AI Final project report

Github code

Team Members:
Mona Zoabi, ID: 212973481
Noor Khamaisy, ID: 212809925
Faheem Francis, ID: 209177575

TABLE OF CONTENTS:

Introduction:

In this project, we aim to optimize the performance of iRobots in a room-cleaning scenario. The challenge involves two iRobots competing to clean a room as efficiently as possible. Each robot aims to gain more points by collecting the awards (dirt) and loses points for taking more than the decided time for making a move or being stuck between cells that have already been cleaned. The objective is to determine which robot model can maximize the cleaning efficiency and minimize the number of moves, effectively comparing different models under the same conditions.

The problem of optimizing iRobot performance in cleaning a room is both interesting and practical. It represents a real-world application of multi-agent search algorithms, where multiple autonomous agents operate in a shared environment with conflicting goals. In this case, the iRobots are the agents that need to navigate and clean the room while minimizing redundant actions and maximizing their cleaning score. This scenario not only mimics real-life situations like household cleaning but also explores competitive dynamics in multi-agent systems, which is a growing area of research in robotics and artificial intelligence.

To address this problem, we have chosen to model it using multi-agent search algorithms, specifically the Minimax algorithm ,its enhancement with Alpha-Beta pruning, and Q-Learning algorithm. These algorithms allow the iRobots to make strategic decisions by considering the potential moves of the opposing robot, effectively planning several steps ahead to maximize their score while minimizing the opponent's score. We also implemented a global time management strategy to handle the limited time each iRobot has for decision-making, ensuring that the robots make timely and efficient moves.

The project involves simulating various models of iRobots, each with a different decision-making algorithm, on a board that represents a real-world environment. The board simulates a realistic scenario where both agents are actively competing to achieve the highest score. By experimenting with different algorithms, we aim to determine which approaches lead to the best performance in terms of the number of cells cleaned, the number of moves taken, and the overall score.

In summary, this project seeks to explore the capabilities of multi-agent search algorithms in a competitive setting, focusing on optimizing the performance of iRobots in a room-cleaning task. Through this work, we aim to contribute valuable insights into the application of artificial intelligence in robotics, particularly in scenarios that require strategic planning and real-time decision-making.

Previous Work

Our game introduces a new concept, yet it shares similarities with existing challenges in the fields of robotics and artificial intelligence, particularly in optimizing autonomous agents. The problem of optimizing the behavior of agents, such as iRobots in shared environments like room cleaning, has been thoroughly explored. Numerous algorithms have been developed to enhance the performance of these agents, focusing on pathfinding, optimization, and decision-making strategies in both single-agent and multi-agent settings. While our game builds on these foundational concepts, it presents unique mechanics and challenges, offering fresh perspectives in a familiar domain.

To enhance performance, alternative algorithms such as **AlphaBeta pruning** have been applied. AlphaBeta builds upon Minimax by eliminating branches of the search tree that are unlikely to affect the final decision, thus significantly reducing the number of states evaluated and improving computational speed. While this technique has been a staple in AI game theory, its efficiency depends on the quality of move ordering and the specific problem's complexity.

Evolutionary algorithms like **Genetic Algorithms (GAs)** have also been explored to manage large search spaces in decision-making environments. GAs begin with a random set of solutions and evolve these through mechanisms like selection, crossover, and mutation, continually refining guesses based on feedback from previous outcomes. While GAs provide flexibility in exploration, they tend to require more moves than Minimax-based methods but can offer improved adaptability in dynamic environments.

Our project also draws on strategies from multi-agent optimization, where algorithms like **Monte Carlo Tree Search (MCTS)** are used. MCTS, known for its success in games like Go, balances exploration of new strategies with exploitation of known good moves by simulating many possible outcomes from the current game state. This algorithm is particularly well-suited for handling the vast decision trees that our game presents, ensuring a more flexible and scalable approach to problem-solving.

