COMP30024 Artificial Intelligence
Project Part B - Report

Felicia Suryawinata 1297668
Chenuli Siriwardena 1244521

Introduction

In this report, we discuss our approach to building an agent to play the game Infexion, using the Minimax search algorithm with alpha-beta pruning alongside the TDLeaf algorithm. Infexion is a fully observable, static, sequential, deterministic and discrete game in which both players play optimally. We discuss the performance of our minimax-based agent against two other agents: Firstly, against an agent that makes random moves, and secondly, a greedy agent that only considers the next best move based on the current state, instead of looking into the future to see how the game would unfold. Upon conducting a series of thorough tests, we concluded that our agent beats both of these agents with a 100% win rate while adhering to the resource constraints, and were able to establish that it is an effective approach to playing the game.

Our Approach

Justification

As mentioned above, we used the minimax algorithm and alpha-beta pruning and to build an agent that plays Infexion effectively. Minimax is used to achieve Nash equilibrium where each player is making the best decision given the strategies used by other players (Harrington, 2015). Infexion is a complete information game, where both players have access to all the information about the game at any given time. Hence, we determined that using the Minimax search algorithm is a better approach to playing the game as opposed to Monte-Carlo Tree search (MCTS), which is more useful in incomplete information games. Moreover, complete simulations of MCTS require more space and time than is allowed by the resource constraints of the game, and fine tuning MCTS to adhere to the constraints is difficult.

Techniques to adhere to resource constraints

1. **Depth selection**

Testing with different depth values for minimax yielded that even depth values produced optimal results, with depth = 2 and depth = 4 performing the best under the resource constraints. However, we identified an instance where testing the minimax agent (as blue) against the greedy agent (red) with depth = 2 resulted in a win for the greedy agent, despite our agent winning against greedy when playing as the red player. However, our minimax agent won with the following modification: call minimax with depth 4 at the start of the game, but if the remaining CPU time for the agent is less than 100s, call minimax with depth 2 in order to save time by not searching deeper in the tree towards the end of the game (less number of computations as there are less number of child boards with lower depth values). This modification yielded in our minimax agent winning all its games against our random and greedy agents, regardless of which colour it played as.

2. **Filtering moves**

Instead of finding all possible legal moves, the getOperators() function only returns spawn actions that spawn in neighbouring cells surrounding the player cells and just barely out of reach of the opponent cells.

Evaluation function

Minimax requires an evaluation function to assign a utility value to each board state in order to determine the most favourable move for each player. We implemented the **Multi-Attribute Utility Theory (MAUT)** in order to construct the utility function (Jansen, Coolen and Goetgeluk, 2011). MAUT is a decision-making theory that assigns weights to attributes based on the relative importance of those attributes. The final utility score is then obtained by multiplying the score of each attribute by its weight and summing up the results. The formula for this additive model of MAUT is as follows:

$$V(a) = \sum_{i=1}^{m} w_i v_i(a)$$

where:

V(a) is the overall value of alternative a

$v_i(a)$ is the value score reflecting alternative a's performance on criterion i

w_i is the weight assigned to reflect the importance of criterion i

In the context of Infexion, an alternative refers to a board state in the minimax search tree. We use MAUT to assign a utility value to each board state and run the minimax algorithm accordingly.

Five attributes were chosen initially, namely;

- Difference in power between the two players (*power_diff)*
- Difference in the number of tokens "eaten" by each player *(eaten_diff)*
- Difference in the number of ally cells of each player (number of tokens that can be spread to) *(ally_diff)*
- Difference in the number of tokens of each player *(token_diff)*
- Minimum distance from player cell to any opponent cell *(min_dist)*

These features were assigned arbitrary initial weights, and we used a machine learning technique, TDleaf, to fine-tune the weights of the features for the evaluation function.

TDLeaf

The "TD" in TDLeaf stands for "Temporal Difference," which refers to the difference between predicted value and actual value over time. In other words, it is the difference between utility value at time t + 1 and at time t (Baxtor et al., 1999). A good approximation of the weights would return no difference. Hence, TDLeaf updates them to obtain a minimum difference between predicted and actual value.

