PBL 3 The Game of Nim

Abstract

This report is on the research and exploration into the game of Nim and how it can be solved for using Artificial Intelligence to determine the best move possible from any given game state (position) to maximise the chances of the Ai winning using a recursive Depth-First-Search minimax algorithm. This report will explore the effectiveness and efficiency of the minimax algorithm in correlation to the heap size and number of objects in that heap. The limitations and constraints of the minimax algorithm will be analysed and compared to the well-known Deep Q-Learning algorithm. This report will explore the design solution and justification for the desired solution to solve the given problem.

Table of Contents

Introduction

Nim is a game of mathematical strategy and has been said to have originated from china with references in Europe dating back to the 16th century. As explained in the presentation there are a variety of variations of the game in existence to date. However, this report will be going to explore the misère variation of the game. The misère variation can consist of a variety of heaps containing objects which will be the playing field for this problem to be solved, the most important aspect of misère however is that the player to take the last object (last to move) loses. When playing the misère variation with varying heaps and heap sizes the player can remove as many objects as they like however only from the one pile/heap they are removing from.

The mathematical strategy to solving Nim is by achieving the nimsum of the piles, basically creating them of equal lengths for your opponent's turn to make the next move that will ultimately lead to their loss.

This report will explore the code that was used to solve for creating an Artificial intelligence to play the game of Nim at an optimal strategy depending on the state the game is in, with us knowing that the nimsum is the optimal way to play, using the minimax approach will most probably use this method aswell without the necessary mathematical checks for nimsum being implemented.

Solving the game of Nim using Artificial Intelligence

The game of Nim is one of a few key aspects in which are vital to take into account when deciding on an Ai algorithm to solve the game by beating the other player depending on the randomised heaps and amount of objects in those heaps.

Nim plays a part in the combinatorial Game Theory in which 2 players know all the moves possible for the current state, there are no randomised variations during the game except for the initialising factors the game generates for the players to play, and is a game of perfect information.

Nim is one of the few well-known games to be considered an impartial game meaning that both players have the same amount of moves and that all winning moves for a position have been solved for. Since Nim is a perfect information game there is at least one best way for each player to solve for.

When taking all of the previous mentioned factors into account and noting that Nim is a Zero-Sum game where no wealth is created or destroyed this means that the wealth for one players move is increased the other players decreases, in other words for what one player wins the other one loses.

All these factors make the game of Nim the perfect theoretical game to be solved for using the minimax algorithm by an ai player component of the 2 player game.

The Minimax Algorithm

What is it?

The Minimax algorithm is a Depth-First-Search algorithm algorithm with backtracking and recursive capabilities. It is usually used in games consisting of 2 players to decide for the next best move for the players given state within the game. The move is evaluated using a position evaluation function within the algorithm to declare how good the players move would be, it will choose the move that maximises their value while minimising the opponents value of their possible moves after the move has been made, this is done so the algorithm will always pick the best move from a winning position to keep ahead and from a losing position to minimise the loss.

How does minimax work

The minimax algorithm works by generating a game tree from its current state and possible moves and the moves possible by the opponent after it has moved and will decide to follow the branch with the most value for its move that minimises the opponents value of their next possible move.

How can minimax be implemented into the game of Nim

The minimax algorithm can be implemented into the game of Nim by using a function to take in the state of the game while it is the computers turn and passing the game state of piles and amount of objects into the minimax algorithm. The minimax algorithm will choose a move then find all the possible player outcomes from that move and recursively traverse through the game tree saving each state, once this has been done for every possible move it can make against every possible move the playing opponent can make it will then follow the branch with the highest maximising value and highest minimising value for the opponent to win the game every time, since this has been mathematically solved we will notice the nimsum xor being found by the computer in games shown further down.

My Implemented solution design for the Game of Nim

I created the Board class that takes in an object being the list consisting of the values representing the amount in the given heap of the list, I then updated the board to itself with the list to be accessed by the player and computer

```
from soloution_search import minimax
import random


# The game board that takes in the list and updates itself
class Board(object):
    def __init__(self, board):
        self.board = board
# player update for their remove amount per pile
    def update(self, piles, num):
        self.board[piles] -= num
#computer update that passes the board state into the solutionsearch minimax function and gets the result back and removes from the board
    def computerUpdate(self):
        self.board = minimax(self.board, -float('inf'), float('inf'), True)[1][1]


# check whether user input is valid
def isValid(remove, board):
    if not remove or len(remove) != 2: return False
    if remove[0] > 0 and remove[1] >= 0 and remove[1] < len(board) and remove[0] <= board[remove[1]]:
        return True
    return False
```

