CS1410 Final Project Writeup

1. Introduction

"Game of Thrones" is a two player game in which players compete for control of territory. Players capture territory by moving over unclaimed tiles. Players can win the game by controlling more than half the territory, or by intersecting the opponent's "trail" tiles ("trail" tiles are created when a player moves through unclaimed territory, and remain on the board until the player returns to their own territory). Players lose the game if their trail is hit by a "white walker," which are "third party" opponents of both players that randomly move across the board. Furthermore, players lose if they collide with the border of the map, or if they collide with their own trail tiles.

This paper outlines two bots for GoT. The first bot uses a modified version of alpha-beta pruning to determine the move it should make. Because it is not possible to enumerate the entire tree of game states in GoT, this bot relies heavily on an explicitly defined heuristic to determine the "value" of intermediate game states. The other bot also uses the same modified version of alpha-beta pruning. However, rather than using a heuristic function with explicitly defined rules and rewards, the second bot relies on a heuristic function generated entirely through machine learning.

2. Bot #1: Adversarial Search Without ML

As mentioned, this bot uses "modified" alpha-beta pruning as well as an explicitly defined heuristic to select moves in the game. This section will discuss the design of this bot, with a particular focus on the design of the heuristic function. This section will also highlight important design choices made during the development of the bot.

2.1 The Game Tree

The game tree used by this bot is fairly straightforward. Nodes of the tree are game states. Each node's children are the set of all game states that are reachable by single moves (either right, left, down, or up) by the player whose turn it is.

The tree is traversed using alpha-beta pruning with a maximum depth of 4 (where the depth of the root node is 0). I chose to traverse to depth 4 simply because traversing any deeper caused the bot to occasionally take longer than one second to make a decision. When the alpha-beta pruning algorithm reaches nodes of depth 4, it calls a heuristic function to assign value to the state. When the algorithm reaches terminal nodes, it assigns a large positive reward to states where the bot wins, and a large negative reward to states where the bot loses. The magnitude of the reward is inversely proportional to the depth of the terminal state.

2.1.1 Selective Pruning of Tree Nodes

While the game tree itself contains all reachable game states for each state, this bot only considers a subset of the possible moves for each game state. Specifically, this bot uniformly excludes moves which will cause the bot to immediately die (moves which cause the bot to collide with its own trail or with the border), and moves which put the bot or its trail tiles dangerously close to a white walker. The decision to design this bot's decision function so that it never makes moves which cause it to immediately die was quite straightforward. However, the decision to design the bot so that it never makes moves which put it in danger of being hit by a white walker was more complex.

There were two major factors behind my decision to design my bot so that it does not consider moves which put it in danger of being hit by a white walker. First, as far as I can tell, simply returning to one's territory when a white walker gets close is a good strategy for handling white walkers. While this strategy may not be the most effective way to absolutely maximize the territory the bot captures, since the bot will occasionally return to its territory when it is not necessary to do so, this strategy drastically reduces the probability that the bot dies from the white walker. Designing the bot so that it doesn't consider moves which put it in danger of being killed by white walkers means that the bot will always return to its own territory when a white walker gets close. Second, and perhaps more importantly, the provided transition function does not update the position of the white walker. This means that when the heuristic function is determining the value of states, it cannot take into account the actual position of the white walker in that state. Simply excluding moves which cause the bot to be in danger of being hit by the white walker circumvents this issue.

After deciding to exclude moves which put the bot in danger of being killed by a white walker from the set of possible moves for my bot, I had to determine the best way to classify states as "dangerous." After some trial and error, I found that comparing the distance from the white walker to my bot and my bot's trail tiles was a good way to determine the "safety" of game states.

2.2 The Heuristic Function

Outside of moves which guarantee that the bot will die, and moves which put the bot in danger of being killed by a white walker, the behavior of the bot is almost entirely dictated by the bot's explicitly defined heuristic function, which assigns rewards to intermediate (non-terminal) game states. The heuristic function is made up of several components, each of which is described in detail below.

2.2.1 Captured Territory

The heuristic function rewards states where the amount of territory controlled by the bot increases, and penalizes states where the amount of territory controlled by the bot decreases. The rationale behind this aspect of the heuristic function is fairly straightforward: capturing territory advances the bot toward victory, and losing territory does the opposite. It is also important to note that the reward for capturing territory is greatly increased when the bot does not yet control a large amount of territory. I decided to increase the reward for capturing territory in the early stages of the game after observing that it was important that the bot stay near its own territory at the start of the game. Before increasing the reward for capturing territory early in the game, my bot would occasionally make it impossible to return to its own territory by surrounding its territory in trail tiles. This would essentially guarantee that my bot would lose.

