INF8215 - Fall 2022 - Project Report

Fischer team

Team’s name

Our team and agent's name on Challonge was Fischer, inspired by the famous Chess player.
Our members are Hugo Barral (2225621) and Khalil Sabri (2178483).

Methodology

Minimax agent (Fischer)

The agent is based on minimax, and uses alpha beta pruning which reduces the amounts of branches heavily, especially the more depth we go.

Board Evaluation :

Similar to Chess, Evaluating a board consists most importantly of counting the pieces on the board, and giving the right value for each piece. In the case of Chess, the oldest derivation of the standard values is due to the Modenese School in the 18th century[1]. In the case of Avalam we had to find out these values by playing many games, and tuning.

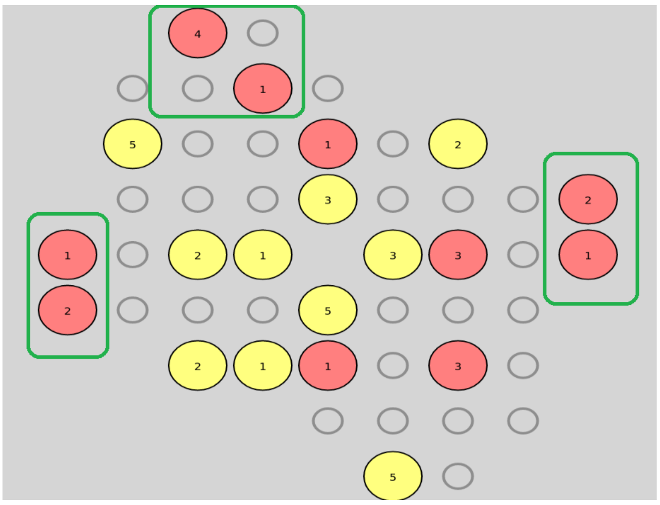
The pieces we have in Avalam are three and we valuated them as follow :

- **Tour of 5 :** **1.55**
- **Isolated pawn :** **1.5**
- **Other Pawns :**
  - **Isolated group of pawns with sum 5 :** **1.55 points**
  - **Isolated group of pawns with sum <5 :** **1.5 points**
  - **Not belonging to an isolated group of pawns :** **1 point**

If we give very high values to Tours and Isolated pawns , the agent will be acting very greedy trying to have as many of them , while the other agent will have more pawns that will turn into Tours and isolated pawns later in the game .

In the case of a draw, Tours of 5 will be considered for the win, for this reason they’re given a slight more value than isolated pawns .

Another very important feature we have added to the Board Evaluation for Minimax agent, is the case where we have an **isolated group of pawns**. For example , in one of the last games , there was a pawn of 1 , and a pawn 4 next to each other . It is very important for the agent to know their real value which is 1.55 , instead of 1+1=2 . Otherwise , the agent will miscalculate the position as an advantage of 0.45 , and until the end will realise so . Having this feature also means that our agent can **create for the adversary such a pawn structure and gain advantage** which also happened in the game the agent won by a difference of two at the end. This exact trick was used against MCTS to win.
The reason for developing this feature was that we discovered this weakness while playing against the agent , and abused it against him and won , while we would lose most of the time before since the agent is better at calculating.



Ranking of Actions :
Since Avalam has a very high breadth that can go to 250 at the beginning of the game , limiting the amount of actions to analyse is necessary for better performance .

Here are the **Heuristics** chosen :

- **If Move yields an isolated pawn :** +5 points .
- **If Move results on tours with higher weight :** +Height of new tour (The higher the height the higher the priority) .
- **Penalty on putting your stone on top of another one of your own stones** : -2.5 points .A penalty is better than removing it , sometimes that can be the best move , for example if such a move results in 3 Tours , which also happened in one of the final games.

Depth Management :
We start by a depth of 5 , and then we increase the depth by one each 9 steps played . For that , around move 18 , we play with depth 7 , and around move 27 we achieve the maximum depth we can get to (depending on how much time left) , which is depth of 8.

Time Management :
Because of the depth 7+ moves which take a long time , in many cases , we can end up in time related issues , for that reason , we have decided to limit the depth to 6 when the time left is less than 135s . Taking moves at depth 6 takes seconds .

