ARTIFICIAL INTELLIGENCE ASSIGNMENT 1

Project 1 - Final delivery

Carolina Couto Viana - 202108802
João Oliveira - 202108737
Sérgio Peixoto - 202108681

WORK SPECIFICATION

The goal of this project is to implement the game **"Chesskoban"** and solve it using uninformed and heuristic search methods. Chesskoban is a **solitaire game** played in custom boards mapped to each level, where the player can **control a special piece "Pawn" to move his own pieces, white knights.**
The Pawn can move horizontally/vertically and, in case of colisions with white pieces, pushes them one square in the direction of the move. The **objective is to position each white knight strategically to capture all black knights** simultaneously, akin to the movement of a knight (L shape) in a chess game.

- **Goal**: develop heuristic search methods that can explore the possible game states and seek the optimal sequence of moves;
- **Analysis of the results**: take into consideration quality parameters such as the number of moves required to complete the level, as well as the time spent to obtain the solution;
- **Comparison of the efficiency**: between uninformed search methods and heuristic search methods.

FORMULATION OF THE PROBLEM AS A SEARCH PROBLEM

State Representation: The state is represented internally by:

- Player (pawn) current position
- White knights current positions

The rest of the information (black knights, tiles, ...) is stored at a class level, aiming to avoid storing unnecessary static information in the state, resulting in less memory spent

Initial State: The player and the knights are in their original position on the tiles. (defined by the current level)

Objective Test: Each white knight should individually be positioned, at the same time, in a valid way to consume a distinct black knight (distancing from it in an L shape).

Operators:

Name	Preconditions	Effects	Cost
Move Player	• There should not be an empty space (lack of a tile) in the adjacent tile in the direction of movement • There should not be a black knight in the adjacent tile in the direction of movement • There should not be more than one white knights in the adjacent tiles in the direction of movement • If there is 1 and only 1 white knight in the adjacent tile in the direction of movement, there should be an empty tile (not wall/black knight) after this white knight in the same direction	• The player is moved from its current position to the new position • If there is 1 and only 1 white knight adjacent to the player in the direction of its movement, the white knight and the player both move in this same direction 1 tile	1

HEURISTICS/EVALUATION FUNCTION

The evaluation function is used by the search algorithm to evaluate the quality of a given state.

To design that evaluation function we designed some heuristics:

1. **H1: Manhattan distance to closest ideal position** (white knight): This is the sum of the Manhattan distances of the closest white knights to a checking position.
2. **H2: Large penalty for white knights stuck in corners**: (Weight: 10000) Acts as a pruning condition given the minimization problem. A white knight stuck in a corner, unless in a checking position, leads to nowhere in the problem
3. **H3: Small penalty for white knights not in a checking position**: (Weight: 4)

Each heuristic will have different weights assigned to it, to prioritize some of them.

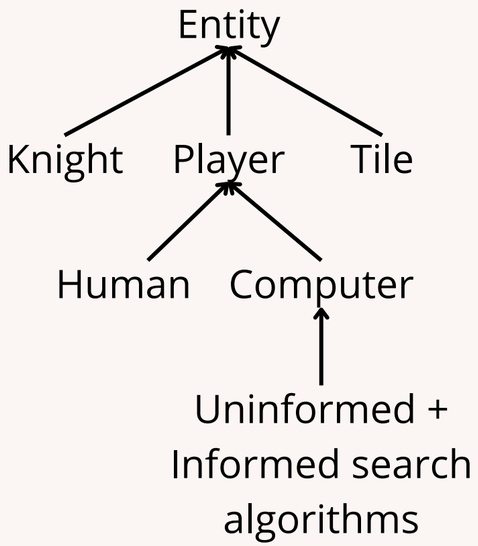
In order to analyze the impact of different heuristics, we tested 4 separate evaluation functions, that are the combination of the different heuristics

1. **Evaluation function 1: H1 + H2**
2. **Evaluation function 2: H1 + H2 + H3 (penalty = 4)**
3. **Evaluation function 3: H1 only**
4. **Evaluation function 4: H1 + H2 + H3 (penalty = 7)**

Later on in this presentation, we will analyze the results of the different evaluation functions.

