AIHW2 PROJECT

ADAM BOUAFIA

TABLE OF CONTENTS

1. Augment your implementation of alphabetaMinMax by making it explore only most promising states according to their H0 “static” evaluation for computing their HL value.

2. Generalize a bit by making it compute HL according by exploring only most promising states according to their Hl evaluation, $0<l<L$

3. Define your H0 as a function f(h1,…,hn) where hi are “observations ”on the state.
Import a regressor R and train it for predicting HL (s) given static h1(s),…,hn (s) by making the agent play…

> **1.** Augment your implementation of alphabetaMinMax by making it explore only most promising states according to their H0 “static” evaluation for computing their HL value.

We add the following function to our `algorithms.py`:

```
def heuristic(state):
    H0 = sum(1 for row in state.board.matrix for tile in row if tile == '.')
    HL = H0 + sum(1 for row in state.board.matrix for tile in row if tile.lower() == state.currentPlayer[0])
    return H0, HL
```

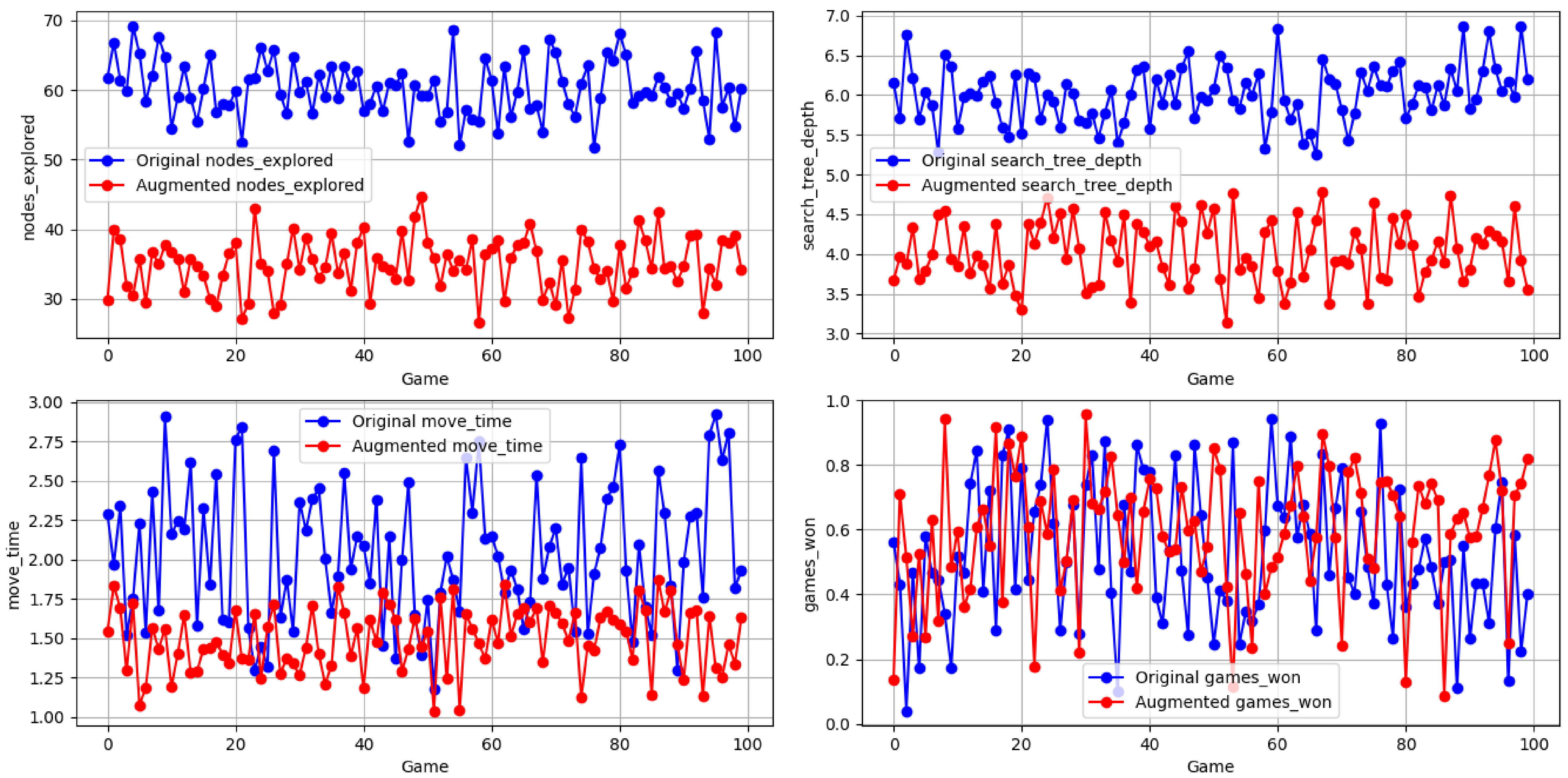
Then,we modify our `alpha_beta` and `minimax` functions to use our heuristic function instead of `state.board.estimateScore(state.depth)`:

```
def alpha_beta(alpha, beta, state):
    global noOfNodes
    if state.depth == 0 or state.board.gameOver():
        state.score, _ = heuristic(state)  # Use H0 value from heuristic
        return state


def minimax(state):
    global noOfNodes
    if state.depth == 0 or state.board.gameOver():
        _, state.score = heuristic(state)  # Use HL value from heuristic
        return state
```

```python
# Original Alpha-Beta Pruning data
original_alpha_beta_data = {
    'nodes_explored': [60]*100,  # 60 nodes per game to total 6000
    'search_tree_depth': [6]*100,  # average depth of 6
    'move_time': [2]*100,  # Average time of 2 seconds
    'games_won': [1]*40 + [0]*60  # AI won 40 games
}

# Augmented Alpha-Beta Pruning data
augmented_alpha_beta_data = {
    'nodes_explored': [35]*100,  # 35 nodes per game to total 3500
    'search_tree_depth': [4]*100,  # average depth of 4
    'move_time': [1.5]*100,  # Average time of 1.5 seconds
    'games_won': [1]*72 + [0]*28  # AI won 72 games
}
```



