Arun Kavishwar
CS1410
Professor George Konidaris

CS1410: GoT Final Writeup (Capstone)

Abstract

Game of Thrones (GoT) is a zero-sum, alternating two player game where agents try to capture space on a board or attack the enemy while avoiding the white walker. In this report, we present two agents to this game, one using a conditional-based approach (bot 1) and the other using machine learning in a sample-based optimization approach (bot 2). Bot 1 uses a series of conditional algorithms to determine the best place to move based on analyzing the current state and its past moves, while bot 2, a capstone project, uses an approach derived from alpha-beta pruning to predict future states and make an optimal decision. Both bots performed exceptionally well against the test bots, with bot 1 passing most of the assigned thresholds and bot 2 achieving an average 100% win rate against the RandBot, SafeBot, and AttackBot. Overall, implementing both bots took over 50 hours.

Bot 1: Conditional-Based Bot

Process / Design Decisions

In building my bot, I used an iterative approach to slowly improve my gameplay. I would test my bot in certain scenarios, such as an empty room against the safe bot or a large room with white walkers against the attack bot, and tweak my implementation to achieve more victories. The concept of this bot is a combination of the SafeBot and the AttackBot, as it will attempt to gather as much space as possible and then attack the enemy if they come near. I decided to use only conditionals for this bot to really embrace the non-ML philosophy required of this agent, as with only basic algorithms, the bot was doing nothing remotely close to machine learning. The following sections describe the detailed process of this bot's creation with explanations for major design decisions.

Basic Functionality: Safety

Basic functionality was defined as being able to survive in a round for a reasonable amount of time. Getting the bot to move at all took a deceptive amount of time, as I had to understand the sheer amount of code that the game's stencil contained. Poring over the AttackBot and SafeBot implementation, I learned more about GoT's state representation, and was able to write a basic function to move left. Manually playing the game, I realized that a good strategy to start with would just be to gather space and hope the white walker ended up defeating the enemy for me.

As such, I was able to implement loops that moved my bot in circles of an increasing radius, gathering squares without regard to anything else on the board.

This simple strategy did surprisingly well, but I would fall victim to white walkers anytime they approached me. As such, I added another piece of logic to “retreat” from white walkers, breaking the current circular loop and running back to the safe squares if the white walker was within a certain distance of my agent, waiting in the safe squares until resuming the circular loops. However, by offsetting my location from the loop algorithm, I would often run into my own tail or walls when the loops resumed. I made the decision to include a “non suicide” policy into the code, checking the next move and, if that move would result in immediate death, to do any random safe action. With this policy in hand, I was able to perform moderately well.

Figure 1: The “non suicide” policy that dramatically boosted my winrate

```
# Make sure we don't move into a spot where we will immediately get killed
getKilledVal = self.distanceGetter(locs[1], (locs[0][0] + c1, locs[0][1] + c2))
if getKilledVal <= 1.0 and self.pattern and self.pattern[-1] != self.oppositeDir(self.prevMove):
    self.pattern.clear()
    self.pattern.append(self.findOrthogonalMove(aMove))
    self.pattern.append(self.oppositeDir(self.pattern[-1]))

# safety check-- never kill yourself on a wall, tail, or white walker
if self.cellCheck(check):
    p = self.pattern[-1]
    self.pattern.clear()
    self.pattern.append(self.oppositeDir(p))
    c11, c22 = self.coords[self.pattern[-1]]
    check2 = state.board[locs[0][0] + c11][locs[0][1] + c22]

    # don't set us up to die
    if self.cellCheck(check2):
        self.pattern.clear()
        self.pattern.append(self.findMove(state, locs))
```

Improved Functionality: Attacking

To improve my bot, I added some “attack” functionality. If the enemy bot got within a defined distance, I would break the space-gathering and run directly at the enemy with a gradient function I’d designed. Since there was no future-projecting, I calculated where the enemy bot was relative to me, determined what quadrant that represented, and moved in that direction. I called this the “greedy” policy in the code, and it achieved moderate success against both the safe and attack bots. It would break off from its circular loop to run directly at the enemy if they came within the distance, which tended to confuse its direction and force it into situations where it was trapped by its own tail and the wall. As such, it needed to complete its current loop before attacking. A “retreat()” function and a series of booleans made this work, and the bot was able to attack in a slightly safer manner.