WORK IMPLEMENTED

- Programming language: Python (pygame: https://www.pygame.org/docs/)
- Development environment: VSCode



Data Structures:

- Dashboard:
  - During game, shows number of moves and other statistics
- Menu:
  - Choose player (Human or AI Algorithms) and level of difficulty (1 through 7)
- Entity, Player, Human, Computer, Tile, Knight
  - Entity defines position and draws the entities;
  - Implemented features (most important):
    - move() -> if possible (handles collisions) moves entity
    - update() -> receives key input, determines new position
    - draw() -> receives coordinates and sprite and draws respective entity
    - checkWinCondition() -> handles winning condition
    - Computer:
      - get_solution() -> final path for computer player to take
      - generate_possible_moves() -> obtain all possible moves at given state
      - heuristic() -> evaluation function
      - checkWin() -> returns True when white knights are correctly placed to win game
- Level: Handles entities and starts game; Checks winning condition; Handles game cicle
- Position: coordinates
- Node and GameState: used for algorithms; represent a tree node (with attributes used on the different algorithms) and the state of the game (pawn and white knights position)

IMPLEMENTED ALGORITHMS

1. Uninformed search

1.1. Breadth First Search — Time Complexity: O(b^d) — Space Complexity: O(b^d)

- Takes an initial state, a goal_function (checkWin()) and a function to obtain child nodes (generate_possible_moves()) and performs a Breadth First Search

1.2. Depth First Search — Time Complexity: O(b^m) — Space Complexity: O(b*m)

- Takes an initial state, a goal_function (checkWin()) and a function to obtain child nodes (generate_possible_moves()) and performs a Depth First Search

1.3. Iterative Deepening DFS — Time Complexity: O(b^d) — Space Complexity: O(b*d)

- DFS Function: Takes an initial state, a goal_function (checkWin()), a function to obtain child nodes (generate_possible_moves()) and a depth D and performs a Depth First Search until depth D
- Iterative Function: Progressively increases depth until goal state has been reached

1.4. Uniform Cost — Time Complexity: O(b^(1+C/ε)) — Space Complexity: O(O(b^(1+C/ε)))

- Always expands the game state (node) with the least cost since the starting point -> g(n) function adds 1 per move made since initial state.
- Same class as A*, but disregards the heuristic function

IMPLEMENTED ALGORITHMS

2. Informed search

2.1. Greedy Search — Time Complexity: O(b^m) — Space Complexity: O(b^m)

- Always expands the game state (node) with the least heuristic value until goal state -> h(n) function defines value according to heuristic function, detailed previously
- Same class as A*, but disregards the cost function

2.2. A* Algorithm — Time Complexity: O(b^d)* — Space Complexity: O(b^d)

- Always expands the game state (node) with the least combined heuristic and cost value until goal state -> f(n) = g(n) + h(n)