In summary, while Minimax serves as the benchmark for strategy optimization in our project, its enhancement through AlphaBeta pruning improves both performance and computational efficiency by reducing the number of evaluated states. Additionally, Q-learning introduces a reinforcement learning approach that adapts strategies over time based on accumulated experience, further enhancing decision-making in dynamic environments. These methods ensure our approach remains robust and efficient across a wide range of game scenarios.

Common Assumptions in previous works:

1. **Static Environments:** Many studies assume that the environment is static, meaning that obstacles and dirty cells do not change during the cleaning process. This assumption simplifies the problem, allowing researchers to focus on optimizing pathfinding and decision-making without the additional complexity of a dynamically changing environment.

2. **Perfect Information:** It is often assumed that the agents have complete knowledge of the environment, including the location of all obstacles, dirty cells, and the opponent. This assumption eliminates uncertainty, making it easier to model the agents' decision-making processes.

3. **Deterministic Actions**: Most models assume that the actions of the agents are deterministic, meaning that each action leads to a predictable outcome. This simplifies the analysis and optimization of the algorithms, as there is no need to account for stochastic or unpredictable elements.

Our project aims to build on this previous work by comparing different iRobot models, each implementing a unique search algorithm. The focus is on evaluating their performance in a competitive room-cleaning scenario, where the environment is not static. Unlike traditional setups, the dirt on the board can change with each round, and the starting positions of the players also vary from one turn to the next. We assume that the iRobot is aware of the dirt positions at all times, which adds an element of strategic decision-making. This dynamic environment introduces additional complexity, making it crucial to test a range of configurations and strategies, including the Minimax algorithm, Alpha-Beta pruning, and potentially other heuristic-based methods. Through this comparative analysis, we hope to identify the best-performing iRobot model and gain valuable insights into the most effective algorithms for multi-agent environments under these changing conditions.

Model Description:

The goal of this project is to compare three AI models—MinimaxPlayer, AlphabetaPlayer, and QLearningPlayer—each designed to control iRobots tasked with cleaning a room. These models use different algorithms to optimize the robots' actions to maximize their cleaning efficiency and overall score. Each agent has a specific way of interacting with the environment, making decisions, and improving its performance over time. Below, we will discuss each of these models, their underlying assumptions, and the criteria for their success.

MinimaxPlayer: The MinimaxPlayer uses the Minimax algorithm, a well-known strategy in game theory and multi-agent systems. In this setting, the robot plays against either another robot or a human-controlled player, and its task is to maximize its cleaning score while minimizing the score of the opponent. The Minimax algorithm simulates all possible moves for both the player and the opponent and evaluates the resulting game states. Based on these simulations, the robot selects the move that maximizes its score while assuming the opponent will also play optimally to minimize the robot's score.

The MinimaxPlayer runs a tree search, evaluating moves based on a utility function, which assigns values to board states depending on the number of cleaned cells and penalties for inefficient actions. By simulating several moves ahead, it can anticipate the opponent's strategy and act accordingly.

Heuristic function value: ***h(s, p, d)***, where s(m*n) is the board. player p positioned at cell (i-p, j-p), the opponent player at cell (i-d, j-d)

Evaluation function components:

Player Position and Dirt Distance:

- The heuristic first assesses the relative positions of the players and the objects (dirts) on the board. The aim is to minimize the distance between the player and the dirt, as acquiring the dirt is likely the goal.

Path Length Between Players:

- The minimax also seems to factor in the number of steps between players. A function
- $y(s,p,d) = 4 - \#posible\ steps\ from\ (i_d, j_d)$ defines the number of possible steps or directions from a player's current position, and the board setup affects the paths that can be taken.
- The formula **`value(dirt) / min(Dist(dirt))`**evaluates the value of a dirt and how far the player is from it. The goal is to minimize the distance to the nearest dirt.
- The function

$$f(s,p,d) = |\{(i,j) | there\ is\ a\ white\ path\ from\ (i_p, j_p) to\ (i,j) \cap s(i,j) = 0\}|$$

checks if there is a direct path between two players. This might be useful for calculating how likely it is for players to intercept each other or block moves.

