Artificial Intelligence - 2023/24

SOKOBOND

3LEIC07 - Group_A1_73

Francisco Campos
Henrique Pinheiro
Sara Azevedo

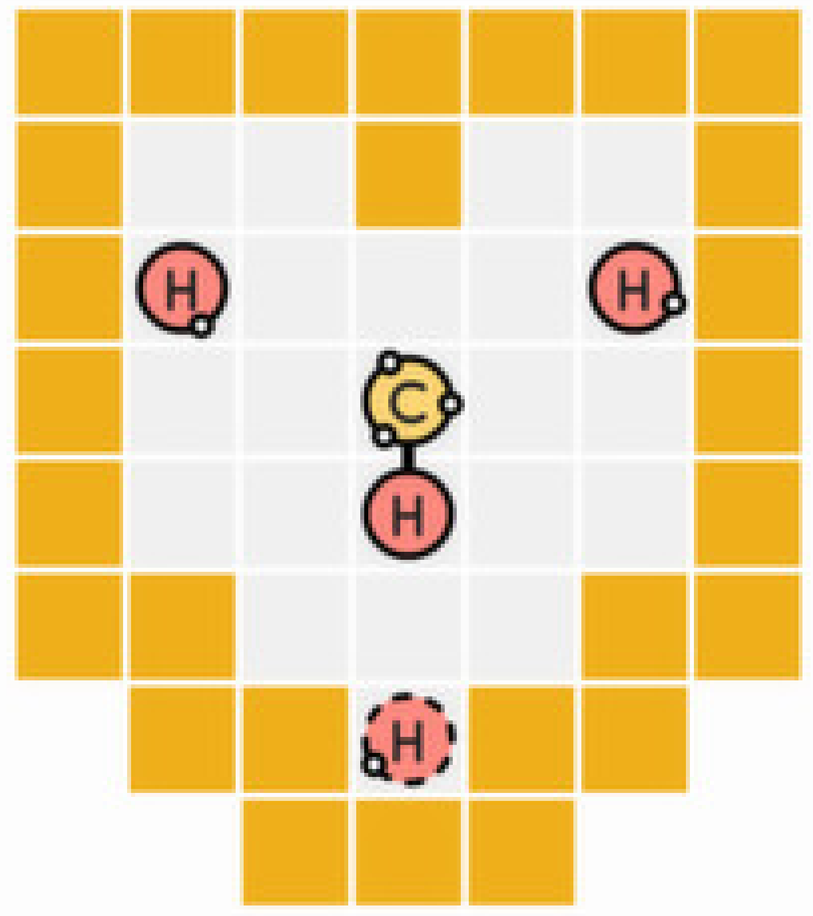
Game Definition

The game Sokobond is a single-player puzzle game featuring multiple levels of varying difficulty.

The aim of the game is to connect atmospheres on a 2D board to create chemical components.

At each level, new pieces may appear that energise the game and add the possibility of splitting, rotating and so on to the component.

Each level ends when the component is completely formed, i.e. there are no spare valence electrons and all the atmos are connected.



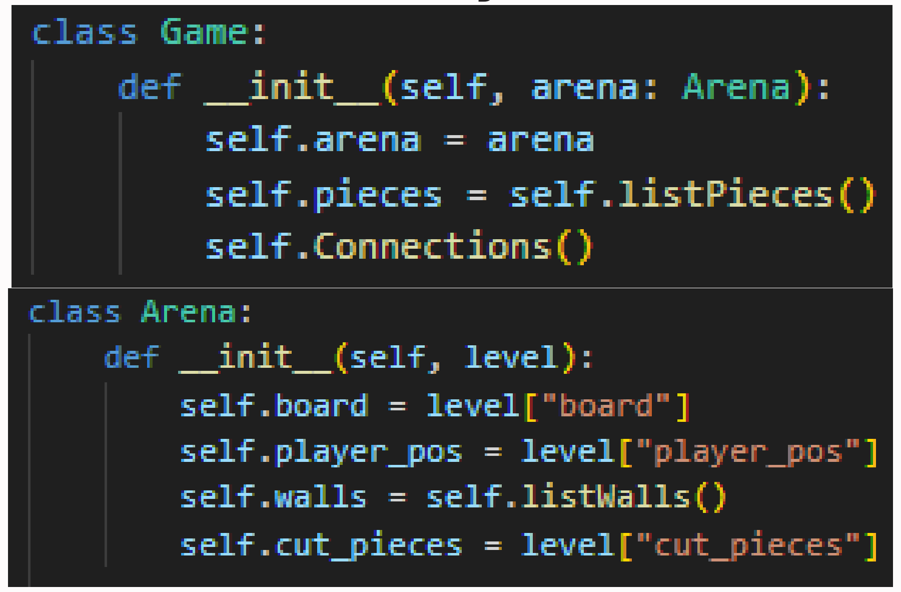
Formulation of the problem

State representation

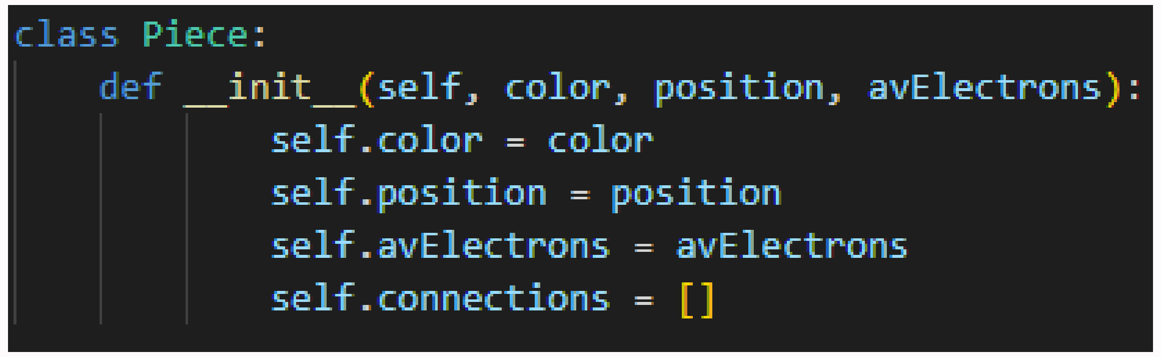
The state of the game is kept in a single instance of the Game class. It holds the game matrix, where the following representative symbols are used:

