Implementation of Alpha-Beta Pruning Algorithm for Backgammon Game Engine

Muhammad Luqman Hakim (13523044)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13523044@std.stei.itb.ac.id muhluqhakim@gmail.com

***Abstract*—This technical report presents an implementation of a backgammon-playing AI that utilizes the Branch and Bound strategy, specifically through alpha-beta pruning applied to an expectiminimax search tree, to make strategic decisions in a probabilistic environment. Backgammon is a complex, stochastic, two-player board game where optimal decision-making must account not only for the current board state but also for the uncertainty introduced by dice rolls. This AI system models the game as a combination of deterministic player moves and probabilistic dice outcomes, structured in a game tree. By using alpha-beta pruning, the AI efficiently explores this tree by discarding suboptimal branches that cannot affect the final outcome, reducing the computational burden without sacrificing performance. Additionally, the use of domain-specific heuristic functions—such as pip count and blockade potential—enhances the AI's ability to evaluate non-terminal positions and choose moves that maximize strategic advantage. This report outlines the theoretical foundation, design, implementation, and evaluation of this approach.**

***Index Terms*—Backgammon, Game AI, Branch and Bound, Alpha-Beta Pruning, Expectiminimax, Heuristic Evaluation, Probabilistic Games, Game Tree Search, Decision Making, Artificial Intelligence**

I. Introduction

Introduction

Backgammon is one of the oldest known board games, with its modern form originating in 17th-century England [1]. The game presents a compelling challenge for algorithmic decision-making due to its integration of stochastic elements and strategic planning. The central problem addressed in this report is developing an artificial intelligence agent capable of competitive performance in backgammon by effectively managing both probabilistic outcomes and adversarial strategic considerations. Unlike deterministic games such as chess, backgammon requires optimal decisions under uncertainty, where each move must account for probabilistic dice rolls while anticipating opponent responses. This stochastic component introduces significant complexity, as players must evaluate expected outcomes across multiple possible dice combinations rather than deterministic sequences. The objective is to develop and evaluate a search-based artificial intelligence approach that accommodates randomness while maintaining computational efficiency for real-time gameplay.

Backgammon is a race game where two players compete to move fifteen checkers from start to finish before their opponent [2]. The game combines tactical positioning with probabilistic outcomes, requiring strategic decisions under uncertainty created by dice rolls. Players must traverse the board according to dice-determined movement rules, with the ultimate objective being complete checker removal. The game incorporates strategic elements including piece capture, defensive blocking formations, and risk management in positioning. As a zero-sum game, any advantage gained by one player corresponds directly to the opponent's disadvantage, creating a strictly competitive environment suitable for adversarial artificial intelligence techniques.

The algorithmic foundation centers on Expectiminimax, an extension of classical Minimax designed for games with probabilistic elements. Traditional Minimax constructs game trees by alternating between maximizing and minimizing nodes representing optimal player choices under deterministic conditions. Expectiminimax introduces chance nodes to model random events, creating trees that alternate between max nodes for agent decisions, min nodes for opponent responses, and chance nodes for dice outcomes. Each chance node contains branches for possible dice combinations weighted by probability, with evaluation computed through expected values across all outcomes. Computational efficiency is achieved through Alpha-Beta Pruning, which reduces evaluated nodes without compromising decision optimality. The mechanism maintains alpha and beta threshold values representing the best discovered values for maximizing and minimizing players respectively. When a node's value falls outside the alpha-beta window, remaining branches can be safely eliminated. This optimization is particularly valuable for Expectiminimax trees, where branching factors increase substantially due to chance nodes representing dice combinations.

Heuristic evaluation functions provide position assessments when exhaustive search becomes computationally infeasible. These functions must capture essential strategic elements determining position strength, including piece distribution, defensive formations, offensive opportunities, and endgame considerations. Function quality directly impacts agent performance, as inaccurate assessments lead to suboptimal decisions despite optimal search algorithms.