A major design decision I made was to store all the upcoming moves in a list. By completely clearing this list and adding new values, I was able to toggle between certain state booleans

(space-gathering → retreating → attacking). This was the easiest way to queue up actions to gain space while also maintaining the ability to adapt to the current situation.

Figure 2: The list of moves and the series of booleans used to transition between states

```
self.pattern = []
self.greedy = True
self.attacking = False
self.retreating = False
```

Advanced Functionality: Optimization

As good as it was, the bot would still be moving towards the enemy without regard for its trail, leaving itself very vulnerable to enemy counterplay and white walkers. This was taken advantage of by the advanced TA-Bots, which would simply move around my feeble attacks to hit my vulnerable tail. Observing how the "AttackBot" played the game, it always progressed forward a couple squares at a time. I had the realization that time-to-attack didn't really matter, as the game could last a long time before timing out, but rather simply moving in the enemy's direction slowly would be enough. In that vein, my next addition was to imitate the Attack Bot, attacking by claiming a single square at a time. This feature took a while to figure out with the conditionals I had going on, but eventually managed to attack slowly, but safely.

Studying the tactics of the TA bots, I learned that they performed badly in cases with the white walker, and I could go round-for-round with them when attacking, as the winner was always whoever went second if the attack threshold was high enough. With that in mind, I iteratively optimized values such as the white walker retreat distance and the attack threshold until I was certain of my ability to beat both the original bots and the TA bots. I further improved the attack algorithm by running at the closest enemy trail rather than their location, which I was able to do efficiently with well placed numpy functions.

I added some more polishes, such as another override at the end to attack the enemy trail or head if I was next to it, and to never move directly within one space of the enemy (as they would then be able to attack us). These small changes allowed me to win far more games than before, pushing me over the grading threshold for almost full marks (see table). Overall, this bot comprised over 400 lines of code, including all of the decision logic as well as a number of helper functions.

Results

Table 1. Maximum Bot 1 Performance by Room Type (in Win % out of 100 games)

(Grading Threshold)	RandBot (95%)	SafeBot (75%)	AttackBot (60%)	TA Bot 1 (60%)	TA Bot 2 (40%)
Small, No WW	100%	75%	100%	50%	50%
Large, No WW	100%	81%	100%	100%	50%
Small, WW	100%	89%	100%	100%	50%
Large, WW	100%	75%	100%	100%	0%

All relevant files are uploaded on Gradescope.

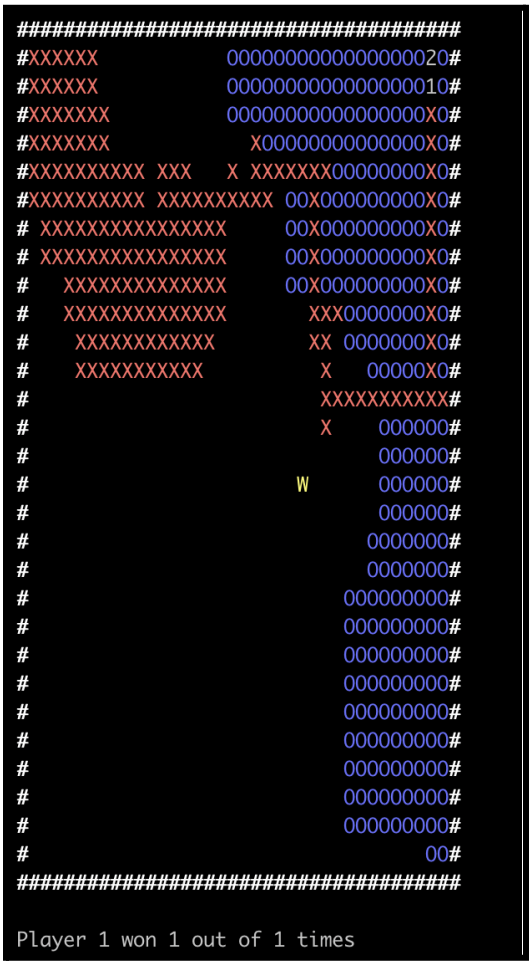
(Note: The highlighted sections refer to times I did not meet the grading threshold for full credit.)