- 'y' - wall
- ‘H’ / ‘O’ - atoms
- None - blank space

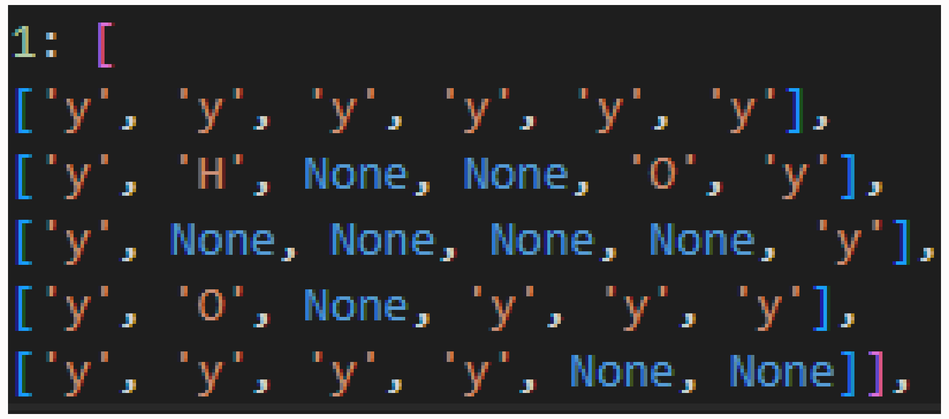
For each atom symbol, an instance of the Piece class is created, where all the necessary information is stored.



```
class Game:
    def __init__(self, arena: Arena):
        self.arena = arena
        self.pieces = self.listPieces()
        self.Connections()
class Arena:
    def __init__(self, level):
        self.board = level["board"]
        self.player_pos = level["player_pos"]
        self.walls = self.listWalls()
        self.cut_pieces = level["cut_pieces"]
```



```
class Piece:
    def __init__(self, color, position, avElectrons):
        self.color = color
        self.position = position
        self.avElectrons = avElectrons
        self.connections = []
```



```
1: [
['y', 'y', 'y', 'y', 'y', 'y'],
['y', 'H', None, None, 'O', 'y'],
['y', None, None, None, None, 'y'],
['y', 'O', None, 'y', 'y', 'y'],
['y', 'y', 'y', 'y', None, None]],
```

Initial State

The initial state is variable, since there are different combinations of starting board for each level.

Formulation of the problem

Objective function

In each level, the game ends when the component is completely formed, that is, when there are no excess valence electrons and all the atmos are connected.

Operators

Moving up, down, right, or left.

Heuristics/evaluation function

We’ll be using the following search algorithms: DFS, BFS, IDS, greedy, A* search

We’ll be using the following heuristics:

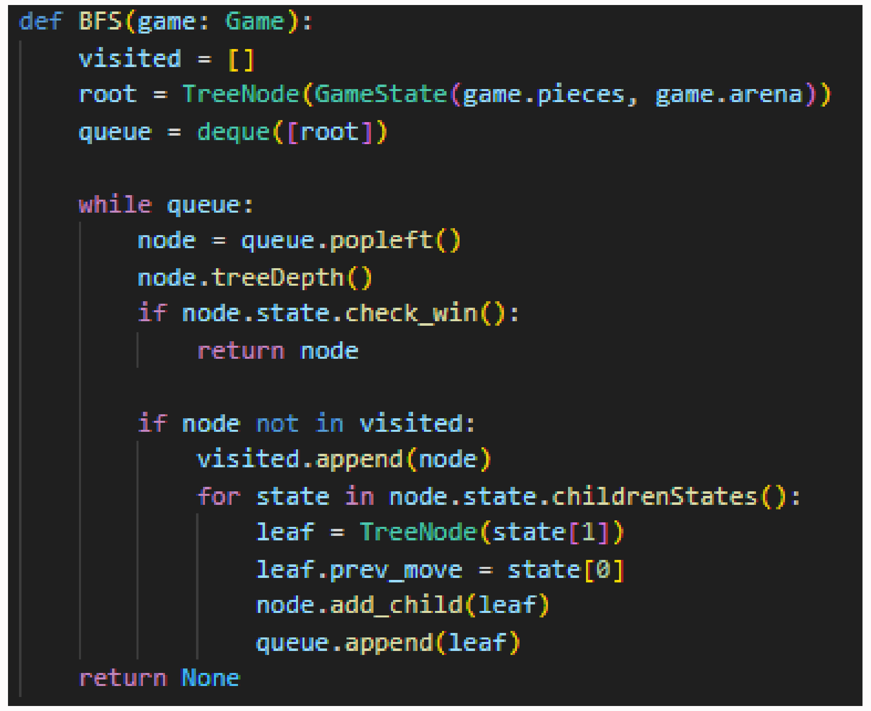
- Heuristic 1 - minimize distance between molecule and free atoms
- Heuristic 2 - minimize distance between molecule and free atoms with higher number of available electrons

Approach

The game comprises three distinct menus, facilitating the initiation of gameplay. Players can choose between playing or delegating control to an artificial intelligence. Then, the players can select the desired difficulty level. All levels are solvable. In addition, during artificial intelligence gameplay, players can select the algorithm for AI decision making.

