Assignment Part B

Rohan Hitchcock and Patrick Randell

Overview of our algorithm.

1 Searching and pruning

Our search of the state space is based on a minimax search with alpha-beta pruning. More explination here?

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.

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 description of move ordering was most effective. 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.

1.3 Pruning heuristics

We also experimented with various pruning heuristics to reduce the search space of minimax. discussion of heursics / why included and not includeded. 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

Very brief description of evaluation function including expression in terms of features and weights

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. 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). Why signed squared difference?, Include discussion of the number of opponents stacks here too?

We experimented with many features to encourage stacking. In the final version we found the feature f_2, defined to be the number of our players stacks

$$f_2(\texttt{state}) = s_p$$

was best at encouraging stacking for our machine-learning approach. 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.

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

$$f_3(\texttt{state}) = \frac{n_p}{d_1(\texttt{com}_p, \texttt{com}_o)} \qquad f_4(\texttt{state}) = \frac{n_o}{d_1(\texttt{com}_p, \texttt{com}_o)}$$

where n_p and n_o are the number of our player's tokens and the opponent's tokens respectively. Discuss why these features.

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 $\lambda =$check lambda val, why this val?. The learning rate started at [val] and was gradually decreased [how?] as performance improved.

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.

Discuss difference in weights for white and black

3 Other aspects

3.1 Opening book

We included a sequence of how many? 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.

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).