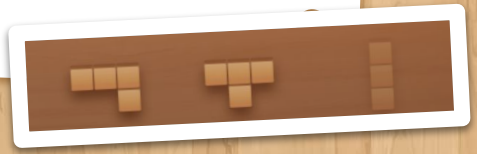
Game Description:

Wood Block Puzzle Game

Wood Block is a puzzle game where the main objective is to clear a specific number of 'target' blocks by creating a complete line or column. This allows players to progress to the next level!



Gabriel da Costa Filipe Carvalho, Guilherme Moura Pinho Andrade Silva e Valentina Pereira Cadime
Group_A1_30

Related Work

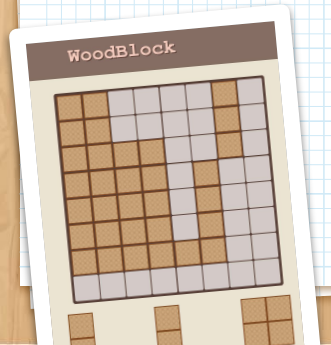
There are similar games, each one with different rules and game modes. However, since most of them are made to be played on the phone, we can't really uncover their code.

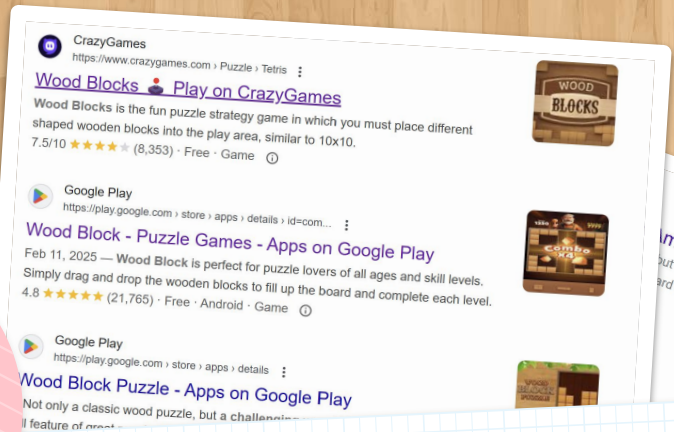
We did find a game using python though:

https://github.com/RedCuckoo/wood-block-puzzle/blob/main/main.py

Also in a single page of html:

https://github.com/hjvogel/websim-ai/blob/main/WoodBlock.html





This github explains what we or an AI should focus on for the wood block infinite gamemode:

https://github.com/gary-z/blokie

Look at all possible board states resulting from places the 3 pieces and choose the move that results in the best "board cleanliness" score. As always, the less blocks placed on board, the more options we can choose to pass to the next round of blocks.

State Representation: We chose to represent the problem at hand with this state

```
[
        [piece1, piece2, piece3],
        [[piece1, piece2, piece3], [piece1, piece2, piece3], ...],
        target_blocks,
        [[0, 1, 2, ...], [0, 1, 1, ...], [0, 1, 0, ....], ...],
        (recent_piece, (x, y))
]
```

```
# List containing the playable pieces the player can select
# List of lists, each containing the following pieces to be used after the player has placed all playable pieces
# Number of target blocks to clear (win condition)
# Matrix representing the board (0 indicates a blank space, 1 a normal block, and 2 a target block)
# Tuple containing the most recently placed piece and a tuple with its position
```

Piece Representation: Each piece is a list of (x, y) coordinates, where origin (0, 0) is the top-left corner of the piece

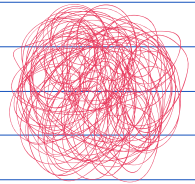
Ex.: [(0, 0), (1, 0), (0, 1), (1, 1)] #2x2 Square

Initial State Example: 4x4 board

```
[
        [[(0, 0)], [(0, 0), (1, 0), (0, 1), (1, 1)], [(0, 0), (1, 0), (2, 0), (3, 0)]],
        [[[(0, 0), (0, 1)], [(0, 0), (1, 0)], [(0, 0), (1, 0)]],  ...],
        4,
        [
         [0, 0, 0, 0],
         [0, 2, 2, 0],
         [0, 2, 2, 0],
         [0, 0, 0, 0]
        ],
        None
]
```

Objective State:

Any state where target_blocks is equal to or less than 0, regardless of the current state of the board.



