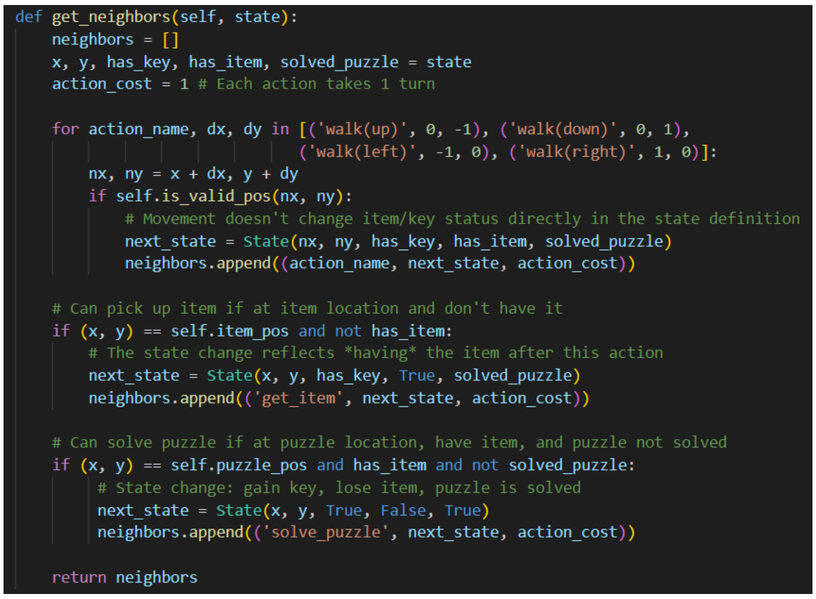
The heuristic function $h(n)$ calculates the estimated Manhattan distance to the goal. Its logic dynamically changes depending on whether the player has the key (state.has_key) or the item (state.has_item), guiding the search towards necessary sub-goals (item, puzzle) before heading to the door.



```python
# La heuristica estima el costo restante hasta el objetivo
def heuristic(self, state):
    current_pos = (state.x, state.y)

    if state.has_key:
        return self.manhattan_distance(current_pos, self.door_pos)

    elif state.has_item:
        dist_to_puzzle = self.manhattan_distance(current_pos, self.puzzle_pos)
        dist_puzzle_to_door = self.manhattan_distance(self.puzzle_pos, self.door_pos)
        return dist_to_puzzle + dist_puzzle_to_door

    else:
        dist_to_item = self.manhattan_distance(current_pos, self.item_pos)
        dist_item_to_puzzle = self.manhattan_distance(self.item_pos, self.puzzle_pos)
        dist_puzzle_to_door = self.manhattan_distance(self.puzzle_pos, self.door_pos)
        return dist_to_item + dist_item_to_puzzle + dist_puzzle_to_door
```