2.3. Weighted A* Algorithm — Time Complexity: O(b^d)* — Space Complexity: O(b^d)

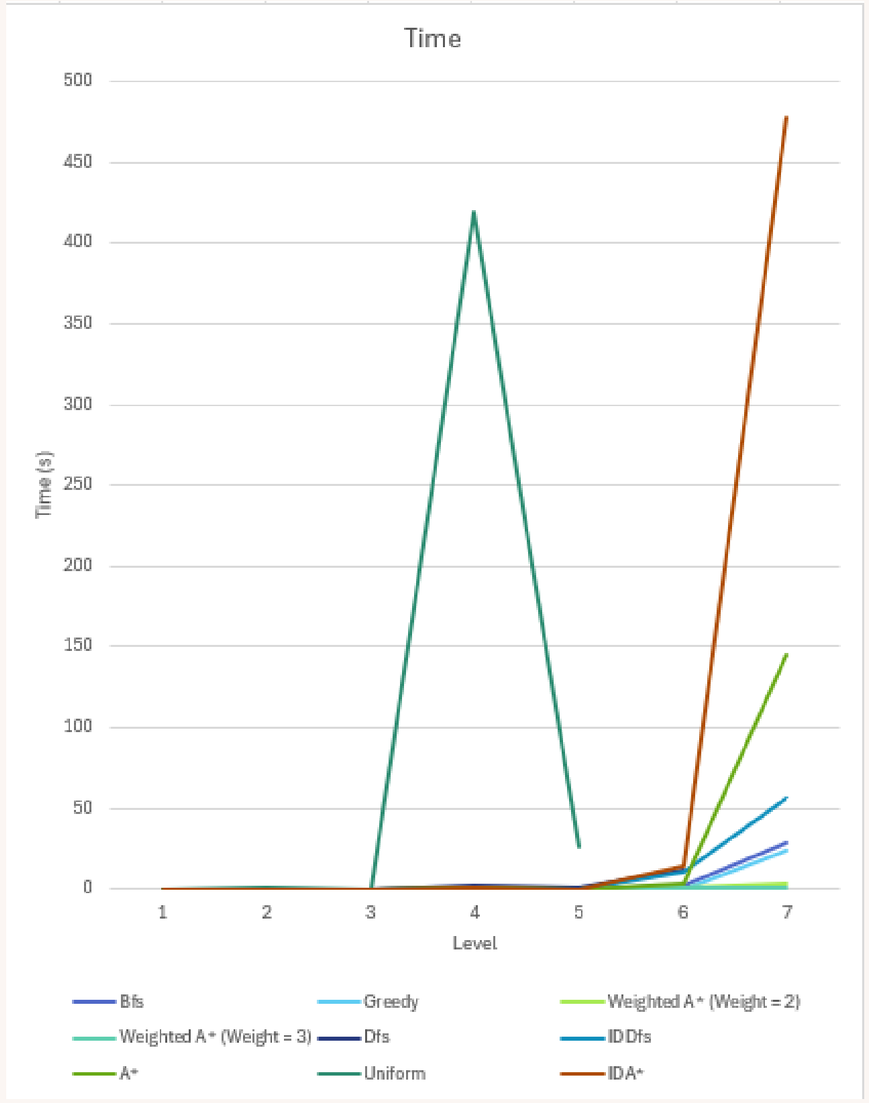
- Same as A* algorithm, except the heuristic function has a weight associated to it, to give it prevailance
- Tested with weights = 2 and 3