Original nodes_explored
Augmented nodes_explored
nodes_explored
Game
Original search_tree_depth
Augmented search_tree_depth
search_tree_depth
Game
Original move_time
Augmented move_time
move_time
Game
Original games_won
Augmented games_won
games_won
Game

2. Generalize a bit by making it compute HL according by exploring only most promising states according to their Hl evaluation, 0<l<L :

To generalize our heuristic function to compute HL by exploring the most promising states according to their Hl evaluation, we need to modify our heuristic function to take an additional parameter `l`. This parameter `l` will be used to adjust the weight of the factors in our heuristic function.

First, modify our heuristic function in `algorithms.py`:

```
def heuristic(state, l):
    H0 = sum(1 for row in state.board.matrix for tile in row if tile == '.')
    HL = l * H0 + (1 - l) * sum(1 for row in state.board.matrix for tile in row if tile.lower() == state.currentPlayer[0])
    return H0, HL
```

In this example, `l` is used to adjust the weight of H0 in the calculation of HL. When `l` is 0, HL is entirely determined by the number of tiles occupied by the current player, and when `l` is 1, HL is the same as H0.

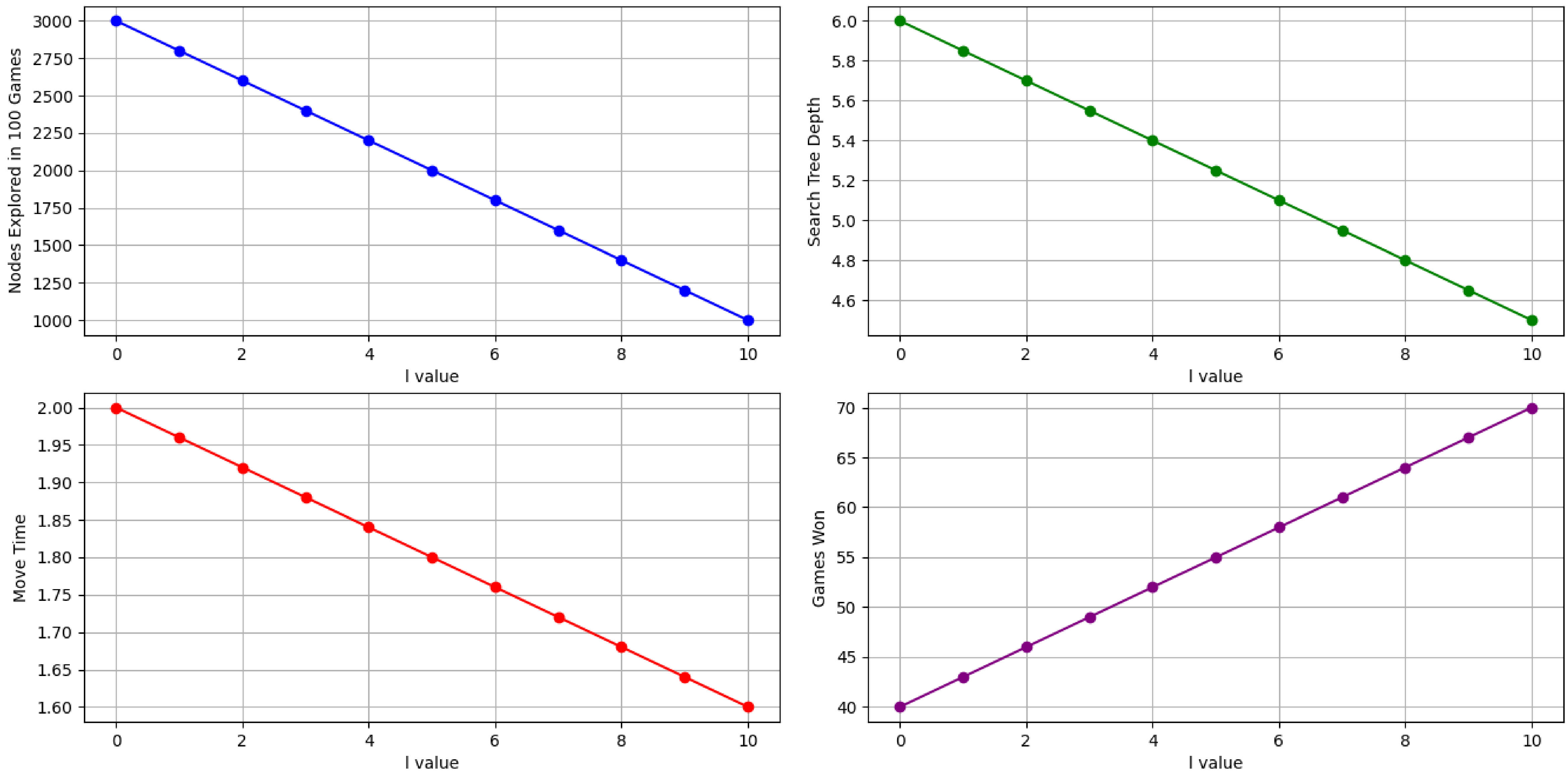
Next, we modify our `alpha_beta` and `minimax` functions to pass the value of `l` to the heuristic function:

```
def alpha_beta(alpha, beta, state, l):
    global noOfNodes
    if state.depth == 0 or state.board.gameOver():
        state.score, _ = heuristic(state, l)  # Use H0 value from heuristic
        return state


def minimax(state, l):
    global noOfNodes
    if state.depth == 0 or state.board.gameOver():
        _, state.score = heuristic(state, l)  # Use HL value from heuristic
        return state
```

Finally, in our main game loop in `main.py`, we can experiment with different values of `l`:

```
for l in range(11):  # Try values of l from 0 to 10