Neighbor Generation

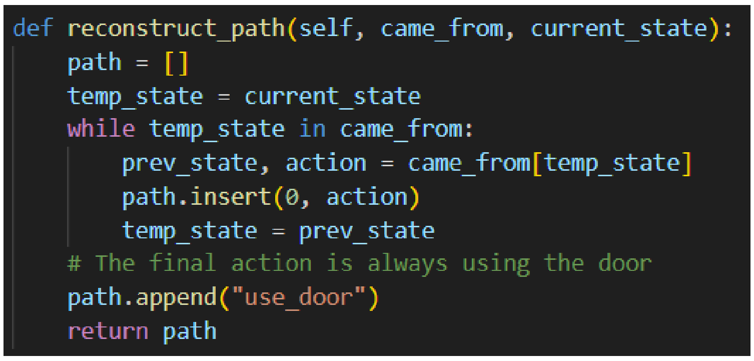
This method calculates all possible successor states (neighbors) from a given state. It considers valid movements ('walk') and interaction actions ('get_item', 'solve_puzzle') that change the player state (inventory/key). Each generated neighbor corresponds to an action costing 1 turn.



```python
def get_neighbors(self, state):
    neighbors = []
    x, y, has_key, has_item, solved_puzzle = state
    action_cost = 1 # Each action takes 1 turn

    for action_name, dx, dy in [('walk(up)', 0, -1), ('walk(down)', 0, 1),
                                ('walk(left)', -1, 0), ('walk(right)', 1, 0)]:
        nx, ny = x + dx, y + dy
        if self.is_valid_pos(nx, ny):
            # Movement doesn't change item/key status directly in the state definition
            next_state = State(nx, ny, has_key, has_item, solved_puzzle)
            neighbors.append((action_name, next_state, action_cost))

    # Can pick up item if at item location and don't have it
    if (x, y) == self.item_pos and not has_item:
        # The state change reflects *having* the item after this action
        next_state = State(x, y, has_key, True, solved_puzzle)
        neighbors.append(('get_item', next_state, action_cost))

    # Can solve puzzle if at puzzle location, have item, and puzzle not solved
    if (x, y) == self.puzzle_pos and has_item and not solved_puzzle:
         # State change: gain key, lose item, puzzle is solved
         next_state = State(x, y, True, False, True)
         neighbors.append(('solve_puzzle', next_state, action_cost))

    return neighbors
```

Path Reconstruction

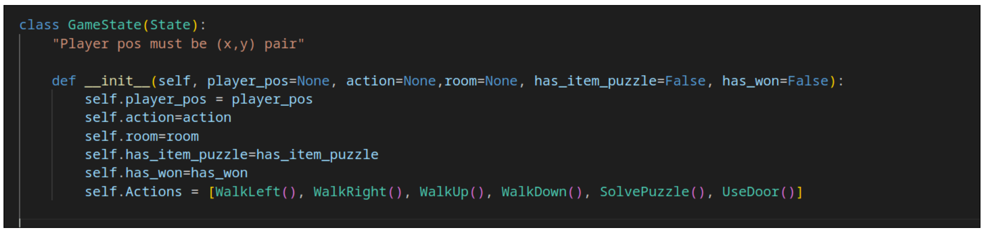
Once A* reaches the goal, this function uses the came_from dictionary to backtrack from the final state to the start, constructing the sequence of actions (path) in the correct order. It appends the final 'use_door' action.



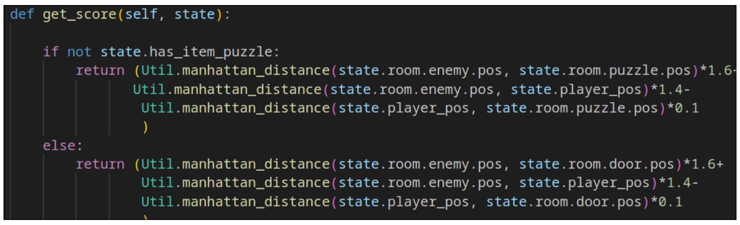
```python
def reconstruct_path(self, came_from, current_state):
    path = []
    temp_state = current_state
    while temp_state in came_from:
        prev_state, action = came_from[temp_state]
        path.insert(0, action)
        temp_state = prev_state
    # The final action is always using the door
    path.append("use_door")
    return path
```

A star and minimax

Creating representation for each game state made it easier to compare states and track progress.



```python
class GameState(State):
    "Player pos must be (x,y) pair"

    def __init__(self, player_pos=None, action=None,room=None, has_item_puzzle=False, has_won=False):
        self.player_pos = player_pos
        self.action=action
        self.room=room
        self.has_item_puzzle=has_item_puzzle
        self.has_won=has_won
        self.Actions = [WalkLeft(), WalkRight(), WalkUp(), WalkDown(), SolvePuzzle(), UseDoor()]
```