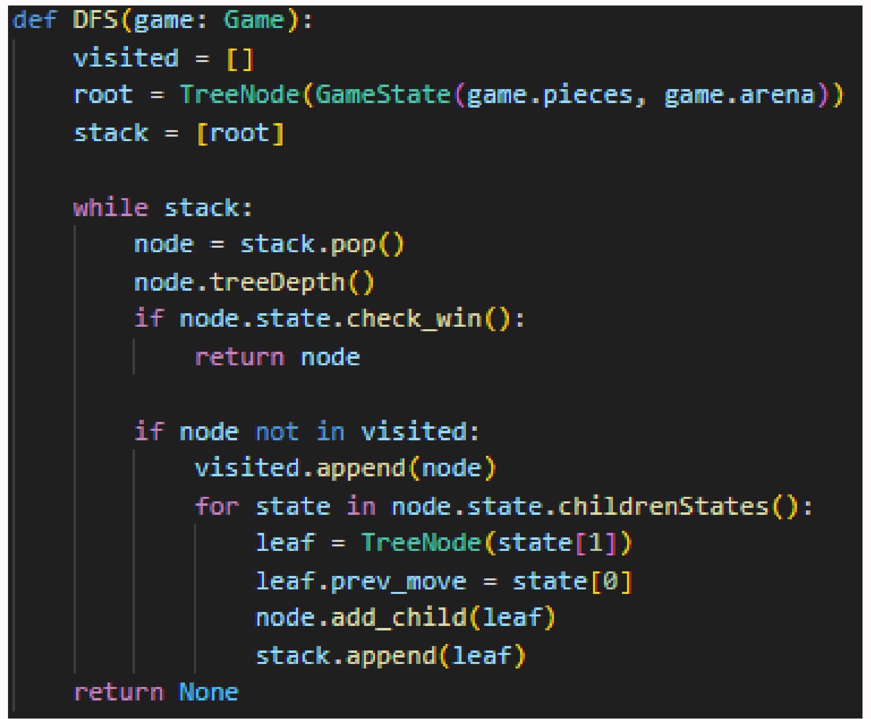
Implemented Algorithm’s

BFS -



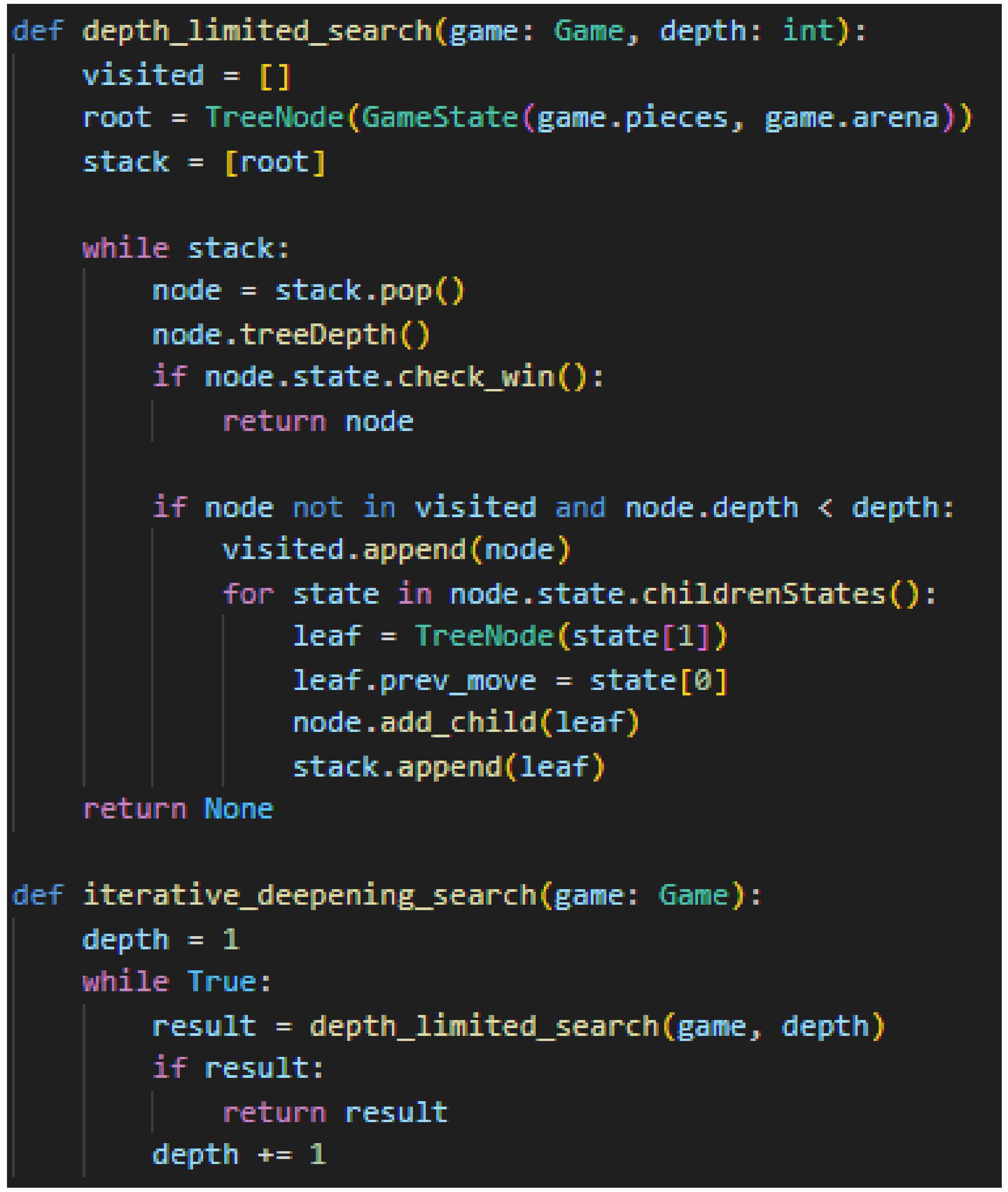
```python
def BFS(game: Game):
    visited = []
    root = TreeNode(GameState(game.pieces, game.arena))
    queue = deque([root])

    while queue:
        node = queue.popleft()
        node.treeDepth()
        if node.state.check_win():
            return node

        if node not in visited:
            visited.append(node)
            for state in node.state.childrenStates():
                leaf = TreeNode(state[1])
                leaf.prev_move = state[0]
                node.add_child(leaf)
                queue.append(leaf)
    return None
```