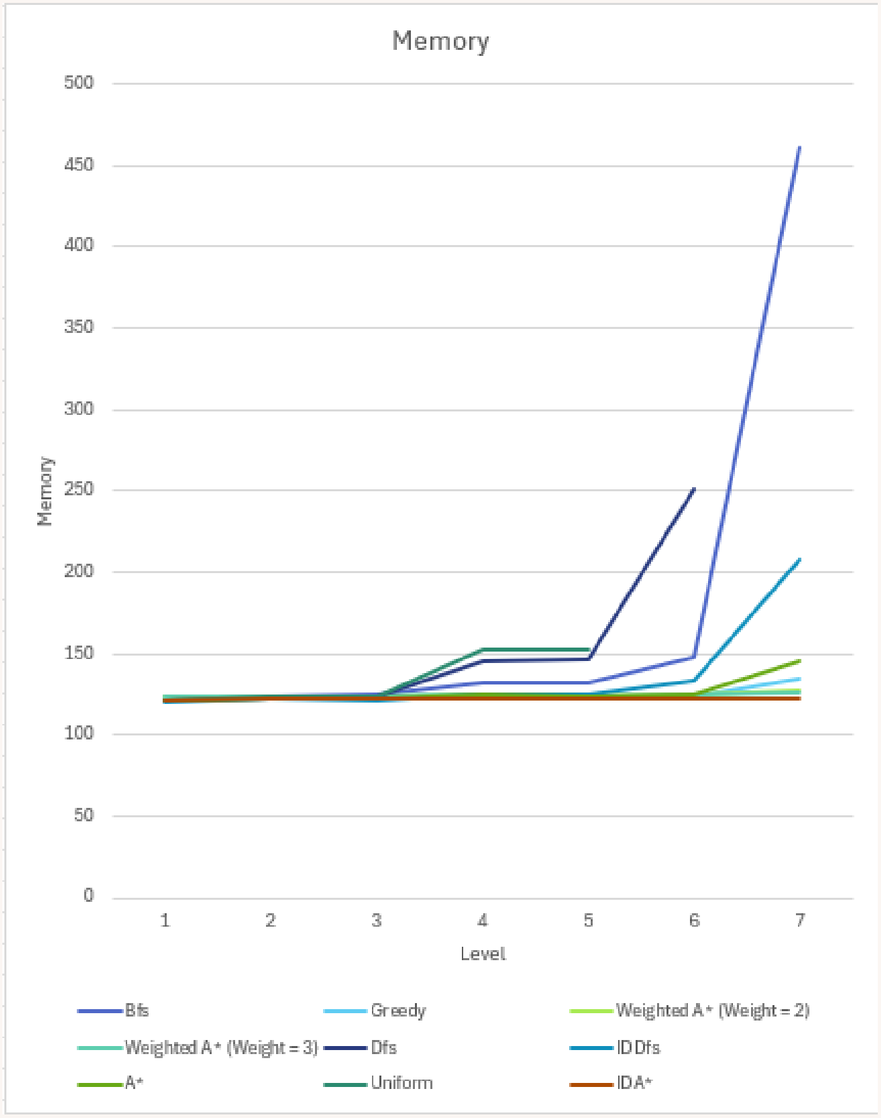
2.4. Iterative Deepening A* Algorithm — Time Complexity: O(b^d)* — Space Complexity: O(bd)*

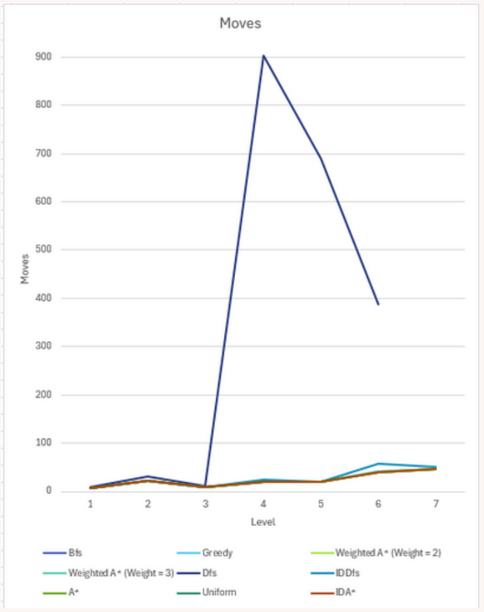
- A* Function: Performs an A* Algorithm search (as described above) until depth D, provided as an argument
- Iterative Function: Progressively increases depth until goal state has been reached

