AA 228 FINAL PROJECT PROPOSAL
REINFORCEMENT LEARNING FOR SIMPLIFIED CHESS POSITIONS

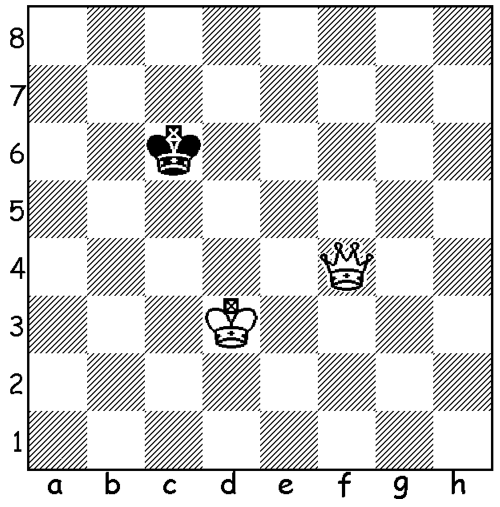
SUNIL DEOLALIKAR

1. Problem Domain

Getting computers to effectively play chess has been an open problem for decades. Most top chess engines rely on searching many moves ahead in the future and also have access to large databases of previous positions recorded from matches between top human players. A reinforcement learning approach to training an AI to play this game is difficult mainly due to the vast number of possible states and actions. A chess board contains 64 squares and a total of 32 pieces, so examining all possible board states is infeasible – the number of possible positions is estimated to be order of magnitude 10^{43}. Additionally with 16 starting pieces per side there is also a very large action space to consider on top of this already vast state space.

Instead of attempting to solve the extremely difficult problem of getting a computer to be able to intelligently play a full game of chess, I wish to examine this in a smaller context, in just getting a chess engine to play in the "endgame". Often chess beginners are taught how to win (also known as checkmate) from simple canonical positions with fewer pieces. Common examples would include King and Queen vs a King or a King and Rook/Castle vs a King [Figure 1.1]. These 3 (or ambitiously 4) piece positions reduce the state space greatly to order of magnitude $\sim 10^6$ states and ~10-20 actions. This space could perhaps be even more drastically reduced by exploiting some aspects of board symmetry (in some cases, the path to checkmate shouldn't differ if you flip the board along a central axis) thus making the number of states much more manageable (but still large) for reinforcement learning and turning this into a problem more reasonable for the scope of this class. A simplified problem such as this should still be interesting to examine though as there is no very easily described deterministic algorithm to reach a checkmate. Most competent chess players will be able to easily win from one of these canonical positions, but due to relying on heuristics rather than a set algorithm, it is unlikely that all players would get to the win with the same moves/action sequence or even in the same number of moves.

Figure 1.1. An Example basic endgame position to evaluate



One of the main challenges here will be to come up with a reasonable reward function such that will encourage the agent to get to a final goal of winning the game. This can be a clearly defined goal (can have a function that returns a large value if the enemy has been checkmated or a function that returns a negative value if a piece is lost), however there will likely need to be some smaller intermediate heuristic rewards for this approach to be successful as the end state position could be many moves away. There seems to be a lot of published research on evaluating a chessboard state to a numerical value for existing computer programs to use when evaluating which move to play so I am planning to also leverage some of the relevant research in this field to help create a good reward function for my simple game state.

Another issue that will need to be addressed is on how to choose the moves of the opponent player to the agent. A basic approach will be to just randomly choose among possible moves, alternatively the board state could be fed to some existing open-source chess engine such as GNUChess or Stockfish and those clients could make the appropriate reasonable opponent moves to train the agent.

2. Success Metrics

The primary measure of success that I plan to use is whether the computer is able to reach a winning state within a to be decided range of moves (<50 for comparision sake, the above problem can be solved in ~10 moves). This is a very clearly defined state in which the enemy king is under attack from a piece and also cannot move. If this proves to be easily accomplished then I will move on to measure success as the number of moves taken to get to checkmate by my engine and compare that to the actual solution for the least number of moves to make for a win (for these simpler positions this can be better obtained from one of the open source chess engines listed above). Then in this case, ideally we can measure success by Δmoves between my engine and well developed open source engines.

3. References

Dazeley, Richard P. “Investigations into Playing Chess Endgames using Reinforcement Learning”

Shannon, Claude E. "Programming a Computer for Playing Chess." Computer Chess Compendium