Roll the Block

Elements of Artificial Intelligence and Data Science

Ana Araújo
Margarida Fidalgo
Pedro Jorge

Introduction

Roll the Block is a puzzle game in which you have to move a rectangular **block** to reach the **destination** by rolling it on the platforms in the **least possible number of movements**. The game is **finished** when the **block falls** (touches a void tile).

There are several game elements that make it more challenging, such as the **glass floor** (in which the block can't stand upright) and **buttons that activate hidden paths**.

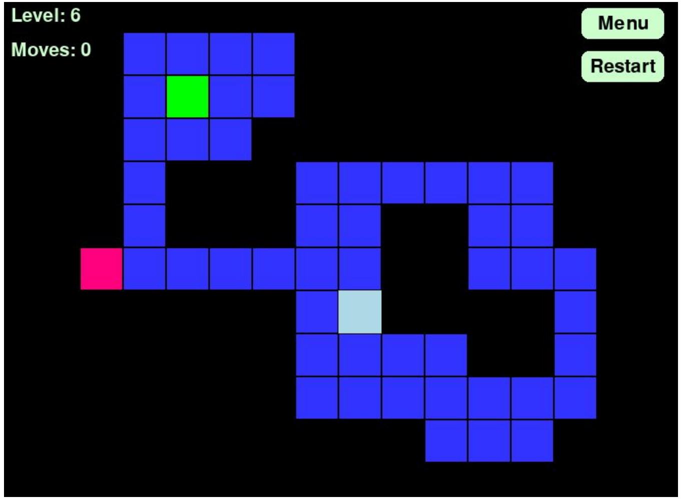


Fig.1 - Level 6 of Roll the Block

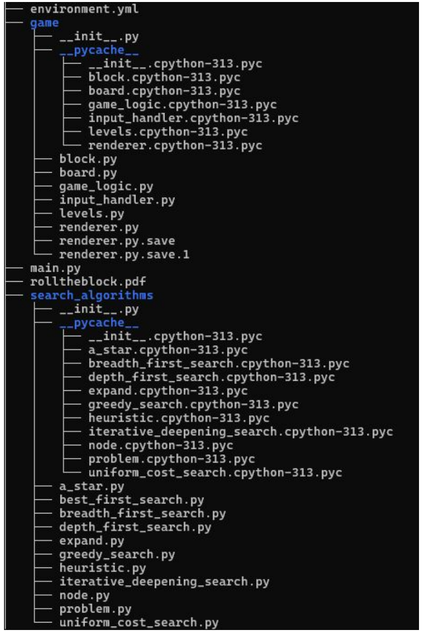
The game as a search problem

- **Initial state:** when the game starts, the block is always upright
- **Goal:** the game is finished when the block steps on the destination
- **Actions:** the block can move up, down, to the left and to the right
- **Possible states:** upright, vertical or horizontal
- **Preconditions:** the tiles to which the block is moving must exist
- **Cost:** the cost is always 1, and it is associated with the block's movement

Games's implementation

All of the game's code was **written modularly**—primarily to ease comprehension and readability, but also to preserve the organization and structure of each component.

Regarding the game's graphic design, all of it was implemented using **Pygame**. Furthermore, we created a **lightweight Anaconda environment** to uniformize and expedite the use of even more libraries: memory_profiler.



```
├── environment.yml
├── game
│   ├── __init__.py
│   ├── __pycache__
│   │   ├── __init__.cpython-313.pyc
│   │   ├── block.cpython-313.pyc
│   │   ├── board.cpython-313.pyc
│   │   ├── game_logic.cpython-313.pyc
│   │   ├── input_handler.cpython-313.pyc
│   │   ├── levels.cpython-313.pyc
│   │   └── renderer.cpython-313.pyc
│   ├── block.py
│   ├── board.py
│   ├── game_logic.py
│   ├── input_handler.py
│   ├── levels.py
│   ├── renderer.py
│   ├── renderer.py.save
│   └── renderer.py.save.1
├── main.py
├── rolltheblock.pdf
├── search_algorithms
│   ├── __init__.py
│   ├── __pycache__
│   │   ├── __init__.cpython-313.pyc
│   │   ├── a_star.cpython-313.pyc
│   │   ├── breadth_first_search.cpython-313.pyc
│   │   ├── depth_first_search.cpython-313.pyc
│   │   ├── expand.cpython-313.pyc
│   │   ├── greedy_search.cpython-313.pyc
│   │   ├── heuristic.cpython-313.pyc
│   │   ├── iterative_deepening_search.cpython-313.pyc
│   │   ├── node.cpython-313.pyc
│   │   ├── problem.cpython-313.pyc
│   │   └── uniform_cost_search.cpython-313.pyc
│   ├── a_star.py
│   ├── best_first_search.py
│   ├── breadth_first_search.py
│   ├── depth_first_search.py
│   ├── expand.py
│   ├── greedy_search.py
│   ├── heuristic.py
│   ├── iterative_deepening_search.py
│   ├── node.py
│   ├── problem.py
│   └── uniform_cost_search.py
```

