Names: Ahrens, CC; Ahrens, Cin

Agent Name: Ducky Wucky, a duck themed agent set on world domination. She knows birds are the best, and she will show it through superior reasoning! Through her self-confident attitude, she will win over those around her and also the game she's playing

Twin: Birdy Wordy, a bird agent who is working through a mid-life, mid-game existential crisis. She's not so concerned with Ducky Wucky or other agents, but rather what philosophy tells her about the nature of her existence. As she ponders her existence, she'll also deign to partake in gameplay (which, interestingly, is the same strategy that Ducky Wucky uses).

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?

Alpha-beta pruning was implemented by passing values of plus and minus infinity into minimax from makeMove(). Then, minimax recursively calls itself once more, switching from being a maximizing node to being a minimizing node and vice versa. When it gets a value back from itself, it then recomputes alpha or beta (depending on if it is a maximizing or a minimizing node, respectively) and then checks to see if beta <= alpha. If this is true, it returns then because we know we do not have to continue looking through this subtree. In summary, the cutoff is detected in minimax. It stops calling itself and returns either when beta <= alpha, when an end of game state occurs, its reached the maximum depth, or it's able to place a center piece.

Here is a table to static eval calls when using alpha beta pruning and a depth cut off of 3 using the standard boards and playing against ourselves:

Player:	Setting:	TTT	FIAR	Cassini
X	using cutoffs	4	17	19
	not using cutoffs	390	254040	278052
O	using cutoffs	3	17	19
	not using cutoffs	240	232512	254040

The shaded rows are the rows played using cutoffs. Obviously, they were significantly better—so much better that it was noticeable when watching even one game be played.

We did not use child ordering by static eval, so we do not know if that would help much.

More detailed description of the persona:
The main persona was Ducky Wucky, an incredibly arrogant, self-aggrandizing duck who is confident in herself a little too much of the time. The twin persona was Birdy Wordy, a slightly more timid and cautious persona who tries to mostly be polite. She also contemplates the meaning of life. Interestingly, the two personas play the game the same way. Ducky Wucky and Birdy Wordy were both duck friends we made during summer travels in Amsterdam for a study abroad program. In the evenings, we would go and feed ducks, and these two, with their silly personalities, were quite recognizable day after day and deserved silly names and to be remembered.

What dialog features if any does your agent use to keep its utterance relevant to the game state or trajectory? Explain how they work.
We used Google's Gemini API to create relevant responses to the game. Sometimes, the model would crash and have to be restarted (in which case we would substitute a statically-generated response). Otherwise, we would include the current game state in the prompt and request a response from the model based on this state. Additionally, when creating our personas, we provided a prompt describing the personas as well as examples of utterances that we had written.

What dialog features if any does your agent use to keep its utterance responsive to the opponent's remarks? Explain how they work.
We pass the most recent remark to each agent so that they know what has been said. We feed this, as well as the current game state, into the prompt we give to the model and request that it create a response based on these two things. They then respond in character with output related to the prompting.

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 had a few phases of development: no AI, settling on a model, and implementing the chosen model.

To get started, we simply used the example utterances supplied in the starter code for this assignment, and we used these for both personas. We did this so that we could ensure the playing portion of our game was working before beginning to implement other features. Then, after getting the rest of our code working, we wrote new dialogue options for each persona by hand.

After our self-written dialogue was to our liking, we began researching AI models. Because it was the first one mentioned in the assignment spec, we started by attempting to use OpenAI's API. However, after implementing the majority of the code using this, we discovered that to do anything meaningful, we would have to pay. We did not want to spend money on doing this assignment, so at this point, we pivoted to using HuggingFace.

With HuggingFace, however, we discovered that the model's necessary size meant that it would be challenging to get this to work without having to pay (although it was likely possible to do so). We did some troubleshooting trying different models that HuggingFace provides, but ultimately decided that this would be too challenging to implement well with what we currently had.

Finally, we settled on using Google's Gemini API, which has a free version and provided the functionality that we needed on the free version. We fed it the lines of dialogue we had previously written for each agent and described each personality. We went through several iterations of prompting. In the beginning, Gemini replied to our prompts by giving us code to produce responses, rather than simply producing responses, which was not what we wanted. We played around with the wording of our prompts and eventually succeeded in getting responses about the agents' next moves, but this commentary often did not line up with the actual move being made. We ultimately told the model explicitly not to mention the next move in its dialogue to prevent incorrect commentary. At this point, it wasn't providing creative enough responses and sometimes they were too long. As a result, we modified our prompt to encourage creativity and specify length of response to be one sentence (rather than the paragraphs it was previously providing). After significant tweaking, we were able to get something that produced the responses that we wanted.

However, settling on dialogue that we liked, the Gemini model occasionally crashed and threw an error approximately every 4–10 utterances (we suspect this was a result of limits on requests per second, minute, and day that are placed on free accounts). To fix this issue, we used a try/except block. If prompting the model was successful, we would simply return the response generated. If prompting the model threw an error, we restarted the model and for that round provided one of our predetermined responses. This succeeded in producing relevant content that was mostly AI-generated that also did not crash.

Finally, for the purposes of testing, we decided to make using our AI optional. We did this because there are a limited number of requests that can be run through the free version of the Gemini API each day, so for big tests that we ran, we wanted to turn off using AI and just use our statically, dumbly generated commentary instead.

Describe each extra credit option you implemented, with a separate subheading and subsection for each one.

Part I

Zobrist Hashing

Zobrist hashing caches previously seen states and their associated static evaluation values. It starts by storing a random 64 bit integer for each possible board position and player. As new states are added, they are stored by XORing all the associated game board positions. In our case, we store the static evaluation function and the depth of the exploration at the same time. When we implemented Zobrist hashing, we noticed a dramatic decrease in static evaluation functions. We did not test it independently of when we implemented alpha-beta cutoffs, but you can view the improvement of before and after in the table above. It improved so significantly that we actually were able to watch the difference in match speed.

Part II

LLM Utterances

To do this, we created a different prompt for each agent. We gave the Gemini API some examples of possible utterances that would be acceptable and described the persona of the agent. Then, at each turn, we gave Gemini the current game state and asked for a one-sentence line of commentary based on the persona, previous prompts, and what the opponent just said. For more information, please refer to the response to the question asking about developing dialogue capabilities.

Tournament

We would like to be entered into the tournament to compete for the top 50% of the class!