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 the greedy agent 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 . 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 utility values 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 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)*

The initial weights were assigned arbitrarily on a scale of 0 - 1 to the features, with 1 being the most important feature. The initial weights that were used can be seen below.

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

Opponent	Time elapsed	Turns
Agent	34.956s	58
Random	51.562s	55
Greedy	3.411s	17

Table 1: Agent vs Opponents, run with initial weights[2]

Since the performance is not optimal in terms of time, TDleaf, a machine learning technique, was then used to fine-tune the weights of the features used in the evaluation functions.

TDLeaf(Lambda)
The TD (Temporal Difference) learning rule is a type of reinforcement learning algorithm that updates values based on the difference between the predicted and actual rewards received from the environment over time. In other words, it is the difference between utility value at time t + 1 and at time t (Baxter 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.

Starting from a set of initial weights, the values are updated at every turn and yield a different set of weights after each iteration of the game against different agents.

The formula used to update the weights is:

$$new\_weights[factor] = weights[factor] + ALPHA * (result - val) * f\_i[factor]$$

where:

ALPHA is the learning rate
f_i[factor] is the value of the feature 'factor' in the current state.
result is the minimax value, which represents the estimated utility of the current state
val is the value calculated using the utility function, which s the actual utility of the current state

[1] min_dist is negative since the agent needs to get the closest distance to opponent
[2] Random has less turns but more time since it generated more tokens on the board and thus, more valid moves are being evaluated which increases the time

The difference between ‘result’ and ‘val’ represents the error in the prediction of the utility value of the state, and this error is used to update the weights of the features.

The implementation process is outlined below:

PHASE 1: Simulated gameplay thrice with TDLeaf.

We ran three iterations of the game using the initial weights against 3 different agents, i.e 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 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 then needed to find the optimal set of weights out of the 3. 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 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	35.074	58
Random	19.235s	37
Greedy	0.221s	5

Table 2: 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	68.043s	84
Random	26.713s	41
Greedy	1.798s	12

Table 3: 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 4: 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' seemed 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 were significant weights. Additionally, exclusion of the redundant features results in reduced computation time, with our agent then becoming faster at making moves.

PHASE 2: Simulate gameplay between our agent and other agents multiple times without TDLeaf (Performance evaluation).

Once we obtain the final learned 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 tested our agent against the random and greedy agents to test our agent's performance.

Performance Evaluation

The agent won all games against both greedy and random opponents whether the agent is red or blue. This proves that the agent is effective at defeating opponents.

Agent playing as the red player

Blue player (Opponent)	Time	No. of Turns
Agent	23.328s	57 turns
Random Agent	5.623s 30.49s 16.5s	29 turns 61 turns 47 turns
Greedy Agent	19.27s	53 turns

Table 5: Agent vs Opponents, run with final weights[3]

Agent playing as the blue player

Red player (Opponent)	Time	No. of Turns
Agent	23.137s	57 turns
Random Agent	3.4s 20.056s 7.13s	26 turns 52 turns 34 turns
Greedy Agent	40.647s	80 turns

Table 6: Opponents vs Agent, run with final weights

Conclusion

In conclusion, through minimax, where the values are obtained using the Multi-Attribute Utility Theory and fine-tuned using TDLeaf, the agent has achieved Nash Equilibrium.

[3] Random is run 3 times due to the randomness of the opponent moves

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