A4 Report

Gong Ziqi; Li Zhehao

- **Agent:**
  The agent is named AlphaGo, an evil AI agent in the K In A Row game. He loves victory and sarcasm, and he enjoys playing with opponents who are destined to be defeated.

- **How the agent's "twin" differs from the basic agent. The twin feature helps when having your agent play against an alternate version of itself. For a simple example of the twin feature, look at the starter-code agent RandomPlayer.py.**
  In both RandomPlayer.py and my agent, the twin feature differentiates two instances of the same agent when they compete against each other. In RandomPlayer.py, setting twin=True changes the agent's nickname from "Randy" to "Randy-Junior" and adjusts the starting point for utterances, ensuring the twin agent behaves differently during interactions. Similarly, in my agent, the twin agent's nickname changes from "Al" to "Al2", and its introduction message varies slightly. This feature prevents identical moves and behaviors, making matches more dynamic when agents of the same type play against each other.

- **How your basic alpha-beta pruning is implemented, and how effective it seems to be. In which function are cutoffs detected? Have you done any tests to find out how many static evals are saved due to alpha-beta cutoffs on search trees of depth 3? If you use child ordering by static eval, how much does that help?**
  In my agent, alpha-beta pruning is implemented in the minimax() function to optimize decision-making by cutting off branches that won't affect the final result.
  Pruning occurs when the beta value becomes less than or equal to alpha, incrementing the self.alpha_beta_cutoffs_this_turn counter. This reduces unnecessary evaluations, improving performance. The agent also tracks the number of static evaluations self.num_static_evals_this_turn to measure pruning effectiveness.
  When running FIAR with search trees of max depth 3(at the beginning from 5 moves to 10 moves, which is at the time of max calculation), it shows on average that for each step, static evaluations are about 5000 and alpha-beta cutoffs are about 240.
  With move ordering based on evaluation scores, pruning becomes even more efficient as better moves are explored first, leading to faster decisions.

- **Describe the persona of your agent in more detail. What is the intended tone of your agent? Is it modeled after a real person, media character, etc?**

  As the name suggests, the agent is respecting the famous GO agent AlphaGo, whose existence changed the world. All of AlphaGo's decisions are believed to be the best solutions. Many world-champions even follow AlphaGo's solutions. We designed this agent based on my impression of AlphaGo, an anthropomorphic inevitable AI who believes that the strong are respected. Therefore, he would taunt and be aggressive.

- **What dialog features if any does your agent use to keep its utterance relevant to the game state or trajectory? Explain how they work.**

  The agent stores both players' historical states. He can access any past states or opponent's current state. Meanwhile, he can get the information of the latest move and evaluation. Therefore, he can generate responses based on the historical or current states.

- **What dialog features if any does your agent use to keep its utterance responsive to the opponent's remarks? Explain how they work.**

  The agent can respond to the opponent's remarks based on different contents. Normally, he would just deny or taunt the opponent based on its remark. When the opponent says "tell me how you did that", the agent will give the latest move information to the opponent. The agent will also respond to the "What's your take on the game so far?", which will generate a short story of the game by the stored historical states and utterances.

- **How did you develop the dialog capabilities of your agent? If you used an iterative design process, describe the phases of this process and what you learned at the end of each phase in terms of what worked or didn't.**

  We develop the dialog capabilities by utilizing the Google Gemini API. The Gemini is initialized and can generate responses in different conditions mentioned above. We hard-coded three different conditions and let Gemini do different reactions in them.

- **Describe each extra credit option you implemented, with a separate subheading and subsection for each one.**
  - **Static Evaluation-Based Child Ordering**

    This optimization is implemented in the minimax() function by setting order_moves=True. When this feature is enabled, the agent evaluates each child state statically and sorts them based on their evaluation scores. If the current player is "X" (the maximizing player), the moves are sorted in descending order

(prioritizing higher scores). If the current player is "O" (the minimizing player), the moves are sorted in ascending order.
This sorting strategy increases the likelihood of alpha-beta cutoffs by evaluating the most promising moves first, allowing the pruning condition (beta <= alpha) to trigger earlier. To measure the effectiveness of this optimization, I added statistics tracking to my code, recording the number of cutoffs (self.alpha_beta_cutoffs_this_turn) and the number of static evaluations (self.num_static_evals_this_turn).

Measuring the Benefit:

A. With child ordering enabled: Recorded the number of alpha-beta cutoffs and static evaluations during a fixed-depth search.
B. With child ordering disabled: Disable order_moves and recorde the same statistics under the same conditions.

By comparing the results of the two experiments, enabling child ordering effectively reduced the number of static evaluations required, significantly speeding up the decision-making process without compromising accuracy. Additionally, the number of cutoffs increased due to the improved ordering.

- **Implement Zobrist hashing**

  This optimization prevents redundant evaluations of previously encountered states by storing their computed evaluation scores in a transposition table (self.transposition_table). In the compute_hash() function, I generate a unique hash value for each board state using random bitstrings stored in a pre-initialized Zobrist table (self.zobrist_table).
  The following statistics are tracked in code:

  A. self.zobrist_table_num_entries_this_turn: Records the number of writes (entries) to the hash table.
  B. self.zobrist_table_num_reads_this_turn: Tracks the number of read attempts from the hash table.
  C. self.zobrist_table_num_hits_this_turn: Counts the number of successful hits (instances where a previously evaluated state was found and its result retrieved).

  By running offline game with Zobrist hashing enabled, advantages can be found in reduced redundant evaluations and faster execution time. When encountering previously evaluated states, Zobrist retrieved their results from the hash table instead of recomputing them, saving computation time. And the overall decision-making speed improved significantly, especially in games with high branching factors or frequent repeated states.

- **LLM implementation:**

The Google Gemini API is utilized. We first import and initialize a client in the prepare function. Then, there is a helper method called the prompt which lets Gemini generate the responses by the given message. In the make_move function, the helper method is called to give a new_remark.

- **Reaction to “Tell me how you did that”**
  In the make_move function, if the current_remark equals the target sentence, the agent will access the information of the latest move and add them to the new_remark.
- **Reaction to “What's your take on the game so far?”**
  In the make_move function, if the current_remark equals the target sentence, the agent will call the prompt function, giving it the historical states and utterances of both players. He asks the Gemini to generate a short story by giving information. The prompt function returns the result and the make_move function initializes the new_remark by that.