We used the TDLeaf learning algorithm to compute the weights by playing the game against itself and updated the weights based on the outcomes of each game. TDleaf updates the weights every turn. The weights gained by TDLeaf are normalised so that it all sums up to 1 to avoid large values that would prevent TDLeaf from effectively adjusting the weights.

The process is outlined below:

PHASE 1: Initialised weights of the features.

As mentioned above, we assigned arbitrary weights to the features before running the tdleaf algorithm. The initial weights we used were:

Initial weights =
{ "power_diff": 0.6, "eaten_diff": 1, "ally_diff": 0.5, "token_diff": 0.3, "min_dist": -0.2 }

PHASE 2: Simulated gameplay thrice with TDLeaf.

We ran three iterations of the game, specifically against our own agent, the greedy agent and the random agent while running the TDLeaf algorithm to update the weights of the features. The final weights obtained after each iteration were normalised to add up to 1 while preserving the signs of the original weights such that our computations are not derailed and made more complicated by the use of large values for the weights.

Each iteration of the game with TDleaf resulted in 3 separate sets of weights, namely weights1, weights2 and weights3. We now needed to find the optimal set of weights out of the 3, that resulted in a faster win for our agent. Therefore, we ran our agent against each opponent (our agent, random agent and greedy agent) for each set of weights obtained, to test which set of weights gave us a faster win on average against the opponents. The statistics are given below:

Minimax Agent vs Random agent weights:

weights1=
{'power_diff': 0.5298628211093089,
'eaten_diff': 1.8436272690352981e-13,
'ally_diff': 9.218136345176491e-14,
'token_diff': 0.40045007157768786,
'min_dist': 0.06968610731292038}

Opponent	Time elapsed	Turns
Agent	68.043s	84
Random	19.235s	37
Greedy	0.221s	5

Table 1: Agent vs Opponents, run with weights1

Minimax agent vs Minimax agent weights:

weights2=
{'power_diff': 0.6346138095733632,
'eaten_diff': 1.8207937231637677e-13,
'ally_diff': 9.103968615818839e-14,
'token_diff': 0.29568628250521206,
'min_dist': -0.06969881111724723}

Opponent	Time elapsed	Turns
Agent	35.074	58
Random	26.713s	41
Greedy	1.798s	12

Table 2: Agent vs Opponents, run with weights2

MInimax Agent vs Greedy agent weights:

weights3=
{'power_diff': 0.4633656408197951,
'eaten_diff': 3.3001207795012265e-13,
'ally_diff': 1.6500603897506133e-13,
'token_diff': 0.24557476587429106,
'min_dist': -0.29105810820857175}

Opponent	Time elapsed	Turns
Agent	43.198s	60
Random	35.299s	47
Greedy	0.458s	7

Table 3: Agent vs Opponents, run with weights3

The weights obtained from the iteration between our agent and the random agent (*weights1*) resulted in the best performance as it beat the random and greedy agents faster than weights2 and weights3. Therefore, the final weights we used were from this particular iteration.

Evidently, 'eaten_diff', 'ally_diff' and 'min_dist' seem to have little to no effect on the final utility value due to their negligible weights. Therefore, we excluded these 3 features in the final runs of the game, and retained only *'power_diff'* and *'token_diff'* and their weights from this iteration, as they had the largest effect on the final weights. Additionally, exclusion of the redundant features results in reduced computation time, with our agent then becoming faster and taking less time to produce a win.

PHASE 3: Simulate gameplay between our agent and other agents multiple times without TDLeaf.

Once we obtain the final weights from running TDleaf iteratively (step 2), we turn TDLeaf off and use those values to build our agent's utility function, using just the 2 features and their weights as described above.

Final weights=
{'power_diff': 0.5298628211093089, 'token_diff': 0.40045007157768786}

We then test our agent against the random and greedy agents to test our agent's performance.

Performance Evaluation

References

Baxtor, J., Tridgell, A., & Weaver, L. (1999). TDLeaf(lambda): Combining Temporal Difference Learning with Game-Tree Search.

Harrington, J. (2009). *Games, Strategies and Decision Making*. Worth Publishers.

Jansen, S. J., Coolen, H. C., & Goetgeluk, R. W. (2011). *The Measurement and Analysis of Housing Preference and Choice*. https://doi.org/10.1007/978-90-481-8894-9_1