Approaching Simultaneous Game-playing techniques

Tyler Wong, Dion D. Tran
University of Melbourne - COMP30024
May 9, 2021

Abstract

Part of the project regarding Artificial Intelligence COMP30024, this report explores simultaneous game theories surrounding the game ROPASCI 360. In particular, the design and implementation of a program that can efficiently decide upon optimal actions to take. We look at backward induction as the main focus of the algorithm, whilst exploring the multitude ways to optimise the algorithm despite the sacrifice on accuracy for space and time constraints.

1 Introduction

ROPASCI 360 revolves around two players simultaneously throwing, moving and capturing tokens, the end goal being the capturing of all the opposing players' tokens. The program implemented carries on the role as one of the players and uses backwards induction to calculate its next optimal move. Backward induction was one of the earlier methods developed which recursively solves multi-stage simultaneous sub games by depth first searching a tree of all possible moves and their predecessors and working back up. Considering the game mechanics, as the game continues, possible number of moves become almost exponential, this property denoted as perfect recall. Thus backward induction algorithms become far computationally heavy, thus we also explore several ways to optimise space and time required.

2 Describing Approach

2.1 What search algorithm have you chosen, and why?

ROPASCI360 can be categorised as a dynamic game with a finite number of moves but such that are general enough to represent scenarios using stochastic events and imperfect information. We looked at previous perfect-recall algorithms such as Backward Induction, Monte Carlo sampling algorithm and Range of Skill Algorithm as well as considering an imperfect recall algorithm, FPIRA. After looking through the algorithm, we concluded to go with the most novel algorithm, backwards induction, as it was not only simpler to implement but easier to exercise and optimise.

Classical backward induction algorithm enumerates states of the game tree in a depth-first manner and after computing values of the surrounding subgames, it solves the game corresponding to the bottom of the trees, and propagates the calculated equilibrium value to the predecessor.

2.1.1 Solving subgames

ROPASCI360

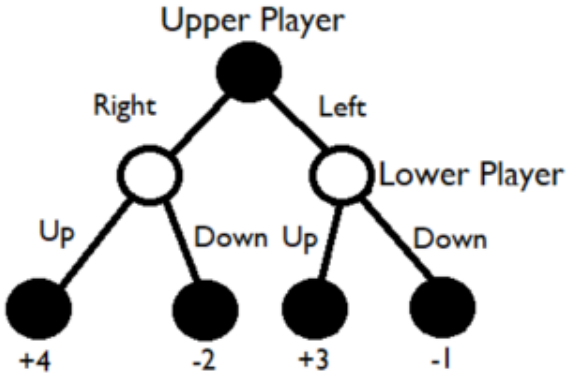

Figure 1.

Figure 1 represents a very simple sub-game example. Using our evaluation function, touched on further ahead, we calculate whether or not the board state favours the player numerically. For instance, if simultaneously, Upper Player moves right as and Lower Player moves up, our evaluation function will say that these moves will favour the Upper Player more by 4.
We can place all potential moves into a payoff matrix.

		Upper Player	
		Right [,1]	Left [,2]
Lower Player	Up [1,]	4	3
	Down [2,]	-2	-1

Figure 2.

We assume that the lower player will minimise points in as to favour itself over the upper player and for this instance, it will consider moving down compared to up as it minimises upper despite upper's move.

To find the optimal strategy we use the sample implementation provided by COMP30024, which takes in a payout matrix and computes equilibrium strategy and row maximiser. Using a mixed strategy, we allow some probability over some actions to reduce predictability when it comes to sub-games that contain no Nash equilibrium.

2.1.2 Optimising Algorithm

Due to the extensiveness of ROPASCI, and the perfect recall backward induction requires, the computational power needed to fully explore game-trees is immense. Thus, optimization of the algorithm that sacrifices accuracy for space and time is required in order to reach any closed conclusions.

2.1.2.1 Cutoff test

Initial assumption in limiting resources is by approaching a cutoff test, giving a depth limit to the search tree. This lower's accuracy depending on the depth. With cutoff by itself, only a depth of 2 can barely be computed. Unfortunately, that means that the algorithm can only look 2 steps ahead when considering an action.

2.1.2.2 Evaluation Function

An evaluation function is created to give an estimation of the true utility function. As the cutoff is now before the end of the game tree can be fully explored, the simple zero-sum utility function is not enough to suggest whether or not the given board state is favouring otherside.
The evaluation function we use consists of features and traits of given player's tokens and its positions to suggest whether the state is best or not. Given features will be explored more thoroughly in 2.3.

2.1.2.3 Filtering / Book learning

ROPASCI 360, have more than not, some clear moves, especially during the beginning of the game. Such as; throwing tokens that can capture instantly and consider actions that move the token towards the target or further away from victims.

2.1.2.4 Aggressive Pruning

As the game is played by both players simultaneously, it isn't possible to alpha-beta prune the tree, thus we decided to prune the payoff matrix by removing suboptimal actions. We do this by prioritising moves that are heuristically more beneficial towards the matrix and only adding the first 10 by 10 moves.

2.2 How have you handled the simultaneous-play nature of this game?

We viewed simultaneous move games as an extensive-form game with imperfect information and also assumed that the opposing player will also make optimal moves similar to paranoid reduction.

2.3 What are the features of your evaluation function, and what are their strategic motivations?

Feature 0 - Gives the highest heuristic towards enemies having no tokens.
Suggested as a base to more heavily decide upon removal of all enemy tokens.

Feature 1 - Number of total present teammates.
Prioritise having more teammates.

Feature 2 - Number of total present enemies.
Prioritise having less enemies.