DFS -



```python
def DFS(game: Game):
    visited = []
    root = TreeNode(GameState(game.pieces, game.arena))
    stack = [root]

    while stack:
        node = stack.pop()
        node.treeDepth()
        if node.state.check_win():
            return node

        if node not in visited:
            visited.append(node)
            for state in node.state.childrenStates():
                leaf = TreeNode(state[1])
                leaf.prev_move = state[0]
                node.add_child(leaf)
                stack.append(leaf)
    return None
```

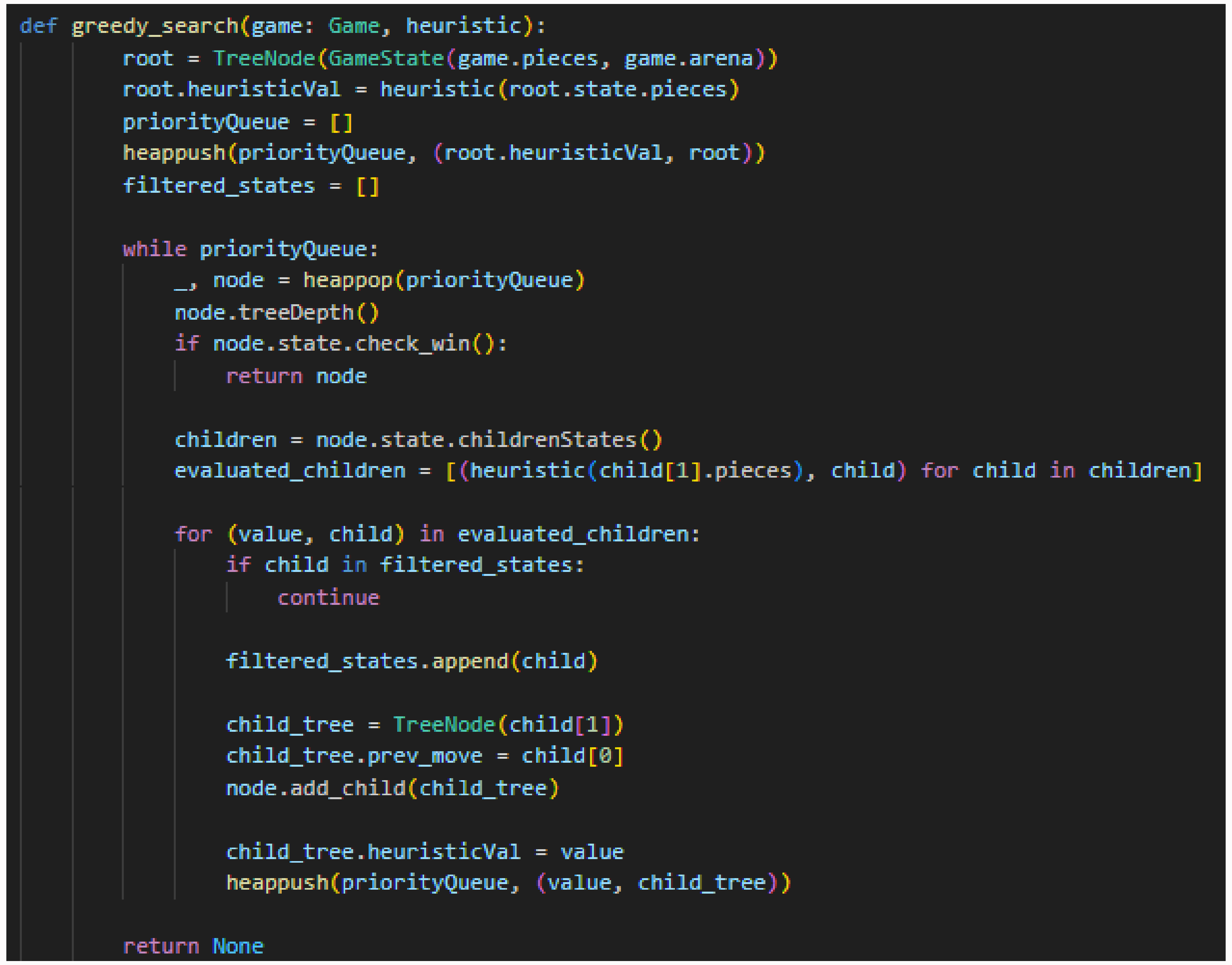
IDS -



```python
def depth_limited_search(game: Game, depth: int):
    visited = []
    root = TreeNode(GameState(game.pieces, game.arena))
    stack = [root]

    while stack:
        node = stack.pop()
        node.treeDepth()
        if node.state.check_win():
            return node

        if node not in visited and node.depth < depth:
            visited.append(node)
            for state in node.state.childrenStates():
                leaf = TreeNode(state[1])
                leaf.prev_move = state[0]
                node.add_child(leaf)
                stack.append(leaf)
    return None

def iterative_deepening_search(game: Game):
    depth = 1
    while True:
        result = depth_limited_search(game, depth)
        if result:
            return result
        depth += 1
```

