# Artificial Intelligence Project Part B
By Alec Yu and Sharan Krishnan

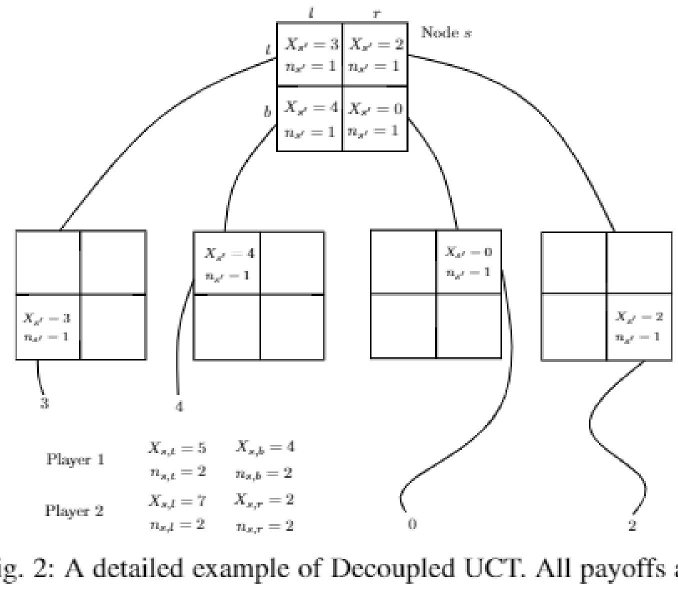
## Problem Representation

An effective adversarial agent in a two player, zero sum game is one which can discern the optimal move to play for any given board state. For most games including RoPaSci, the problem of finding the optimal move can be reduced to some form of tree traversal where board states corresponding to a move being played can be expanded and traversed.

Hence, the clear choice was to represent the game as a tree, with the root node corresponding to the current board and the branching nodes corresponding to the boards reached from playing out a single Upper and Lower simultaneous move. Note that each board can be uniquely identified as the set of upper and lower pieces on the board and their corresponding (r,q) positions, the number of remaining throws for each player and the number of turns played to reach the board.

## Simultaneous Monte Carlo Tree Search (MCTS) using Decoupled UCT (DUCT)

Traditional Monte Carlo Tree Search methods are used in sequential turn-based games. Monte Carlo Tree Search aims to build up a statistics search tree without requiring a form of heuristic or evaluation function, rather, it builds an “evaluation” for a given move based on many Monte Carlo Simulations. This statistics tree is rooted at the current state of the actual played game and branches out with potential moves from the current state. The main idea is that a winning move will lead to a winning position, and thus will score higher when doing Monte Carlo (random) play outs from that position.



$l$
$r$
Node $s$
$t$
$b$
$X_{s'} = 3$ $n_{s'} = 1$
$X_{s'} = 2$ $n_{s'} = 1$
$X_{s'} = 4$ $n_{s'} = 1$
$X_{s'} = 0$ $n_{s'} = 1$
3
4
0
2
Player 1
$X_{s,t} = 5$ $n_{s,t} = 2$
$X_{s,b} = 4$ $n_{s,b} = 2$
Player 2
$X_{s,l} = 7$ $n_{s,l} = 2$
$X_{s,r} = 2$ $n_{s,r} = 2$

Fig. 2: A detailed example of Decoupled UCT. All payoffs are shown in view of Player 1 (Max).

The algorithm used in our RoPaSci agent uses a simultaneous variant of the Monte Carlo Tree Search algorithm called Decoupled UCT. This model ignores the presence of simultaneous moves, and applies cumulative utilities gained for each of their own moves as if there was no joint dependency on these rewards. After all possible move combinations are explored, these rewards are tallied together, and each player chooses separately their final move based on the Monte Carlo simulations.

A high-level explanation for the Simultaneous Move MCTS algorithm can be explained in four main stages:

1. **Selection**
   Given the Decoupled UCT (D-UCT) formula, generate values for each child node based on the number of simulations done and the total utility score and traverse down the best child node.
2. **Expansion**
   When we are unable to apply the D-UCT formula to find a successor node, we expand the game tree by generating all children from the current node. We represent an arc between nodes as the simultaneous move required to be played to reach from one position to another.
3. **Simulation/Rollout**
   After expanding a node with children, the algorithm picks a child node arbitrarily and simulates the entire game from the selected node's position until it reaches an end state.
4. **Backpropagation**
   At the end of the rollout, it evaluates the state of the board and generates a payout for the players (win, loss or tie). This result is then back propagated up to the root node of the tree.

At the end of the algorithm, each move branching from the root node should have an estimated payout. The best next move from the root node will then be the move with the largest utility score generated by the algorithm. Upon reaching the next position, this whole process is reconstructed with the new position as the root node.

## Why MCTS?

MCTS was one of the main algorithms considered for this project as it had many advantages over other tree search methods for game playing such as minimax.