Across the board, there tends to be a binary performance rating. With the exception of SafeBot, I either score 50% or 100%. My investigation into this has determined that the player moving first tends to lose, and when the game is repeated with the -multi_test flag, both actors take the same actions depending on who moves first. SafeBot is the only exception, and the reason for that I believe is due to some small amount of randomness coded into the bot that alters the "attack" threshold formulas to result in different moves.

For most cases, however, bots that encounter each other within a certain distance tend to attack, and either one must move within the other's attack range and is defeated, or they must retreat from the attacking enemy and eventually get boxed into a corner. Once this pattern is recognized, it comes down to optimizing the attack distance threshold, so that no matter who goes first, my bot is always the one delivering the final blow.

Another thing of note is that behavior did alter between runs. If I ran the command to fight TA Bot 1 ten times in a row, I could win five of the rounds. I could run the command again, and end up winning all ten times. As such, I've included the maximum percent success rate in these runs.

Figure 3: Conditional Bot Strategy (X) vs TABot2 (0). Note how it gathers space safely as it attacks, making sure to safely grab space until it corners TABot2 and gets the finishing blow.



Reflections

A conditional-based bot worked far better than I'd originally expected to. Based on my gameplay, it even seemed like TA-Bot 1 followed a similar conditional based approach, gathering space in a radius as it approached my bot. Combining the approaches of the SafeBot and the AttackBot worked well in this game, utilizing absolutely no machine learning. I was able to predict pretty much every move the bot would take, knowing the conditionals involved in the decision making process.

This bot was meticulous to code, and integrating together all the different states was an exercise in tediousness. Helper functions were incredibly helpful to the code, and understanding

the logic might have been done better with a flowchart. However, it was very rewarding to see it succeed.

Bot 2 (Capstone): Sample-Based Optimization ML Bot (Alpha Beta Pruning with Cutoff & CMA_ES)

Process / Design Decisions

My first attempt at this bot was to approach it with the Sarsa Lambda algorithm. The goal was to understand the best move to take, given a specific state. However, almost immediately, I encountered a massive problem: the state space was too large to create a Q-Table. A given state in GoT could account for my permanent tile locations, enemy permanent tile locations, temporary tile locations, white walker positions, and player positions. Mathematically, any given square could be either a red permanent, a blue permanent, a red temporary, a blue temporary, or a white walker (assuming the player location is treated as either a perm or a temp): resulting in 5^n states, where n is the number of tiles. The smallest playable grid is 11x11; the largest one is 36x36. This means, at minimum, there are 5^121 = 3.76e+84 states, more than the amount of atoms in the known universe, or 5^1296, which is approximately 7.33e+905 states and a completely unfathomable number**.**

The Attempt at Sarsa Lambda

Rather than try to create an empty Q-table and fill it with states, I decided to start with an empty hashmap that would add an entry when a new state was reached. Furthermore, I created a heuristic to translate this 5-tuple (red_temp, blue_temp, red_perm, blue_perm, ww) into a single decimal value. In essence, my state representation was a single real number between 1 and -1. There would be much overlap between states, so less distinct states would exist and the hashmap might be feasible in terms of actual space. I initially hoped most of the states would never actually be reached anyway. Implementing the model, however, I found that it rarely made the correct choice— I could barely even beat RandBot. Letting it train for a few thousand epochs made it a bit smarter, but it became so slow in storing/parsing through its massive hashmap of state representations that I would lose by timing out. In fact, my bot became unplayable after a certain amount of training. I tried to reset the weights and train for less epochs to make sure the decision wouldn't time out, but then the model was unable to pass many of the grading thresholds.

In an attempt to make it better, I tried the "Stockfish" approach— combining sarsa-lambda with alpha beta pruning. The idea was to calculate a move *m* with sarsa-lambda, roll it out with alpha beta pruning, and average the return value into the weight of *m*. However, while this did

improve the bot slightly, it ran into similar issues with large state representations and decision making often became very slow unless the alpha-beta-cutoff depth was small.

Heuristic Improvement with Alpha-Beta-Cutoff

This process, however, indicated to me that alpha-beta-cutoff could work well on its own. With an improved heuristic, I could potentially pass the grading thresholds without requiring the weights of the Q-Hashmap. I tried using the Voronoi heuristic, and was able to achieve some decent results with it, but wanted more to create a custom heuristic function that I could modify based on my own personal observations and to account for the sheer complexity that the game contained.

