# Wood Block - T11G5

Guilherme Ribeiro de Campelo Teixeira
Rafael Sousa Cunha
Júlio Duarte Pinto dos Santos

# Specification

- **Game Concept:**
  - A **one-player solitaire game** with a board and various shaped pieces
  - Earn points by **forming complete lines**
  - Game ends when **no more moves** are possible

- **Game Modes:**
  - **Levels Mode:** Destroy **preset red squares** to win
  - **Infinite Mode:** Score as many points as possible
  -
- **Project Objective:**
  - Develop different **search algorithms** to optimize gameplay strategies and decision-making

- **Related Work:**
  - Known board game bots such as Stockfish, Deepblue
  - Student-Developed bots for games in the PFL course
  - Online Block Blast Solver, a web-app that helps win a game with the same concept as ours

# Problem Formulation

**State Representation**

- **State = (Pieces, Grid, Score)**
  - **Pieces** → List of remaining pieces to be played
  - **Grid** → (Red blocks, Blue blocks, Empty spaces)
  - **Score** → Current score (In **infinite mode**, it can increase infinitely)

**Initial State**

- **Levels (Special Levels added for algorithm testing purposes):**
  - Level 1: `(["P1", "P2", "P3"], (4,0,96), 0)`
  - Level 2: `(["P1", "P2", "P3"], (12,0,88), 0)`
  - Level 3: `(["P1", "P2", "P3"], (20,0,80), 0)`
  - Special 1: `(["P1", "P2", "P3"], (3,0,13), 0)`
  - Special 2: `(["P1", "P2", "P3"], (5,0,20), 0)`
  - Special 3: `(["P1", "P2", "P3"], (5,0,31), 0)`
  - Special 4: `(["P1", "P2", "P3"], (2,0,34), 0)`
- **Infinite Mode:**
  - `(["P1", "P2", "P3"], (0,0,100), 0)`

**Objective State**

- **Levels:**

  `(_, (0, _, _), _)`
- **Infinite Mode:**

  `(_, (_, _, _), INF) (Human only)`

# Problem Formulation

**Operators**

Each move consists of placing a piece on the grid:

- **Name:** `Place(Piece, Position)`
- **Preconditions:**
  - Positions occupied by piece must be empty
  - Piece must be available
- **Effects:**
  - Piece is removed from `Pieces`
  - Grid updates (Reds, Blues, Empty spaces)
  - Squares disappear if full lines are made
  - Score increases if full lines are made
- **Cost Function**

**Search Algorithm Heuristics (2 heuristics)**

**Greedy Heuristic**

- Evaluates states based on a score: remaining red squares (1000x), red square alignments (100x), blue square alignments (10x), and final score (2x)

**A* Heuristic**

- Returns the maximum between number of rows and columns containing red squares
- Admissible because destroying a line/column requires at least one block
- By not adding the number of rows and columns with target squares, accounts for cases where a single piece destroys both a row and column

# Implemented Algorithms

b - branching factor
d - solution depth
m - maximum tree depth

**Uninformed Search Algorithms:**

Breadth-First Search (BFS):

- Generates & explores states breadth-first using the typical queue implementation, returns the first goal state found, which guarantees an optimal solution
- Time and Space complexity: O(b^d)

Depth-First Search (DFS):

- Generates & explores states depth-first using the typical stack implementation, returning the first goal state found. Does not guarantee optimality, as it only backtracks when no further moves are possible.
- Time and Space complexity: O(b^d), O(b*d)

Iterative Deepening:

- A DFS with an increasing depth limit. Restarts the search with a higher limit if no goal is found. Slower than standard DFS but finds optimal solutions with much lower memory usage, making it faster
- Time and Space complexity: O(b^d), O(b*d)

# Implemented Algorithms

b - branching factor
d - solution depth
m - maximum tree depth

**Informed Search Algorithms:**

Greedy Search (Best-First Search)

- Evaluates states using heuristic function with priority queue for fastest performance but often suboptimal solutions due to fast but imprecise heuristics and lack of cost function.
- Time and Space Complexity: O(b^m) (worst case)