The implementation addresses backgammon's combinatorial complexity through Expectiminimax search with Alpha-Beta Pruning and domain-specific heuristic evaluation func-

tions. The large branching factor due to dice outcomes and movement possibilities necessitates careful algorithmic design for practical performance. Subsequent sections detail game state representation, backgammon rules and mechanics, heuristic evaluation methodologies, and experimental evaluation across various search depths and optimization parameters.RetryClaude can make mistakes. Please double-check responses.

II. Rules of Backgammon

Backgammon is played on a board consisting of twenty-four narrow triangular points arranged in two opposing rows of twelve points each. The board is divided into four quadrants, with each player having a home board and outer board on their respective side. Players begin the game with fifteen checkers positioned according to a standard initial setup, where each player has two checkers on their twenty-four point, five checkers on their thirteen point, three checkers on their eight point, and five checkers on their six point. The objective is to move all checkers into the home board and subsequently bear them off the board entirely before the opponent accomplishes the same goal.

Movement in backgammon is governed by the roll of two six-sided dice, with each die representing an independent movement opportunity that must be utilized if legally possible. Players move their checkers in opposite directions around the board, with each checker advancing a number of points equal to the value shown on one die. When a player rolls doubles, they receive four moves of the indicated value rather than two. Each checker must land on a point that is either empty, occupied by the player's own checkers, or occupied by exactly one opponent checker. A point occupied by two or more opponent checkers is considered blocked and cannot be used as a landing destination.

The hitting mechanism allows players to capture opponent checkers that occupy points alone. When a checker lands on a point occupied by a single opponent checker, the opponent's checker is hit and placed on the bar, which is the raised ridge running down the center of the board. A player with checkers on the bar must enter all such checkers into their opponent's home board before making any other moves. Entry from the bar is accomplished by rolling the dice and moving the checker to a point in the opponent's home board corresponding to the die value, provided that point is not blocked by two or more opponent checkers. If a player cannot enter a checker from the bar because all corresponding points are blocked, they forfeit their turn.

The blocking strategy involves blocking points, which prevents opponent checkers from landing on those points. A sequence of consecutive blocked points forms a prime, which can significantly impede opponent movement. The most effective prime consists of six consecutive blocked points, which completely prevents opponent checkers from passing through that section of the board. Strategic blocking requires careful consideration of checker distribution and timing, as overly defensive play can result in insufficient progress.

The bearing off phase begins when a player has successfully moved all fifteen checkers into their home board. During this phase, checkers are removed from the board according to dice rolls, where each die value corresponds to the numbered point in the home board. A checker on the six point can be borne off with a roll of six, a checker on the five point with a roll of five, and so forth. If no checker occupies the point corresponding to a die value, the player must make a legal move with a checker from a higher-numbered point. If no checkers remain on higher-numbered points, the player may bear off a checker from the highest occupied point. The bearing off process continues until all checkers are removed from the board.

The game concludes when one player successfully bears off all fifteen checkers. A normal win occurs when the opponent has borne off at least one checker, while a gammon results when the opponent has not borne off any checkers. A backgammon, the most valuable win, occurs when the opponent has not borne off any checkers and still has checkers in the winner's home board or on the bar. These different victory conditions affect scoring in match play and determine the stakes in money games.

The doubling cube introduces an additional strategic element that allows players to increase the stakes during the game. The cube displays the values two, four, eight, sixteen, thirty-two, and sixty-four, representing the current stake multiplier. Either player may offer a double before rolling the dice, proposing to double the current stakes. The opponent must either accept the double and continue playing at the higher stakes, or decline and immediately forfeit the game at the current stake level. Once a double is accepted, only the player who accepted the double may offer the next double, creating a sequence of ownership that alternates between players throughout the game.

Strategic considerations in backgammon encompass multiple competing objectives that must be balanced according to the current position and game phase. Early game strategy typically focuses on establishing advanced anchor points in the opponent's board while developing home board structure for potential attacks. The middle game emphasizes the tension between safety and timing, where players must decide whether to prioritize defensive positioning or aggressive advancement based on the race situation and tactical opportunities. Late game strategy centers on the bearing off process, where mathematical calculations of pip counts and efficient checker distribution become paramount. Throughout all phases, players must continuously assess risk-reward trade-offs, considering both immediate tactical opportunities and long-term positional advantages in their decision-making process.