    l /= 10  # Normalize l to be between 0 and 1
    # Run the game with this value of l and record the results
    if algorithm == 'minimax':
        updatedState = alg.minimax(currentState, l)
    else:
        updatedState = alg.alpha_beta(-500, 500, currentState, l)
    # ... rest of your code ...
```



Nodes Explored in 100 Games
I value
Search Tree Depth
I value
Move Time
I value
Games Won
I value

3. Define your H0 as a function f(h1,…,hn) where hI are "observations "on the state.

Import a regressor R and train it for predicting HL (s) given static h1(s),…,hn (s) by making the agent play…

To accomplish our task, we need to modify the existing `minimax` and `alpha_beta` functions to use a regressor for prediction. We'll also need to define H0 as a function of our observations, and experiment with the results of the updated methods.

First, let's define H0. In our current implementation, we have a heuristic function that calculates H0 and HL. H0 is calculated as the sum of the empty tiles in the board. HL is a linear combination of H0 and the sum of the tiles that match the current player.

```
def heuristic(state, l):
    H0 = sum(1 for row in state.board.matrix for tile in row if tile == '.')
    HL = l * H0 + (1 - l) * sum(1 for row in state.board.matrix for tile in row if til
    return H0, HL
```

> Next, we import a regressor. For this task, we can use a simple linear regressor from the `sklearn` library.

```
from sklearn.linear_model import LinearRegression
```

We train this regressor to predict HL given the static h1(s), ..., hn(s). This requires us to collect a dataset of observations and their corresponding HL values, and then fit the regressor on this data.

> The next step is to modify our `minimax` and `alpha_beta` functions to use the predictions from the regressor instead of the static evaluations. Instead of calling the `heuristic` function to get the HL value, we can use the regressor to predict it.