Improving the search algorithm :
Here , we try to implement ideas to make the search faster :

1. One of the main ideas that improved the robustness and speed of the agent , is saving the last move in a list , so that instead of copying a new board in each branch , you actually keep the same board , play the move , and then redo the move.
2. Using the right data structure , sets for already visited elements in a stack…

MCTS agent

The agent is based on Monte-Carlo Tree Search.
Some changes were made to the fundamental phases seen in class which will be described here.

Tree saving :
Instead of rebuilding a different tree at each turn, our algorithm saves the successors of the chosen action at each turn end.
At the beginning of the next turn, the algorithm will identify which action was chosen by the opponent and retrieve the current node from the saved ones.
Doing this we can keep our previous simulation and parameters which will be significant in the next phases.

Time management :
As MCTS performances are heavily based on time budget allowed, allocating to each step the correct time before choosing an action is primordial.
We decided to allow more time in mid-game (beginning approximately between 10-12th step) where there's less possible actions and still enough time to turn the game on one side or another.

Our allocation formula was inspired by Huan et al.[2] which looked like this.

$$Allocated \ = \ \frac{TimeLeft}{(b + max(0, MidStep - Step))}$$

With $MidStep$ being the estimated mid-game phase of the game (12 for Avalam) and b a constant for the basic divider to partition the time left (4 in our algorithm).

Selection phase and upper confidence bound :
We used a slightly different selection inspired by the Minimax algorithm.
As our score can range between -10 and 10, the first player would want to choose nodes with the highest estimation and the second one the lowest.

Depending on the current node, our selection will choose the successor with the highest, for the first player, or the lowest, for the second player, upper confidence bound.
To take into account the current node perspective, the UCB1 formula was also modified.

$$UCB1(n) \ = \ \frac{U(n)}{N(n)} \ + \ Sign(Parent(n)) \ \times \ C\sqrt{\frac{ln\, N(Parent(n))}{N(n)}}$$

With $Sign(n)$ being 1 for the first and -1 for the second player.

Exploration parameter :
The exploration parameter C has a theoretical value of $\sqrt{2}$ but we decided to tweak this value a little.
When our agent estimate it's falling behind the opponent be choosing an action with a weak or bad estimation value ($Q(n) \ = \ \frac{U(n)}{N(n)}$), the value of C will go up until we get a satisfying estimation.

To do that we save the estimation of our selected child and on the next turn we modify the exploration value according to this formula.

$$C \ = exp^{\ max(0, w - Q(Parent(n))/Sign(n))} \ \times \ \sqrt{2}$$

With n being the current node and w a constant experimentally chooses to decide when an estimation is considered weak (0.25 in our algorithm).

Expansion phase :
When we reach the last simulated node of our tree, we try to expand it.

Doing so we add all its successors to the tree and then estimate a base Q and UCB score value using a similar board evaluation described in Minimax algorithm.
It is kept more simple (not implementing isolated groups) as it's just an evaluation to determine which one will be simulated first.
From this score we also add the C exploration parameter in order to not bias our choice too much. Even if they are not simulated yet, these values are recalculated if the C value appends to change.

Simulation phase :
When a simulation is launched, it keeps playing until it reaches a final board to retrieve the exact score to backpropagate.
These simulations behaviours are inspired by the e-greedy policy taught in Reinforcement Learning, except we found it's preferable to use more randomness here.

At each stage of our simulation, if we fall in the probability ϵ (40% in our algorithm) our algorithm will select k (5 in our algorithm) movable pawns randomly, even if they are the same ones, and choose the best action possible from this selection.
This choice is made with a score prediction, inspired by the given greedy agent, which estimates the given score from this action, penalises covering a pawn of the same colour of the player and action leading to a pawn which can be captured next turn.

When we don't fall in probability ϵ, our algorithm simply chooses a random pawn and a random action associated with it.
The probability and random selection is calculated with the random Python module and no seed.