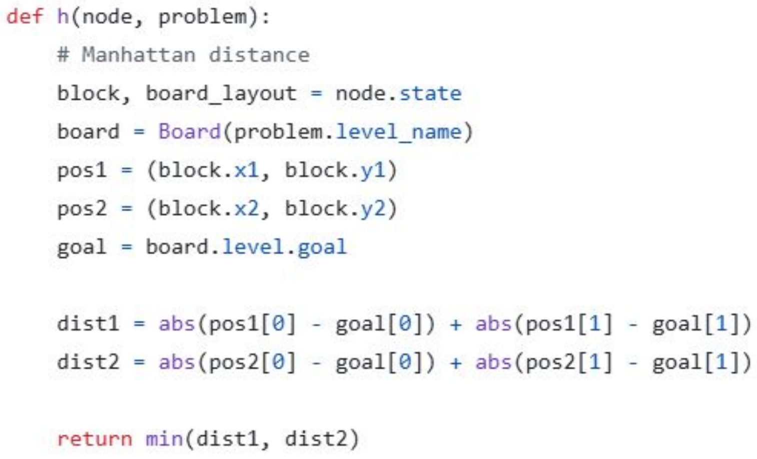
Fig.2 - Project structure

Implemented algorithms

We implemented the following **search algorithms**:

- A* (A-star)
- Breadth-first-search
- Depth-first-search
- Greedy-search
- Iterative deepening search
- Uniform cost search

The **heuristic** used on the **informed search algorithms** (A* e Greedy Search) was the **Manhattan's Distance between the block and the goal**.

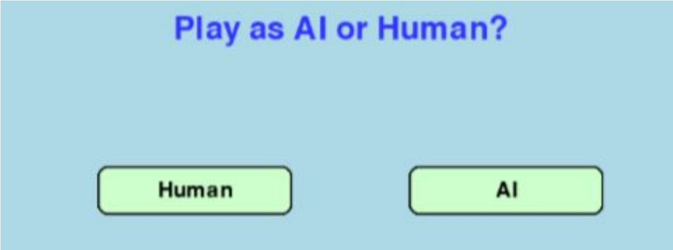


```
def h(node, problem):
    # Manhattan distance
    block, board_layout = node.state
    board = Board(problem.level_name)
    pos1 = (block.x1, block.y1)
    pos2 = (block.x2, block.y2)
    goal = board.level.goal

    dist1 = abs(pos1[0] - goal[0]) + abs(pos1[1] - goal[1])
    dist2 = abs(pos2[0] - goal[0]) + abs(pos2[1] - goal[1])

    return min(dist1, dist2)
```

Fig.3 - Heuristic cell

Search algorithms - Game

In order to **incorporate the search algorithms** into the game itself, an option was included to watch **each algorithm execute** all nine implemented levels, thus allowing the user to **analyse their performance**.



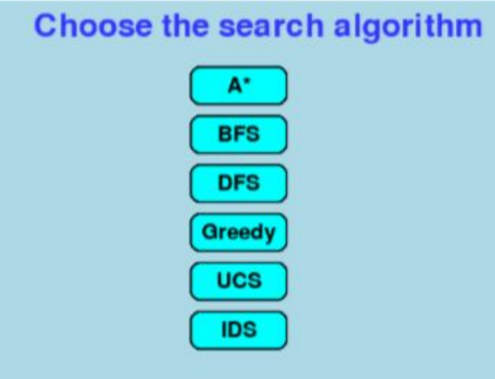


Fig.4 and Fig.5 - Search algorithms' implementation

Average Time Results

ms	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8	Level 9
A-star	22.384	98.429	379.889	574.891	149.729	405.930	791.622	168.138	1,260.092
BFS	49.161	89.751	362.418	630.450	132.208	395.391	759.786	156.505	1,111.486
DFS	52.869	102.809	340.867	643.147	117.661	359.067	840.746	150.765	801.749
Greedy	27.553	85.467	367.842	400.682	137.830	382.721	695.201	146.324	1,082.693
UCS	66.057	86.832	342.619	606.166	133.757	359.087	700.642	162.643	1,026.791
IDS	95.522	1,781.940	14,364.973	13,844.410	2078.238	8,020.108	24,841.639	3,917.418	43,231.887

Average Peak Memory Usage Results

MiB	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8	Level 9
A-star	149.321	151.172	151.404	151.074	150.624	153.938	156.148	151.030	155.142
BFS	152.164	151.538	151.504	154.320	150.492	153.064	152.550	151.904	154.078
DFS	149.512	151.026	152.626	155.090	149.756	150.384	155.094	154.702	155.960
Greedy	151.282	151.170	152.954	152.486	150.116	152.870	153.792	151.766	154.494
UCS	151.838	151.486	153.250	154.750	151.740	154.124	153.434	151.896	154.832
IDS	149.754	151.268	154.502	152.570	149.966	152.570	153.870	153.140	157.708

Conclusion

This project successfully combined the ***Roll the Block*** game with **search algorithms**, and we gained practical insights into both **game development and AI pathfinding**.

The hardest part was possibly the search algorithms' implementation into the game, since it required both Pygame skills and a deep understanding of each algorithm.

Webgraphy/Bibliography

- Book: Artificial Intelligence: A Modern Approach, 4th Edition, 1st Part - Chapter 3
- Pygame Tutorial: https://www.youtube.com/watch?v=AY9MnQ4x3zk&t=306s
- Claude: https://claude.ai/