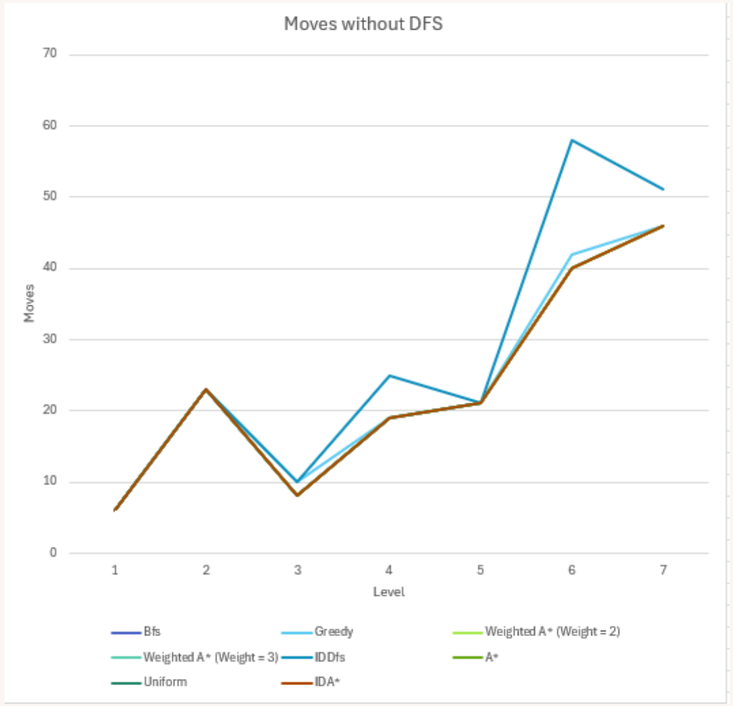
*: Worst-case scenario, actual complexity depends on heuristic function

ANALYSIS

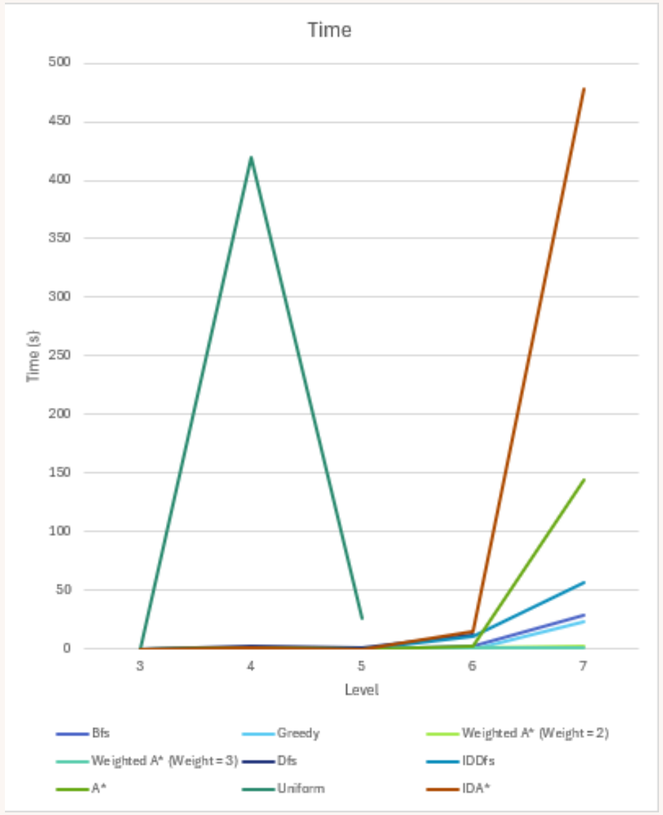

Time
Time (s)
Level
Bfs
Greedy
Weighted A* (Weight = 2)
Weighted A* (Weight = 3)
Dfs
IDDfs
A*
Uniform
IDA*

