**Evaluating the effects of the evaluation functions on the Go games**

Vo Quoc Huy

Nghiem Van Tien

# Abstract

This report focuses on evaluating the performance and outcome differences when using two evaluation functions—**stone count** and **liberty count**—within the **minimax algorithm with alpha-beta pruning** to solve the **game of GO**.

# 1. Introduction

**1.1. Problem Introduction**

GO is an ancient strategic board game originating from China, now widely played around the world. The game is played on a square grid board, typically 19x19 (though smaller sizes like 9x9 or 13x13 are common for beginners), where two players alternately place black and white stones on the intersections. The objective is to control more territory than the opponent. When a stone or a group of stones is completely surrounded—meaning it has no remaining “liberties” (adjacent empty points)—it is captured and removed from the board. The game ends when both players agree to stop placing stones, and the winner is the player who controls the most territory after subtracting captured stones.

Despite its simple rules, the vast state space of GO makes it extremely challenging to solve computationally. A commonly used strategy to address this problem is the **minimax algorithm with alpha-beta pruning**, which simulates decision-making by searching future game states up to a limited depth. This approach requires an **evaluation function** to estimate the desirability of intermediate states. Two basic heuristics are often used for this purpose: **stone count** and **liberty count**. This raises a key question: **Does the choice of evaluation function significantly affect the performance and outcome of the minimax algorithm in GO?**

**1.2. Why Do We Need to Address This Problem?**

The choice of evaluation function in the minimax algorithm plays a crucial role in the efficiency of search processes for strategic games like GO. A well-designed evaluation function not only reduces computational resource usage but also improves the quality of decisions—especially in situations where the full state space cannot be explored due to depth limitations.In this context, comparing two common evaluation heuristics—**stone count** (which measures the number of stones controlled) and **liberty count** (which measures the number of adjacent empty points around stones)—holds both practical and academic significance. Stone count reflects the player’s physical control over the board, while liberty count indicates the potential survivability and strategic flexibility of stone groups. Analyzing the effectiveness of these two approaches offers deeper insights into how a machine “perceives” a game state, paving the way for the development of more sophisticated or even learned evaluation strategies.

In artificial intelligence and computer science, this comparison serves as a typical case of the trade-off between **simplicity and effectiveness** in heuristic design. The findings not only contribute to the enhancement of GO-playing algorithms but also have broader implications for the application of minimax and heuristic search in large-state decision problems such as automated planning, robotic pathfinding, and multi-agent strategic analysis.

**1.3. Our Work**

In this report, we developed a **minimax algorithm with alpha-beta pruning**, based on the foundational knowledge acquired from our artificial intelligence course, to address the decision-making problem in the game of GO. The algorithm was implemented in **Python**, with two distinct versions using separate evaluation classes: **StoneCountEvaluator** and **LibertyCountEvaluator**.

* **StoneCountEvaluator** estimates the game state based on the number of stones each player controls.
* **LibertyCountEvaluator**, on the other hand, evaluates the number of liberties (empty adjacent points) surrounding each group of stones, reflecting their potential survivability and strategic flexibility.

After completing the implementation, we conducted a series of **benchmark tests** to assess and compare the practical performance of the two evaluation functions. The measured metrics included:

* **Win rate** across simulated matches,
* **Average time per move**,
* **Memory usage per move**, and
* **Total memory consumption** over an entire game.

The results allowed us to analyze the strengths and limitations of each evaluation heuristic in the context of a depth-limited minimax search. More importantly, the findings provided clearer insights into how the choice of heuristic impacts the overall performance and decision quality of AI systems in games with large state spaces such as GO.

# 2. METHOD

In this study, we applied the **Minimax algorithm combined with Alpha-Beta pruning** to optimize decision-making in the game of GO. Minimax is a classical decision-making algorithm used in two-player games, where one player aims to **maximize the score** while assuming the opponent will **minimize it**.