2.2.2 Trail Tiles

The heuristic function rewards states where the number of trail tiles increases, and penalizes states where the number of trail tiles decreases. The rationale behind this component of the heuristic is similar to the rationale for the "Captured Territory" component of the heuristic function described above. The magnitude of the reward for each additional (or fewer) trail tile is similar to the magnitude of the reward for captured territory, except for at the beginning of the game, when the reward for captured territory is far greater. The magnitude of the reward for trail tiles is also increased when the trail tile is surrounded by whitespace. In other words, states where trail tiles are created which are not adjacent to the bot's own territory are rewarded more generously than states where the new trail tiles are adjacent to the bot's territory.

By assigning reward in this way, my heuristic function incentivizes my bot to venture away from its territory. I found that getting the bot to venture away from its territory was essential to the bot's success, especially in large maps against the second TA bot.

2.2.3 Distance From Territory

This component of the heuristic is another piece of how my heuristic balances incentivizing the bot to venture away from its own territory (high risk, high reward), and incentivizing the bot to stay near/within its own territory (low risk, low reward). In short, this component of the heuristic assigns no penalty or reward when the bot is close enough to its own territory ("close enough" depends on map size – it is within 4 tiles for the large map), and increasingly penalizes states where the bot is far from its own territory. My rationale behind this part of the heuristic function is that in the vast majority of cases, the risk of straying "very" far from one's own territory heavily outweighs the reward of doing so. I added this component of the heuristic function later on in the development process, after seeing that my bot was frequently losing games as a direct result of straying too far from its own territory.

2.2.4 Ratio of Trail Tiles to Territory Tiles

This component of the heuristic function is yet another way in which my heuristic ensures that the bot is neither excessively risk-averse nor excessively risk-loving. As implied by the title of this section, this part of the heuristic function assigns value to states based on the ratio of the number of the bot's trail tiles on the board to the size of the bot's territory. States with a trail tile to territory size ratio of under 0.5 are neither rewarded nor penalized. States with a trail to territory size ratio over 0.5 are penalized, and the magnitude of the penalty increases as the trail to territory ratio continues to increase. My thinking behind this component of the heuristic is that, based on my observations, it is generally not a good strategy for a GoT player to create an excessively long trail, as doing so puts the player at significant risk of being killed by an opponent or by the white walker.

2.2.5 White Walker and Opponent Proximity

This component of the heuristic a) penalizes states where the opponent or white walker is close to my bot or its trail and my bot is outside its territory, and b) rewards states where the opponent or white walker is nearby and my bot is inside its own territory. The idea behind this part of the heuristic function is to penalize states where my bot is in danger of being killed, and to reward states where my bot is keeping itself out of harm's way. Penalties for states where the bot is close to a white walker or opponent while outside its territory increase as the distance between the opponent/white walker and my bot decreases.

It is important to note that I decided to incorporate the position of the white walker into my heuristic despite the "selective pruning" of tree nodes which generally prevents my bot from getting too close to a white walker (described in section 2.1.1). My thinking here is that even if my bot doesn't directly benefit from the way in which the position of the white walker is incorporated in the heuristic function, incorporating the position of white walkers into the heuristic function increases my bot's capability to predict the moves its opponent will make.

2.2.6 Board Position Heuristic

This component of the heuristic simply assigns a small reward to states where the bot is further from its starting position. I created this part of the heuristic after seeing that my bot would occasionally move back and forth between two tiles for many consecutive turns. My thinking being this part of the heuristic is that if my bot is uniformly incentivized to venture away from its starting position, it is less likely that my bot will end up getting "stuck" in a single position.

2.3 Reflections on Development Process

My strategy for this bot was always to use alpha-beta pruning with an explicit heuristic, so most of the development process simply involved iteratively improving the heuristic function until the bot met the performance requirements. My strategy for the heuristic function changed

significantly throughout the time I spent working on the project. Initially, my heuristic function only took into account the *total number* of territory tiles and trail tiles, (rather than the change in the number of trail/territory between the root of the game tree and the state being evaluated), as well as the proximity of the opponent and the proximity of the white walker. I also made an attempt to incorporate the amount of territory controlled by the opponent in the heuristic function, but was not able to do so successfully.