I then created the environment using the randomly generated requirements presented, 2,5 heaps, 2n+1 values for each and appended them to a list and updated the board to take in the list object saving it to the game variable.

```
if __name__ == "__main__":

    print("Starting Nim!")

    # randomly creating piles between 2,5
    numofpiles = random.randint(2, 5)
    # print the piles
    print('Piles: ', numofpiles)
    n = 1 # Declaring n
    list = [] # making a list
    for number in range(0, numofpiles): # iterate through the list adding adding the value of n depending on the randomised heaps generated
        list.append(n)
        n = (2 * n + 1) # updating n to be 2n + 1 for the next heap to contain the amount of the previous heap with n in it

    game = Board(list)
    print("Enter the number to remove '()' and then the index")
    print("The person player who moves the nims the last object loses")
```

I then created the playing conditions as documented below, this allowed for the player and computer moves to be relayed back to the board through “game.update” which takes the index of the heap and the amount to remove from that heap. Also consists of conditions for each player and computer to break out of once the game has been won or must be continued.

```
player_win = True
while True:

    print("Piles: %s" % (game.board))

    # player's turn
    user = str(input("Player turn: "))                # Take in the player input to be used in the player remove function to be related to the game board through the update function
    player_remove = [int(i) for i in user.split(' ')]

    while not isValid(player_remove, game.board):              # Conditions to check if it is a valid move
        print("Invalid move! Please input again.")
        user = str(input("Player turn: "))
        player_remove = [int(i) for i in user.split(' ')]

    game.update(player_remove[1], player_remove[0])

    if sum(game.board) == 0:        # If the list is not empty and the player did not win move to computers turn
        player_win = False
        break
    elif sum(game.board) == 1:  # if player won then break to the if player_win condition below
        break

    print("Computer turn")
    game.computerUpdate()

    if sum(game.board) == 0:               # if on computers turn the sum is zero then break  beacuase player won
        break
    elif sum(game.board) == 1:          # if sum is 1 after computers turn then player win does not execute and instread prints board saying computer won
        player_win = False
        break

if player_win:
    print(game.board)                  # Print player wins
    print("You won!")
else:
    print(game.board)                      # Print computer wins
    print("You lost, The computer won!")
```

When it is the computers turn it calls the imported minimax function from solution using the computer update function.

The solution has the minimax algorithm:

```python
# applies the Minimax and Alpha-beta pruning to find the best strategy

# decide based on the current state
def minimax(position, alpha, beta, maxPlayer):
    # determining final state
    if (sum(position) == 1 and maxPlayer) or (sum(position) == 0 and not maxPlayer): return (-1, [position])
    if (sum(position) == 1 and not maxPlayer) or (sum(position) == 0 and maxPlayer): return (1, [position])

    if maxPlayer:
        maximumEvaluation = -float('inf')
        result = None
        for i in find_next(position):
            # recursively search for the next possible move
            val, temp = minimax(i, alpha, beta, not maxPlayer)
            if val > maximumEvaluation:
                maximumEvaluation = val
                result = temp
            # update the upper bound
            alpha = max(alpha, val)
            # pruning
            if alpha >= beta:
                break
        return maximumEvaluation, [position] + result
    else:
        minimumEvaluation = float('inf')
        result = None
        for i in find_next(position):
            val, temp = minimax(i, alpha, beta, not maxPlayer)
            if val < minimumEvaluation:
                minimumEvaluation = val
                result = temp
            beta = min(beta, val)
            if alpha >= beta:
                break
        return minimumEvaluation, [position] + result
```

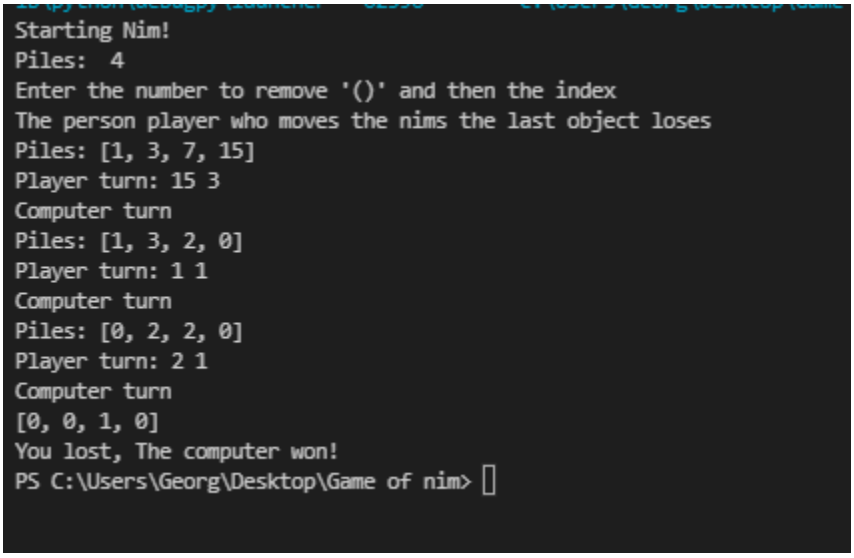
Which recursively calls the defined function find_next(position) to find the next position and branch all possible moves and returns the result:

```python
# find all possible next moves
def find_next(position):
    visited = set()
    res = []

    for i in range(len(position)):
        for m in range(1, position[i] + 1):
            temp = list(position[:])
            temp[i] -= m

            # check if the stage already exists
            rearranged = tuple(sorted(temp))
            if rearranged not in visited:
                res.append(temp)
                visited.add(rearranged)

    return res
```