However, due to the **enormous state space** of GO, traversing the entire game tree is computationally infeasible in terms of time and memory. To address this, we integrated **Alpha-Beta pruning** into the minimax algorithm to eliminate unnecessary branches, significantly reducing the number of states that need to be evaluated—**without affecting the final decision outcome**.

**Working Principle:**

* **Alpha:** the best value that the MAX player can guarantee at the current point.
* **Beta:** the best value that the MIN player can guarantee at the current point.
* **During tree traversal:**
  + At a MIN node, if a value less than Alpha is encountered, the branch is pruned (skipped), as MAX would never allow that move.
  + Similarly, at a MAX node, if a value greater than Beta is found, the branch is pruned, since MIN would prevent that move.

**Pseudocode of Minimax with Alpha-Beta Pruning:**

*def alphabeta(node, depth, alpha, beta, maximizingPlayer):*

*if depth == 0 or node is terminal:*

*return evaluate(node)*

*if maximizingPlayer:*

*value = -infinity*

*for child in node.children:*

*value = max(value, alphabeta(child, depth-1, alpha, beta, False))*

*alpha = max(alpha, value)*

*if alpha >= beta:*

*break # Beta cut-off*

*return value*

*else:*

*value = +infinity*

*for child in node.children:*

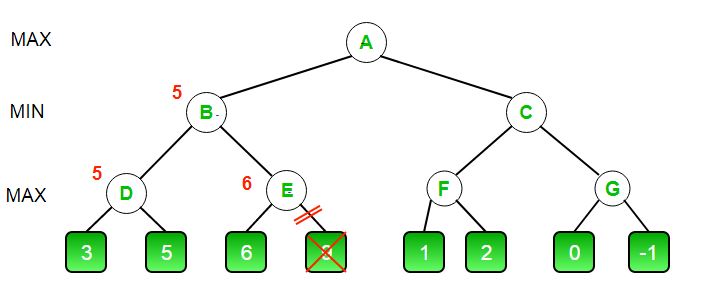
*value = min(value, alphabeta(child, depth-1, alpha, beta, True))*

*beta = min(beta, value)*

*if beta <= alpha:*

*break # Alpha cut-off*

*return value*



**State Evaluation and the Role of the Evaluation Function**

In games with large state spaces like GO, reaching terminal game states at every move is impractical due to time and resource constraints. Therefore, instead of waiting until the end of the game to determine a win or loss, an evaluation function is required to estimate the value of a game state at a certain depth in the search tree.

The values used by the minimax algorithm at leaf nodes or at the cutoff depth are those returned by the evaluation function. For the MAX player, the algorithm aims to select the move that yields the highest value; conversely, the MIN player chooses the move with the lowest value.

Designing an effective evaluation function is critical to the quality of decisions made by the algorithm. A well-crafted heuristic accurately reflects the strategic advantage of the current state and leads to smarter and more competitive moves during gameplay.

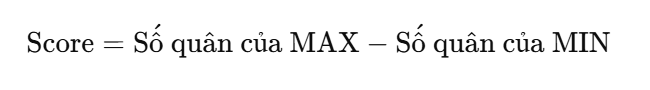
**Evaluation Functions Used in This Study**

In this study, we designed and implemented two simple yet meaningful evaluation functions tailored to the context of the game of GO:

**StoneCountEvaluator**

This function estimates the value of a given game state based on **the difference in the number of stones** placed on the board by the two players:

This function reflects the player's **physical control** over the board.



**LibertyCountEvaluator**This function evaluates the game state based on **the total number of liberties** that each player’s stones currently possess: Liberties are the empty points surrounding a stone or a group of stones, determining whether they are “alive” or at risk of being captured. Therefore, this function evaluates the **survivability** and the **potential for future strategic development**.



**Summary of Roles**

Both functions aim to provide a temporary estimate of each game state to enable the minimax algorithm to make appropriate decisions. While StoneCountEvaluator is simple and fast, LibertyCountEvaluator offers a deeper strategic perspective. Comparing the effectiveness of these two functions is the central focus of this study.