Feature 3 - Total number of potential victims specific to token type.
Prioritise tokens that can lead to the largest amount of captures.

Feature 4 - Total number of potential predators specific to token type.
Helps suggest not throwing tokens that can easily be captured.

Feature 5 - Calculating 'wellbeing' of each team mate tokens.
For each token, we calculate the distance between their victims and their predators and base a value on it.

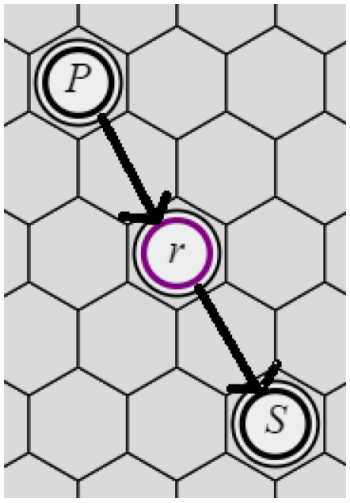

Figure 3.

In figure 3, we see the lower rock being in between the upper's paper and scissors. Using manhattan distance, we see that the rock is 2 hexagon away from both the tokens. Thus we would calculate by using formulas.

(predator_distance - 8) + (8 -victim_distance)

Thus we would have a score of (2-8) + (8-2), which would be 0. However, if the rock was to have moved towards the scissors by one.

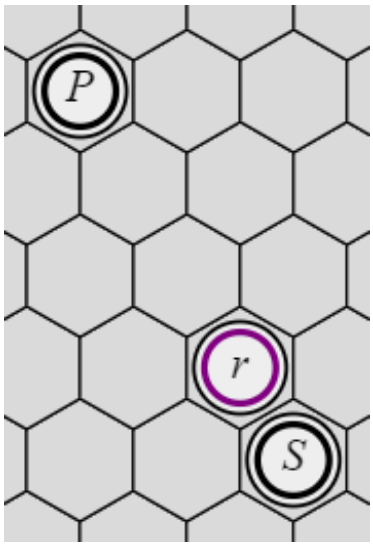

Figure 4.

As per figure 4, rock would be one hexagon away from capturing scissors and 3 hexagons away from paper, thus we would have a score of (3-8)+(8-2), being +2.

Feature 6 - Calculating 'wellbeing' of each enemy tokens.

Similar to feature 6, we also calculate the wellbeing of enemy tokens and focus on the lowering of it.

Feature 7 - Calculates just the closest victim.

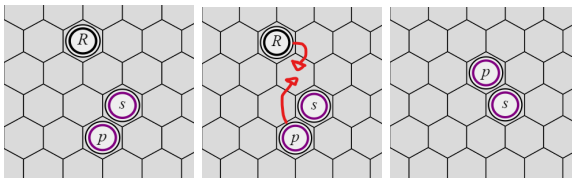

Figure 5.

We added this feature as we found out that some scenarios would not capture the ability to defend by attacking. Following Figure 5, a rare but very possible scenario, previous evaluation functions would consider to simply move the lower scissors away which isn't the best move however this feature helps the algorithm focus more on attacking if a victim is close enough.

With these eight features, we conclude on an evaluation function and test different weights to see which features impacted the backward inductions the most positively.

Initially, we had planned on implementing machine learning and supervised learning on board states that we had labelled but we later decided that since the reduction of computational power was more important than accuracy, it was not to be implemented. This led to manually testing various weights and judging the accuracy of moves.

The Evaluation Formula and its weights.

$$Eval(s) = w_0 f_0(s) + w_1 f_1(s) + w_2 f_2(s) + w_3 f_3(s) + w_4 f_4(s) + w_5 f_5(s) + w_6 f_6(s) + w_7 f_7(s)$$

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	Dep	R.T	T.C	W/L/D
Test1	0	0	0	0	0	0	0	0	0	8.03	75	L
Test2	1	1	1	1	1	1	1	1	1	9.4	360	D
Test3	0.1	1	1	0.1	0.1	5	1	0.5	1	7.1	47	W
Test4	0.1	1	1	0.1	0.1	5	1	0.5	2	977.4	38	W

Table 1. 4 Interesting Evaluation functions and its results against RandomPlaying AI.

Dep: Depth of backward induction cutoff.
R.T: Run-Time of entire algorithm (seconds).
T.C: Amount of turns it took to finish the game.
W/L/D: Win or Loss or Draw

Table 1, shows 4 sample tests we deem important. Test 1 represents the simplest of evaluation functions alongside a backward induction of 1 depth, only looking one move ahead, however the program ends with a draw after many inefficient moves.
Test 2, is the best weight we found considering a depth of 1 in which it wins against the random playing AI in a bit over 7 seconds and 47 turns, the lowest we found.

We conclude that a depth of 2 for backward induction is computationally still not possible despite efforts in filtering and aggressive pruning. Thus the evaluation function will progress with Test 2.

3 Performance Evaluation

3.1 How have you judge your program's performance

In evaluating our program, we placed it against multiple enemies. In testing its ability to function we simply tested amongst ourselves.

When implementing an evaluation function with multiple features, we put it up amongst random A.I's and also itself. To finalised the program we placed it against other students programs, in hopes to see if it could win.

Properties in a sound program that we looked for were.

- Winning or losing over drawing
- Capturing when possible
- Throwing the right pieces
- No eating of its own pieces.
- No repeating moves.

Testing was vigorous, as it was very manual, i.e. looking at game state outputs and results many times, until all 5 tests were passed.

4 Conclusion

Creating an A.I that could efficiently play ROPASCI 360 to a high level proved difficult due to the computational power required to run the backward induction algorithm, despite efforts in optimising it.