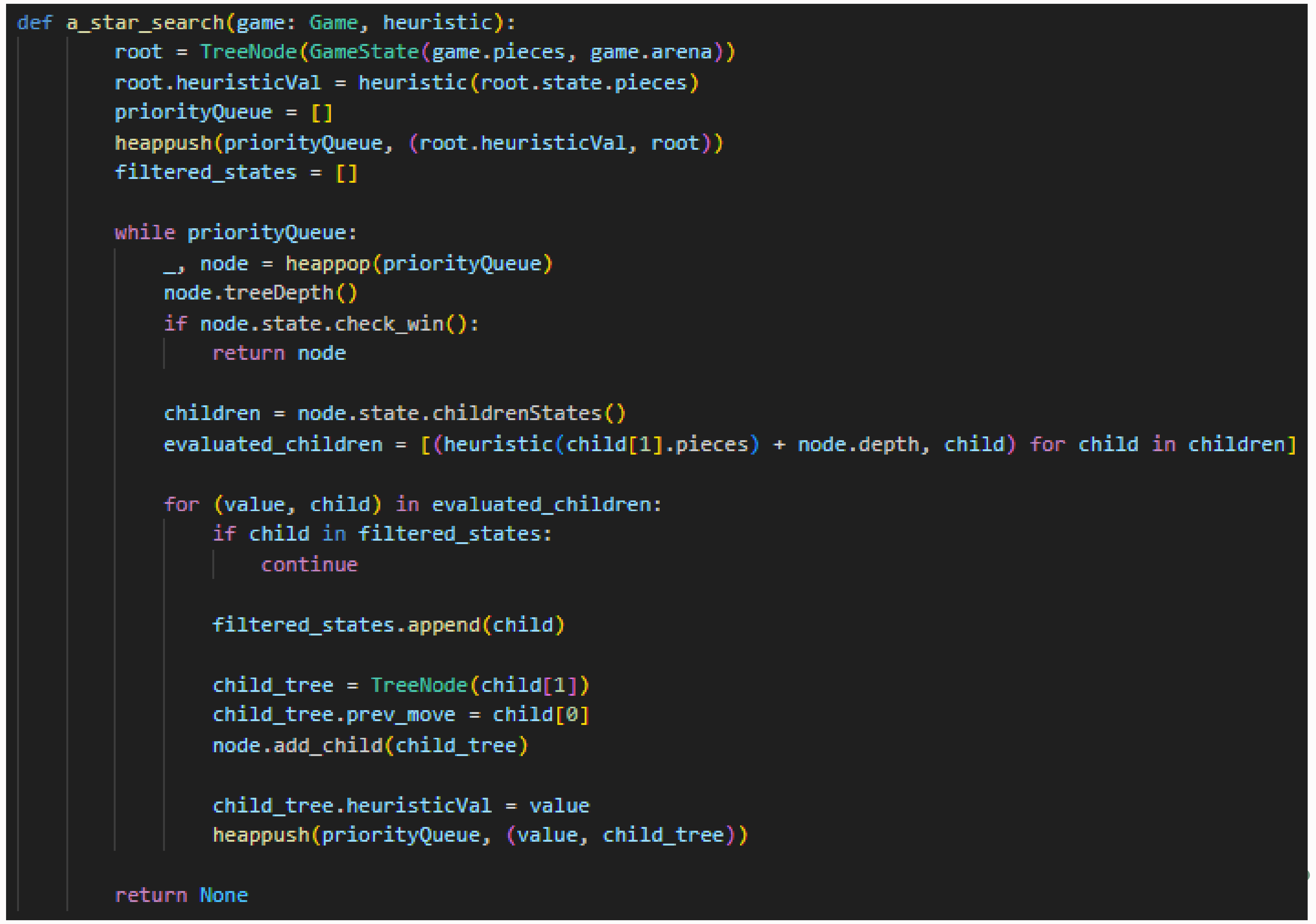
Implemented Algorithm’s

Greedy



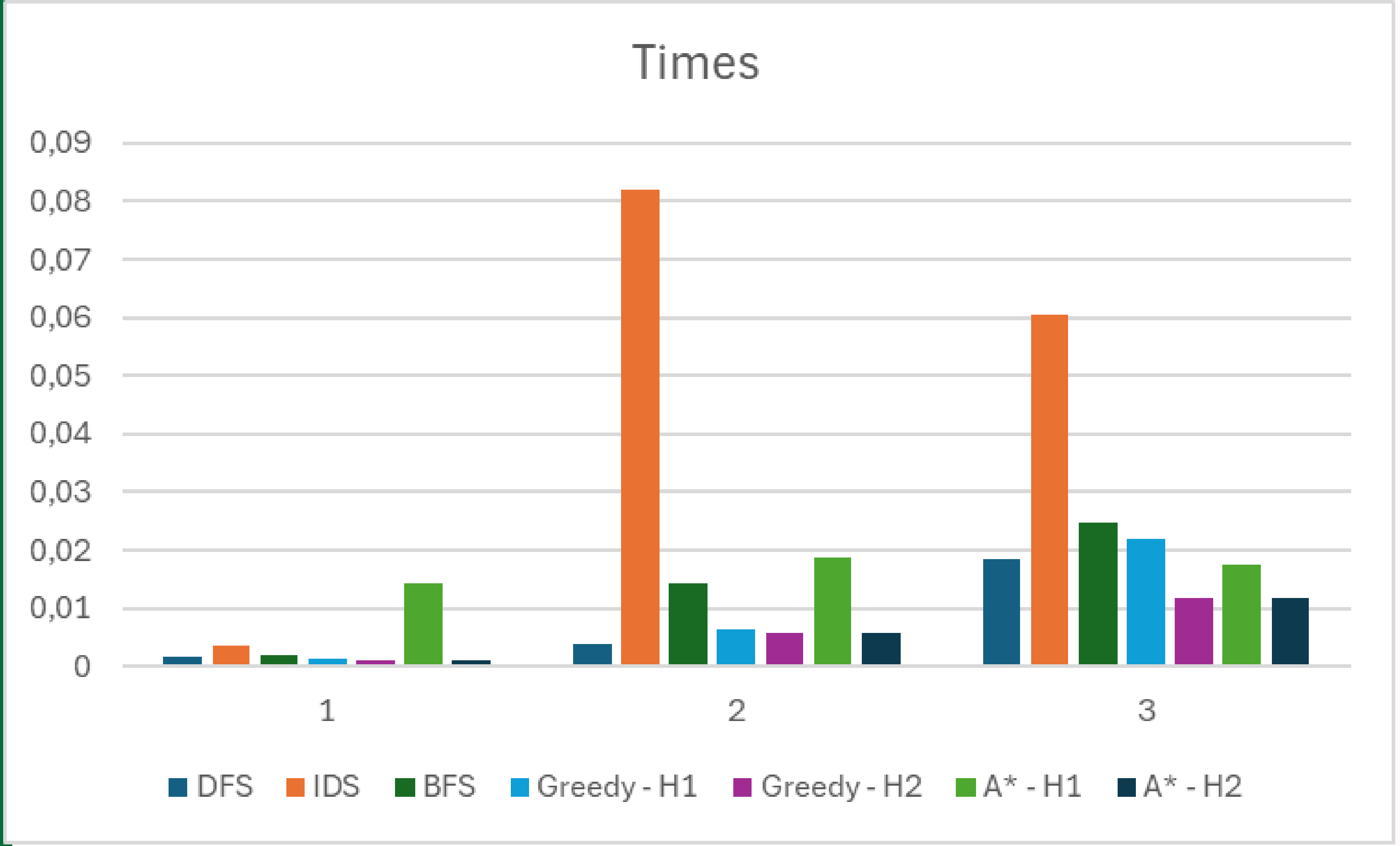
```python
def greedy_search(game: Game, heuristic):
        root = TreeNode(GameState(game.pieces, game.arena))
        root.heuristicVal = heuristic(root.state.pieces)
        priorityQueue = []
        heappush(priorityQueue, (root.heuristicVal, root))
        filtered_states = []

        while priorityQueue:
            _, node = heappop(priorityQueue)
            node.treeDepth()
            if node.state.check_win():
                return node

            children = node.state.childrenStates()
            evaluated_children = [(heuristic(child[1].pieces), child) for child in children]

            for (value, child) in evaluated_children:
                if child in filtered_states:
                    continue

                filtered_states.append(child)

                child_tree = TreeNode(child[1])
                child_tree.prev_move = child[0]
                node.add_child(child_tree)

                child_tree.heuristicVal = value
                heappush(priorityQueue, (value, child_tree))

        return None
```