# 3. Experimental results and analysis

**3.1. Experimental Setup**

The experiments were conducted on **Windows 10** operating system using **Python 3.11**. The execution environment included essential libraries such as:

* **numpy** (for data processing),
* **pygame** (for board rendering and game interface),
* **matplotlib** (for plotting result charts),
* **psutil** (for measuring memory usage and runtime performance).

The algorithm was implemented to run on a **9x9 GO board**, with a fixed **search depth of 2**. Each experiment consisted of either **3 or 10 matches**, depending on the corresponding result charts, and involved two agents using different evaluation functions: **StoneCountEvaluator** and **LibertyCountEvaluator**.

**3.2. Evaluation Methods**

We evaluated the performance of the two AI agents using four key metrics:

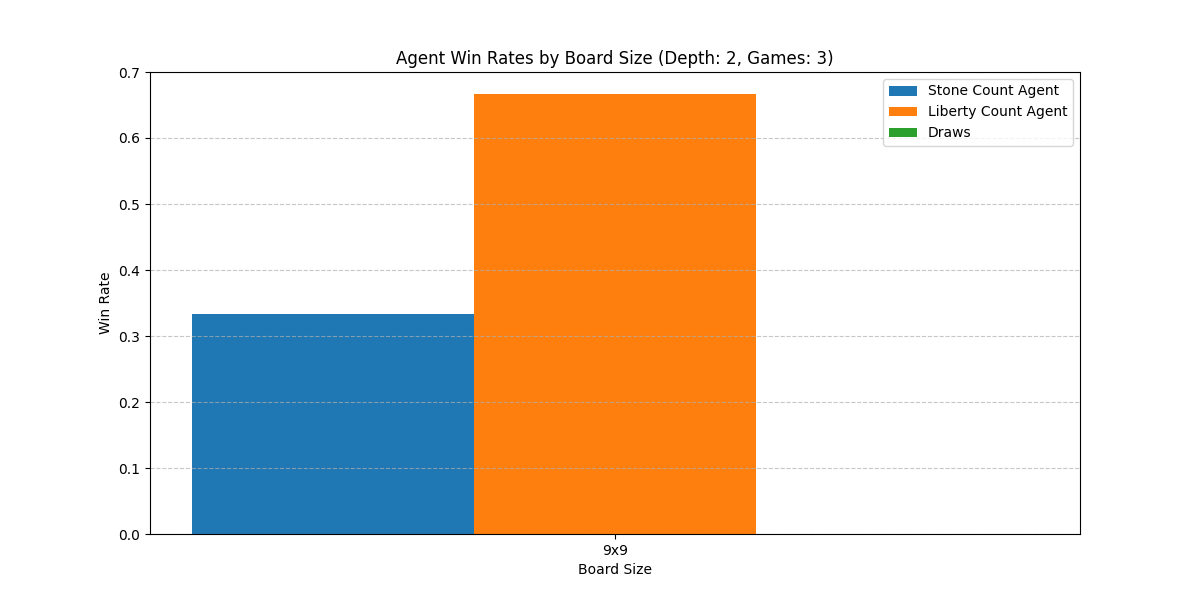
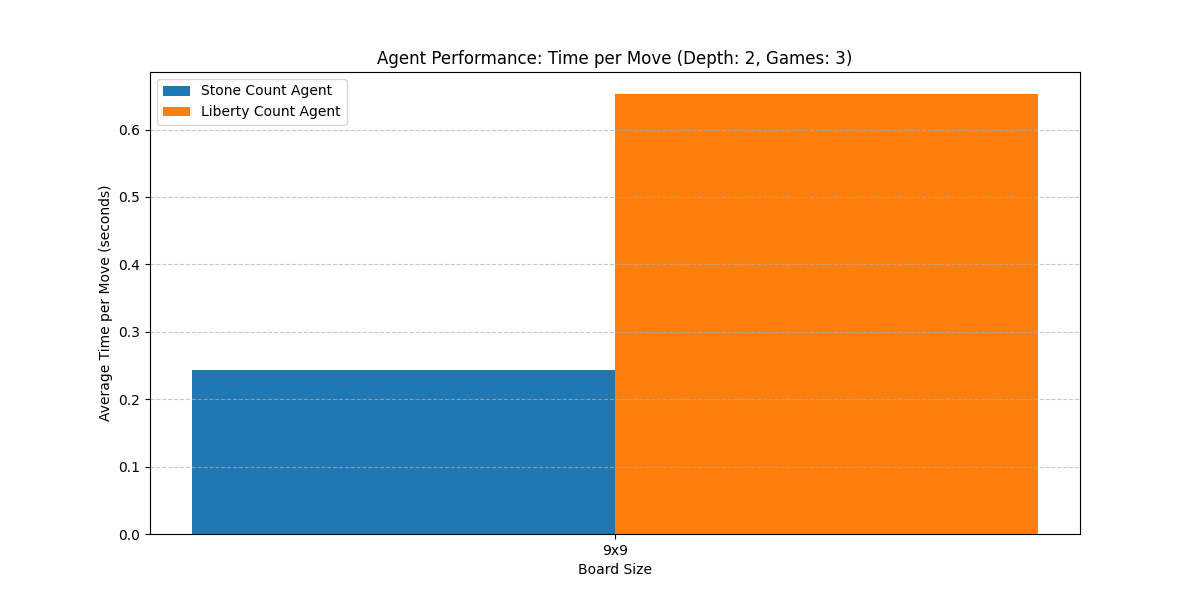
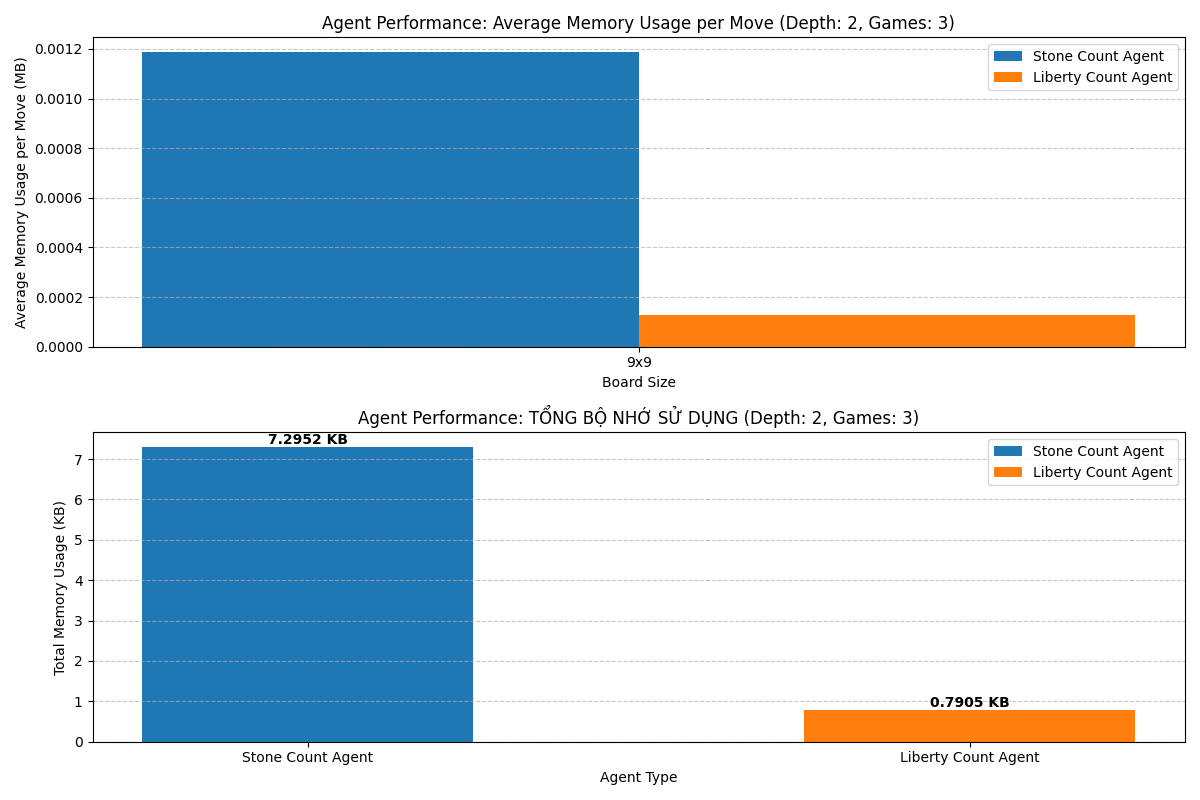
* **Win Rate**: the number of games won divided by the total number of games.
* **Average Time per Move**: measured in seconds.
* **Average Memory Usage per Move**: measured in megabytes (MB).
* **Total Memory Usage**: measured in kilobytes (KB) for the entire match.