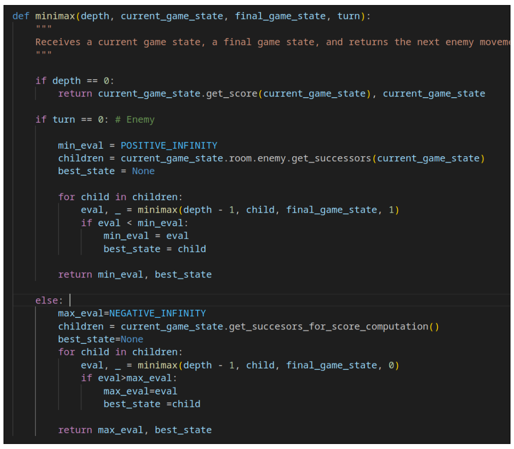
```python
def get_score(self, state):

    if not state.has_item_puzzle:
        return (Util.manhattan_distance(state.room.enemy.pos, state.room.puzzle.pos)*1.6-
                   Util.manhattan_distance(state.room.enemy.pos, state.player_pos)*1.4-
                    Util.manhattan_distance(state.player_pos, state.room.puzzle.pos)*0.1
                    )
    else:
        return (Util.manhattan_distance(state.room.enemy.pos, state.room.door.pos)*1.6+
                    Util.manhattan_distance(state.room.enemy.pos, state.player_pos)*1.4-
                    Util.manhattan_distance(state.player_pos, state.room.door.pos)*0.1
```

The game state also includes a scoring function, which provides a numerical evaluation of the situation. The score plays a crucial role in guiding the minimax algorithm by helping it determine the most promising moves based on the player's and the opponent's positions.

The get_score` function checks how good or bad a game state is for the enemy by looking at the distances between the player, the enemy, and important spots in the room. A lower score means a better situation for the enemy.

If the puzzle item hasn't been picked up yet, the score mainly depends on how close the enemy is to the puzzle (so it can protect it) and to the player (to keep pressure on them). The puzzle is given more weight. On top of that, a small part of the distance from the player to the puzzle is subtracted, so the player is slightly rewarded for getting closer to it.

After the puzzle is collected, the focus changes. Now the score depends on how close the enemy is to the door and to the player. Again, the function slightly lowers the score if the player is near the door, since that means they're getting closer to escaping. The enemy tries to keep the score low, while the player wants it to go up. This keeps both sides with opposite goals.

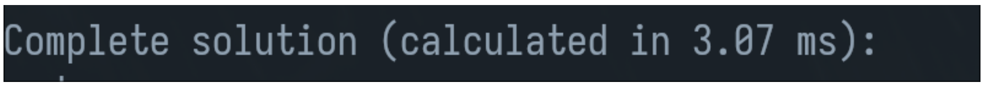


```python
def minimax(depth, current_game_state, final_game_state, turn):
    """
    Receives a current game state, a final game state, and returns the next enemy moveme
    """

    if depth == 0:
        return current_game_state.get_score(current_game_state), current_game_state

    if turn == 0: # Enemy

        min_eval = POSITIVE_INFINITY
        children = current_game_state.room.enemy.get_successors(current_game_state)
        best_state = None

        for child in children:
            eval, _ = minimax(depth - 1, child, final_game_state, 1)
            if eval < min_eval:
                min_eval = eval
                best_state = child

        return min_eval, best_state

    else: 
        max_eval=NEGATIVE_INFINITY
        children = current_game_state.get_succesors_for_score_computation()
        best_state=None
        for child in children:
            eval, _ = minimax(depth - 1, child, final_game_state, 0)
            if eval>max_eval:
                max_eval=eval
                best_state =child

        return max_eval, best_state
```

The minimax function simulates several turns ahead to decide the enemy's next move. It alternates between the enemy (who wants to minimize the score) and the player (who wants to maximize it). At each depth level, it explores possible future game states and picks the one that leads to the best outcome for the current turn. The process stops when the depth limit is reached, and the score of the state is returned. The enemy's children are states based on its walking actions, and the player's children are states based on the player's walking actions.