Results and evolutions

- Before we started building any of the agents , we played a game between ourselves and **developed the first heuristics at the time** . (Tours and isolated pawn should have more score , putting your stone on another is a bad move , moves that lead to Tours or isolated pawns should be analysed first ….)
- Both of the agents (Minimax and MCTS) were developed in parallel , and each time we would play the two agents between themselves.
- Each time an agent would win , and we would discover weaknesses found in the agent that lost , and work on improving them .
- Two weeks before the tournament , Minimax just improved heavily , and doesn't lose anymore to MCTS on red , and most of the time still wins on yellow .
- Last results between the agents (MCTS vs Minimax) : 7-8 , 6-8 , 5-7 , 7-8 . Minimax wins !
- **Main Result : We won the competition !**

Evolution of Minimax :

- **First version of minimax** had a **depth of 3** , and it was still very slow .
- **Removed cloning of the board** and saving the last move made the agent much faster.
- Implementation of **alpha beta pruning** , its results for depth 3 were not as significant than later on when we worked with more depth.
- **Reducing the breadth of actions** to analyse by ranking actions . This allowed the depth to achieve 6 or 7 depending on the breadth chosen .
- **Evaluation of board** back then was very basic , with Tours of 5 and isolated pawns valued at 5 and pawns to 1 . This led to a greedy agent that gathered as many Tours and isolated pawns in the beginning and lost at the end .
- **Improving** the ranking of actions and the evaluation of board **(Writing better Heuristics & Tuning)**.

Evolution of MCTS :

- **First version of MCTS** was very biassed as the estimation of the expanded nodes where not taking the exploration value into account.
- Time management was **completely static in the beginning**, using our mid-game allocation function showed some improvement in the exploration results.
- Expansion phase **did not use an evaluation at first**, which leads to more uncertainty on which node to choose first, the old functions are still available in our files.
- Simulation phase was also **entirely random before**, it evolved using different functions still available in our files.
- **We stopped cloning the board** for each node just like Minimax, instead we only used one clone to play the simulation as it consumed too much memory.
- **Normalising the score** given by the simulation and the board evaluation was tried but it ended up exploring too much, so we stuck to using the normal score to add some bias which would accelerate our algorithm.
- The tweaking of the exploration parameters was implemented to reup the exploration when falling behind but at first we simply used a value of $\sqrt{2}$

Discussion

The biggest advantage of Fischer(minimax agent) is the way it evaluates the board and selects the actions to analyse by the use of solid and generalistic Heuristics . The other advantage is algorithmic making it fast and robust .

The biggest disadvantage of Fischer is time management, it's harder for minimax to manage time , as it definitely needs to finish its calculations differently than MCTS where you can set up time allocated for a specific move.

Some of the improvements that can be done :

- Improve more the algorithmic part , at each node of the minimax Tree , we evaluate the board again and again and we select the same actions to analyse , a big part of the board doesn't change , and many positions have already been evaluated . Easier said , but harder in reality , our attempt to do so resulted actually in a slower agent that abused memory .
- Similar to the weakness found in 'isolated groups of pawns' , there are probably some other weaknesses that we haven't noticed yet . Basically , Improving more the evaluation of the board .
- Setting up the depth needed depending on the position instead of which step we are in , this will allow some endgames to take the maximum depth possible.
- Implement another agent based on deep RL, similar to AlphaGo[3].

References

[1] “Chess piece relative value,” *Wikipedia*. Nov. 19, 2022 [Online]. Available: https://en.wikipedia.org/w/index.php?title=Chess_piece_relative_value&oldid=1122712884. [Accessed: Dec. 08, 2022]

[2] S.-C. Huang, R. Coulom, and S.-S. Lin, “Time Management for Monte-Carlo Tree Search Applied to the Game of Go,” in *2010 International Conference on Technologies and Applications of Artificial Intelligence*, Hsinchu City, TBD, Taiwan, Nov. 2010, pp. 462–466, doi: 10.1109/TAAI.2010.78 [Online]. Available: http://ieeexplore.ieee.org/document/5695493/. [Accessed: Nov. 09, 2022]

[3] D. Silver *et al.*, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, doi: 10.1038/nature16961. [Online]. Available: https://www.nature.com/articles/nature16961. [Accessed: Dec. 08, 2022]