**3.3. Experimental Results**

The experimental results are illustrated in the following charts:

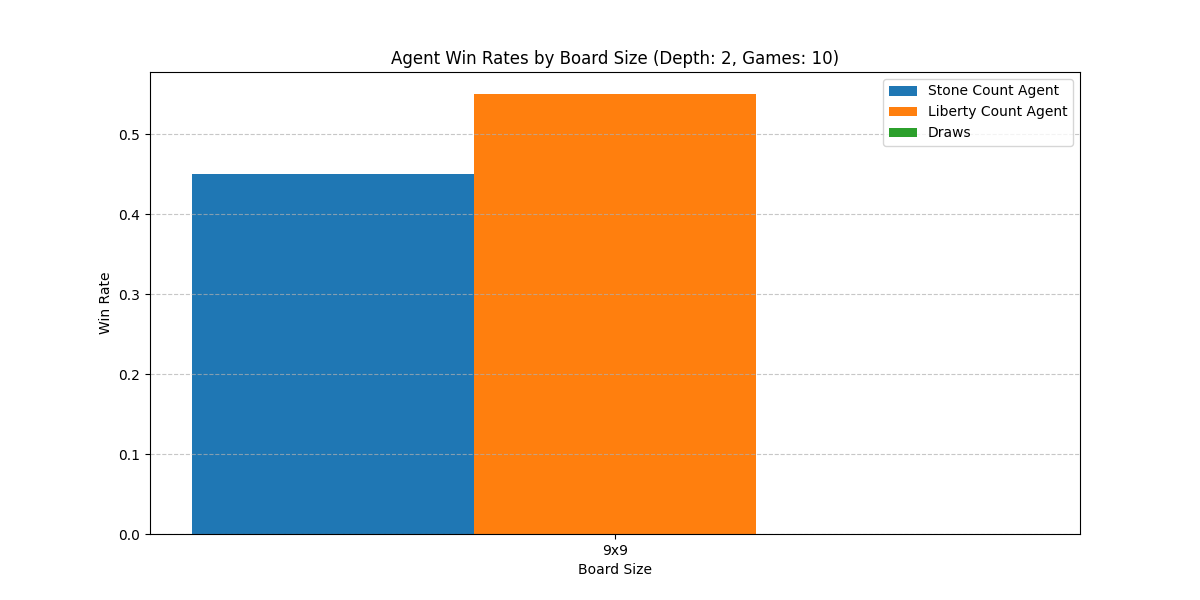