In Monte Carlo Tree search, there is no need for a heuristic function to estimate the value of a position. This is a key characteristic of a Monte Carlo Tree search - rather than constructing a heuristic function, we can just estimate the value of a given move or position by doing random playouts and expect that this value will converge to the "true" value of that position. This meant that we don't need to spend much time playing the game to understand the nuances of it and developing a good heuristic function wouldn't be a bottleneck for our algorithm.

Moreover, MCTS can also handle larger branching factors better than minimax. As we will discuss further below, the performance of minimax is severely hindered by large branching factors, and this was especially relevant in RoPaSci, with the board being relatively open and the addition of "Throw" moves increases the search space drastically. As we were bound by a time constraint per game, MCTS was also chosen for its ability to favour moves that are more likely to be good, making its search asymmetric.

Finally, MCTS does not need to run to "completion" to generate strong output, unlike minimax. MCTS outputs stronger moves the longer it is run, however the search can be stopped at any point. This flexibility proved to be a lot more comfortable to deal with when trying to fit within the time and space constraints of the task.

## SM-MCTS - Heavy rollouts

Usually in a Monte-Carlo tree search algorithm, purely random moves are chosen during the playout phase. This aims to form an “expected” value for a given position without the need of any heuristic. However, this caused a few problems in our practical use:

1. If we are taking a uniform distribution for each possible move, throw moves are extremely biased - We have 3 possible throw moves for each tile that we can throw to.
2. Simulating a random game often results in many draws - since the game is called off past 360 moves, and it’s very possible for random games to go past this move threshold. These simulations don’t add any value to our algorithm.
3. Because the simulations took so many moves, this resulted in a lot of computational resource in our algorithm

Instead, we took a subset of “greedier” moves and randomly chose the moves from there.
This not only ended the games faster, but also it still preserved most of the properties of MCTS, and randomly played out games according to our strategy.

## Shortcomings of Decoupled UCT MCTS

Our implementation of the Decoupled UCT variant of MCTS for simultaneous games has the following shortcomings:

1. We assume no joint dependency between moves - this may not fully capture the nuances of simultaneous gameplay that a Nash equilibria solver may be able to do.
2. Since we determined that taking the whole action space still was too much for our algorithm, our algorithm’s performance is heavily contingent on our pruning method. Here we’ve used a rule-based greedy pruning method, and we’ve discovered that there are very obscure scenarios that our agent will play poorly.
3. We’ve limited the number of simulations heavily in order to fulfill the time constraints. There may be some noise due to not enough simulations
4. We don’t consider possible strategies that our opponent can have - we assume that both players share the same strategy

## An Alternative Strategy: Simultaneous Move Minimax using Alpha-Beta Pruning

In the simultaneous move variant of minimax, the assumption that players can see the opponent’s previous move no longer applies and nodes now represent boards created from both players moving. Node utilities from the maximiser’s perspective are computed as the mixed strategy Nash equilibrium (MSNE) produced from the matrix of sub-node utilities, which are then computed recursively in a depth first traversal, terminating at a leaf node. The move made by the player in response to the current board is randomly chosen with the probabilities assigned by the MSNE.

Alpha-beta pruning in a simultaneous context becomes more complicated since the element of randomness in both players’ strategies implies that the assumptions on their behaviour towards utility must relax significantly. Now, the only guarantee for a player is that they will never play a move that is strictly dominated. More precisely, if move *a* always leads to at least as good a payoff as move *b* no matter what the opponent plays, then *b* cannot be a part of the players mixed strategy.

Letting $o_{a,b}$ and $p_{a,b}$ be an admissible lower and upper bound for the utility of the board after the utility maximiser and minimiser play moves *(a,b)* respectively, we can assess the feasibility of the following inequality to determine if move *a* is strictly dominated.

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_{a-1} \\ x_{a+1} \\ \vdots \\ x_m \end{pmatrix}, P = \begin{pmatrix} p_{1,1} & \cdots & p_{1,n} \\ \vdots & & \vdots \\ p_{a-1,1} & \cdots & p_{a-1,n} \\ p_{a+1,1} & \cdots & p_{a+1,n} \\ \vdots & & \vdots \\ p_{m,1} & \cdots & p_{m,n} \end{pmatrix}, e = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

$$f = \begin{pmatrix} o_{a,1} & \cdots & o_{a,n} \end{pmatrix}$$

$$x^t P \geq f, 0 \leq x \leq 1, \textstyle\sum_i x_i = 1$$

That is, move *a* is strictly dominated if there exists a mixed strategy *x* for the game where move *a* is not considered, where the pessimistic utility for each of the minimiser's moves $(x^T P)$ is at least as good as the optimistic utility of *a* for each of the opponents moves. If the system of inequalities is feasible, then *a* is dominated by the mixed strategy *x* and can be pruned. We can assess a similar linear problem to determine if the minimiser's action *b* is dominated.

For RoPaSci, a cut-off depth is also necessary to preserve efficiency in the depth first-traversal. This requires terminating search after a maximal depth is traversed and estimating the player's utility for the board state reached at that depth. Attempts to build an evaluation function for a board are discussed below.