Operator:

Name: place_piece
Precondition: selected piece must fit in the selected position
Cost: 1
Effect: places piece in the selected position (updating the **board** and **blocks_to_break** accordingly) and removes the piece from the selectable pieces list

Heuristic Evaluation:

We developed multiple heuristic functions, each tailored to meet the unique needs of its respective algorithm. The heuristic functions we created are:

Greedy heuristic: This heuristic evaluates the score that the current move will produce focusing on target block clearing, inheriting some of the score of its predecessor node

A* heuristic: This heuristic needed to be admissible, never overestimating the number of moves necessary to reach the optimal goal state, so it evaluates the current state of the board and returns the minimum number of lines/columns to clear

Infinite heuristic: This heuristic was made for the infinite gamemode where the objective is to survive the longest, so the heuristic analyzes the current state of the board and evaluates its level of “cleanliness”

Implemented Algorithms:

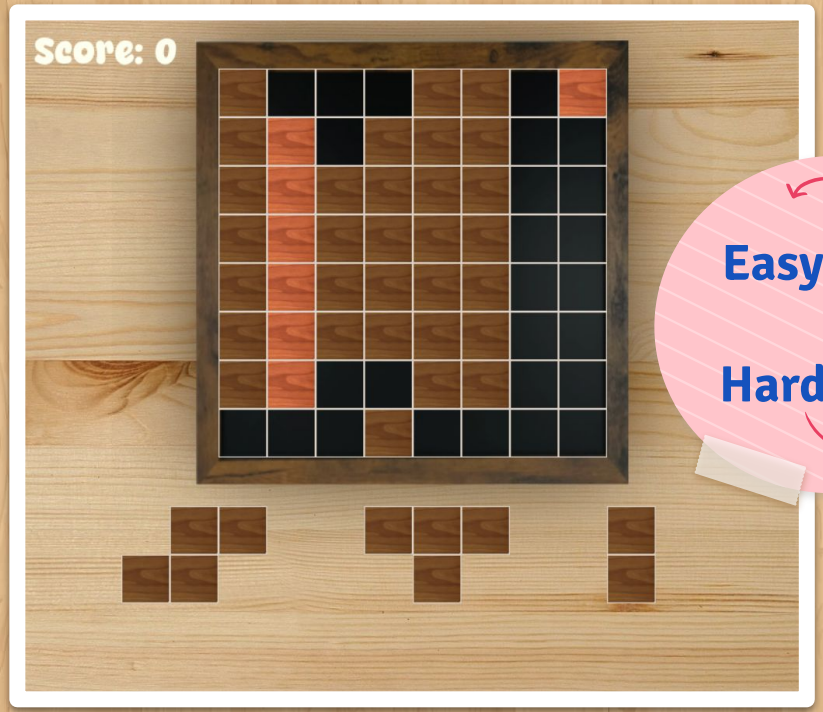
Where:
b is the branching factor
d is the depth of the solution
m is the maximum depth of the tree

We implemented a variety of algorithms throughout our project:

Implemented Algorithm	Time Complexity	Space Complexity	Complete	Optimal Cost	Reason to Implement
BFS	O(b^d)	O(b^d)	Yes	Yes	Simple algorithm, useful for comparison and guaranteed to find best solution
DFS	O(b^m)	O(b*m)	No	No	Simple algorithm, useful for comparison
Iter Deep (DFS)	O(b^d)	O(b*d)	Yes	Yes	A complete implementation of a DFS that computes the optimal solution
Greedy	O(b^m)**	O(b^m)	No*	No	We needed a fast and feasible solution for a problem of any size
A*	O(b^d)**	O(b^d)	Yes	Yes	Most complex algorithm, useful for comparison with simpler ones
Weighted A*	O(b^d)**	O(b^d)	Yes	No	Useful to understand the relevance of the admissibility of the a* heuristic when given higher priority

* According to slides. In our case when no more pieces can't be placed (and state is not a goal), the algorithm backtracks to find another path, so we can consider the algorithm to be complete.
** Heuristics greatly reduce the time complexity

