REPORT PLAN

STRUCTURE:

Minimax search with alpha-beta pruning to depth n. At depth n, if the evaluations of multiple nodes are equal, we use Monte Carlo Tree Search on said nodes to create another evaluation which we use as a tiebreaker.

The main classes that we have created and used are our MCTS, Minimax, Features, MCTS Node and Minimax Node. Our features class is a class which contains the board state as various attributes. From this class we built our MCTS Node and Minimax Node, which act as a wrapper class for the Features class yet have extra necessary functionality that allow it to work in its respectively search.

DECIDING ON ACTION:

Our game playing program dealt with the 3-player nature of the game by changing its own colour each turn. This allowed us to implement maxn simply by recursively finding our player’s best move for each turn.

Our bot uses two main methods when deciding what action to take. Our main search algorithm is minimax with a hand-crafted evaluation function. This chooses an action by assuming that the other players will be playing optimally to our evaluation function, and from there we choose the action that results in the best possible state. In the cases where there were multiple game states which were equally as good as each other, we used a Monte Carlo Tree Search to determine which of these we should take. Our MCTS took each of the board states, and repeated simulated moves until it reached a terminal state (game is over), and from there updated the ‘rewards’ of all the states leading to that state with an appropriate ‘reward’, depending on the outcome. This was done repeatedly, many of the time exploiting common paths that lead to a positive reward and occasionally branching out to explore other possible paths to try find potentially better paths. The move to make was then the move that had the most visits, as this was the path that many of the time lead to a positive outcome.

We decided on an action by assuming each other player would be playing optimally to our evaluation, and from there we chose the action that would give us the best possible outcome.

EFFECTIVENESS:  
We began by building 2 separate game playing bots, one that used minimax and another that used MCTS. We found that our minimax was a very strong player even with a simple evaluation function and a low depth. Our MCTS was also a strong player, however as it did not have an inbuilt evaluation function and generated one by simulating playthroughs, it took quite a bit of time for there to be enough simulations so that the evaluation was decent – requiring 7 seconds per turn to create an evaluation that was better than the evaluation of our minimax.

Our goal was to optimise the MCTS’ speed and reward system so that on average, it could defeat the minimax bot within the time restrictions (60 seconds per game). After optimising (?) using techniques such as changing the amount of time allowed depending on how far along the game is as well as changing the nodes to contain data in numPy arrays (both techniques which are discussed in depth below), our bot was able to defeat the minimax bot with 4 seconds per turn. However, due to the time restrictions, this meant that our bot was only allowed approximately 20 moves before it ran out of time (20 because at the beginning where moves weren’t as important, it was given 0.5 seconds), which majority of the time we found was not enough to complete a game.

This lead us to our final implementation which actually used both of these searches. Currently, our evaluation function for minimax would sometimes result in a tie between multiple states, and from that would choose one at random. Realising that MCTS was an effective search algorithm, but just could not be used very often due to time constraints, we opted to use MCTS in the case of evaluation ties which did not happen too often, allowing us to use a longer, and therefore more effective, Monte Carlo Tree Search.

CREATIVE TECHNIQUES:  
- allowing more time to simulate moves depending on the point of the game  
-having nodes contain the visits and rewards of children so that addition of arrays would be done quicker – as well as saving memory  
-having each piece cycle through being all other pieces

A creative technique that we implemented was instead of each player having a set colour as would seem to be the obvious approach, we had a single player which contained the game state and every turn it changes its colour and in turn its goals.

Another creative technique that was used was to split the game into separate ‘phases?’, as during different parts of the game some moves would be more important than others, and so should be calculated more effectively. With this, we allowed our MCTS to run with more time depending on the current phase. We decided that at the beginning of the game, moves are not too important as they do not have too much of an impact on the game and so the difference between two good pieces is not game-changing. In contrast, moves in the midgame can possibly make or break the result of the game, and so the difference between good moves decided by the evaluation function can change the entire outcome of the game. This resulted in us allowing much more time for MCTS after a certain marker which decided that we were now in the ‘midgame’ to ensure the move picked was the optimal one.