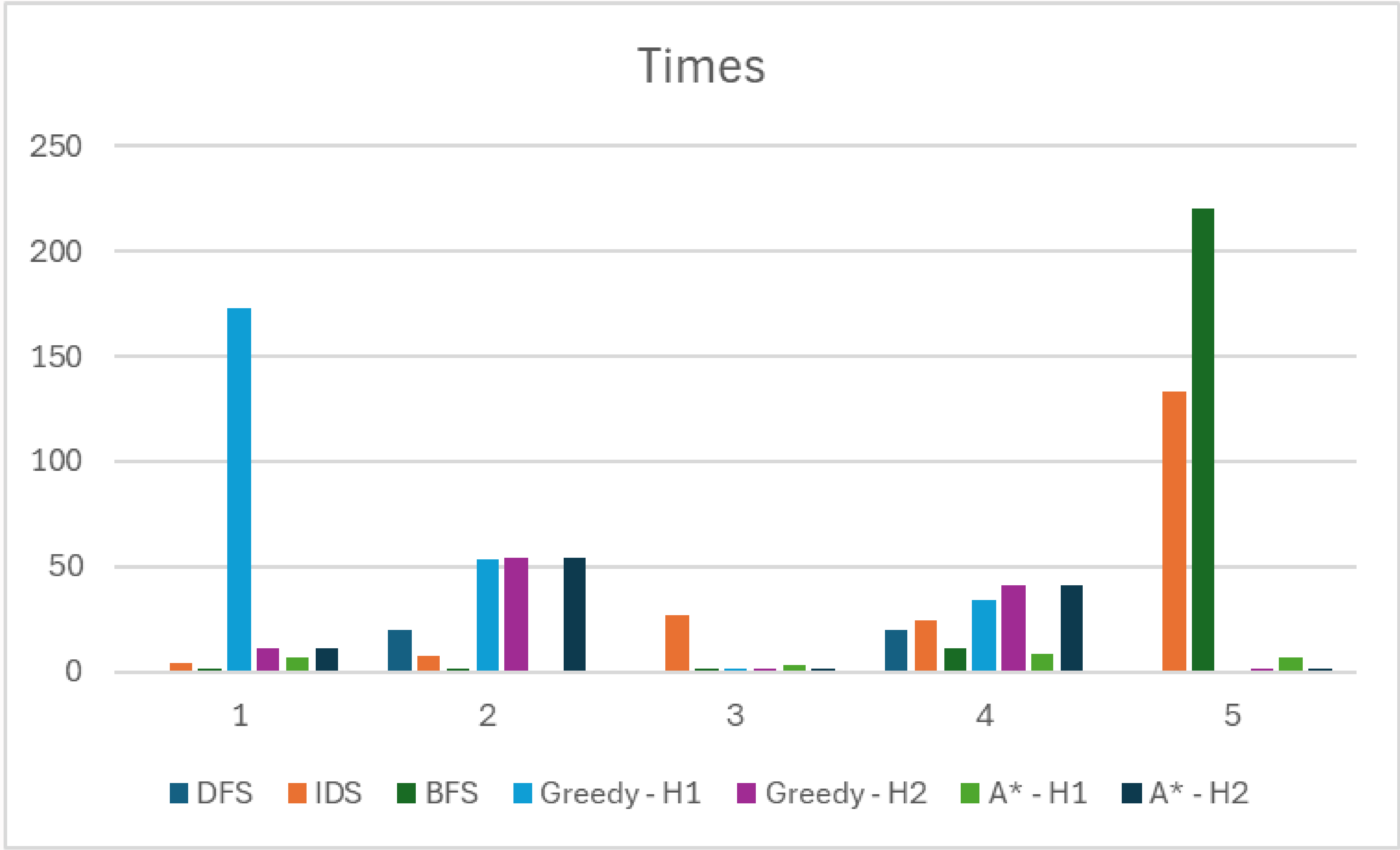
A*

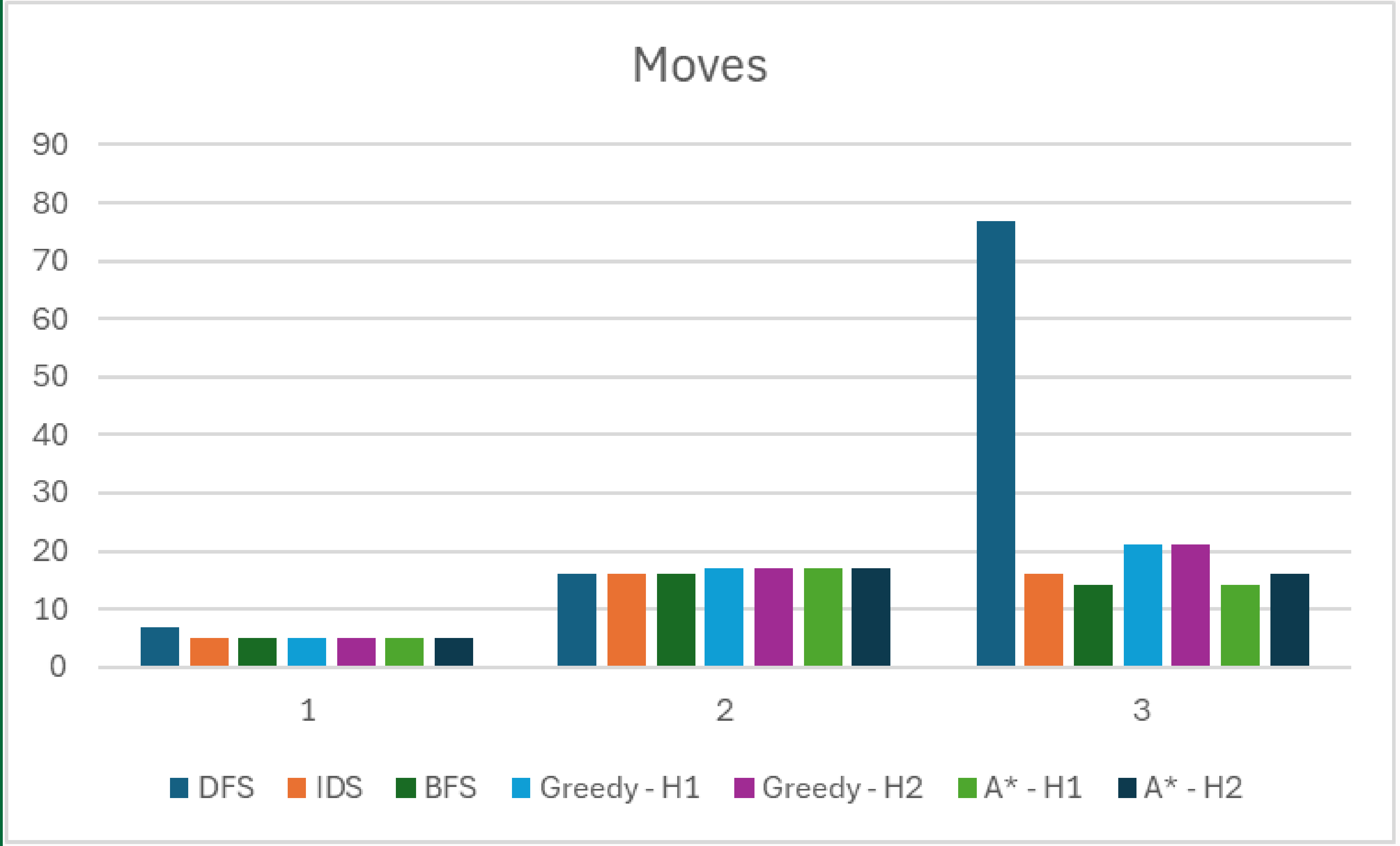


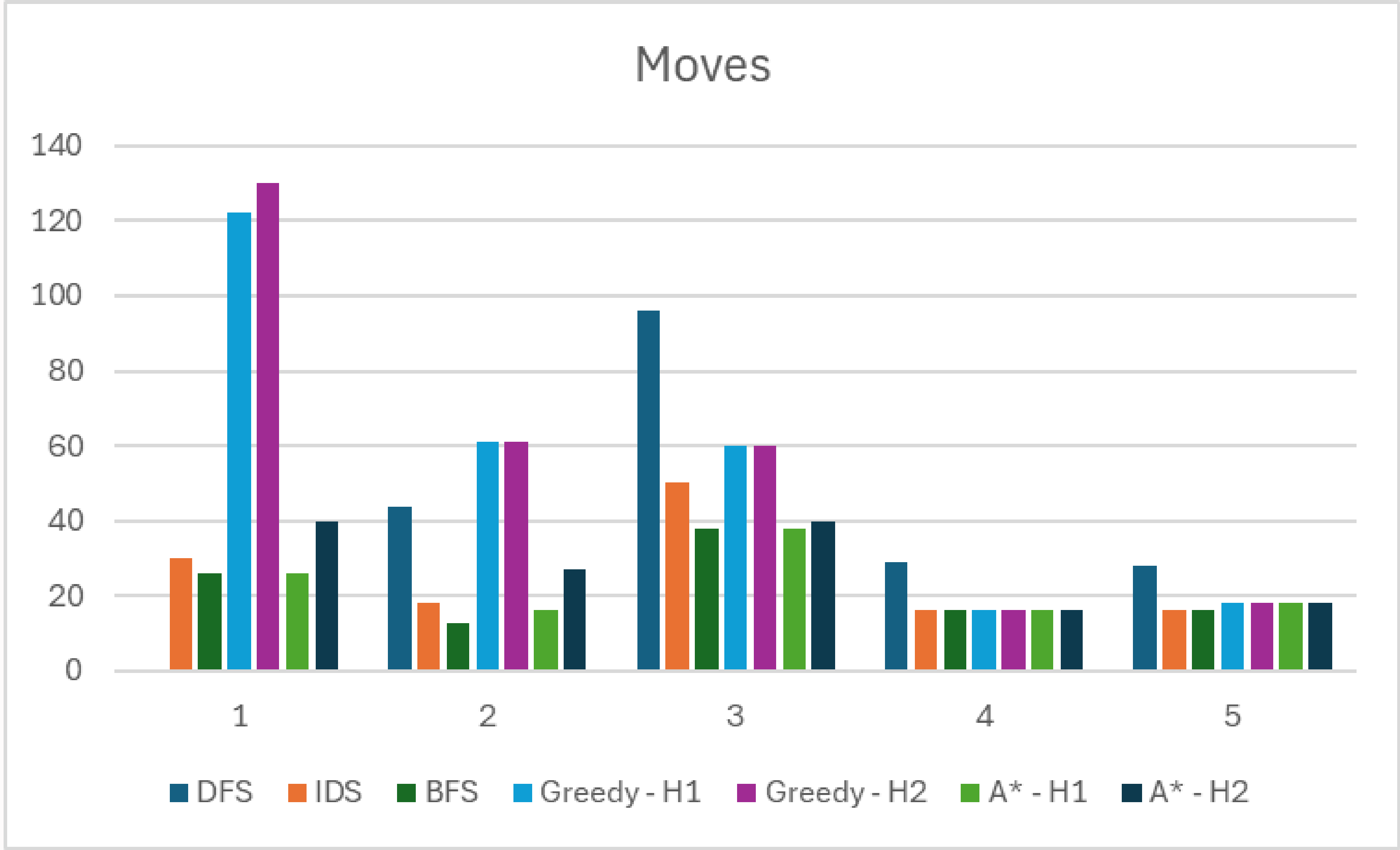
```python
def a_star_search(game: Game, heuristic):
        root = TreeNode(GameState(game.pieces, game.arena))
        root.heuristicVal = heuristic(root.state.pieces)
        priorityQueue = []
        heappush(priorityQueue, (root.heuristicVal, root))
        filtered_states = []

        while priorityQueue:
            _, node = heappop(priorityQueue)
            node.treeDepth()
            if node.state.check_win():
                return node

            children = node.state.childrenStates()
            evaluated_children = [(heuristic(child[1].pieces) + node.depth, child) for child in children]

            for (value, child) in evaluated_children:
                if child in filtered_states:
                    continue

                filtered_states.append(child)

                child_tree = TreeNode(child[1])
                child_tree.prev_move = child[0]
                node.add_child(child_tree)

                child_tree.heuristicVal = value
                heappush(priorityQueue, (value, child_tree))

        return None
```

Experimental Results









Conclusion

Through this project, we aim to consolidate our understanding of various topics, including search algorithms and the development of single-player games. We realize the importance of heuristics and effective implementations to obtain optimal solutions in terms of time efficiency and memory usage. In addition, this effort allowed us to gain insights into the application of artificial intelligence (AI) in gaming environments.

Related work

The Sokobond website shows information and resources related to the game.
- https://sokobond.com

A repository on GitHub that contains an implementation of Sokobond.
- https://github.com/vpelss/Sokobond_JS/blob/master/index.html

A repository on GitHub that contains a resolution of Sokobond with an A* Algorithm in Java.
- https://github.com/moonawar/SokobondSolver

Implementation work

Programming technologies: Python + Pygames

Develop environment: VS Code

Data structures:

- matrices to keep the state of the board
- a logic similar to a graph with its nodes to keep the connections between atoms, with adjacency lists