Assignment Part B

Rohan Hitchcock and Patrick Randell

Our approach for this assignment is built upon the basics of adversarial search introduced in lectures. We developed our algorithm using some machine learning techniques, and tried to tailor our strategy to the specific components of the game, learning as we went along. A guide to the code we have developed can be found in the ReadMe file of our submission.

1 Searching and pruning

Our search of the state space is based on Minimax search with Alpha-beta pruning. This seemed like the most intuitive algorithm for Adversarial search, and it allowed us to pass in different expansion strategies, evaluation functions and other conditions to experiment with their affect on our player.

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 3.1), 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. There are instances in a game where searching an extra depth would prove extremely beneficial, but achieving this would require an overhaul of our approach, targeted at optimal move ordering and simple evaluation, which is a different strategy entirely.

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 "BOOM" moves first, followed by moving 1 token from a stack of n, n positions (then 2 tokens, n positions), down to moving the entire stack 1 position, was most effective. 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 rather moderate speed improvements however. We believe that this is because the best moves change significantly throughout the game, and than an approach which improves significantly on this would require a dynamic evaluation order depending on the current state of the the game. We experimented slightly with this idea, and found that simply playing as black meant starting on the defensive, and thus moving larger stacks their maximum range away were usually the best moves at the start of the game. Many of the experimental changes ended up being slower later in the game, and were not kept for that reason. However, we did decide on a dynamic strategy in the end, which is used when there are 5 or less remaining stacks on the board. This strategy generates "BOOM" moves

first, followed by moving an entire stack of tokens its maximum range, down to moving 1 token from the stack, 1 position. This strategy also only generates moves for every second distance, to reduce the expansion order. For example, with a stack of size 4, the only distances generated were 4, 2 and 1. For 5, it would be 5, 3 and 1. 1 is always generated. We found that this sped up the end game process slightly, without affecting performance. Should we talk about PVS here?

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 own tokens without any opponent tokens in the explosion radius. These moves are clearly not beneficial, and this seemed to be a pretty clear reduction of the search space. As discussed in the previous section, we also experimented with skipping certain distances for a stacks in the late game. This change was reasonably effective when skipping every second distance. However, when more distances were skipped, our player began to lose against moderate players. 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 function was carefully designed to include the most important aspects of the game without sacrificing speed. It has 5 normalised features, with slightly differing weights when playing as black vs white.

$$eval(s, W) = \tanh \sum_{i=1}^{5} w_i * f_i(s)$$

where s is the current state, and W is the weight matrix.

$$\mathrm{W}_{black} = [4.48799476, 4.0206573, 3.77726706, -0.07250787, 0.06310117]$$

$$\mathrm{W}_{white} = [4.48799476, 4.0206573, 3.77726706, -0.03250787, 0.06310117]$$

The features are described in detail below.

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.

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

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

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

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

where n_p and n_o are the number of our player's and the number of the opponent's tokens respectively. Perhaps sum of their stack heights is more accurate? 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). The squared difference means that the more our player is winning by, the further encouraged it is to increase its lead and close out the game. Conversely, the more our player is losing by, the more it is encouraged to reduce the opponents lead.

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

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

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.(Again I think sum of their stack heights explains it better, thats actually what were using). These features proved to be effective, for the most part. Due to a limited depth of 3, we encountered an issue where, without a large stack, and without being within a 1 column/row distance from an opponent, the first feature would not be applied to states reachable in 3 moves time. Feature f_2 is extremely important in these scenarios, as it acts as a fallback feature (as with f_4) when taking opponent tokens is not a foreseeable option. This features encourages moving our tokens closer to the opponents, in hopes that taking opponent tokens will then become both available, and the primary objective, due to the relative weights of the features. f_3 acts as the defensive counterpart to f_2. To help explain,

imagine that the weights for f_2 and f_3 were equal but opposite. They could then be combined as shown:

$$\frac{n_p - n_o}{d_1(\texttt{com}_p, \texttt{com}_o)}$$

Here, it can be seen that if we are losing (the sum of the opponent stack heights is greater than ours), the numerator would be negative. A better state would then be one where the COM distance is greater, reducing the magnitude of the fraction and effectively encouraging our to increase the distance. If we are winning, we should reduce the distance. The slight difference in the actual weights of these features results in a more attacking and opportunistic player. Our player will not run unless it is losing badly.

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,

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

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, with λ =0.8. This value was changed many times throughout the learning process. As we played more games, we gained a better idea of how critical early game moves were, and so this value started lower and was adjusted to be higher towards the end. The learning rate started at [0.1], as we started the weights in the regions in which we thought would make sense. It was gradually decreased to [0.01] as performance improved. This gradual change was quite manual, a lot of time was spent watching the performance of our algorithm, and it was often "rewinded" to previous weights, given a lower learning rate, and set off to learn once more.

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.

After a lot of training, it was interesting to find that with a given set of weights, our player would win as one colour, but lose as the other, against the same opponent. This lead to the idea of training different weights for different colours. It became pretty clear why this would be, as the first move advantage given to white leaves black starting on the defensive foot (This also lead to the adoption of the Opening book moves). Our final weights differ only in that for f_4, where black is given a more negative weight, encouraging stacking slightly more than white. This makes sense, as black usually needs to stack in order to escape the intial attacks of a good white player, and strike back.

3 Other aspects

3.1 Opening book

We included a sequence of 3 - 4 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 slightly 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 occured (within reason), and creates a defensive stack of 4 in the opposite direction. This has proven to be extremely effective in reducing the effectiveness of enemy attacks in the beginning, and allowing us 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).