[DT0171] ARTIFICIAL INTELLIGENCE 2024/2025

HOMEWORK 1: Heuristic Search

Student: Tien Dung Nguyen – 301142 – tiendung.nguyen@student.univaq.it

1. Implementation

a. Agent module

The **Agent** module mimics a pathfinding agent designed to navigate a grid-based board. The agent uses various algorithms to solve path-planning problem from a starting point $(0,0)$ to an endpoint $(n-1, n-1)$, avoiding obstacles along the way. The **Agent** module contains some attributes with information about an agent: state, possible moves and solution. It also has some methods, namely:

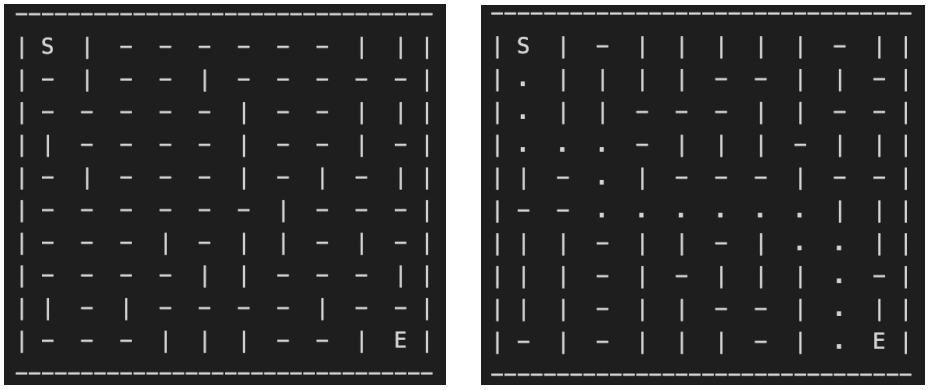
- **reset_solution:** Reset agent solution to an empty state
- **heuristic:** Calculate the heuristic value based on the given **option** parameter, which can be used to select heuristic function of choice. Some heuristic functions implemented are:
  - Euclidean distance from current state to final state, defined by:
    $$h(x,y) = \sqrt{(x-(n-1))^2 + (y-(n-1))^2}$$
  - Manhattan distance from current state to final state, defined by:
    $$h(x,y) = |x-(n-1)| + |y-(n-1)|$$
  - Minus of the distance to the nearest obstacle from a specific direction and sum of the direction coordinator, defined by:
    $$h(x,y,\vec{n}) = -(|x - x_{(x,y),\vec{n}}| + |y - y_{(x,y),\vec{n}}| + x_{\vec{n}} + y_{\vec{n}})$$
    where $(x_{(x,y),\vec{n}}, y_{(x,y),\vec{n}})$ is the closet obstacle from (x,y) in direction $\vec{n}(x_{\vec{n}}, y_{\vec{n}})$.
    This guarantees that priority is for the direction that is furtherst from an obstacle and the direction that the agent move is closer to the ending point
- **mark_solution:** Mark solution path on the game board

b. Game module

The **Game** module models a grid-based environment for pathfinding algorithms. It includes functionality for defining the grid, generate obstacles, validating paths, and visualizing the game board. The **Game** module contains information about size of the grid, starting point, ending point, the grid representation and list of obstacles. It also contains some methods:

- **clear_board:** Resets all grid cell, including obstacles, keeps only the start and end points intact

- **reset_board:** Clears the paths marked while preserving obstacles, start, and end points
- **mark_solution:** Updates the grid with a solution path
- **generate_obstacles:** Generates a specified number of obstacles randomly on the grid while ensuring the obstacles do not overwrite the start or end points and a valid path exists between the start and end points
- **is_path_exists:** Uses Breadth-First Search (BFS) to check if a path exists between the start and end points
- **visualize:** Prints the grid to the console in a readable format
  Example of a generated grid with obstacles (l) and a grid with solution (.)



(S: starting point, E: ending point, - : feasible point)

c. Algorithms

- **Breadth-first search:** From the starting point, explore all neighbors at the current depth before moving to the next depth level. This algorithm guarantees the shortest path in an unweighted grid but is nefficient in large grids due to exploring many unnecessary nodes.
- **Depth-first search:** From the starting point, explore as far as possible before backtracking. This algorithm can be more memory efficient than BFS but does not guarantee the shortest path.
- **Best-first search:** From the starting point, explore neighbor nodes based on the heuristic function. It is implemented using a priority queue.
- **A*:** A* combines the advantages of BFS (guaranteeing the shortest path) and Best-First Search (using a heuristic for efficiency). It uses a priority queue where the priority of a node is determined by the sum of:
  - **Cost so far (g):** The actual distance from the start node to the current node
  - **Heuristic (h):** The estimated cost to the goal

$$f(x, y) = g(x, y) + h(x, y)$$

where $h(x, y)$ can be the heuristic function of choice.