Obstacles and Penalties:

- Penalties are introduced when no valid paths are available between players or towards the dirt. The formula includes a term **`r(s, p, d`**) that assigns a penalty score when a player is blocked from proceeding along a valid path.

Score and Distance Maximization:

- The function **`h(s,p,d)`** calculates the score based on a combination of factors: the player's current position, the distance to dirt, and potential blocking paths by the opponent.

This heuristic is designed to help the agent make strategic decisions based on both the player's positioning and the current state of the game.

AlphabetaPlayer: AlphabetaPlayer is an optimized version of MinimaxPlayer. It uses the Alpha-Beta pruning technique to reduce the number of game states that need to be evaluated. Alpha-Beta pruning eliminates branches in the decision tree that cannot influence the final decision, effectively speeding up the decision-making process without compromising the quality of the outcome.

In a real-world scenario like this, where computational time is limited, Alpha-Beta pruning allows the robot to consider deeper game trees within the same time constraints. This leads to more informed decisions in competitive environments. The core mechanics of AlphabetaPlayer remain the same as MinimaxPlayer, with the key difference being its improved efficiency in evaluating potential moves.

QLearningPlayer: The QLearningPlayer is based on a reinforcement learning (RL) framework, where the robot learns from its interactions with the environment rather than relying on a predefined strategy. The Q-learning algorithm allows the robot to adapt its behavior based on rewards received for its actions. The agent starts with little knowledge of the environment and gradually learns the optimal policy by exploring different actions and receiving feedback in the form of rewards or penalties.

In this case, the QLearningPlayer receives a positive reward for cleaning a dirty cell and a negative reward for attempting to clean an already cleaned cell. Over time, the agent learns which actions yield the highest cumulative reward by updating its Q-values, which represent the expected future rewards for each action in a given state. Unlike MinimaxPlayer and AlphabetaPlayer, the QLearningPlayer does not rely on lookahead simulations but learns through experience, making it highly adaptable to changing room layouts and opponent behavior.

Q-Learnings' parameters:

`learning_rate` (default = 0.2):

- The learning rate (α) controls how much new information overrides the old Q-values. A learning rate of 0.2 means the agent will update its knowledge gradually, balancing between what it knows from past experience and the new information from the environment.
- A lower value (like 0.2) helps to stabilize learning, avoiding large shifts in knowledge, but may require more time to converge to an optimal policy. This allows it to balance past experience with new data and can help prevent large fluctuations in the Q-values, but it may require more iterations to converge to the optimal policy.

`discount_factor` (default = 0.5):

- The discount factor (γ) dictates how much the agent values future rewards versus immediate rewards. A **discount factor (γ) = 0.5** means that the agent values immediate rewards more than future rewards, but it still considers future rewards to some extent.
- This parameter must be high in environments where long-term planning yields better results (e.g., complex games where short-term rewards might lead to suboptimal long-term outcomes).

`exploration_rate` (default = 0.5):

- This is the initial exploration rate, determining how often the agent takes random actions instead of the best-known action (according to its Q-values). Exploration helps the agent discover new strategies that could lead to better long-term results.
- A value of 0.5 means that, initially, the agent will explore new actions 50% of the time. This encourages a balance between exploration and exploitation early in training.

Assumptions:

To build and evaluate these models, several assumptions are made about the environment and the robots' interactions:

Fully observable environment: Each agent has complete information about the room layout, the location of dirt on the floor, and the positions of all players. This assumption simplifies the decision-making process by eliminating the need for hidden state variables or uncertainty in observation.

Deterministic actions: Every action taken by an agent (such as moving to a cell or cleaning it) leads to a deterministic outcome. There is no randomness in the game mechanics, which means that agents can fully predict the consequences of their actions, making strategic planning more effective.

Turn-based competition: Agents alternate moves, and only one agent can clean a cell at a time. If both agents attempt to clean the same cell, a predefined rule determines which one succeeds.