III. Algorithm Overview

The artificial intelligence agent for backgammon employs a sophisticated search algorithm that combines game-theoretic principles with computational optimization techniques to handle the game's inherent probabilistic nature. The core algorithmic framework is built upon Expectiminimax search enhanced with Alpha-Beta Pruning, creating a robust system capable of

evaluating complex game positions while maintaining computational tractability for real-time gameplay.

A. Expectiminimax

Expectiminimax represents a fundamental extension of the classical Minimax algorithm specifically designed to accommodate games that incorporate stochastic elements alongside strategic decision-making [3]. The algorithm addresses the limitation of traditional Minimax, which assumes deterministic outcomes and perfect information, by introducing probabilistic reasoning into the search process. In deterministic games such as chess or checkers, the game tree consists solely of decision nodes where players alternate between maximizing and minimizing their respective outcomes. However, backgammon's inclusion of dice rolls necessitates a more sophisticated approach that can properly model and evaluate uncertain outcomes.

The Expectiminimax framework constructs a game tree that alternates between three distinct types of nodes, each serving a specific purpose in modeling the game's structure. The value of each node type is computed according to the following formulas:

For max nodes representing the maximizing player's decisions:

$$V(n) = \max_{a \in A(n)} V(\text{child}(n, a)) \tag{1}$$

For min nodes representing the minimizing player's decisions:

$$V(n) = \min_{a \in A(n)} V(\text{child}(n, a)) \tag{2}$$

For chance nodes representing probabilistic outcomes:

$$V(n) = \sum_{o \in O(n)} P(o) \cdot V(\text{child}(n, o)) \tag{3}$$

where $A(n)$ represents the set of available actions at node n, $O(n)$ represents the set of possible outcomes at chance node n, and $P(o)$ represents the probability of outcome o [4].

Max nodes represent decision points for the artificial intelligence agent, where the algorithm seeks to select the move that maximizes the expected outcome given the current game state. These nodes correspond to positions where the agent has rolled the dice and must choose how to move its checkers according to the rolled values. The evaluation at max nodes involves comparing all legal moves available for the given dice roll and selecting the option that yields the highest expected value when considering all possible future developments.

Min nodes represent the opponent's decision points, where the algorithm assumes the opponent will select moves that minimize the agent's expected outcome. This assumption reflects the adversarial nature of backgammon, where optimal play requires anticipating that the opponent will make the strongest possible responses to the agent's moves. The evaluation at min nodes involves identifying the opponent's best available move, which from the agent's perspective represents the worst-case scenario that must be planned for. This adversarial modeling ensures that the agent's strategy remains robust against competent opposition.

Chance nodes constitute the unique feature that distinguishes Expectiminimax from classical Minimax, representing the probabilistic outcomes of dice rolls. Each chance node contains branches corresponding to all possible dice combinations, with each branch weighted according to the probability of that specific outcome occurring. In backgammon, there are thirty-six possible dice combinations when rolling two six-sided dice, though only twenty-one distinct outcomes exist due to the commutative property of the dice values. The probability distribution for dice outcomes is given by:

$$P(\text{dice outcome } (i,j)) = \begin{cases} \frac{1}{36} & \text{if } i = j \text{ (doubles)} \\ \frac{2}{36} = \frac{1}{18} & \text{if } i \neq j \text{ (non-doubles)} \end{cases} \tag{4}$$

The evaluation of chance nodes involves computing the expected value across all possible dice outcomes, weighted by their respective probabilities. This calculation ensures that the algorithm makes decisions based on probabilistic reasoning rather than assuming specific dice sequences. The expected value computation at chance nodes represents a critical component of the algorithm's ability to handle uncertainty, as it allows the agent to make informed decisions even when future dice rolls cannot be predicted deterministically.

B. Alpha-Beta Pruning

Alpha-Beta Pruning serves as a crucial optimization technique that dramatically reduces the computational complexity of the Expectiminimax search without affecting the quality of the final decision. The pruning mechanism operates by maintaining two threshold values throughout the search process that represent bounds on the minimax value of the current node [4]. Alpha represents the best value that the maximizing player can guarantee along the current path from the root to the present node, while beta represents the best value that the minimizing player can guarantee along the same path.