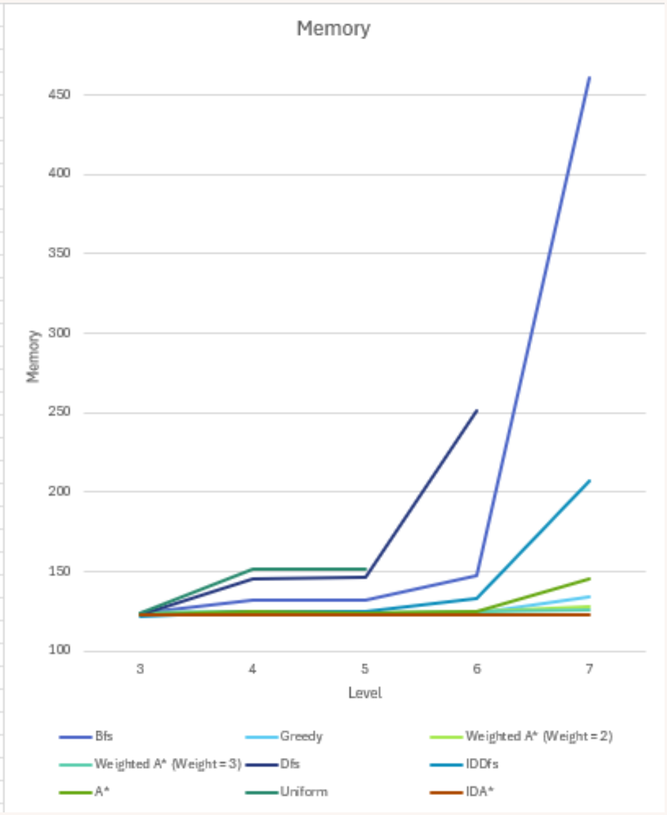

Memory
Memory
Level
Bfs
Greedy
Weighted A* (Weight = 2)
Weighted A* (Weight = 3)
Dfs
IDDfs
A*
Uniform
IDA*

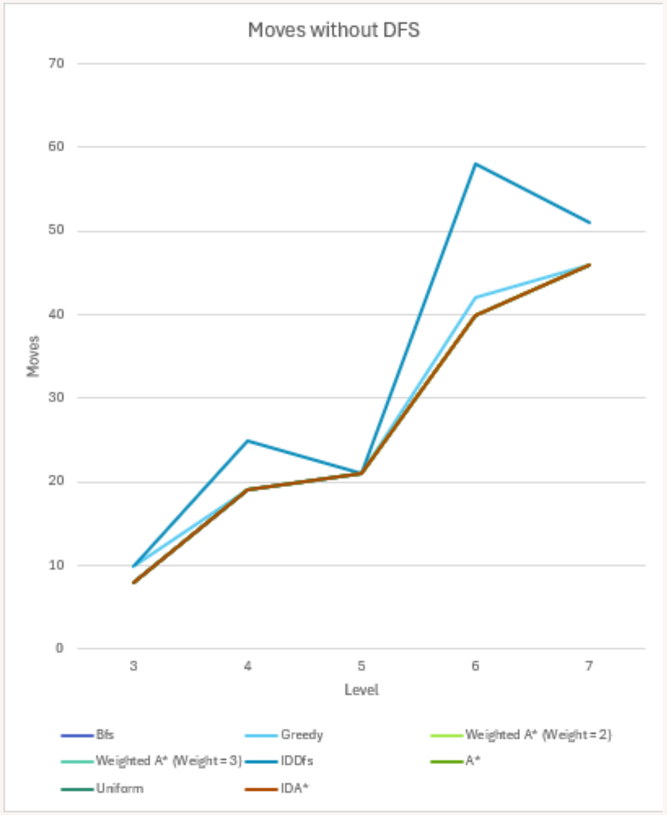

Moves
Moves
Level
Bfs
Greedy
Weighted A* (Weight = 2)
Weighted A* (Weight = 3)
Dfs
IDDfs
A*
Uniform
IDA*


Moves without DFS
Moves
Level
Bfs
Greedy
Weighted A* (Weight = 2)
Weighted A* (Weight = 3)
IDDfs
A*
Uniform
IDA*

ANALYSIS – CLOSEUP AND CONCLUSIONS







Most efficient time wise - Weighted A* (=3)

