Assignment Part B

Rohan Hitchcock and Patrick Randell

Our approach for this assignment is built upon minimax search, with an evaluation function trained using machine learning. In this report we discuss searching and pruning strategies in Section 1, the evaluation function in Section 2.1, and our (limited) opening move book in Section 3. Explanation of specific implementation details can be found in the ReadMe file of our submission (if required).

1 Searching and pruning

Our search of the state space is based on minimax search with alpha-beta pruning, plus some additional pruning heuristics to further reduce the search space.

1.1 Depth of minimax

In our final version, our player uses a minimax search to depth three until the time resource budget is almost exhausted, when it reverts to a minimax search of depth one. We considered approaches in which the depth varies, using a shallow search early in the game and a deeper search on moves in the mid and late game which are considered to be critical. This approach was rejected however, because when combined with our opening book (see Section 3), we believe the early game states are strategically important and offers important opportunities to attack. Searching too deep in the early game however may result in exhausting the time budget before the game is finished. Improving the depth of minimax, even for some moves, would likely be the single biggest improvement we could make to our player however.

1.2 State expansion order

Alpha-beta pruning is most effective when better moves are evaluated first in the search tree. With this in mind, we experimented with different expansion orderings and found that generating explosion moves first, followed by prioritising moves which moved a small amount of tokens a long way. This ordering favours a more attacking strategy, as usually the most attacking moves are those in which 1 token is sent out from a larger stack into a group of enemy tokens. This only resulted in moderate speed improvements however which we believe is because the best moves change significantly throughout the game. It follows that an approach which improves significantly on this would require a dynamic evaluation order depending on the current state of the the game.

We included simple dynamic expansion strategy which changes the expansion order when there are less than five stacks on the board. In this case, explosion moves are generated first, followed by prioritising moves which move the largest amount of tokens the farthest distance.

1.3 Pruning heuristics

We also experimented with various pruning heuristics to reduce the search space of minimax. Firstly, we did not generate states that would explode our a player's own tokens when no opponent tokens were nearby. These moves are clearly not beneficial.

When there are less than five stacks on the board, we only generate moves of every other possible distance for each stack (starting with the largest distance). This is because moving to one square is often just as good as moving to one of the adjacent squares, and experiments found it had no noticeable effect on performance. When more distances were skipped, however, our player's performance was significantly degraded.

In order to account for the 4-repeated-state game rule we do not consider any state which our player has already encountered in the game. This is under the assumption that if a state did not result in an improvement of our player's position the first time (such as by removing some opponent's tokens from the board) it will also not do so this time, and so is not worth exploring again. In addition to marginally improving the performance of minimax by reducing the search space, this also helps our player avoid repeating states which the evaluation function overestimates the potentially reward of. This addition particularly helped our player finish games faster in the late game.

2 The evaluation function

Our evaluation is of the form

$$r(\texttt{state}, w) = \tanh\left(\sum_{i=1}^{n} w_i f_i(\texttt{state})\right)$$

where $w = (w_i)_{i=1}^n$ is a vector of weights and $(f_i)_{i=1}^n$ a vector of feature functions. In our final player we had five features (so $n = 5$) and the final weights when playing as white and black were (to 2 decimal places)

$$w_{\text{white}} = \begin{pmatrix} 4.49 & 4.02 & 3.78 & -0.07 & 0.06 \end{pmatrix} \qquad w_{\text{black}} = \begin{pmatrix} 4.49 & 4.02 & 3.78 & -0.03 & 0.06 \end{pmatrix}$$

These weights were obtained via training using the TD-Leaf(λ) algorithm (discussed in Section 2.2). In our implementation, each feature was normalised to (approximately) lie between -1 and 1 to prevent the tanh function being evaluated as 1 for precision reasons. Therefore the features used in the implementation differ from those discussed in Section 2.1 by some constant factor.

2.1 Feature selection

When selecting features we focused on encouraging three types of behaviour in our player:

Taking tokens: This is required to win the game.

Controlling space: Spreading our stacks around the board limits restricts the opponent's moves and provides more opportunities to attack.

Stacking: Improves mobility, providing an advantage in both offensive and defensive play.

To encourage taking tokens we defined the feature f_1 as the signed squared difference in our player's and opponents tokens, that is:

$$f_1(\texttt{state}) = \operatorname{sgn}(n_p - n_o) \cdot (n_p - n_o)^2$$

where n_p and n_o are the number of our player's and the number of the opponent's tokens respectively. Note that by *number of tokens* we mean the sum of the heights of the relevant player's stacks. We would expect this feature to be weighted positively since it is positive when our player has more tokens than the opponent (our player is winning) and negative when our player has less tokens than the opponent (our player is losing). Squaring the difference means that it is better for our player to extend its lead by one token when it is already two tokens up compared to when it is only one token up, thereby encouraging increasingly aggressive play as our player extends its lead. Similarly in a losing game, as our player falls further behind increasingly evasive play is encouraged.