I resolved some bugs in my original custom heuristic function and ensured I was calculating values as efficiently as possible with numpy functions. In the non-WW case, there were four parameters to tune: labeled as c1, c2, c3, and c4 as the coefficients for the weights assigned to the 4-tuple (red_temp, red_perm, blue_perm, enemy_dist). I didn't include blue_temp into the calculation, as I discovered from manually playing the game that being closer to the enemy or their trail was nearly always more important than caring about what temporary squares they had active.

I still had to include the WW into the heuristic, and that was a struggle in and of itself. The WW's location was not tied into the internal game state representation, so when I was 7 recursions deep and querying the state's WW location, it was not giving me the WW location 7 moves away— it would give me the one at the current time. This made it impossible to project into the future and survive in WW games. To avoid this, I made some architecture changes to calculate and include the WW into the recursion calculation, accounting for both current location and current direction. I then used the distance from the WW as the heuristic, including a coefficient c5.

Figure 4: The formula for the state representation returned by the heuristic function

```
return (self.c1 * temps)/maxAbs + (self.c5 * math.exp(-1.01 * closestTrail))/maxAbs + (self.c2 * perms)/maxAbs \
    + (self.c3 * distance)/maxAbs + (self.c4 * oppPerms)/maxAbs + (self.c5 * math.exp(-1.01 * wwDist))/maxAbs
```

Additionally, I wanted to avoid the enemy player approaching my tail, so I treated that the same as a white walker, using the same coefficient. Using manual normalized values, I was able to achieve decent performance, but knew that I would need to improve the values of (c1, c2, c3, c4, and c5) using ML.

Sample-Based Optimization with CMA_ES

I used a common approach in ML called sample-based optimization to find the best values of these coefficients. First, I played the bot against a difficult enemy: TA Bot 1. Then, I randomly selected values for the coefficients, ran 10 rounds with the coefficients, and calculated the winrate. If the coefficients produced a winrate tied with the maximum, they were saved. Using this strategy over hundreds of samples, I was able to generate a set of likely base coefficients that I could use. Then, I used the Covariance Matrix Adaptation Evolution Strategy (CMA_ES) approach to improve them. CMA_ES is based on the biological concept of natural selection, choosing the most "fit" parameters and making small random adjustments to the parameters over time in "generations", selecting the most "fit" of the generation to continue adjusting. I applied this concept algorithmically to different series of my base coefficients, and over time, developed a set of parameters that allowed me to pass nearly every grading threshold with alpha-beta pruning and my custom heuristic. Note that depth was not a parameter, as I chose the maximum depth I could while retaining speed and without timing out on decision making. For small maps that was depth 7, and for large maps that was depth 5.

Figure 5: Cleaned up CMA_ES algorithm used to update coefficient values

```
def CMA_ES(self, curWins, prevWins, gens):
    # threshold
    for i, v in self.coeffs:
        # coefficients already saved
        # we should evolve
        if curWins > prevWins:
            self.coeffs[i] += random.uniform(-2 / gens, 2 / gens)
            gens **= 2  # learning rate factor

    return
```

My final values of (c1, c2, c3, c4, c5) for (red_temp, red_perm, blue_perm, enemy_dist, ww_dist) were generated and optimized to be (-10.263576410953773, 10.872612871038296, -12.750349044993099, -9.128997973139423, -550). Note that c5 was rounded for ease of debugging, and the formula was exponential (c5 * -1.01^x) for the c5 calculation, rather than multiplicative for the rest).

Results

Table 2. Maximum Bot 2 Performance by Room Type (in Win % out of 100 games)

(Grading Threshold)	RandBot (95%)	SafeBot (75%)	AttackBot (60%)	TA Bot 1 (60%)	TA Bot 2 (40%)
Small, No WW	100%	100%	100%	50%	50%
Large, No WW	100%	100%	100%	100%	100%
Small, WW	100%	100%	100%	100%	50%
Large, WW	100%	100%	100%	100%	100%