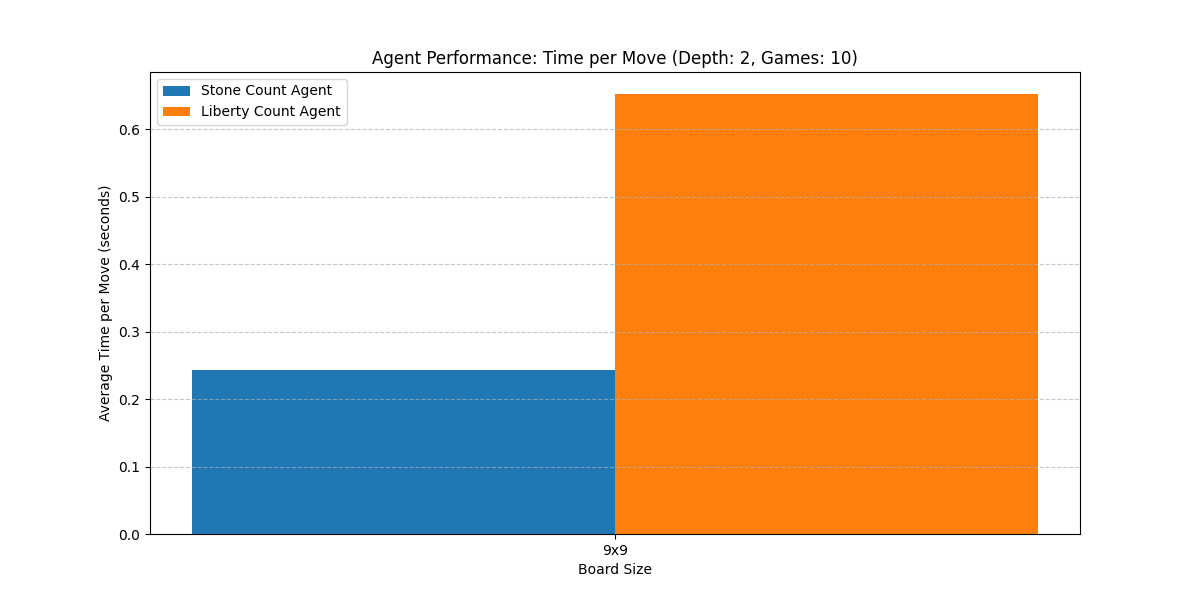
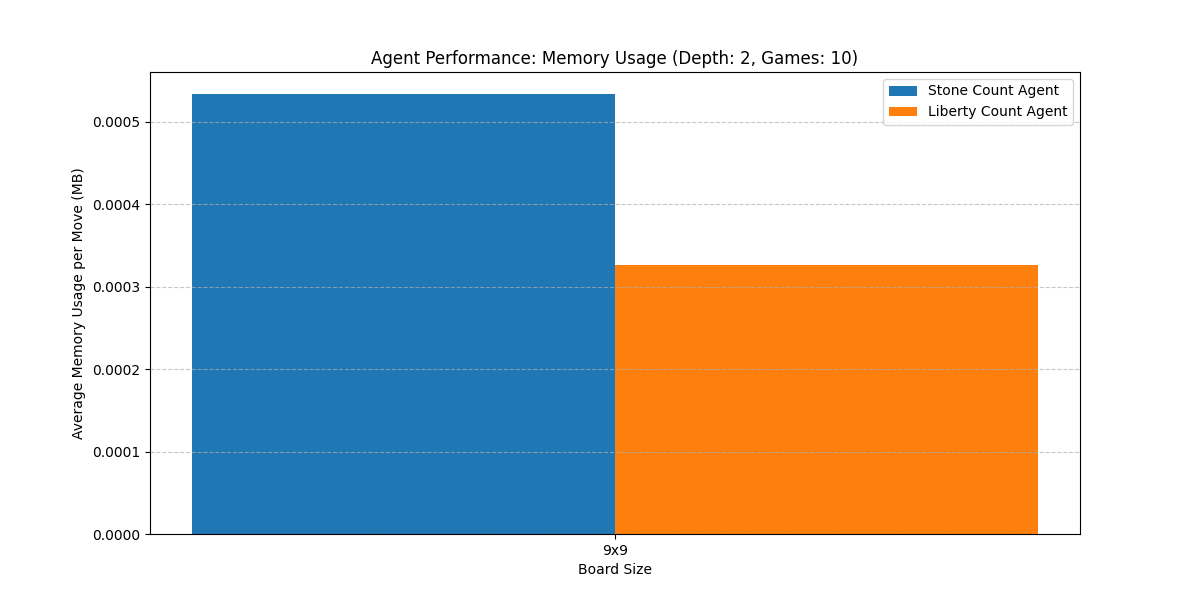
**a. For 3 matches:**