My findings

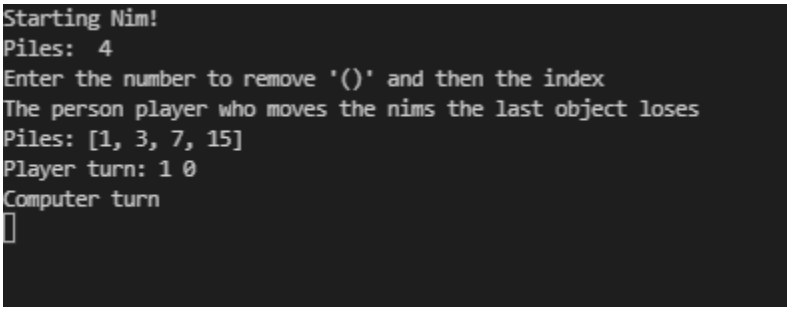
From the implemented solution to the problem it was clear that the computer did know the next best move and what move it should take to win, in this occurrence of the game it can be seen that the

computer was able to find the nimsum mathematical choice for the optimal play without being mathematically structured to follow and generate the conditions to do so which further verifies that the nimsum mathematical approach is in fact the best way to play this mathematical game to win.



This can be seen when the computer made the piles at index 1 and 2 both equal to 2 by removing 1 therefore putting the player in the losing position to be left with the last piece to take (last move)

My findings also led me onto discover that when there was a heap size of 15 and 3 heaps left the computer would take a very long time to process the next move as opposed to how it instantly did for a 3 heap game with the max number of 7.



This clearly shows how inefficient the algorithm is for bigger branches, more possible moves and deciding the best move from that. This could've been further improved if a nimsum heuristic was included as even with alpha beta pruning the computer was still unable to find a move for over 30 minutes.

Limitations

From the above findings it is clear the the minimax algorithm is although perfect for the game of nim theoretically, it is however not efficient and no feasible for heaps above 3 with values above 7 as the possible moves the ai could take and the following moves the player could take and branching them to therefore calculating which branch to traverse is too much computation for this algorithm to run

efficiently. However heap sizes of 7 and heap piles of 3 the algorithm works perfectly and near instant to the naked eye. Even for 4 piles the algorithm will work if the first player removes the pile with 15 first.

Another approach to this problem could be the use of Deep Q-Learning, as the computer will play against itself to the point that it will teach itself how to play the game from any position or game state for the optimal strategy so that once it does come to the state in the game that minimax was inefficient the Deep Q-Learning algorithm would already have a move saved and find that move in a near constant time complexity. Making for a much more efficient algorithm then minimax for larger game that would generate a game tree consisting of branches which the minimax would be inefficient at traversing.

All Knowledge gained was through class, lectures, lecture slides of Deakin and the below mentioned references.

Bibliography

Darby, G. (2018, May 15). *Nim, Gametrees, and Minimax*. Retrieved from delphiforfun.org: http://delphiforfun.org/programs/nim_minimax.htm

Ivan Koswara, M. M. (2021, May 16). *Combinatorial Games - Definition*. Retrieved from brilliant.org: https://brilliant.org/wiki/combinatorial-games-definition/

Ivan Koswara, M. M. (2021, May 18). *Nim*. Retrieved from Brilliant.org: https://brilliant.org/wiki/nim/

javatpoint. (2018). *Mini-Max Algorithm in Artificial Intelligence*. Retrieved from javaTpoint: https://www.javatpoint.com/mini-max-algorithm-in-ai

Lague, S. (2018, April 20). *Algorithms Explained – minimax and alpha-beta pruning*. Retrieved from Youtube: https://www.youtube.com/watch?v=l-hh51ncgDI

Stanford. (n.d.). *Zero-Sum Games*. Retrieved from Stanford University Computer Science: https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/game-theory/zero.html

Yang, Y. (2019, September 30). *Nim*. Retrieved from GitHub: https://github.com/kevinyang372/Nim