# **CITS3001 Assignment Report – ThreeChess**

Lachlan D Whang (22466497), Tye Bridgeman (22326712)

There exist a wide variety of research and algorithms developed for the standard, 2-player game of Chess. 3-Player Chess is an interesting variant on this widely popular game which introduces a new degree of entropy and a vast number of new board layouts, and the difficulty involved with training intelligent agents to play a game of N-Player Chess grows exponentially as one observes the state-space explosion with increasing values of N. Thankfully, there exist a number of algorithms which can be used to generalize the agent learning process for any arbitrary number of players in a single game of chess. While a perfect game-playing agent may still be some distance away from our current efforts, this document will explore some of the suitable techniques which we can use to implement a rudimentary agent for 3-Player Chess.

**Greedy Algorithms**

All game-playing algorithms explored in this document work to maximize some utility based on the information available. Basic greedy algorithms constitute the most basic, game-playing agents whose moves are not random nor uninformed. At each turn, greedy agents make a move which maximizes the utility value in the short-term, in hopes that such moves will lead to maximized utility values in the long run. This utility value could be the number of pieces taken by the agent, or the total value of the points of pieces taken, or the total number of pieces remaining on the board that belong to the agent – the list goes on.

**Active Learning & Q-Learning**

The active reinforcement learning problem is characterized by a lack of a fixed policy, unknown reward and unknown transition models, with the latter 2 properties being present in Markov Decision Processes. An intelligent, game-playing agent in an active reinforcement learning environment needs to be able to select the “best” move based on the information it currently has access to at every turn, by calculating the utilities of states (and actions, depending on the algorithm), and selecting the action out of all possible actions in the current state that will produce the maximum known utility value.

Basic greedy algorithms can be applied to the active learning space. One simple greedy algorithm which undergoes the active learning process could involve the following 3 steps in the main game-playing loop.

1. Sense the current state and associated reward .
2. Call the **Adaptive Dynamic Programming** algorithm (a passive learning algorithm) to update the estimates for all previous utilities.
3. Choose the next, optimal action using the same equation for selecting an optimal action in Markov Decision Processes…

… where is the set of actions that can be performed in the current state .

The main problem with this basic, active learning, greedy algorithm is that it **does** **not** **converge on the optimal policy**. The algorithm actively exploits to the maximum possible extent permissible by the bounds of its knowledge, and exploration is minimal.

Q-Learning is the de-facto algorithm for solving the active reinforcement learning problem. The algorithm maintains a set of state-action pairs and their utilities, updating them after every move. Q-Learning is superior to basic greedy algorithms in that it balances **exploitation** with **exploration**, artificially inflating the utility values of state-action pairs that have been visited some number of times less than a predefined hyperparameter, such that when the algorithm attempts to select the action which maximizes this utility, it will explore the results of these actions. Exploration occurs most frequently towards the start of the learning process, and as the algorithm learns the outcome of more actions or moves, the -function responsible for the inflation of these utilities will start to present the actual, calculated utility value to the algorithm, instead of the inflated value, indicating a shift towards exploitation once the algorithm has “matured”.

The biggest challenge facing implementation of a Q-Learning agent is the method by which the algorithm’s knowledge will be **persisted** between training and playing sessions. Once the agent’s state is torn down at the end of a game, the information it has learnt needs to be stored. **I/O methods** are clearly the most straightforward and simplest way to persist this knowledge, but the functions are slow in comparison to the rest of the algorithm, and if the agent is unable to determine whether a position represents the end of a game, it will have to update stored utilities on disk after performing each move – a major detriment to its performance, although it is possible that this situation can be avoided depending on the information available.

**Implementation 1: Q-Learning *(Agent22466497)***

The beauty of the Q-Learning algorithm lies in its simplicity and adaptability. Q-Learning is based on the **Temporal Difference Learning** passive learning algorithm, which seeks to update only the utility value of the most recent state visited at the end of every action performed; Q-Learning similarly updates only the most recent state-action pair utility. While the algorithm itself is indeed comparably dense in comparison to other algorithms such as the **Monte Carlo Tree Search** or the **MiniMax** algorithm, given enough time and the right training data, the performance of a Q-Learning agent can surpass agents implementing either of the aforementioned algorithms, although such performance is not guaranteed for this particular problem instance.

**The Q-Learning Update Algorithm**

There are 2 main functions associated with a Q-Learning agent. The Q-Learning update algorithm is called after each action to update the Q-utilities for the impacted state-action pair.

function Q_Learning_Update(s, r, a, s', r' , y, n, Q, Nsa) 
if s' is a terminal state: 
Q[s', None] = r' 
// s is null if no states have been visited before 
if s is not null: 
if Nsa[s, a] exists: Nsa[s, a] 1 
else: Nsa[s, a] = 1 
c = a]) // we call function with input Nsa[s, a] 
= (1 — c)Q[s,a] + c (r + ) 

The following input arguments to the Q-learning update algorithm are fixed.

- The previous state-reward-action tuple.  
 - The current state and reward.  
 - The discount factor.  
 - A function which specifies a **learning rate** which **decreases over time**.

The following arguments to the Q-Learning update function have values that may **change** over the course of a single iteration.

- The table of Q-values, i.e. the table of state-action pair utilities.

- A 2D table, where keeps track of the number of times action was performed while in state .

**The Q-Learning Agent Model**

The Q-Learning update step is just 1 function which fits into the wider agent model as a whole. On its own, the function above is just another passive learning function - the overall agent code is what makes this an agent which is actively learning. The Q-Learning agent model captures the overall behavior of the agent in a general setting; in the context of the threeChess environment, the agent’s playMove function corresponds directly to the innermost while(true) loop.

function AgentModel_Q_Learning(y, 71) 
// Initialization 
Initialize Q, NS a to empty tables, 
// Main loop: Execute mission after mission 
while (true) 
// no previous state at start 
Initialize variables s, r, a to null 
s' = initial state (chosen randomly, if multiple initial states exist) 
// Execute one game, from start to end 
while (true): 
(s' , r') = SenseStateAndReward() 
Q_Learning_Update(s, r, a, s' , r', y, n, Q, NS a) 
if s' is terminal: break // Reached a terminal state 
a = argmaxf(Q [s', a'], NS a [s', a']) // Choose next action 
ExecuteAction(a) 

*Agent22466497* demonstrates an implementation of the Q-Learning algorithm above.

**The Q-Learning Reward Function**

Certain environment situations possess innate rewards that the agent is provided with at the end of every action. This is not true regarding the threeChess environment, and as such, designing a suitable reward function for this agent is both a necessity, and extremely challenging to perfect. Simple reward functions could include the difference between the number of, or value of, the pieces from the previous state. More complex reward functions could consider the positions of pieces and the number of attacking lines from the agent’s pieces to their opponents. An immense combination of different factors makes designing such a reward function relatively challenging.

**Implementation 2: XXX *(Agent22326712)***

**References**

Govindan, S. and Wilson, R., 2007. *Faculty & Research › Working Papers › A Decomposition Algorithm For N-Player Games A Decomposition Algorithm For N-Player Games*. PhD. Stanford Graduate School of Business.

Schrittwieser, J. et al., 2019. *Mastering Atari, Go, Chess And Shogi By Planning With A Learned Model*. PhD. Cornell University.