Time constraints: Each agent is limited by a global time budget, meaning they must make their moves quickly and efficiently to maximize their performance. This introduces an additional layer of complexity, as the agents must balance exploration with decision-making speed.

Success Criteria:

To evaluate the success of each player, we define a set of criteria based on the reward function and the overall performance of the agents in cleaning the room. The reward function, which is central to evaluating the quality of each agent's decisions, is composed of the following factors:

Points for cleaning dirty cells: Every time an agent successfully cleans a cell with dirt on it, they receive a positive reward. The total number of cleaned cells directly contributes to the agent's score, making this the primary objective for each player.

Losing points for delayed decisions ensures that agents are penalized for taking too long to act. If an agent exceeds the set time limit for making a move, it incurs a penalty, encouraging faster decision-making. This rule helps balance decision quality with speed, ensuring agents prioritize timely actions.

Final score: The overall performance of each agent is measured by its final score, which is a combination of the points gained from cleaning cells and penalties incurred. The agent with the highest score at the end of the game is considered the most successful.

Evaluation:

To evaluate the quality of the solution, the following key factors should be considered:

1. **Agent Performance Metrics**:

   Score Calculation: The final score reflects an agent's ability to clean dirty cells while minimizing penalties. Higher scores indicate more effective cleaning with fewer penalties.

   Time Management: Agents that consistently make decisions within the time limit demonstrate effective time management, balancing decision quality and speed.

2. **Efficiency**: Decision-Making Speed: Agents that make faster decisions without sacrificing performance indicate an efficient use of computational resources.
3. **Strategy Optimization**:Consistency: An agent that maintains a high score over multiple rounds suggests a reliable strategy.

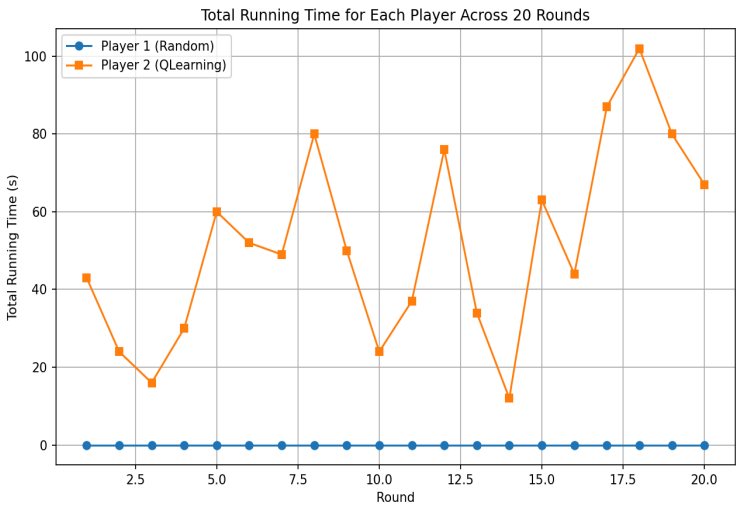
4. **Comparison to Random player**:The solution should be compared to random player, focusing on both the quality of decisions and computational efficiency.

By considering these factors, we can assess whether the solution successfully balances decision-making accuracy, speed, and adaptability to real-time constraints.

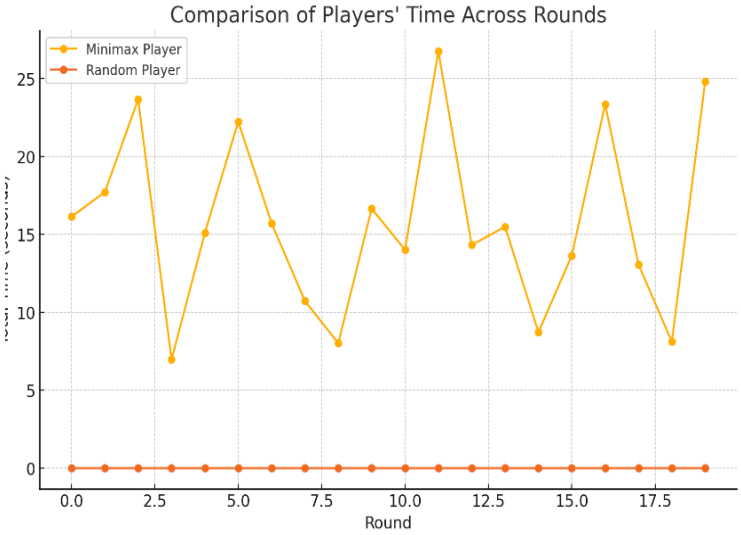
Results:

Time Comparison:

(Plot 1)



(Plot 2)

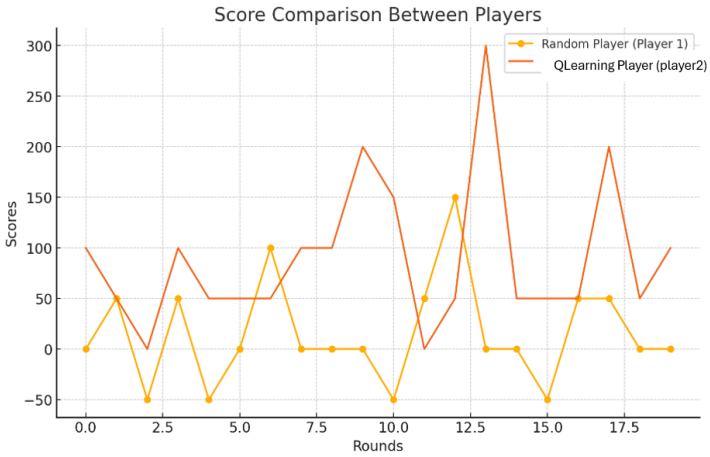


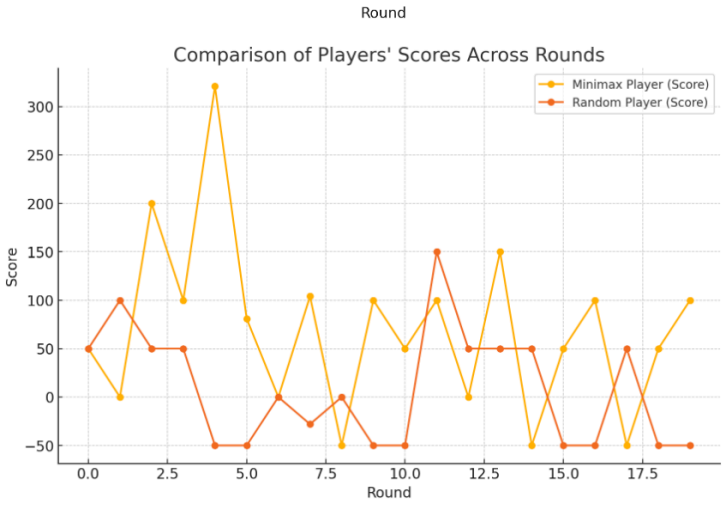
The reason QLearning has a higher running time compared to Minimax lies in the inherent complexity and nature of the algorithms. QLearning, being a reinforcement learning algorithm, involves both exploration and exploitation, especially in its early stages, where it takes time to explore various actions and learn optimal policies. This process leads to more computation, as it updates the Q-table for each state-action pair based on the environment's feedback. Additionally, QLearning adapts dynamically to the opponent's strategy over time, which adds to its computational cost. In contrast, Minimax follows a predefined strategy, using a game tree to compute the best move without needing to learn from experience. This deterministic approach leads to more predictable and consistent running times, especially when optimized with techniques like Alpha-Beta pruning. Thus, QLearning's adaptive learning nature generally results in higher running times compared to Minimax's direct and structured decision-making process.

In plot 1 and 2 The Random player, on the other hand, makes quick decisions, as it simply selects moves randomly without deep computation.

