CS 221 P-Progress

Connor Settle

November 2018

1 Introduction

To make my game playing agent for Abalone, I've decided to use a combination of TD Learning, the Minimax Algorithm, Depth Limited Search, and Alpha Beta Pruning to provide an efficient, yet flexible way to teach my agent to play the game. The algorithm will train against itself by playing the game against itself, trying to either minimize or maximize the value of the board depending on which side it's playing.

More specifically, my agent will use an Alpha-Beta pruned Minimax Algorithm with limited depth to choose each move, resulting in a fairly efficient way to look a few turns ahead. It will then use the state, action, reward, new state loop to update the weights vector associated with the features I'll define for use with TD Learning. This way, my algorithm has the capacity to learn and update its weights vector over training time, but can also just run as a game playing agent if I pass in a weights vector for use in evaluating each game state, so I can show off the results once it's done training.

2 Algorithm Specifications

2.1 Overview

As specified in P-Proposal, a game state will consist of a 2D array representation of the game board. 0 means no ball is at that position, -1 is a black ball, and 1 is a white ball. As a concrete example, the initial game state for my smaller test version of the Abalone board is as follows:

	0	1	2	3	4
0			1	1	1
1		0	1	1	0
2	0	0	0	0	0
3	0	-1	-1	0	
4	-1	-1	-1		

A transition, also specified in P-Proposal, is a tuple containing information about where to move which balls. The tuple is in the form of ((row, col), movement direction, direction of attached balls, number of balls attached). As a concrete example, one of the transitions that my current algorithm often uses as its first move is ((1, 3), 3, 0, 2). This means that starting from position (1, 3), we will move 2 balls in direction 3, starting with the ball in position (1, 3) and taking another ball from direction 0 with it. This move is specified further as follows:

	0	1	2	3	4			0	1	2	3	4
0			1	**1**	1		0			1	0	1
1		0	1	**1**	0		1		0	1	**1**	0
2	0	0	0	0	0	=>	2	0	0	0	**1**	0
3	0	-1	-1	0			3	0	-1	-1	0	
4	-1	-1	-1				4	-1	-1	-1		

Every game state will have a value associated with it, calculated using the function $V(s; w) = w \cdot \phi(s)$, in which s is the game state, w is the weights vector, and $\phi(s)$ is the feature vector detailed later in this report.

As the algorithm operates, it will use $V(s; w)$ to estimate the value of future game states, and apply the Minimax Algorithm to calculate the best next move. Alpha-Beta pruning will be used to limit the number of states that need to be searched, and the search will also be depth limited at a depth $d = 3$. This depth is necessary due to the ludicrously large state space for Abalone games.

Additionally, at any given move, there will be a slight chance ϵ that a random move will be taken rather than the minimax move. This is because we need the algorithm to explore a bit more, or else it could get stuck exploiting local minima/maxima.

After taking a move, the algorithm will analyze the current s, a, r, s' and use it to update the weights vector based on the calculation $w \prec w - \eta[V(s; w) - (r + (s'; w))]\nabla_w V(s; w)$.

Using this algorithm, what we're doing is training the weights vector, since the weights vector defines the value of each game state. Thus, once training has been completed, the algorithm can simply output its current weights vector for use as input later on, and can use an inputted weights vector as the driving force behind which state to choose during gameplay.

2.2 Features

Initially, I had a set of features designed to specifically target optimal or desirable strategies for the game. I wanted to incentivize certain things, like clumping and attacking.

My initial features were as follows:

(a) [**number of shared edges**]

this feature was designed to incentivize clumping, as the more edges are shared between balls of the same color, the more clumped up the balls are

(b) [**ratio of black:white**]

this feature was used as a tracker for how far the game had gone. Balls being removed would alter the ration of black balls to white balls.

(c) [**distance from the center**]

adding up the distance from the center of all the balls on each team was a way if trying to get the agent to move it's pieces closer to the middle, away from the dangerous edges

These features were not much more than an extension of my initial baseline, only reformatted into a feature vector. They were just as ineffective as my baseline, perhaps even more so. They aren't very expressive, and don't provide enough breadth for the algorithm to optimize. After reconsidering my approach and researching other implementations of TD Learning, I created a new set of features, which is defined as follows:

(a) [**black ball in position (x, y)**]

[**white ball in position (x, y)**]

having two features for every space in the board, one indicator of a black ball in that spot and one indicator of a white ball, is a great way to keep my features generaland let the learning figure out which slots are better

(b) [**black's turn**]

[**white's turn**]

the TDGammon example from the slides used which player's turn it was as a feature, and I can see how that would be useful, so I included it in my feature set

(c) [**# of black balls lost / number needed to win**]

[**# of white balls lost / number needed to win**]

this one is not an indicator feature, and it measures the progress towards defeat for both colors

I may add back in some of the old features in the future, especially the one regarding the number of shared edges, if I want to push the algorithm in a specific direction. For now, though, I'm trying to keep it general and let the learning do most of the work.