The pruning conditions are formally defined as follows. At a min node, pruning occurs when:

$$v \leq \alpha \tag{5}$$

At a max node, pruning occurs when:

$$v \geq \beta \tag{6}$$

where v is the current node value, α is the best value for the maximizer, and β is the best value for the minimizer.

The pruning process exploits the property that if a node's value is determined to lie outside the current alpha-beta window, then the remaining unexplored branches of that node cannot possibly influence the final decision at the root. When the algorithm discovers that a min node has a value less than or equal to alpha, it can immediately terminate the search of that node's remaining children, since the maximizing player would never choose a path that leads to such a poor outcome. Similarly, when a max node has a value greater than or equal

Algorithm 1: Expectiminimax Algorithm

Data: Current game state s, search depth d, player type $player$
Result: Best value for the current position
if $d = 0$ *or* s *is terminal* **then**
  return *Evaluate(s)*;
end
switch $player$ **do**
  case *MAX* **do**
    $bestValue \leftarrow -\infty$;
    foreach *action* a *in GetLegalMoves(s)* **do**
      $newState \leftarrow$ ApplyMove(s, a);
      $value \leftarrow$ Expectiminimax($newState$, $d-1$, CHANCE);
      $bestValue \leftarrow \max(bestValue, value)$;
    end
    return $bestValue$;
  end
  case *MIN* **do**
    $bestValue \leftarrow +\infty$;
    foreach *action* a *in GetLegalMoves(s)* **do**
      $newState \leftarrow$ ApplyMove(s, a);
      $value \leftarrow$ Expectiminimax($newState$, $d-1$, CHANCE);
      $bestValue \leftarrow \min(bestValue, value)$;
    end
    return $bestValue$;
  end
  case *CHANCE* **do**
    $expectedValue \leftarrow 0$;
    foreach *dice outcome* o *in GetPossibleDiceOutcomes()* **do**
      $newState \leftarrow$ ApplyDiceRoll(s, o);
      $value \leftarrow$ Expectiminimax($newState$, $d-1$, GetOpponent(s.currentPlayer));
      $expectedValue \leftarrow expectedValue + P(o) \times value$;
    end
    return $expectedValue$;
  end
end

to beta, the search can be terminated because the minimizing player would never allow the game to reach a state where the maximizing player could achieve such a favorable result.

The alpha and beta values are updated during the search process according to the following rules:

$$\alpha = \max(\alpha, v) \quad \text{at max nodes} \tag{7}$$

$$\beta = \min(\beta, v) \quad \text{at min nodes} \tag{8}$$

The effectiveness of Alpha-Beta Pruning depends significantly on the order in which moves are explored during the search process. When better moves are examined first, the alpha and beta bounds become tighter more quickly, leading to

Algorithm 2: Expectiminimax with Alpha-Beta Pruning

Data: Current game state s, search depth d, player type $player$, α, β
Result: Best value for the current position
if $d = 0$ *or* s *is terminal* **then**
  return *Evaluate(s)*;
end
switch $player$ **do**
  case *MAX* **do**
    $bestValue \leftarrow -\infty$;
    foreach *action* a *in GetLegalMoves(s)* **do**
      $newState \leftarrow$ ApplyMove(s, a);
      $value \leftarrow$ Expectiminimax($newState$, $d-1$, CHANCE, α, β);
      $bestValue \leftarrow \max(bestValue, value)$;
      $\alpha \leftarrow \max(\alpha, value)$;
      if $\beta \leq \alpha$ **then**
        break ; `// Beta cutoff`
      end
    end
    return $bestValue$;
  end
  case *MIN* **do**
    $bestValue \leftarrow +\infty$;
    foreach *action* a *in GetLegalMoves(s)* **do**
      $newState \leftarrow$ ApplyMove(s, a);
      $value \leftarrow$ Expectiminimax($newState$, $d-1$, CHANCE, α, β);
      $bestValue \leftarrow \min(bestValue, value)$;
      $\beta \leftarrow \min(\beta, value)$;
      if $\beta \leq \alpha$ **then**
        break ; `// Alpha cutoff`
      end
    end
    return $bestValue$;
  end
  case *CHANCE* **do**
    $expectedValue \leftarrow 0$;
    foreach *dice outcome* o *in GetPossibleDiceOutcomes()* **do**
      $newState \leftarrow$ ApplyDiceRoll(s, o);
      $value \leftarrow$ Expectiminimax($newState$, $d-1$, GetOpponent(s.currentPlayer), α, β);
      $expectedValue \leftarrow expectedValue + P(o) \times value$;
    end
    return $expectedValue$;
  end
end