Score Comparisons :

Score comparison between Q-Learning player vs. Random player and between Minimax player vs. Random player:



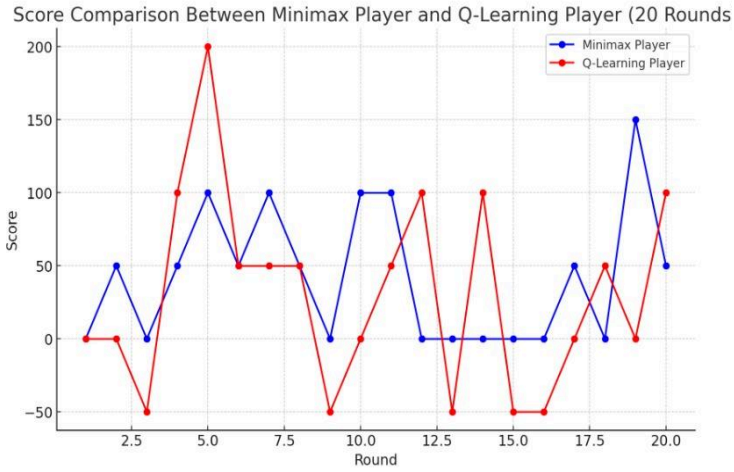


In the comparison between Q-learning and Minimax playing against a random player, the higher

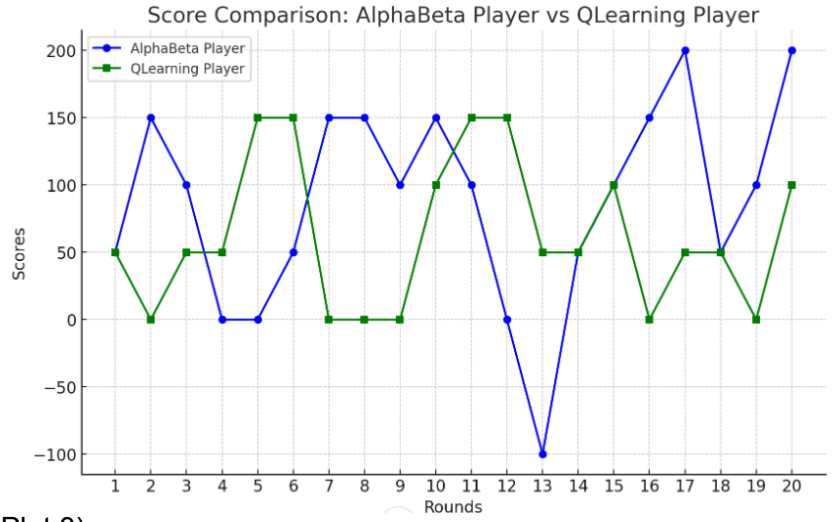
score achieved by Q-learning is primarily due to the nature of how these algorithms function. Q-learning is a reinforcement learning technique that improves its strategy over time by learning from its experiences in each round. It adjusts its actions based on rewards received, which helps it to optimize its performance gradually as it explores and exploits the most effective moves against the random player's unpredictable actions. On the other hand, the Minimax algorithm is more static and works by anticipating the worst-case scenario moves of its opponent, which is highly effective when playing against a rational, strategic opponent. However, when playing against a random player, Minimax may not be as adaptable, as it cannot effectively predict the random player’s moves and thus cannot fully optimize its strategy. Therefore, Q-learning's ability to learn and adapt over multiple turns allows it to perform better and achieve higher scores against a random player.