Most efficient memory wise - IDA*

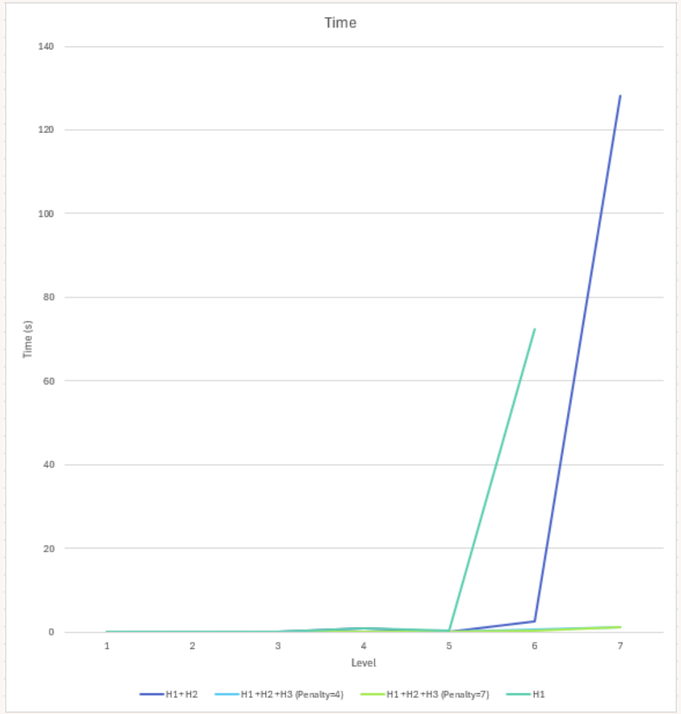
Optimal - BFS, Weighted A* (both), A*, IDA* (aren't all visible in the graph)

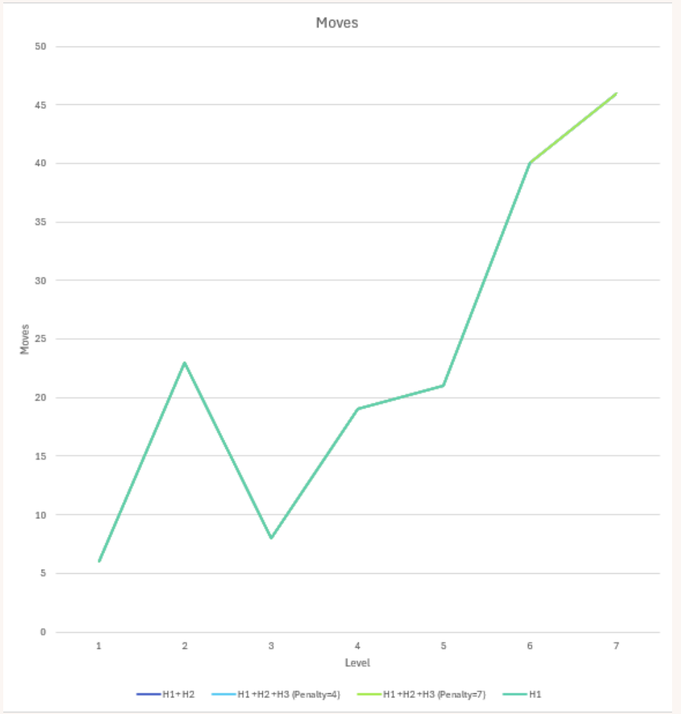
All around "best" - A* (although the weighted A* gave really good results, the weight added to the heuristic made it assume inadmissible values, which might result in non-optimality in more complex levels)

