Department of Computer Science
Technical University of Cluj-Napoca

# Artificial Intelligence

*Laboratory activity*

Name: Olaru Sebastian, Semerian Bogdan
Group: 30431
Email: sebyolaru464@gmail.com

Teaching Assistant: Adrian Groza
Adrian.Groza@cs.utcluj.ro

# Contents

# Chapter 1

# A1: Search

## 1.1 Dijkstra's Algorithm in Python

In this section, we present the Python implementation of Dijkstra's algorithm for solving a given problem. The algorithm prioritizes nodes based on their total cost and explores the node with the lowest cost first.

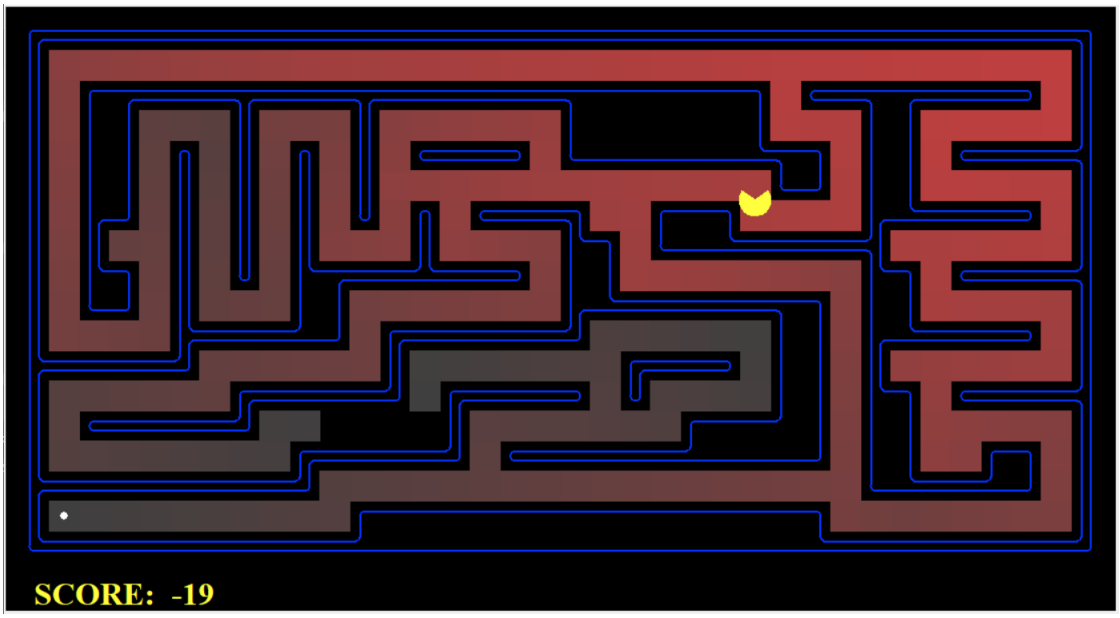


Figure 1.1: Dijkstra applied on medium Maze

## 1.2 Corner Heuristic

The `cornerHeuristic` is a heuristic function used in pathfinding algorithms, particularly for maze-solving problems. It estimates the cost from the current state to the nearest goal state based on the corners of the maze.

The function takes two arguments: `state` representing the current position and `problem` which is optional and assumed to have a `getCorners()` method that returns a list of corner positions.

If the current position is already at a corner, the function returns a cost of 0, indicating that no additional cost is needed to reach a corner.

Otherwise, the function calculates the Manhattan distance to the nearest corner. It iterates over the list of corners and computes the distance as the sum of the absolute differences in x and y coordinates.

The minimum distance found is returned as the heuristic value.

## 1.3 eatingAllFoodHeuristic

The `eatingAllFoodHeuristic` is a heuristic function used in pathfinding algorithms, particularly for Pac-Man games. It estimates the cost from the current state to the nearest goal state based on the objective of eating all the food in the maze.

The function takes two arguments: `state` representing the current position and `problem` which is optional and assumed to have a `getFood()` method that returns a grid with food positions.

If there is no remaining food, indicating that the goal has been reached, the function returns a cost of 0.

Otherwise, the function calculates the sum of Manhattan distances from the current position to all remaining food pellets. It iterates over the list of remaining food and computes the Manhattan distance for each. The total distance is returned as the heuristic value.

## 1.4 Manhattan Distance Heuristic

The `manhattanDistance` heuristic is a distance-based heuristic used in pathfinding algorithms. It estimates the cost from the current state to the goal state based on the Manhattan distance.