2.3 Reward

My initial iteration did not have any rewards, relying solely on features with predetermined weights to operate. After some time, I introduced a set of basic rewards based on which player is which. Following my earlier setup in which white is trying to maximize and black is trying to minimize, my current rewards are as follows:

(a) +20 for a black ball being removed

(b) -20 for a white ball being removed

(c) +100 for white winning the game

(d) -100 for black winning the game

I'm considering adding other rewards in the future, if these ones are either not expressive enough or don't happen often enough. My future rewards are as follows:

(a) +5 for pushing a single black ball

(b) +10 for pushing 2 black balls

(c) -5 for pushing a single white ball

(d) -10 for pushing 2 white balls

These rewards would more closely incentivize the act of forcing your opponent around the board, and might help accelerate stagnant play. I'll begin testing with these rewards if my current efforts don't yield strong enough play.

The reason I'm hesitant to use these is that the actual game does not reward the act of pushing balls, other than the fact that pushing is generally a way to force your opponent into a worse position. I feel that providing a numerical reward for performing such an action might move the agent's focus away from maximizing the value of the actual game. However, if it helps the agent be more aggressive in play, then it's something worth pursuing, as Abalone agents often tend to settle into more defensive patterns [1].

3 Preliminary Testing

I began my testing on the default Abalone board, which has a hex width of 5 (meaning each side of the hexagonal board is of length 5) and started with 14 black balls and 14 white balls. 6 balls must be removed on one side in order to win the game. Later, I switched to a smaller hex width of 3, with only 2 balls needed to win, and another version of hex width 4 and 3 balls pushed to win. For preliminary metrics of success, I used the number of turns taken to finish the game. This isn't exactly the best metric, since it could vary greatly depending on the effectiveness of the agent, and also because an agent might actually be better if it can last longer, prolonging the game, but since I doubt my agent has learned enough for such strategies to be relevant, I went with it. The tests I ran are below:

First test

- Initial features

- Board hex width: 5
- Games to play: 1

Result: not a single game even finished. I let the program run for about 5 minutes before cancelling it. It seemed that my features were too vague to push the game towards victory, so I began changing things.

Added max moves per game before calling it a draw

- Initial features
- Board hex width: 5
- Games to play: 15
- Max moves: 10000

Result: all games were draws. Over the 15 or so tests I ran, neither side ever emerged victorious before the max moves was reached.

Lowered board size

- Initial features
- Board hex width: 3
- Games to play: 100
- Max moves: 10000

Result: all games now finished. With only 5 balls of each color, and 2 pushed balls needed to win the game, the game now finishes every time. The average number of turns taken until victory over 100 games is 10.67 turns.

Added reward

- Initial features
- Basic rewards
- Board hex width: 4
- Games to play: 100
- Max moves: 10000

Result: I added the basic rewards system, and even tried increasing the hex width to 4. All the games finish, and even with the increased complexity, the average number of turns to finish over 100 tests is 27.23 turns. The reward system seems to be helping push the agent towards more aggressive play.

Updated features

- Newer, more general features
- Basic rewards
- Board hex width: 4
- Games to play: 100

- Max moves: 10000

Result: This actually seemed to hurt the algorithm at first, as it didn't have specific goals to go after. What I found interesting is that the number of moves it took to finish the game actually slightly decreased as the number of games being played continued. In the first iterations, games would take anywhere from 75 to 300 moves. It seemed as if the game was only won randomly. By the end, the number of moves was in the 50 - 150 range, which is not exactly a stellar improvement, but does indicate a level of learning beyond what my initial features provided.

4 Conclusion

Based on this preliminary testing, I now have a way forward. My algorithm should be able to train my agent on smaller scale boards with no trouble, so I'll be focusing primarily on boards of hex width 3 or 4 to begin with. I still haven't gotten the agent to win either way on a board with the default Abalone hex width of 5, but I'll try training the agent on smaller boards first before applying this learning to a larger board to see if that helps. I'll continue refining my features as well, since the ones I have now are expressive but generic, and I'm sure I can come up with some more interesting ones to help push the algirhtm towards success. I can also try tweaking the rewards as well, if need be. Lastly, there are a lot of parameters used in TD learning, including the discount factor, the chance of making a random move, or then step size for updating the weight vector based on the gradient. I can tweak all of these parameters and retrain, to see how different values change the outcomes of training.

5 References

[1] Lee, Benson, and Hyun Joo Noh. "Abalone - Final Project Report." www.cs.cornell.edu/ hn57/pdf/AbaloneFinalReport.pdf.