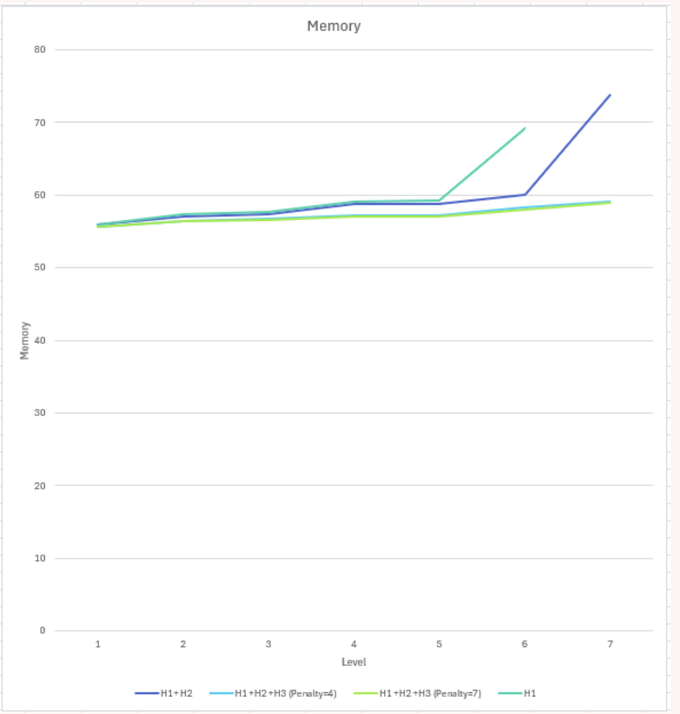
For these tests, we chose Evaluation Function 1. Although we could have used better heuristics, as evident by the next slide, we took this approach to better be able to notice the differences between each algorithm (as times would be very similar all throughout the tests with some of the other heuristics, even at level 7) and be able to make conclusions regarding the algorithms.

It's worth mentioning that, even though the Uniform Cost and A* are quite similar in their implementation, their results vary wildly as the Uniform Cost wasn't able to run level 6 and 7, either by taking too long or by using too much memory

ANALYSIS – HEURISTICS AND CONCLUSIONS







Observations:

- **Evaluation function 2 and 4 (H1 + H2 + H3) gave, by far, best results, followed by Evaluation function 1 (H1+H2)**
- **Evaluation function 3, composed of only one heuristic (H3) could not provide results for the last level,**

Conclusions:

- **The inclusion of penalties greatly influenced the overall results of the evaluation functions**
- **Not all heuristics were admissible -> the penalties were overoptimistic - this led to some very fast results**
- **Despite this, all the heuristics gave optimal results for most informed algorithms - probably due to the relatively simple levels - and near optimal results for the rest.**

Note: all ran with the A* algorithm

CONCLUSIONS

- With the project, we were able to better understand the works of informed and uninformed algorithms, when applied to a single player game.
- Regarding the “best” algorithm (better overall quality) we picked the **A* search algorithm**. The algorithm showed optimal results with the chosen Evaluation function and overall good time/memory usage.
- Regarding the heuristics, we picked the **Evaluation function 2 (H1 + H2 + H3, penalty = 4)** because the trade off optimality/execution time was extremely benefitial. This heuristic was extremely fast and presented optimal/near optimal results.

NOTE ON THE MONTE CARLO TREE SEARCH

- Initially, we tried to implement the Monte Carlo Algorithm and make it work for single player.
- Although not usually done, there are papers that account for it working on certain single player games: https://dke.maastrichtuniversity.nl/m.winands/documents/CGSameGame.pdf
- We developed the algorithm, however, despite two separate tries and 4 days of debugging, it did not work.
- We maintained one version of the algorithm in the code, to show for our effort, but did not allow for the AI player to run it.

REFERENCES

- MonteCarlo: https://dke.maastrichtuniversity.nl/m.winands/documents/CGSameGame.pdf; https://github.com/sergeim19/SinglePlayerMCTS
- A* Algorithm: https://en.wikipedia.org/wiki/A*_search_algorithm; https://gist.github.com/Nicholas-Swift/003e1932ef2804bebef2710527008f44
- ChessKoban: https://steamcommunity.com/app/1784220

The End.

2023/2024; 2nd Semester