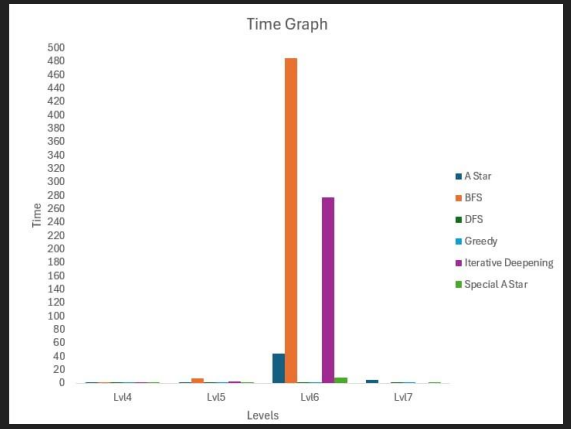
A* Search:

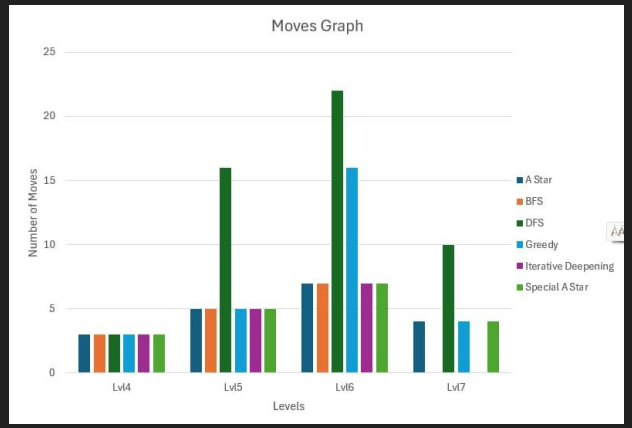
- Same idea as the Greedy Search, but uses an admissible heuristic and includes path cost (current depth). Slower than the greedy approach, but guarantees an optimal solution.
- Time and Space Complexity: O(b^d) (worst case)

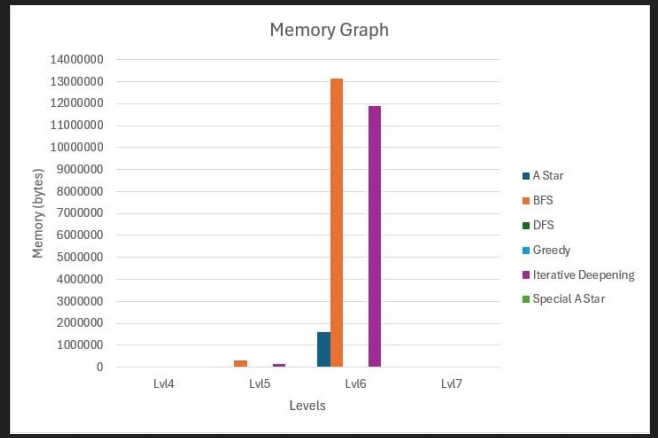
A* Search Variant:

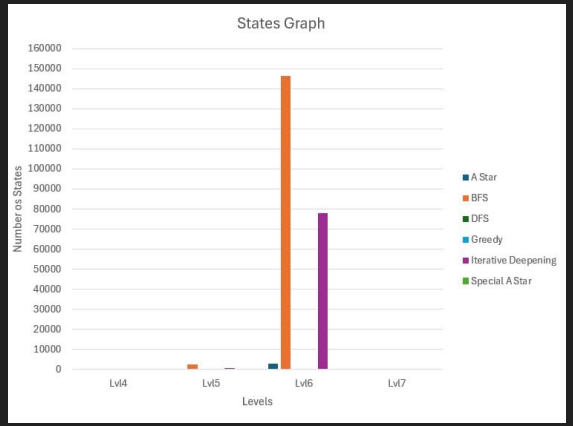
- Distinguishes states by proximity to specific objectives rather than entire grid, providing faster optimal solutions in most cases but potentially failing in complex scenarios (higher grid size, more depth necessary, etc.).
- Time and Space complexity: O(b^d) (worst case)