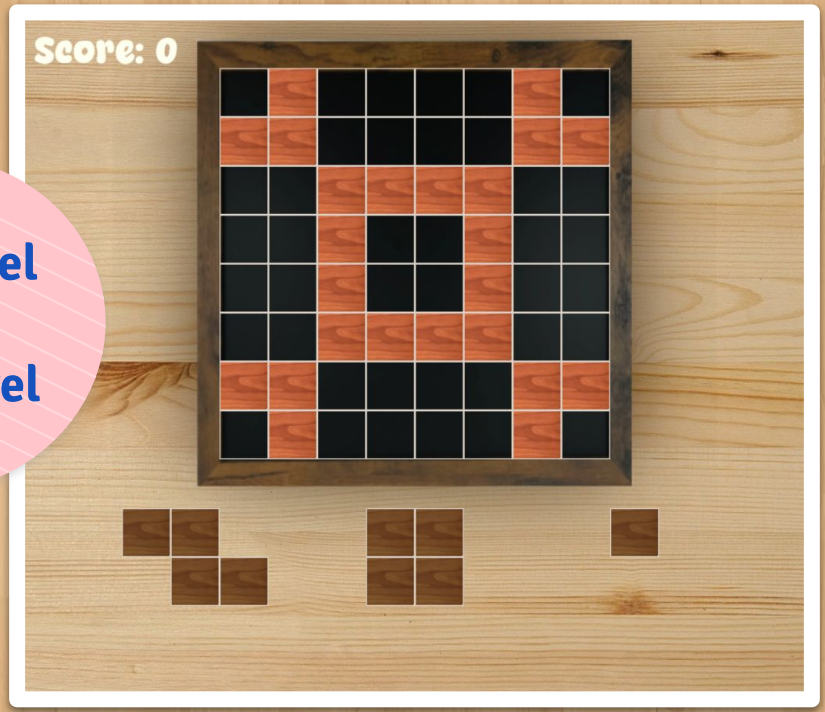
Experimental Results:

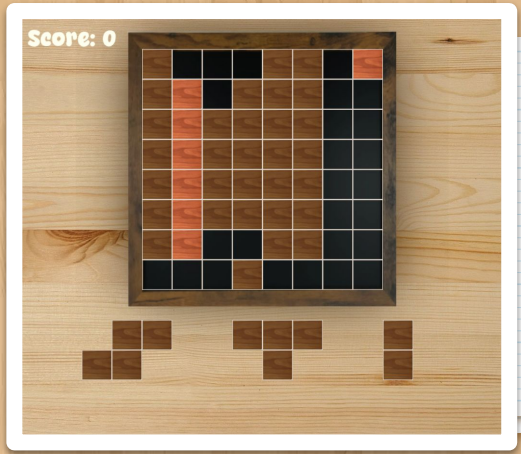


Easy Level

-

Hard Level





Easy Level

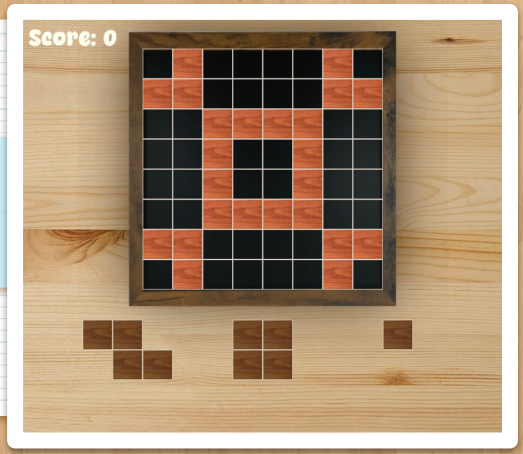
Obtained results:

BFS: 3 moves, 7.5 seconds
DFS: 62 moves, 2.6 seconds
Iter Deep: 3 moves, 0.9 seconds
Greedy: 5 moves, 0.9 seconds
A*: 3 moves, 1.3 seconds
Weighted A*: 4 moves, 6 seconds (outlier for specific level)

Hard Level

Obtained results:

BFS: Not Feasible (> 1000s)
DFS: Not Feasible (> 1000s)
Iter Deep: Not Feasible (> 1000s)
Greedy: 20 moves, 7.8 seconds
A*: Not Feasible (> 1000s)
Weighted A*: 19 moves, 145 seconds



Experimental Results

Conclusions

In this project, we consolidated our understanding of search algorithms used in single-player games.

We explored the trade-off between the time invested in searching and the quality of the solution found. Additionally, we recognized the crucial role of designing an effective heuristic that aligns with the specific requirements of the problem at hand.

References consulted and materials used

Documents:

We mostly used the slides from AI classes, mainly **Lecture 2b - Solving Search**

Websites:

We used **StackOverflow** to answer some of our more complex questions regarding the a* algorithm and its heuristic creation process.

Others:

No scientific articles, third-party software or other materials were used to develop the game.