The function takes two arguments: `state` representing the current state and `problem` which is optional and assumed to have a `goal` attribute representing the goal state.

The function assumes that both the current state and the goal state are represented as square grids. It converts the 1D representation of states into 2D grids for easier calculation.

The total distance is initialized to 0. The function then iterates over each tile in the grid, skipping the empty tile, and calculates the Manhattan distance between the current position and its goal position.

The Manhattan distances for all tiles are summed up and returned as the heuristic value.

## 1.5 Chaos Maze Layout

The ‘chaosMaze‘ layout is a complex maze filled with obstacles, points, and ghosts. The layout presents a challenging environment for our Pac-Man agent. The goal of the agent in this maze is to navigate through the maze, capture points represented by ’%’, and avoid ghosts represented by ’G’. The ’P’ marks the starting position of our Pac-Man agent.

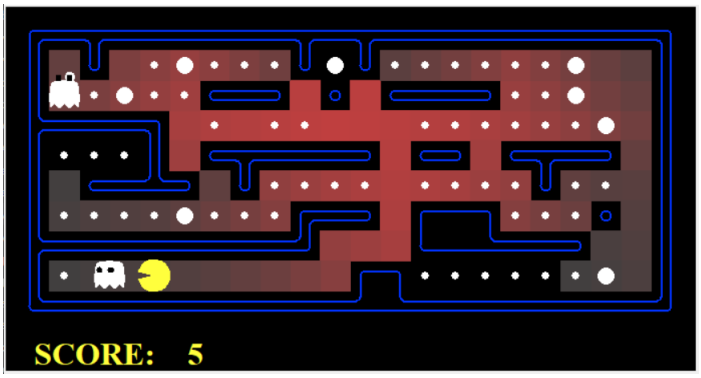


Figure 1.2: chaosMaze

## 1.6 Escape Room Layout

The ‘escapeRoom‘ layout is a simple room with a narrow passage leading to certain points. It serves as a proving ground for our jumping algorithm. The agent needs to demonstrate its ability to navigate through tight spaces and reach all the target points.

# Chapter 2

# A2: Logics

## Overview

This Python script generates a logic configuration for a Minesweeper game with First Order Logic (FOL) hints. It ensures each hint aligns with the actual placement of mines on a grid. The script outputs a JavaScript file (`items.js`) compatible with a Minesweeper game implementation in JavaScript.

## Features

- **Grid Initialization:** Sets up a Minesweeper grid of a specified size.
- **Random Mine Placement:** Randomly distributes a specified number of mines across the grid.
- **Diverse Hint Generation:** Generates various types of FOL hints based on the real state of the Minesweeper grid.
- **Verified Hints:** Each hint is checked for accuracy against the mine configuration on the grid.
- **Export to JavaScript File:** Outputs the mine and hint configuration in a format suitable for import into a JavaScript-based Minesweeper game.

## Key Functions

- `is_mine(grid, x, y)`:
  - **Purpose:** Checks if a given cell contains a mine.
  - **Parameters:** `grid` (the Minesweeper grid), `x`, `y` (coordinates of the cell).
  - **Returns:** Boolean value (`True` if there's a mine, `False` otherwise).
- `count_adjacent_mines(grid, x, y)`:
  - **Purpose:** Counts mines adjacent to a specified cell.
  - **Parameters:** `grid` (the Minesweeper grid), `x`, `y` (coordinates of the cell).
  - **Returns:** Integer count of adjacent mines.
- `all_diagonal_mines(grid, x, y)`:

  - **Purpose:** Checks if all diagonal cells relative to a given cell contain mines.
  - **Parameters:** `grid` (the Minesweeper grid), `x`, `y` (coordinates of the central cell).
  - **Returns:** Boolean value (`True` if all diagonal cells have mines, `False` otherwise).

- `generate_minesweeper_logic(grid_size, num_mines, num_hints)`:
  - **Purpose:** Generates the Minesweeper game logic including mine placement and hints.
  - **Parameters:** `grid_size` (size of the Minesweeper grid, NxN), `num_mines` (number of mines), `num_hints` (number of hints).
  - **Returns:** A list of dictionaries, each representing a mine or a hint with its coordinates, type, and value.

- `save_to_js_file(items, filename)`:
  - **Purpose:** Saves the generated game logic to a JavaScript file.
  - **Parameters:** `items` (list of mines and hints), `filename` (name of the output file).
  - **Output:** A JavaScript file (`items.js`) containing the game logic.