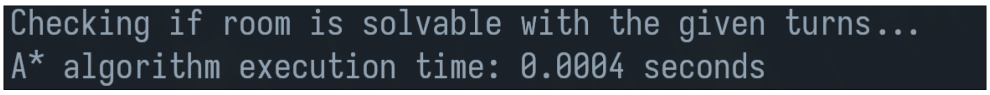
RESULTS AND COMPARISONS

Prolog



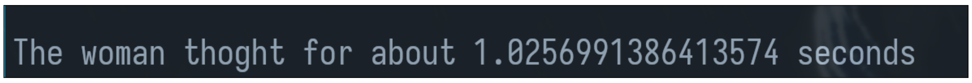
```
Complete solution (calculated in 3.07 ms):
```

A star



```
Checking if room is solvable with the given turns...
A* algorithm execution time: 0.0004 seconds
```

A star and minimax



```
The woman thoght for about 1.0256991386413574 seconds
```

The game version that includes the A star algorithm developed in python had the best performance. Then, there is the prolog approach and finally we have the minimax version.The Prolog implementation is slow because Prolog is designed to be good at declarative reasoning and logical deduction. Since our solution does not fully exploit these strengths and relies instead on intensive state exploration, it is more appropriate to use other languages better suited for this type of problem.

On the other hand, the fact that the third version took into consideration an ‘intelligent’ enemy produced a significant time increase. The cost of computing the shortest path is higher because every player movement also produces an enemy response that must be computed using minimax.

DISCUSSIONS

Compared to the original A* and Minimax strategies described in the report, this Prolog-based system represents a more declarative and efficient approach. While it may lack advanced optimization techniques, it is highly performant, scalable, and easier to reason about due to its rule-based nature. The decision to rely on state progression rather than global evaluation functions is especially suited for deterministic puzzles where player and environment behavior can be tightly controlled.

It was desirable to maintain the shortest path requesting functionality while using the minimax algorithm for enemy decision-making. One approach could have been to block the enemy's shortest path to the goal, but this could create a situation where the shortest path itself depends on the enemy's position, and the minimax algorithm depends on the player's movement. This circular dependency could lead to an infinite loop where the algorithm keeps recalculating without making progress. To avoid this, we opted for a simpler approach by creating a score function that doesn't block the enemy's shortest path, preventing unnecessary complexity and potential performance issues. This highlights the challenges in integrating purely optimal pathfinding (like A*) with adversarial search (like Minimax).
The chosen A* algorithm itself proved highly effective for the player's path calculation due to its efficiency and optimality guarantees, especially when combined with a state-dependent heuristic that adapted to the player having the item or key. The significant difference in execution time observed between the pure A* version and the Minimax version underscores the substantial computational cost introduced by simulating an 'intelligent' opponent turn by turn. However, at the end of the project, we also realized that it was possible to improve the performance of the Minimax component even more by saving (memoizing) minimax returns in memory, potentially mitigating some of this cost.

CONCLUSIONS

Sometimes, using less accurate but computationally cheaper functions, like the developed scoring function for the Minimax enemy, is worth it. It allows the system to run efficiently without overloading resources, even if it means sacrificing some precision or true predictive capability in decision-making. Balancing this tradeoff ensures smooth performance without unnecessary complexity. This project confirmed that for the specific task of optimal pathfinding with state dependencies (like inventory), the A* algorithm provides an excellent and highly efficient solution. A key takeaway was the importance of comprehensive state representation; simply tracking coordinates is insufficient when actions and goals depend on items or keys. Furthermore, while integrating different AI techniques offers richer gameplay, it necessitates careful design to manage interactions and performance impacts. Looking ahead, implementing memoization for the Minimax calculations presents a clear path to performance improvement for the adversarial version of the game.