# Experimental Results (general)

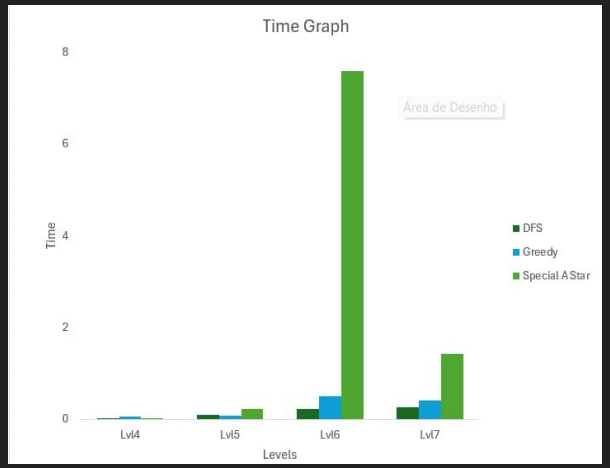


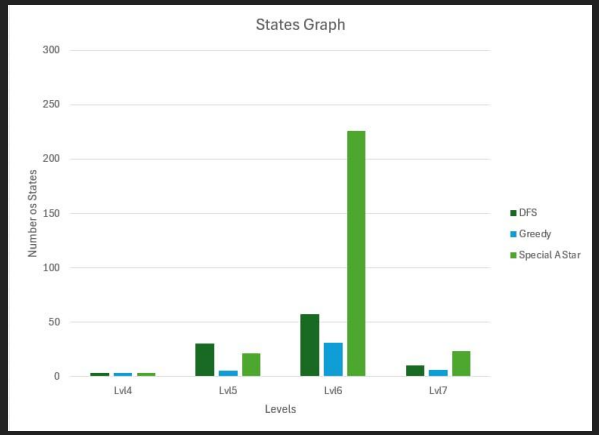



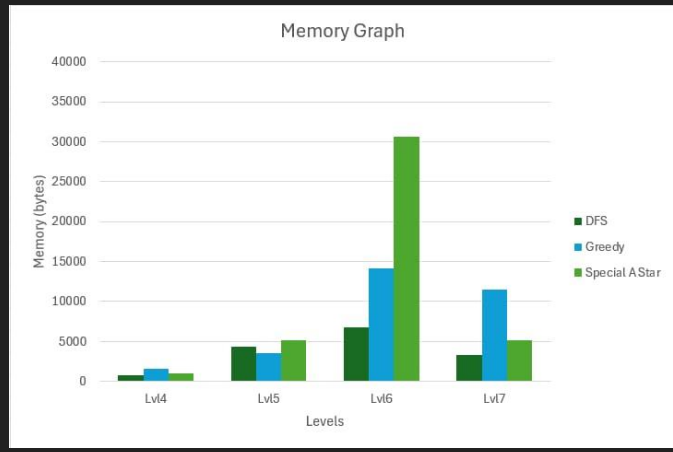



# Experimental Results (Non-Optimals)



# Conclusions

**Uninformed Search**

- BFS and Iterative Deepening are not feasible for realistic-sized boards if we want optimal solutions.
- For slightly smaller boards, they are feasible for up to a depth of 5, jumping from 10/- seconds to several minutes if we add more depth.
- DFS is extremely fast but tends to return highly non-optimal solutions due to its depth-first nature.

**Informed Search**

- Greedy Search is the best conventional algorithm for fast, good-enough solutions — ideal for hint generation in our game's context.
- Our A* Variant performs very well but may not yield a result in some cases due to aggressive state filtering (usually, only if we extend the board to at least 11x11). In cases where it doesn't fail, it seems to always return an optimal path.
- Our A* heuristic appears to be *admissible* — in all our tests it never overestimated the cost to reach the goal. No case was found where exhaustive BFS found a better solution than our conventional A*.

# References

Materials used:

- Theoretical slides
- A* application: https://iopscience.iop.org/article/10.1088/1742-6596/1898/1/012047/meta
- Iterative Deepening: https://researchmgt.monash.edu/ws/portalfiles/portal/333066936/330802887_oa.pdf

Tools used:

- Language: Python
- PyGame library
- Various libraries for data structures representations (ex.: heapq for priority queue)