Features to encourage good control of space and proximity to the opponent were the hardest to define. Many features which would likely encourage this behaviour – such as the number of opponent tokens within a fixed radius of our player's stacks, or the shortest distance from our player's stacks to the opponent – made minimax search to a sufficient depth computationally infeasible. We found that the centre-of-mass of our player's and the opponent's stacks was a good balance between being easy to calculate and capturing enough information about the positioning of each player. We defined the centre-of-mass for player $x \in \{p, o\}$ as

$$\texttt{com}_x = \frac{1}{|S_x|} \cdot \sum_{(a,b) \in S_x} \binom{a}{b}$$

where S_x is the set of stack positions for player x. We experimented with several features involving centre-of-mass including the Manhattan distance between our player's centre-of-mass $\texttt{com}_p$ and the opponent's centre-of-mass $\texttt{com}_o$, and the Manhattan distance between our player's centre of mass and the centre of the board. In the final version we used two features f_3 and f_4 where

$$f_2(\texttt{state}) = \frac{n_p}{d_1(\texttt{com}_p, \texttt{com}_o)} \qquad f_3(\texttt{state}) = \frac{-n_o}{d_1(\texttt{com}_p, \texttt{com}_o)}$$

where n_p and n_o are the number of our player's tokens and the opponent's tokens respectively. These two features encourage our player to get reduce the distance to the opponent when it is more than three moves from being able to take any tokens (beyond the depth of the minimax search), and hence feature 1 does not apply. The center-of-mass distance also encourages the player to surround the opponents tokens if possible, since this brings the centre-of-masses close together. The use of two features here, weighted by the number of tokens of each player, is intended to give our player the flexibility to adopt different behaviour in this regard when it is winning and losing. The relative weighting of these two features was left to be learned in TD-Leaf(λ).

We experimented with many features to encourage stacking. In the final version we found the feature f_4 and f_5 defined to be the number of our player stacks and opponent stacks respectively,

$$f_4(\texttt{state}) = s_p \qquad f_5(\texttt{state}) = -s_o$$

were best at encouraging stacking for our machine-learning approach. A slight negative weight for f_4 meant that we were encourages to reduce our spread on the board by stacking, whilst a slight positive weight for f_5 meant that states were better for us when the opponent was not stacked. Another option we perused was using twelve separate features which were the number of stacks of each height (up to a height of twelve). This is an attractive option because it provides the evaluation function a great deal of flexibility to determine optimal stack heights, in balance with other features. However, we found this was not suitable for machine learning: stacks of specific heights occurred quite rarely so those features were often zero resulting in their weights not being updated. If the weight of one stack feature became too low, our player would never make stacks of that height, and thus the weight of that feature would never be improved in training. We found that the weight of the simpler f_2 was much easier to train, and resulted in better stacking behaviour.

2.2 Training the evaluation function

We trained the weights of our evaluation function using the TD-Leaf(λ) algorithm described in Baxter et al. 1999. The value of λ was adjusted throughout the learning process to experiment with the effect of changing this parameter, although ultimately this means that our weights were not obtained using a single value of λ. The learning rate was adjusted manually between 0.1 and 0.01, decreasing it as our player's performance improved. In the event performance began to degrade the weights were reset to an earlier version and learning was resumed with a lower learning rate.

Ideally, our player would train against a series of varied opponents, increasing in skill as our player improved. Baxter et al. 1999 attribute the significant improvement of KnightCap (a chess-playing agent trained using TD-Leaf(λ)) in-part to the varied nature of its opponents which became better along with the agent. They discuss how learning via self-play was not as effective, and even after many more games of self play resulted in worse performance.

In light of this – and lacking a large pool Expendibots players of varying skill levels – we aimed to include as much variation as possible in training. This included training against:

- A player making random moves.
- Other players we were experimenting with.
- Self-play and versions of itself with different feature weights.
- Other groups players in the battlefield environment.

We tried to include as much play in the battlefield environment as possible, but at times these games were hard to come by. Variation in the opponent helped avoid over-fitting a particular player's strategy.

Another factor Baxter et al. 1999 attribute to the success of KnightCap was that it started with a good set of starting weights. We also found this to be the case with our player. In cases where the initial weights were not tuned well enough we found that the player found local maxima where games were very long and it drew frequently (this can be thought of as the player adopting a hyper-defensive strategy). TD-Leaf(λ) doesn't necessarily encourage players to win quickly, but this was an important when considering our resource constraints.

2.3 Using different weights for white and black

After a lot of training, we found that for the learned weights, our player would win consistently when playing as white but often lose or draw as black. This lead to the idea of training different weights for different colours. After making this change we observed an improvement in the performance of the player when playing as both colours. We suggest that this is because playing as black and white require different strategies. White moves first and so it may play more aggressively, while black must be more defensive.

3 The opening book

We included a sequence of three to four (depending on the colour) of fixed opening moves. These moves were found by playing games in real life, and chosen so that they are at least guaranteed to not be detrimental to our player. Initially this was done to help manage our resource budget. The opening moves were some of the longest to evaluate, likely because many branches had similar

evaluations, reducing the capacity for pruning. With a relatively shallow minimax search, the early game is hardest to navigate since the tokens are so far apart and all stacks are size one.

Adding an opening book also had a positive impact on the playing performance of our player. It served to make the transition from early to mid-game quicker, where our evaluation function performs better, good moves are at a depth our minimax can reach, and our evaluation ordering in minimax is better optimised for pruning. It also meant that we always entered the mid-game on our terms, rather than because our opponent had developed their position and advanced, resulting in better performance in the rest of the game.

The opening book for Black is conditional on the opponent. If our opponent has advanced quickly, or formed a stack of 4, black determines on which side of the board (left or right) this has occurred , and creates a stack of 4 in the opposite direction. This has proven to be extremely effective in defending against the opponent's attacks in the early-game, and allowing our player to counter-attack effectively.

References

Baxter, J, A Tridgell, and L Weaver (Jan. 4, 1999). "TDLeaf(lambda): Combining Temporal Difference Learning with Game-Tree Search". *arXiv:cs/9901001*. arXiv: `cs/9901001`. URL: `http://arxiv.org/abs/cs/9901001` (visited on 05/10/2020).