- **Score comparison between Minimax player vs. Alphabeta player, and between Ahpabeta player vs. QLearning player:**

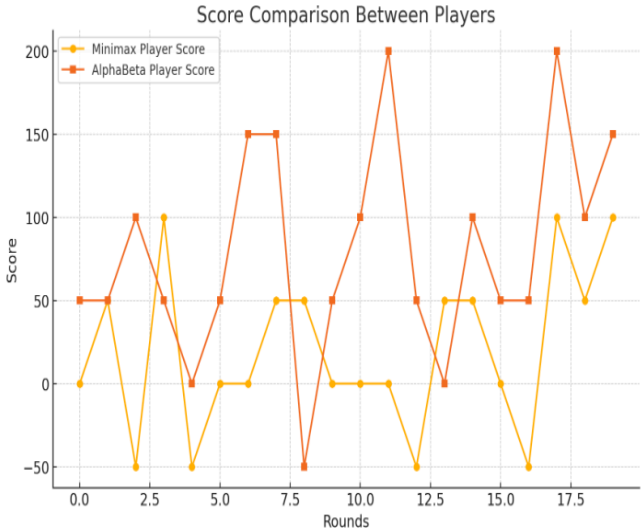
(Plot 1)



(Plot 2)



(Plot 3)



Plot 1:

The results are due to the fundamental differences between Q-Learning and Minimax strategies. Q-Learning, being a reinforcement learning algorithm, initially explores various strategies, leading to the volatile performance seen with large score spikes and dips as it learns through trial and error. This explains the high variability in the Q-Learning Player's scores. In contrast, the Minimax algorithm is deterministic, making decisions based on a pre-defined strategy that minimizes potential losses by evaluating possible future states. As a result, the Minimax Player's performance is more consistent but lacks the dramatic swings seen in the Q-Learning Player.

Plot 2:

The AlphaBeta player performs better because it uses a more sophisticated search algorithm that prunes unnecessary branches in the decision tree, allowing it to focus on the most promising moves and anticipate future actions more effectively. AlphaBeta pruning optimizes the Minimax algorithm, enabling the player to evaluate deeper game states, leading to better decision-making in complex scenarios. On the other hand, Q-Learning, being a reinforcement learning method, takes longer to converge to an optimal strategy as it relies on trial and error to update its policy. While Q-Learning improves over time, its performance can be less effective in short-term competitions where strategic foresight, as used by AlphaBeta, offers an advantage.

Plot 3:

The Score Comparison Between Players plot highlights the performance of both Minimax and AlphaBeta players across multiple rounds. As shown in the graph 3, the AlphaBeta player consistently outperforms the Minimax player in most rounds. This is evident from the fact that the AlphaBeta player's score (represented by the line with square markers) frequently exceeds that of the Minimax player (shown with circular markers).

In several instances, the AlphaBeta player not only scores higher but also maintains a significant lead. This indicates that AlphaBeta's pruning strategy, which reduces the search space, allows for more efficient decision-making compared to Minimax's exhaustive search approach. The result is that AlphaBeta wins more frequently, as reflected by the higher scores in the majority of rounds.

In a few rounds, the scores of both players are closer, and occasionally, Minimax performs better. However, the overall trend shows that AlphaBeta's optimization gives it a competitive edge, leading to more consistent victories.

Summary:

Problem Overview

This project aims to optimize the performance of **iRobots in a room-cleaning task**, where two robots compete to clean a room as efficiently as possible. Each robot is tasked with maximizing its score by cleaning dirty cells while minimizing redundant moves and avoiding penalties for exceeding time limits or getting stuck. The project tests and compares three different algorithms: ***MinimaxPlayer***, its enhancement: ***AlphabetaPlayer, and QLearningPlayer***, using multi-agent search techniques to determine which approach results in superior room-cleaning efficiency.

The main goal is to evaluate the performance of these algorithms under identical conditions, focusing on their ability to clean as many cells as possible within the time constraints, and to avoid penalties associated with inefficient movements. By simulating multiple scenarios, the project highlights the strengths and weaknesses of each algorithm.

Model and Methodology

1. MinimaxPlayer:

The MinimaxPlayer uses the **Minimax algorithm**, a strategy rooted in game theory. It evaluates each potential move by considering all possible outcomes, both for itself and for the opponent. MinimaxPlayer aims to maximize its own score while minimizing the potential gains of its competitor.