2. Statistics

a. Test case description

For this experiment, 10 different test cases will be generated for each of the obstacle percentage defined in the list: [10, 20, 30, 40, 50]

```
MAP_SIZE = 10
NUM_OF_SAMPLES = 10
PERCENTAGE = [10, 20, 30, 40, 50]
game_list = {}
for percentage in PERCENTAGE:
    games = []
    for i in range(NUM_OF_SAMPLES):
        random.seed(str(percentage) + str(i))
        game = Game(MAP_SIZE)
        game.generate_obstacles(int(percentage / 100 * MAP_SIZE * MAP_SIZE))
        games.append(game)
    game_list[percentage] = games
```

[686] ✓ 0.0s Python

Defined criteria for statistics are:

- Average path length found by the algorithm
- Average runtime of the algorithm

All algorithms will run all the test cases, then the average path length and average runtime will be calculated.

b. Algorithms configuration

In this experiment, the best-first search algorithm will use the distance to the closet obstacle from a direction to be the heuristic function. The same heuristic function is also used in A* algorithm.

c. Path length statistics

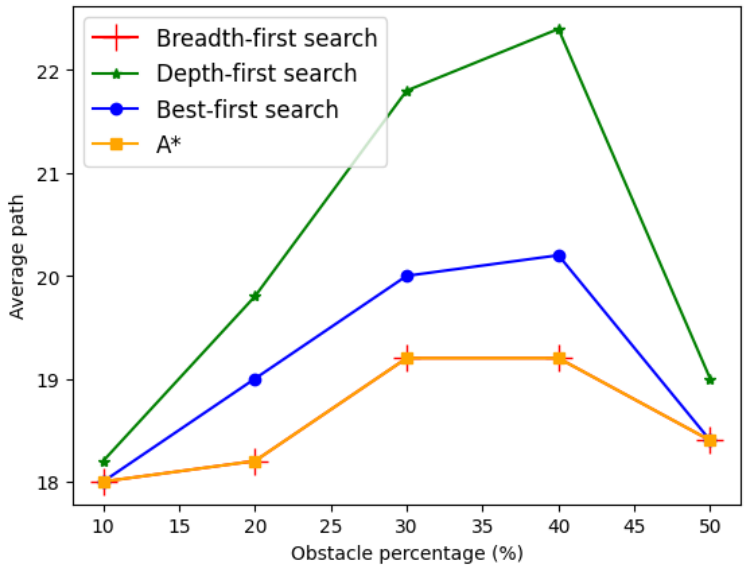


Figure 1. Average path found by different algorithms with respect to percentage of obstacle

As the graph represent, with 10% of area covered with obstacles, the difference between algorithms is not significant. However, as the number of obstacles increase, the Depth-first search shows a sharp peak compared to other algorithms, while best-first search found better path than DFS; A* and bread-first search produced the same average optimal path. Overall, BFS and A* demonstrate greater robustness in maintaining optimal or near-optimal paths across different obstacle densities. In the meantime, best-first serach with the selected heuristic function only use heuristic guidance, which can lead to non-optimal path.

d. Runtime statistics

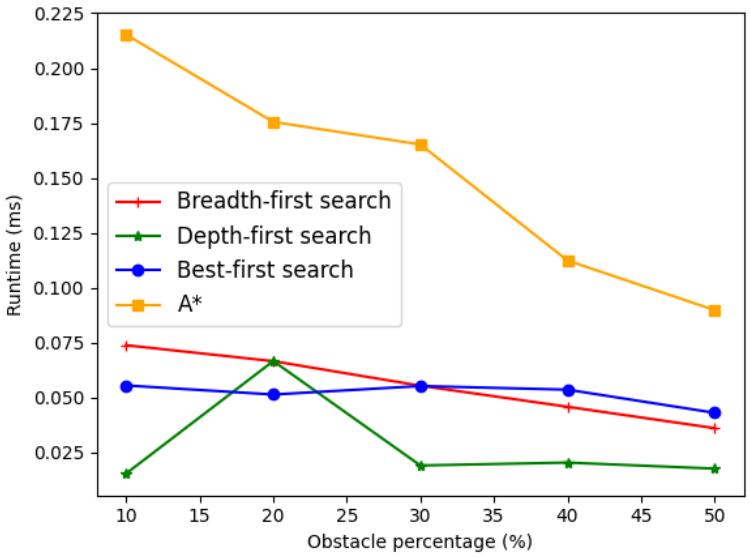


Figure 2. Average executed time by different algorithms with respect to percentage of obstacle

A noticeable trend from the graph is that as the percentage of obstacle increase, the runtime of algorithms tends to decrease because of narrow feasible search space. Except for the case where DFS likely gets "stuck" exploring inefficient paths of test cases that have 20% obstacle density. Best-first search has a lower runtime compared to A*, as it prioritizes heuristic evaluation without considering the full path cost. In average, uninformed search algorithms are faster due to their simplicity and lack of heuristic overhead, while informed search runtimes are higher, especially A* trades runtime efficiency for better path quality.