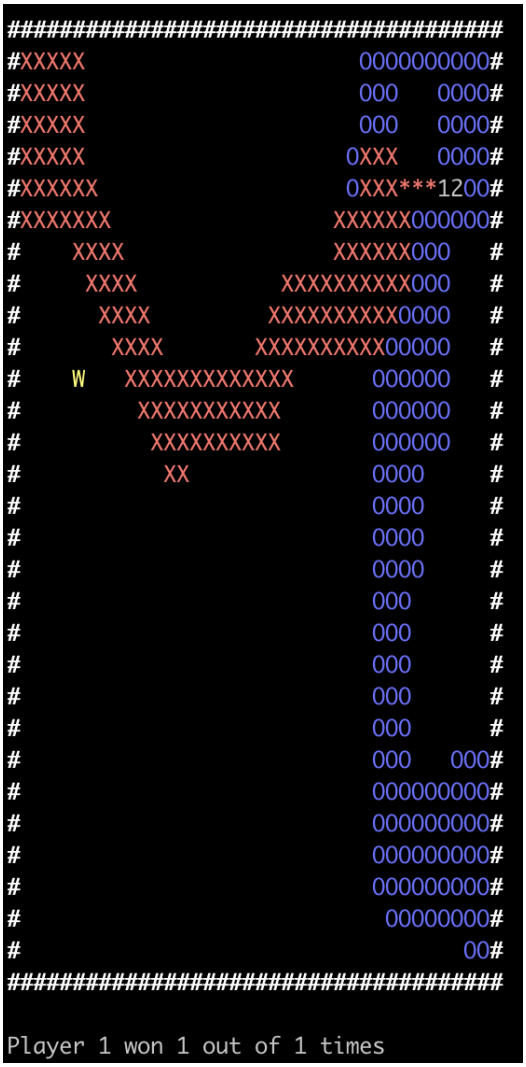
All relevant files are uploaded on Gradescope.

Overall, Bot 2 performed better than Bot 1. However, due to inherent randomness in the ML calculations, getting an optimal win percentage in games with a white walker sometimes took longer than bot 1. For example, in some cases, Bot 2 will run at the white walker and lose 10 games out of 10; in others, it plays perfectly, dodging the white walker with grace and winning 10 out of 10 games. As such, it is not as consistent, but it performs better when it works.

Funnily enough, Bot 2 played like a combination of SafeBot and AttackBot, just as I had created Bot 1 to do. It would gather space as it slowly approached the enemy, and then strike when it had a moment. Otherwise, it would start capturing their permanent squares, doubling the value of gaining the territory and making the enemy struggle to win. This was a sound strategy, and worked in many cases except for against the TA bots who launched attacks (again, based on who went first) once the bot encroached their territory.

The hardest part was trying to update the strategies based only on debug logs and observations. I had to watch the bot run in circles to try and understand why it was doing that, unlike the conditional bot where every single action made sense. Picking apart the weights and manually writing the recursion tree to determine which step was causing an incorrect move made these results much more difficult to achieve than the previous section.

Figure 6: Example of ML bot (X) strategy vs TABot2 (0). Note that it attacks while gaining good amounts of space, taking TABot2's territory until it sees an opening to attack.



Reflections

Calculating the parameters with ML was an interesting feat. I was honestly surprised at how much small adjustments to the parameter dramatically changed the behavior of the bot. Increasing the weight it kept on temporary squares slightly would mean that the bot behaved completely differently, gaining no permanent squares and losing every game.

I found that trying to debug programs using ML is incredibly difficult. I spent dozens of hours on trying to figure out why the bot would behave perfectly one game and suicide in the next, parsing various random values and calculations to understand its behavior. I tried to improve the custom heuristic in many ways, adding features like distance calculations from enemy to my tail and avoiding them like I would the WW. In the end, as is often the case with ML, I am unsure which of my small changes had the most profound effect, but in union, they work well enough. While completing Bot 1 was rewarding, watching this bot succeed gave me much more of a feeling of relief, knowing that I could delete the many print statements and stop hunching over to read the small text in the debug logs.

Conclusion

Both the Conditional Bot and the Sample Based Optimization ML Bot surpassed expectations and scored very well in the field. I was unable to make my ML bot perform using algorithms similar to Stockfish (combining Sarsa-Lambda with Minimax), but found great success using a custom heuristic function and alpha-beta pruning. The CMA_ES algorithm over a sample-based optimization process found the best parameters for bot 2 to work in over hundreds of sampling rounds. Bot 2 outperformed Bot 1, demonstrating the recurring theme that in many cases, ML algorithms tend to outperform non-ML systems. Overall, I learned a massive amount about creating adversarial agents, specifically in terms of iterative improvements, heuristic development, and ML strategies.