As I progressed through the project and started facing stronger opponents, my heuristic function became more and more complex. I had to do quite a lot of trial and error to determine the best relative weights for different components of the heuristic function. I also had to do a lot of experimentation to derive a heuristic function which caused the bot to effectively adapt its strategy depending on the size of the map and the opponent's strategy.

3. Bot #2: Adversarial Search with ML

Like bot #1, this bot uses alpha-beta pruning to determine which move it should make. This bot also uniformly excludes moves which cause it to immediately die, and moves which put it in danger of being killed by a white walker. However, unlike bot #1, this bot's heuristic function is not an explicitly defined set of rules and rewards. Instead, this bot's heuristic function is a neural network which takes in information about the game state, and outputs a predicted reward for that state.

3.1 The Heuristic Function

The inputs of the neural network are several pieces of information about the game state. Specifically, the inputs are: 1) the difference between the amount of territory claimed by the bot in the state being evaluated and in the state at the root of the game tree, 2) the difference between the number of the bot's trail characters in the state being evaluated and in the state at the root of the game tree, 3) the difference between the number of the bot's trail tiles *which are surrounded by whitespace* in the state being evaluated and in the state at the root of the game tree, 4) the distance from the bot to the border of the map, 5) the distance from the bot to the opponent's trail, 6) the distance between the bot and the bot's trail to the white walker, and 7) the distance between the opponent, the bot, and the bot's trail.

My approach for determining the input types for the network was informed by what I learned while working on bot #1 (the non-ML bot). As a result, many of the inputs to the neural network are similar to the "inputs" considered by bot #1's explicitly defined heuristic function.

The network itself consists of two hidden layers. The first hidden layer has 64 elements, and the second hidden layer has 12 elements. The output of the network is a single number (since the network outputs the value of the state being evaluated). The values of the hidden layers are also activated using the tanh function. I chose the tanh function in part because it has a range of (-1,1), unlike other popular activation functions like sigmoid, which has a range of (0,1). I felt

that it was important for it to be possible for the values of the hidden states to be negative, given the nature of the inputs and output of the network.

3.2 Training

In order to train the heuristic function, I used an "evolutionary" process where I create a "generation" of bots, run games with each bot in the generation, and use the weights and biases from the bot which performs best as a "starting point" for the weights and biases for the next generation of bots. My thinking in choosing to pursue this strategy was that it would be a reasonably straightforward way to learn a heuristic function. The following is a step-by-step overview of this process:

1. Load the weights and biases from the best performing bot from the previous generation, or instantiate new weights and biases. If instantiating new weights and biases, instantiate weights using the normal distribution with mean=0, and instantiate the biases as 0.
2. Construct the new generation of bots. Each generation has 16 bots, where each bot has a different set of weights and biases. Instantiate the weights and biases as follows:
    a. Instantiate 4 bots with the same weights and biases as the best bot from the previous generation
    b. Instantiate 8 bots with weights and biases generated via the normal distribution with mean = (weight/bias values from best bot from previous generation) and standard deviation = learning rate. For most of my training, I used a learning rate of .03
    c. Instantiate 4 bots with weights and biases generated via the normal distribution with mean = (weight/bias values from best bot from previous generation) and standard deviation = learning rate * c (where c is some value > 1).
3. After creating the new generation of bots, run a/several game(s) for each bot against different opponents, and/or on different maps. I did most of my training by having each bot play against the second TA bot on the large maps, and against the attack bot on the small map.
4. Using the *reward function* (described in detail in section 3.2.1), determine how well each bot did in all of the games it played.
5. Repeat steps 1-4 using the bot which was given the highest reward by the reward function.

**Note: The training process for this bot was very incremental. I would frequently save weights and biases generated from a couple rounds of training, modify my hyperparameters, or even modify the games which my bots played, and then re-start training starting from the saved weights and biases.

3.2.1 The Reward Function

Perhaps the most important part of my training process for my heuristic function is the reward function. The way in which the reward function assigns value to end-of-game states is fairly straightforward. First, the function assigns a reward based on the difference between the amount of territory controlled by my bot and by the opponent. My thinking here is that in general, controlling more territory leads to winning the game. Second, my function assigns reward based on whether my bot won. Importantly, reward is only assigned to a bot for winning when that bot wins all of the games it played during that generation. So, bots had to win against the second TA bot in the large map, and against the attack bot in the small map, in order to be rewarded for winning.