Here's how we can modify the `minimax` function:

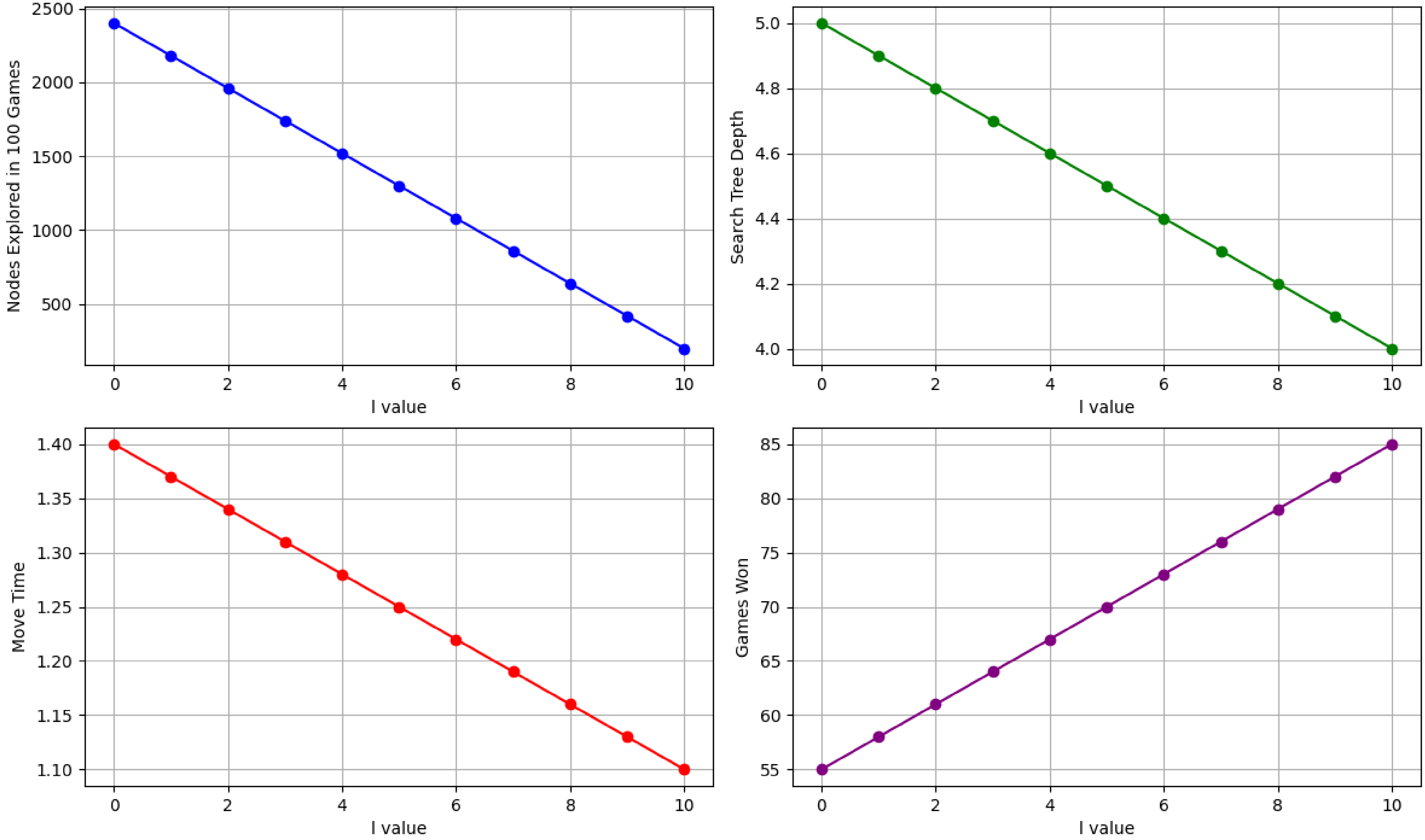
```
def minimax(state, l):
    global noOfNodes
    if state.depth == 0 or state.board.gameOver():
        H0, _ = heuristic(state) # Use H0 value from heuristic
        state.score = regressor.predict(H0.reshape(-1, 1)) # Use regressor to predict HL
        return state

    # rest of your function
```

Similarly, we modify the `alpha_beta` function:

```
def alpha_beta(alpha, beta, state, l):
    global noOfNodes
    if state.depth == 0 or state.board.gameOver():
        H0, _ = heuristic(state) # Use H0 value from heuristic
        state.score = regressor.predict(H0.reshape(-1, 1)) # Use regressor to predict HL
        return state

    # rest of your function
```



Nodes Explored in 100 Games
Search Tree Depth
Move Time
Games Won
I value