## Shortcomings of Minimax

Although guaranteeing an optimal solution, provided an accurate evaluation function at cut off depths, for the purposes of the RoPaSci and the time and space restrictions imposed on agents, simultaneous minimax is far too inefficient. In the worst case, where we are past turn 9 of the game (the entire board is available for throws for both players) and no moves are pruned, the branching factor can reach in excess of 200^2 nodes. With a cut-off depth as shallow as 2, the search space is 200^4 = 1.6*10^9, requiring far too much time to compute relative to the time restrictions.

The inclusion of pruning brings its own issues. The simultaneous alpha-beta pruning discussed above is only effective when bounds, $o_{a,b}$ and $p_{a,b}$, are tight so that the system of equations is feasible for more moves and they can be pruned, however, finding tight bounds that are admissible proves challenging. In most cases for RoPaSci, the bounds can only be set to the default width of *(-1,1)* and in some special cases where a player has an invincible token, the bounds can be tightened to *(0,1)* or *(-1,0)*. However, in either of these cases, an insufficient set of moves are pruned, and the algorithm remains inefficient.

## Pruning Strategies

### Rule based pruning

While the SM-MCTS algorithm deals with larger search spaces better than minimax, still, the search space is too large to consider all the moves within a 60 second period. We use a greedy rule-based method to prune moves down to make the search space smaller, based on our strategic understanding of the game. In order of priority: we consider "greedy" moves in this order of importance:

1. Slide/swing capture moves:
2. Throw capture moves if we haven't thrown many pieces
3. Slide/swing distance closing moves
4. Escaping moves

We then take a subset of these moves to some number. This ensures that we consistently can make moves in a certain time frame.

## Evaluation Function and Attempts at Machine Learning

Evaluation functions use board attributes to form an estimate of the associated utility for the maximiser. For this project, the primary applications of an evaluation function would be to estimate utilities of boards at the cut-off depth for the simultaneous minimax solution and define a relation amongst the set of sub nodes which algorithms may use to greedily prune moves corresponding to low estimated utilities. The following evaluation function was chosen:

$$\widehat{U}(q) = w_0 + w_1 TokensDiff + w_2 ThrownDiff + w_3 UnthrownDiff + w_4 DominanceDiff$$
$$+ w_5 SpreadDiff + w_6 MinDistDiff$$

Where,

- $\widehat{U}$ is the utility estimate for board state $q$
- $w_i$ represent intercept and slope coefficients chosen to minimise error
- $TokensDiff$ is the difference in Upper and Lowers' remaining tokens.
- $ThrownDiff$ is the difference in the number of thrown tokens between Upper and Lower.
- $UnthrownDiff$ is the difference in Upper and Lower's remaining throw moves
- $DominanceDiff$ measures the difference in dominance between Upper and Lower token type. For example, the dominance difference for scissors is $\frac{S_U}{R_L} - \frac{S_L}{R_U}$
- $SpreadDiff$ measures the difference in spread between Upper and Lower thrown tokens. Spread is measured by taking the central token of a player and computing the average token distance from this.
- $MinDistDiff$ is the difference between the minimum distance to a capture between Upper and Lower

Given a dataset containing board attributes and its corresponding utility, gradient descent machine learning can be applied to learn weights $w_i$ that minimises the sum of errors $U(q) - \widehat{U}(q)$. However, a difficulty that arises from this task is computing true values for a diverse set of boards. With RoPaSci's immense branching factor and maximal depth of 360 (where a draw is concluded), finding true utilities for mid and early game board states (where more pieces are present) is infeasible even with approximations such as taking a random sample of sub nodes and decreasing maximal depth to 30. As such, weights could not be learned.

## Performance Evaluation and Selection of Algorithms

The algorithms discussed in this report can all be evaluated and ranked purely by win-rate, where a win is defined as either a victory specified in the RoPaSci rules or by an opponent breaching time and space constraints. Hence by assigning two agents' different algorithms and matching them in many games, the stronger agent can be confidently selected based on their win rate.

For the algorithms discussed, the experiments found that the SM-MCTS has a higher win rate against the simultaneous move minimax algorithm and an SM-MCTS agent which prunes their nodes using a rule-based pruning performs better against the one with an evaluation function. Hence, the agent selected for submission was one which employed SM-MCTS and greedily prunes nodes based on a rule-based system.

## References used

Lanctot, M., Wittlinger, C., Winands, M. and Den Teuling, N., 2021. *Monte Carlo Tree Search for Simultaneous Move Games: A Case Study in the Game of Tron*. The Netherlands.

Saffidine, A., Finnsson, H. and Buro, M., 2012. Alpha-Beta Pruning for Simultaneous Moves. Sydney.

Tak, M., Lanctot, M. and Winands, M., 2021. *Monte Carlo Tree Search Variants forSimultaneous Move Games*. Maastricht University.