3.3 Reflections on Development Process

My approach for this bot changed significantly over the course of the development process. Initially, I attempted to implement a perceptron-like algorithm which would learn weights for each of the components in the heuristic function used by Bot #1 (unlike the final version of Bot #2, the early versions of the bot used an explicitly defined heuristic function as well as learned weights). However, I wasn't very successful with this approach.

I then decided to switch to the "evolutionary" approach described above where the heuristic function is entirely learned. Initially, I used a neural network with only one hidden layer, and I used sigmoid for my activation function rather than tanh. However, I found that my algorithm was more successful with two hidden layers and tanh activation. My reward function also changed significantly over the course of the design process. At first, I took into account how many moves it took for my bot to win/lose, with the idea being that a bot which wins more quickly is "better" than a bot which takes longer to win. I also initially awarded bots for winning games even if they did not win all of the games they played in a given round of training.

4. Results

Results were generated by running games between my bots and each of the opponent bots (RandBot, SafeBot, AttackBot, TA1Bot, TA2Bot). For each combination of bots, I ran ten games (5 where my bot started first, 5 where the opponent started first) on the large map with the white walker, and on the small map without white walkers.

4.1 Bot #1 (non-ML)

This table shows Bot #1's win-rate for each of the opponent bots, on the large map with a white walker, and on the small map without a white walker.

Map:	Starting Order:	Opponent:				
		RandBot	AttackBot	SafeBot	TA1Bot	TA2Bot
Large with White Walker	My bot first	100%	100%	100%	100%	100%
	My bot second	100%	100%	100%	100%	100%
Small w/o White Walker	My bot first	100%	100%	100%	40%	100%
	My bot Second	100%	100%	100%	60%	40%

4.2 Bot #2 (ML)

This table shows Bot #2's win-rate for each of the opponent bots, on the large map with a white walker, and on the small map without a white walker.

Map:	Starting Order:	Opponent:				
		RandBot	AttackBot	SafeBot	TA1Bot	TA2Bot
Large with White Walker	My bot first	100%	40%	100%	100%	0%
	My bot second	100%	60%	20%	100%	0%
Small w/o White Walker	My bot first	100%	100%	100%	40%	40%
	My bot Second	100%	100%	100%	100%	100%

**Note: the win-rates reported above are win-rates from a test I did on my laptop. The win-rates here are not necessarily reflective of the win-rates my bots achieved on Gradescope.

5. Discussion and Conclusion

I was quite pleased with the performance of Bot #1.With the exception of the test for the first TA bot on the small map (where I lost 1.2 points), I was able to meet all of the performance thresholds outlined in the handout. I was also particularly pleased with the way in which my bot adapted its strategy based on the size of the map. On the small map, my bot is fairly conservative, and rarely goes farther than 1-2 tiles away from its territory. On the other hand, while playing on the large map, my bot is far more aggressive, and sweeps up territory quickly. I think that Bot #1's ability to adapt its strategy contributed significantly to Bot #1's excellent performance on both large and small maps, particularly against the second TA bot, which tends to capture terrain much more quickly than all of the other bots

Bot #2's performance was clearly not as good as Bot #1's. However, I was still fairly pleased with the bot's performance, despite the fact that Bot #2 ended up missing a lot of points in the autograder tests for the second TA bot. I think that the biggest issue with Bot #2 was that unlike Bot #1, it was not really able to adapt its strategy to different map sizes. As a result, while Bot #2's play-style was successful in the small map, Bot #2 was simply too risk-averse to perform well in games on the large map, particularly against the second TA bot.

I think there are a number of ways in which the performance of each bot could be improved. For Bot #1, there is certainly more fine-tuning which could be done to improve the heuristic function. In particular, I think performance could be significantly improved by optimizing the weights of each of the various components of the heuristic function. I also think performance could be improved by adding entirely new components to the heuristic function. For instance, a component which encourages the bot to attack the other bot could significantly improve the bot's win-rate, particularly on small maps where games are often won by one bot killing the other. A component which, in some way, takes into account the number of remaining steps in the game could also improve the performance of the bot.

For Bot #2, there are many ways in which performance could be improved. First, there are probably several ways in which to optimize the architecture of the neural network model used for the heuristic function. Changing the input types, changing the number of hidden layers, changing the size of the hidden layers, and changing the activation function are all ways in which the heuristic function could be improved. Second, and perhaps more significantly, I think that optimizing the reward function could significantly improve the bot's performance.