more aggressive pruning and consequently better performance. The best-case scenario for Alpha-Beta Pruning reduces the effective branching factor from b to $\sqrt{b}$, where b is the average branching factor, resulting in a time complexity improvement from $O(b^d)$ to $O(b^{d/2})$ for a search of depth d [3].

This principle motivates the use of move ordering heuristics that attempt to prioritize the exploration of moves that are likely to be optimal or near-optimal. In the context of backgammon, effective move ordering might prioritize moves that advance checkers safely, create or extend blocking structures, or improve overall position mobility. Common move ordering techniques include:

$$\begin{aligned}\text{Move Score} = w_1 \cdot \text{Safety} + w_2 \cdot \text{Advancement} \\ + w_3 \cdot \text{Blocking} + w_4 \cdot \text{Hitting}\end{aligned} \tag{9}$$

where w_i are weights determined through empirical analysis or machine learning techniques.

The integration of Alpha-Beta Pruning with Expectiminimax requires careful consideration of how pruning decisions interact with chance nodes. Traditional Alpha-Beta Pruning applies directly to the max and min nodes in the search tree, but chance nodes require special handling since they represent probabilistic rather than adversarial choices. The pruning bounds must be propagated through chance nodes using the expected value calculations, ensuring that the optimization remains sound while preserving the probabilistic reasoning that makes Expectiminimax suitable for stochastic games.

The computational complexity of the combined Expectiminimax with Alpha-Beta Pruning approach depends on several factors including the search depth, the branching factor at each node type, and the effectiveness of the pruning. In backgammon, the branching factor for chance nodes is fixed at twenty-one distinct dice outcomes, while the branching factor for decision nodes varies significantly based on the current position and can range from very few legal moves in constrained positions to dozens of possible moves in open positions. The best time complexity can be expressed as:

$$O(d) = O(b_{chance}^{d/2} \cdot b_{decision}^{d/2}) \tag{10}$$

where $b_{chance} = 21$ is the branching factor for chance nodes, $b_{decision}$ is the average branching factor for decision nodes, and d is the search depth.

IV. Implementation

A. Game State Modeling and Expectiminimax Mapping

Backgammon is modeled as a game tree where each node represents a complete game state $s = \langle B, p, d \rangle$, with B encoding the board configuration across all 24 points, $p \in \{1, 2\}$ indicating the current player, and $d = (d_1, d_2)$ representing the dice roll. The critical insight lies in recognizing that backgammon alternates between deterministic decision phases where players choose moves and stochastic chance phases where dice outcomes determine available actions.

The game tree structure consists of two interleaved node types that form the backbone of the expectiminimax approach. Decision nodes V_D represent positions where players must select from available move sequences, while chance nodes V_C represent the probabilistic dice rolling phase. This creates a repeating pattern where each player's turn begins with a chance node that determines the dice outcome, followed immediately by a decision node where the player selects moves based on those dice values.

The expectiminimax algorithm naturally maps onto this structure by treating decision nodes with standard minimax evaluation and chance nodes with expected value computation. At decision nodes, the maximizing player seeks $\max_{m \in \mathcal{M}(s,d)} V(\delta(s, m))$ while the minimizing player pursues $\min_{m \in \mathcal{M}(s,d)} V(\delta(s, m))$, where $\mathcal{M}(s, d)$ represents all legal move sequences for dice roll d and $\delta(s, m)$ denotes the resulting state after executing move m. At chance nodes, the expected value is computed as $\sum_{d \in D} P(d) \cdot V(\text{child}(s, d))$, where D encompasses all 21 distinct dice combinations and $P(d)$ the probability of each dice roll.

