Section 1) A 500-word summary about what you did and how it adheres to the requirements of the project CS 4649 or 7649.

The task of this project was to build an AI that can intelligently play Recon Blind Multi-Chess, with the only requirement being that, as a CS 4649 team, our AI must utilize MCTS in some capacity. We approached this problem by first having a meeting in which we discussed what kinds of approaches we could take. We quickly decided that we wanted to take the most straightforward approach that we could, as this would allow for the fewest errors in implementing MCTS which we haven't had a lot of experience programming before. Thus, we chose to use a pure MCTS approach for determining the move for the agent to take. Since each board is likely completely unique regarding where the opponent's pieces are, we chose to initialize a new MCT with every move. This MCT would then be used to determine a move based on the current board via the standard select-->expand-->simulate-->backpropagate process, running for a user-specified amount of time before termination. While this does not allow the MCT to grow and learn over many plays of the game, we believe that with the inherent variability of the board, this was the most straightforward approach. A move that appears valuable in one board configuration may be absolutely terrible in another board configuration due to the uncertainty of the board state, as only half of the pieces' positions are known with absolute certainty. Because of this, the simplest, most straightforward approach appeared to be to initialize a new tree for searching every time. Beyond the MCTS itself was the task of maintaining some idea of what the current board looks like. This problem stumped us at first. Even if the opponent's policy was known (random movement), the problem we ran into was that the uniform distribution of move probability meant that there was never a "most likely" move that we could plan for. A neural net didn't seem useful for predicting a move when we knew it was a uniform distribution at all times. Then, Dr. G suggested particle filtering in the lecture about approaches for the RBMC project, and things seemed to fall into place. Particle filtering wasn't a central topic of this class, but it was a topic covered in CS 3600, and the opportunity to incorporate an approach from a previous class felt like a great way to synthesize our learnings from multiple classes, while also adhering to the idea of "the most straightforward approach" since we already had code for particle filtering from CS 3600. It seemed like a great idea, but unfortunately, we simply could not create enough particles to represent the extremely large number of possible board states midway through a chess game, and the particle filtering began to break down about 10-20 moves into the game. So after days of working on particle filtering, we scrapped it entirely and started a new with a much simpler sense algorithm which reacts to the sense results and nothing else. While still flawed and not at all predictive, it turned out to be much more effective. Unfortunately, by the time we realized that we needed to scrap and restart our board sense algorithm, the due date was upon us, and we didn't have the time to refine our agent to something that's as effective as we know it can be.

*Section 2) A 500-word summary about what you *learned* from doing the project that enhanced your experience above and beyond the lectures, PSets, and midterms.*

The RBMC project has made it very clear the impact variability/randomness can have on a heuristic or problem solving tactic. When we started the project, we overestimated the 'thinking' abilities of MCTS and believed it would be the main focus of our coding. We thought that modeling the randomness of the board would not be the biggest problem we had to tackle, yet we quickly learned of its importance. In our first couple of meetings we were making the mistake of assuming human-like opponents. For example, when discussing how to implement choosing where to sense, we assumed it would be logical to sense towards the center of the board at the beginning of the game because most human players look to control the center early. However, we soon realized that random agents do not act in this manner and by using this logic our sense was not effective. Without an exact (or even closely related) representation of the board, the moves selected by MCTS may not actually move you towards a winning position on the true board state. This problem we learned gets worse with time. As the board begins to vary more and more from the actual true state, the chosen moves from MCTS get worse and worse until it is essentially making random moves on the true board. This is when we started looking into maintaining some sort of accurate board state. At first, we settled upon particle-filtering in an attempt to mitigate this problem. Although this topic was not strictly covered in the course, two of us have seen it before in Intro to AI (3600), while the third never really looked into this topic, so it was nice to get some exposure and practice actually implementing it in code. We ran into some difficulty getting decent results from our particle filtering, so we tried to make some additional changes but ended up scrapping it for something much more hard-coded to the situation. Regardless, this was one of our biggest learning experiences beyond the scope of the class. After coding this up it became more evident what some of the default functions in my_agent.py are intended for, specifically the ones that informed you about piece positions like handle_opponent_move. In general, most of those functions were a sort of way to sense the world state and then take what you've learned/sensed and build your understanding of the truth. Rethinking what we had done under these guides helped us understand how to fix our sensing agent. This idea can be translated to any scenario with unknown world state and sensors that can tell you about the state. In all cases we want to minimize the uncertainty/how much your understanding of the state varies from the true state using your sensors as a way to gain knowledge about what is actually happening. Even though we turned in our assignment a little late, we learned a lot about modeling random agents and minimizing the uncertainty that encompasses them.

Section 3) Each of your team members will include a 150-word summary on how you contributed to the team.

Jeremy's summary: I was given the task of choosing our agents next move. To begin, I researched MCTS a little on the internet to get my bearings and understand what coding challenges would need to be tackled. I then split the problem up into 4 parts modeled after the 4 main steps of MCTS (selection, expansion, simulation, and backpropagation). Expansion and simulation proved to be quite easy with the python-chess library since it provided built in functions to transition a board state given a certain move. I struggled deciding how to represent a node in the tree and every change I made to these nodes required a change to selection and backpropagation. After talking it over with the group, we chose a dictionary because of its look up speed and ease of use/access. Finally settled on a node representation I could now finish up the search algorithm, yet there still seemed to be something wrong with selection. Every tree produced resembled a linked list with seemingly no expansion of unvisited moves. After a little more research I stumbled upon the UCT heuristic for game trees which helped balance the exploitation of winning moves versus the expansion of non-visited moves.

Raveen's summary: After our initial meeting, I was tasked with two methods: handle_game_start and handle_sense_result. We spent a long time discussing how to handle sense results and eventually settled on implementing a particle filter, and so handle_game_start became "initialize the particle filter," and handle_sense_result turned into "update the particle filter." As a result, after writing these two methods, I was the one who understood it the best. From there, it was a short step to me implementing the entire particle filter, and I take full responsibility for its failures and eventual replacement with our new reactionary algorithm. One of the big issues with the particle filter was that it severely underestimated the frequency with which the random opponent would choose an illegal move and thus execute no move at all. Thus, this new algorithm doesn't ever assume that the opponent moved unless 1) it is told that a piece was captured, or 2) the sense result is different from the current board model, in which case it will update the board model to reflect the new information then resume assuming the opponent never moves.

Heather's Summary: After our initial team meeting, I took the lead on the functions handle_opponent_move_result, choose_sense, and handle_game_end. My main focus was on choose_sence. Initially, I wrote choose sense to pick from the middle section of the board at the beginning of the game, and then a larger area as the game continues. In each section, I would choose the square furthest from the average square of all our pieces as the square to sense. This was in an attempt to model after a human agent before we realized this was not the goal. After realizing this, I remodeled choose_sence so that it did not include the sectioning but just the square furthest from our average, not including the edges. Then, as we tested our algorithm, we realized that random sensing best fit our model. Therefore, we decided to pick randomly from any square not on the edge, and check that we do not have 5 or more of our pieces around that square.