* **Memory**:
  + **StoneCount** used approximately **7.3 KB**, while **LibertyCount** only consumed around **0.79 KB**.
  + **Average per move**: StoneCount used about **0.0012 MB**, whereas LibertyCount’s usage was **significantly lower**.
* **Time**:
  + The **average time per move** for **StoneCount** was approximately **0.25 seconds**.
  + **LibertyCount** was **noticeably slower**, averaging around **0.65 seconds** per move.
* **Win Rate**:
  + **LibertyCount** won about **66.7%** of the games, while **StoneCount** achieved only **33.3%**.



**b. For 10 matches:**

* **Memory:**
  + StoneCount consumed an average of approximately 0.00053 MB per move, while LibertyCount used about 0.00033 MB per move.
  + The total memory usage followed a similar trend as in the 3-game experiment, with LibertyCount being more memory-efficient.
* **Time:**
  + StoneCount maintained a stable execution time of around 0.25 seconds per move.
  + LibertyCount continued to require more time, averaging 0.65 seconds per move.
* **Win Rate:**
  + LibertyCount won approximately 55% of the games, while StoneCount won 45%.



**3.4. Analysis and Discussion**

The results reveal clear differences between the two evaluation functions:

* **In terms of computational efficiency**, *StoneCountEvaluator* runs faster but consumes more memory due to its simplicity (merely counting the number of stones).
* **Strategically**, *LibertyCountEvaluator* is slower but achieves better win rates, indicating its ability to make more informed decisions by assessing the survivability potential of stones in future moves.
* **For system optimization**, in environments with limited resources (e.g., RAM), *LibertyCountEvaluator* may be the more suitable option if longer processing time is acceptable.

Thus, depending on the intended use—whether prioritizing **speed** or **strategic accuracy**—developers can choose the most appropriate evaluation function. This study also opens up new directions for designing **hybrid heuristics** that combine the strengths of both approaches.

# 4. Conclusion

In this report, we developed and implemented a GO-playing AI system using the **Minimax algorithm with Alpha-Beta pruning**, along with two distinct evaluation functions: **StoneCountEvaluator** and **LibertyCountEvaluator**. The experiments were conducted on a 9x9 GO board with a fixed search depth, and performance was assessed using clear metrics such as **win rate**, **computation time**, and **memory usage**.

The results showed that **StoneCountEvaluator** offers advantages in speed and simplicity, while **LibertyCountEvaluator**, though more time-consuming, provides better strategic outcomes—as reflected in its higher win rate and more accurate evaluation of board positions. The choice of evaluation function ultimately depends on the specific objective of the system: **computational efficiency** or **decision quality**.

A limitation of this study is that it was only tested on a 9x9 board with a fixed depth; larger board sizes or deeper searches were not explored. In the future, we hope to develop **hybrid evaluation functions**, or apply **machine learning techniques** to train data-driven evaluation models, with the goal of improving AI performance in more complex games like GO.

# 5. References

[1] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed., Pearson, 2020.

[2] M. Müller, “Computer Go,” *Artificial Intelligence*, vol. 134, no. 1-2, pp. 145–179, Jan. 2002.

[3] D. Silver, A. Huang, C. J. Maddison et al., “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[4] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized Experience Replay,” in *Proceedings of the International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, 2016.