Queen's Gambit

SoC Report

Balaji Karedla

2 June 2024

Contents

1 Week 1

1.1 Greedy or not

1.1.1 Introduction

There is a list of n numbers and two players who move alternately. On each move, a player removes either the first or last number from the list, and their score increases by that number and that number gets deleted. Both players try to maximize their scores and play optimally. A certain player wins if their score is strictly greater than the score of the other person.

1.1.2 My Attempt

I tried to solve this question first using recursion and *Minimax Algorithm*, where I check the maximum sum each player can get by choosing each extreme element and recurse along until the array has a single element.

Then, for Dynamic Programming, the script loops through the array, but in reverse direction, all the while saving the results found in a dictionary with the tuple of the beginning index of the sub-array(`beg`) and the length of the sub-array(`length`) as the keys, with the tuple containing the maximum scores of the players as the values.

The time-complexity in both the cases (recursion and loop) with the dictionary was $O(n^2)$.

1.2 Pawnscape

1.2.1 Introduction

In this game, you're provided with an $N \times N$ board where each player has N pawns positioned on both ends in the first and last rows. Victory is achieved by getting one of your pawns to the opposite end. If no feasible moves remain, the game results in a draw. Pawns are restricted to moving one step forward, even for their initial move.

1.2.2 My Attempt

I gave it a try manually and could eliminate two out of four ({`2030, 2131, 2232, 2333`}) moves possible for white at the beginning of the game. The game is symmetric at the beginning and when the first move is `2232` or `2131`, black has a move to win the game irrespective of white's next move.

But, for the case of the first move being `2333`, finding the possibilities manually turned out to be tiresome. So, I made a engine that goes through all the possibilities of games with the *Minimax Algorithm* and finds if any position is a winning position, losing position or atmost a draw by returning `1`, `-1` or `0` respectively.

For making the strategy easier, I saved `json` files for both the players with every state and their best child state mapped, so that the player plays optimally. A function that reads these `json` files gets the best child state for every state possible in a valid game.

1.2.3 My Answers to the Questions

Question 1: For $n = 4$ find if there is a strategy that ensures that white never loses irrespective of what black does.

Answer: The *Minimax Algorithm* gave a value of `0` for the beginning position of the game, hence there is a strategy or a sequence of moves that ensures the white never loses irrespective of what black does.

Question 2: For $n = 4$ find if there is a strategy that ensures that white always wins irrespective of what black does. If the answer of the first part is NO then you can ignore this part.
Answer: As the *Minimax Algorithm* gave a value of `0`, no there doesn't exist a strategy where white always wins irrespective of what black does.

Question 3: Code you strategies for questions 1 and 2.
Answer: The code for this I wrote in the file `engine.py` in the function `run_with_strategy()` where the strategy is based on the *Markov Decision States* created using the *Minimax Algorithm*.

Question 5: For $n = 4$ find if there is a strategy that ensures that black never loses irrespective of what white does.
Answer: As the value of the value of the beginning position is `0`, black does have a strategy to never lose irrespective of what white does.

Question 6: Code your strategy for question 5.
Answer: The code for this strategy also exists in the file `engine.py`.

Question 8: Implement this game on your own in python. You can implement it in any way you like. Also code your strategies as well.
Answer: My implementation of the game exists in the file `engine.py`.

1.3 The Game of Nim

1.3.1 Introduction

The number of piles, i.e., the rows in the image, of the game is finite, and each pile contains a finite number of matchsticks. These numbers can vary for different instances of Nim. Each player, during her turn, chooses exactly one pile, and removes any number of matchsticks from the pile she has selected (she must remove at least one matchstick). The player who removes the last matchstick wins the game.

1.3.2 My Attempt

I found out about *Sprague-Grundy Theorem* and *Bouton's Theorem* and tried to implement the strategy using the *XOR property* or the *Nim-Sum*.The code of the same is in the `engine.py` file.

Figure 1: Diagram of a Nim game with four heaps

1.3.3 My Answers to the Questions

Question 1: Does there always exist a sequence of moves such that the player making the first move wins?
Answer: No, there doesn't always exist a sequence of moves such that the player making the first move wins. A very simple counter-example is the game where there are two heaps with one matchstick each. The player who makes the first move loses.

Question 2: Given a Nim position, is there a way to determine whether it is possible to win for any player with perfect play? If not, why? If yes, how?
Answer: No, given a position, if the *Nim-Sum* of the matchsticks in the heaps is zero, there is no way the matchsticks can be taken to make the *Nim-Sum* zero again. But, there always exists a way to make the *Nim-Sum* zero if it is not zero already, from Bouton's Theorem. Hence the next player can always force the next position we get to have a *Nim-Sum* of zero. As the game should end, the first player would never get a state with just one heap of non-zero matchsticks.

Question 3: If a sequence of move which guarantees a win does exist for a Nim position, how will you determine the sequence of moves? That is to say, how is the computer coded to play the most optimum way?
Answer: The computer can be coded to find the number of matchsticks to remove to make a *Nim-Sum* of zero. By making the *Nim-Sum* zero all the time for the second player, we can force

him to always have no matchstick to pick at the end.

1.4 Chomp

1.4.1 Introduction

Chomp is a two-player strategy game played on a rectangular grid composed of smaller square cells, resembling the blocks of a chocolate bar. Players take turns selecting and "eating" (removing) one block along with all blocks positioned below it and to its right. The top left block is "poisoned," and the player who consumes it loses the game.

Figure 2: Sequence of a Chomp game starting from a 4×7 grid

1.4.2 My Attempt

I tried analysing the game manually and found out that

- If the game has an *L-shaped position* with legs of equal length, the second player always has a move by *Mirror-Image Strategy* and the first player always loses.
- If the game has a *Square Grid*, the first player can force the position to an *L-shaped position* with legs of equal length and win.
- If the game has an *L-shaped position* with legs of unequal length, the first player can force the position to an *L-shaped position* with legs of equal length and win.

I am still trying to make a strategy for the general case.