B. Heuristic Evaluation Function

1) Pip Count Heuristic: The pip count heuristic $H_{pip}(s)$ measures the racing advantage between players by computing the total distance all checkers must travel to bear off. This fundamental metric captures the pure racing aspect of backgammon, where the player with fewer pips remaining holds the advantage in positions where contact play has ended or when both players are running their back checkers to safety.

For player p, the pip count is calculated as:

$$\text{Pip}(p) = \sum_{i=1}^{24} \text{checkers}_p(i) \cdot \text{distance}(i) \tag{11}$$

where $\text{checkers}_p(i)$ denotes the number of player's checkers on point i and $\text{distance}(i)$ represents the minimum distance from point i to bearing off. The pip count heuristic then becomes

$$H_{pip}(s) = \text{Pip(opponent)} - \text{Pip(current player)} \tag{12}$$

, making positive values favorable for the current player. This heuristic becomes increasingly important as the game progresses toward the endgame, where positional considerations give way to pure racing dynamics.

2) Blot Vulnerability Heuristic: The blot vulnerability heuristic $H_{blots}(s)$ quantifies the exposure risk by evaluating unprotected checkers that can be hit by the opponent. A blot represents a single checker on a point, making it vulnerable to attack and potentially sending it to the bar, which significantly disrupts the player's development and timing.

$$H_{blots}(s) = \sum_i \text{blot}(i, \text{opponent}) - \sum_i \text{blot}(i, \text{current player}) \tag{13}$$

where $\text{blot}(i)$ is 1 when exactly one of the player's checker is in that point and 0 otherwise. The heuristic indicates that

blots represent a liability for the current player, hence the current (maximizing player) should minimize his number of blots. This heuristic is particularly crucial during the opening and middle game phases when contact between opposing forces is frequent and tactical hitting opportunities abound.

3) Blockade Strength Heuristic: The blockade strength heuristic $H_{blockade}(s)$ evaluates the defensive and offensive potential of consecutive occupied points, which form prime structures that restrict opponent movement. Blockades serve dual purposes in backgammon strategy: they prevent opponent checkers from advancing and create safe landing zones for the player's own checkers.

The blockade evaluation rewards longer consecutive sequences of controlled points:

$$H_{blockade}(s) = \max(\text{length of blockade(current player)}) - \max(\text{length of blockade(opponent)}) \quad (14)$$

The most valuable blockades are those that form a continuous prime of six points, creating an impenetrable barrier that completely blocks opponent movement.

C. Combined Heuristic Framework

These three heuristic components can be effectively combined into a unified evaluation function that captures the multifaceted nature of backgammon strategy. The combined approach takes the form:

$$H(s) = w_1 \cdot H_{pip}(s) + w_2 \cdot H_{blots}(s) + w_3 \cdot H_{blockade}(s)$$

where the weights w_1, w_2, and w_3 can be dynamically adjusted based on the current game phase and position characteristics. During opening play, blot safety and blockade formation typically receive higher emphasis, while racing elements become dominant in endgame scenarios. The adaptive weighting allows the evaluation function to shift strategic priorities as the game evolves, ensuring that the AI's decision-making remains contextually appropriate throughout all phases of play. This modular approach also facilitates the incorporation of additional heuristic components as needed, making the evaluation framework extensible and refinable through empirical testing and domain expertise.

V. Result

The experimental evaluation demonstrates that the alpha-beta pruned expectiminimax algorithm provides improvements over the baseline greedy approach, even when constrained to shallow search depths. The performance metrics were obtained through testing against a greedy algorithm baseline that employed identical heuristic evaluation functions, ensuring that any observed performance differences could be attributed solely to the search strategy rather than evaluation function disparities.