# Example Usage

```
grid_size = 8
num_mines = 10
num_hints = 8  # Adjust based on desired hint frequency
items = generate_minesweeper_logic(grid_size, num_mines, num_hints)
save_to_js_file(items)
```

# Output Format

The output `items.js` contains an array of objects, each representing either a mine or a hint. Each object has `coordinates`, `type`, and `value` (or `hint`).

# Customization

- The grid size, number of mines, and number of hints can be adjusted as needed.
- Additional hint types can be added by extending the `generate_minesweeper_logic` function.

# Dependencies

- Python Standard Library.
- No external libraries are required for this script.

# Limitations

- The script currently supports a basic set of hint types. Complex logic patterns might require additional development.
- The randomness in mine placement and hint generation may lead to varying levels of game difficulty.

# Chapter 3

# A3: Planning

# Bibliography

# Appendix A

# Your original code

Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more. This section should contain only code developed by you, without any line re-used from other sources. This section helps me to correctly evaluate your amount of work and results obtained.

Listing A.1: Dijkstra's Algorithm

```
def dijkstra(problem):
    """Search the node that has the lowest total cost first."""
    from util import PriorityQueue

    que = PriorityQueue()
    que.push((problem.getStartState(), []), 0)

    visited = set()

    while not que.isEmpty():
        currentState, path = que.pop()

        if currentState in visited:
            continue

        visited.add(currentState)

        if problem.isGoalState(currentState):
            return path

        successors = problem.getSuccessors(currentState)
        for nextPosition, direction, cost in successors:
            if nextPosition not in visited:
                totalCost = problem.getCostOfActions(path + [direction])
                que.push((nextPosition, path + [direction]), totalCost)

    return []  # Return an empty list if no path is found
```

Listing A.2: Corner Heuristic

```
def cornerHeuristic(state, problem=None):
    """
    A heuristic function that estimates the cost from the current state
    to the nearest goal state based on the corners of the maze.
```

```
    """
    corners = (
        problem.getCorners()
    )  # Assuming getCorners() returns a list of corner positions
    current_position = state

    if current_position in corners:
        return 0  # Already at a corner, no additional cost

    # Calculate the Manhattan distance to the nearest corner
    min_distance = float("inf")
    for corner in corners:
        distance = abs(corner[0] - current_position[0]) + abs(
            corner[1] - current_position[1]
        )
        if distance < min_distance:
            min_distance = distance

    return min_distance
```

Listing A.3: Eating All Food Heuristic

```
def eatingAllFoodHeuristic(state, problem=None):
    """
    A heuristic function that estimates the cost from the current state
    to the
    nearest goal state based on eating all the food in the maze.
    """
    remaining_food = (
        problem.getFood().asList()
    )  # Assuming getFood() returns a grid with food positions
    current_position = state

    if not remaining_food:
        return 0  # No remaining food, already at the goal

    # Calculate the sum of Manhattan distances to all remaining
    food pellets
    total_distance = sum(
        abs(food[0] - current_position[0]) + abs(food[1] -
        current_position[1])
        for food in remaining_food
    )

    return total_distance
```

Listing A.4: Manhattan Distance Heuristic

```
def manhattanDistance(state, problem=None):
    """
    A heuristic function that estimates the cost from the current state to th
    goal state based on the Manhattan distance.
```

```
    """
    current_state = state
    goal_state = problem.goal

    size = int(len(current_state) ** 0.5)  # Assuming the state is a square g

    # Convert the 1D state into a 2D grid for easier calculation
    current_state = [current_state[i:i+size] for i in range(0, len(current_st
    goal_state = [goal_state[i:i+size] for i in range(0, len(goal_state), siz

    total_distance = 0

    for i in range(size):
        for j in range(size):
            if current_state[i][j] != 0:  # Skip the empty tile
                value = current_state[i][j]

                # Find the goal position of the tile
                goal_i, goal_j = divmod(goal_state.index(value), size)

                # Calculate Manhattan distance
                total_distance += abs(i - goal_i) + abs(j - goal_j)

    return total_distance
```

```
%%%%%%%%%%%%%%%%%%%%%%%%
% % .o...%o%......o  %
%..o..%%% % %%%%..o  %
%%%%% .G..P  ......o %
%...% %%%%%% %% %%%% %
% %%%% %....o....%.. %
%....o...%%% %%%...% %
%%%%%%%%%%G..%%%%%%  %
%.          %%......o %
%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%
%P                     .%
%%%%%%%%%%%%%%%%%%%%%%%%
% %      %            % %
%.%      % .          % %
% %      %            % %
%%%%%%%%%%%%%%%%%%%%%%%% %
%.                     %.%
%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing A.5: Generator for FOL games