The algorithm runs a tree search that simulates several moves ahead and uses a heuristic to decide the best strategy.

2. AlphabetaPlayer:

AlphabetaPlayer builds on Minimax by incorporating Alpha-Beta pruning, an optimization that cuts off branches of the decision tree that are unlikely to influence the outcome. This drastically reduces the number of states evaluated and improves computational efficiency.

By focusing only on the most promising moves, AlphabetaPlayer can delve deeper into the decision tree, leading to smarter decisions in the same amount of time.

3. QLearningPlayer:

QLearningPlayer relies on Q-learning, a reinforcement learning approach. Unlike Minimax and Alphabeta, which use lookahead strategies, QLearningPlayer learns from its environment. It adjusts its policy over time based on the rewards it receives from successful or failed actions.

The robot adapts its behavior over several rounds, improving its strategy by balancing exploration (trying new strategies) and exploitation (capitalizing on what it has learned).

Results and Comparisons

1. Time Efficiency:

- MinimaxPlayer and AlphabetaPlayer showed more consistent decision-making times because of their deterministic nature. MinimaxPlayer, while slower than AlphabetaPlayer, was stable, owing to its comprehensive but slower tree evaluations.

QLearningPlayer had a higher running time, especially early in the process, due to its exploration phase. Over time, as it learned the optimal strategies, the computational load decreased but remained higher than its counterparts.

2. Score Comparison:

- QLearningPlayer outperformed both Minimax and Alphabeta when playing against a random opponent. Its adaptability allowed it to explore and exploit strategies effectively, adjusting to the random player's unpredictable moves.

- In contrast, the MinimaxPlayer did not perform as well against a random player because its approach is based on predicting an opponent's rational behavior, which is less effective when faced with random decisions.

AlphabetaPlayer consistently outperformed MinimaxPlayer in head-to-head matches. The pruning technique allowed Alphabeta to make more informed decisions, enabling it to explore deeper possibilities and optimize moves more efficiently.

Critique and Limitations:

One limitation of the study is the assumption of perfect information. In real-world applications, agents may not have complete knowledge of their environment or opponent's moves. Removing this assumption could introduce more complexity and affect performance.

Another limitation is that the environment is static, meaning that the dirt on the floor and obstacles do not change over time. Real-world scenarios might involve dynamic changes, like obstacles moving or dirt reappearing, which could impact the effectiveness of the models.

The heuristics used by MinimaxPlayer and AlphabetaPlayer are based on non static factors like distance to dirty cells and pathfinding. If these heuristics were to vary or adapt based on dynamic factors, the performance might change significantly.

Alternative Approaches:

Several alternative algorithms could potentially address the limitations or offer new avenues for optimization:

- **Genetic Algorithms (GA):** could be applied to evolve strategies across multiple iterations, providing flexibility in exploration and adaptability in changing environments.
- **Monte Carlo Tree Search (MCTS):** offers a way to handle vast decision trees by randomly sampling outcomes, which could improve performance in more complex or uncertain environments.
- **Deep Reinforcement Learning (DRL) techniques like Deep Q-learning or policy** gradients could help the QLearningPlayer adapt even better, especially in environments with more variables.

Conclusion

The results of this project highlight the effectiveness of **Alpha-Beta pruning** in improving the performance of search algorithms, with AlphabetaPlayer consistently outperforming MinimaxPlayer. Meanwhile, QLearningPlayer, despite its slower start and higher computational costs, adapted well to more dynamic or unpredictable environments, showing the potential of reinforcement learning. While the project demonstrates varying levels of success among the models, the findings emphasize the importance of optimization techniques and hint at promising directions for future research, including genetic algorithms and deep reinforcement learning.

Bibliography:

Go Game - Mastering the Game of Go with Deep Neural Networks and Tree Search

Monte Carlo Tree Search

Q-learning algorithm

*We used ***ChatGPT*** to resolve bugs and errors in the code and to correct grammar in the report.