With a maximum search depth of 2 plies and 500 independent game trials, the Branch and Bound implementation achieved a win rate of $56.4\%(p = 0.002)$ against the greedy baseline. This represents a meaningful competitive advantage, with the AI system winning 282 out of 500 games compared to the greedy algorithm's 218 victories. The average normalized score differential further substantiates this performance gap, with the alpha-beta implementation averaging 0.592 points per game compared to the greedy algorithm's 0.458 points per game. This 29.2% advantage in average scoring demonstrates consistent strategic superiority across the test suite.

The statistical significance of these results is particularly noteworthy given the limited computational depth employed. The shallow 2-ply search constraint was deliberately chosen to evaluate the algorithm's efficiency in time-critical scenarios while maintaining practical applicability for real-time gameplay. Despite this limitation, the expectiminimax approach with alpha-beta pruning successfully identified superior move sequences that the myopic greedy strategy failed to discover.

In addition to its performance against the greedy baseline, the algorithm was also evaluated against a purely stochastic opponent that selected moves uniformly at random. In this configuration, the alpha-beta pruned expectiminimax agent demonstrated overwhelming dominance, achieving a win rate of 95.4% over 500 simulated matches. The disparity in effectiveness is further highlighted by the average normalized score: the algorithm achieved a mean score of 2.0 points per game, while the random agent averaged just 0.048. This result underscores the algorithm's ability to consistently exploit uncoordinated play patterns, identifying and capitalizing on suboptimal moves with high precision. The pronounced scoring gap reflects not only a high frequency of wins but also a consistent pattern of decisive victories. These findings validate the robustness of the expectiminimax strategy in both adversarial and chaotic environments, where the absence of coherent opposition allows the heuristic-guided search to dominate through calculated positioning and probabilistic foresight.

The domain-specific heuristic functions, including pip count differential, positional advantage metrics, and blockade potential assessment, proved effective in guiding the search toward strategically sound positions. The integration of these heuristics with the probabilistic expectiminimax framework successfully balanced immediate tactical gains against longer-term positional advantages, contributing to the observed performance improvements.

VI. Conclusion

This study has shown that framing Backgammon decision-making as a bounded expectiminimax search enhanced by alpha-beta pruning yields a practical yet robust AI capable of strategic play under uncertainty. The Branch and Bound paradigm, realized through alpha-beta pruning, effectively constrains the search space, allowing the agent to reach deeper evaluation horizons and thereby improve move quality. Heuristic functions capturing pip distance and blockade strength provide meaningful guidance at depth-limited leaf nodes, producing outcomes that outperform simpler evaluators. Future work will explore adaptive heuristics via reinforcement learning, transposition tables for state re-use, and dynamic depth adjustment based on real-time performance metrics,

further advancing the efficacy of algorithmic strategies in stochastic adversarial games like Backgammon.

VII. Acknowledgment

I would like to express my sincere gratitude to God Almighty for His guidance, who made the completion of this paper possible.

References

[1] Cram, D., & Forgeng, J. L. (2017). Francis Willughby's Book of Games. https://doi.org/10.4324/9781315255040
[2] Parlett, D. S. (1999). The Oxford history of board games. http://ci.nii.ac.jp/ncid/BA41511831
[3] Russell, S. J., Norvig, P., & Davis, E. (2010). Artificial intelligence: A Modern Approach. Prentice Hall.
[4] Michie, D. (1966). GAME-PLAYING AND GAME-LEARNING AUTOMATA. In Elsevier eBooks (pp. 183–200). https://doi.org/10.1016/b978-0-08-011356-2.50011-2

Appendix A

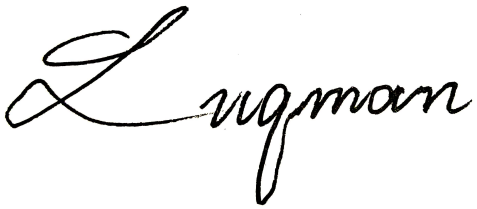
Complete Implementation of the Program

The complete implementation of the program can be accessed at https://github.com/carasiae/bg-pmo

STATEMENT

I hereby declare that the paper I wrote is my own writing, not an adaptation, or translation of someone else's paper, and not plagiarized.

Bandung, 2 January 2025



Muhammad Luqman Hakim