```
import random
import json
```

```
def is_mine(grid, x, y):
    return 0 <= x < len(grid) and 0 <= y < len(grid) and grid[x][y] == "M"


def count_adjacent_mines(grid, x, y):
    count = 0
    for dx in range(-1, 2):
        for dy in range(-1, 2):
            if dx == 0 and dy == 0:
                continue
            if is_mine(grid, x + dx, y + dy):
                count += 1
    return count


def all_diagonal_mines(grid, x, y):
    for dx in [-1, 1]:
        for dy in [-1, 1]:
            if not is_mine(grid, x + dx, y + dy):
                return False
    return True


def generate_minesweeper_logic(grid_size, num_mines, num_hints):
    # Initialize the grid
    grid = [["-" for _ in range(grid_size)] for _ in range(grid_size)]
    mines = set()

    # Place mines randomly
    while len(mines) < num_mines:
        x = random.randint(0, grid_size - 1)
        y = random.randint(0, grid_size - 1)
        if (x, y) not in mines:
            mines.add((x, y))
            grid[x][y] = "M"

    items = []

    # Add mines to items
    for x, y in mines:
        items.append(
            {
                "coordinates": {"x": x + 1, "y": y + 1},
                "type": "MINE",
                "value": "asd",
            }
        )

    # Always add a hint at the first position (0,0)
```

```
if not is_mine(grid, 0, 0):
    hint = f"-mine(1,1)"
    items.append(
        {
            "coordinates": {"x": 1, "y": 1},
            "type": "HINT",
            "value": hint,
            "hint": hint,
        }
    )

# Generate additional valid FOL hints
hints_generated = 1  # Start with 1 due to the initial hint at (0,0)
hint_types = [
    "no_mines_row",
    "no_mines_col",
    "specific_mine",
    "no_adjacent_mines",
    "adjacent_mine_count",
    "all_diagonal_mines",
]
while hints_generated < num_hints:
    x, y = random.randint(0, grid_size - 1), random.randint(0, grid_size

    if grid[x][y] == "M" or (x == 0 and y == 0):
        continue  # Skip mine cells and the first cell

    hint_type = random.choice(hint_types)

    if hint_type == "no_mines_row" and all(
        grid[row][y] != "M" for row in range(grid_size)
    ):
        hint = f"all-x-mine(x,{y-+-1})"
    elif hint_type == "no_mines_col" and all(
        grid[x][col] != "M" for col in range(grid_size)
    ):
        hint = f"all-y-mine({x-+-1},y)"
    elif hint_type == "specific_mine":
        if is_mine(grid, x, y):
            continue
        hint = f"-mine({x-+-1},{y-+-1})"
    elif hint_type == "no_adjacent_mines":
        if any(
            is_mine(grid, x + dx, y + dy)
            for dx in range(-1, 2)
            for dy in range(-1, 2)
            if dx != 0 or dy != 0
        ):
            continue
        hint = f"all-dx-dy-(-mine({x-+-1}+dx,-{y-+-1}+dy)-&-!(dx-=-0-&-dy
```

```
        elif hint_type == "adjacent_mine_count":
            adjacent_mines = count_adjacent_mines(grid, x, y)
            hint = f"sum adjacent mine({x + 1},{y + 1}) = {adjacent_mines}"
        elif hint_type == "all_diagonal_mines":
            if not all_diagonal_mines(grid, x, y):
                continue
            hint = f"all diagonal mine({x + 1},{y + 1})"
        else:
            continue

        items.append(
            {
                "coordinates": {"x": x + 1, "y": y + 1},
                "type": "HINT",
                "value": hint,
                "hint": hint,
            }
        )
        hints_generated += 1

    return items


def save_to_js_file(items, filename="items.js"):
    with open(filename, "w") as file:
        file.write("module.exports = Object.freeze(")
        file.write(json.dumps(items, indent=2))
        file.write(");")


# Example usage
grid_size = 8
num_mines = 3
num_hints = 5  # Adjust based on desired hint frequency
items = generate_minesweeper_logic(grid_size, num_mines, num_hints)
save_to_js_file(items)
```