T2-Juegos-SI

July 14, 2020

1 Algoritmo Minimax y Alpha-Beta prunning

1.0.1 Minimax

El algoritmo de minimax en simples palabras consiste en la elección del mejor movimiento para el computador, suponiendo que el contrincante escogerá uno que lo pueda perjudicar, para escoger la mejor opción este algoritmo realiza un árbol de búsqueda con todos los posibles movimientos, luego recorre todo el árbol de soluciones del juego a partir de un estado dado, es decir, según las casillas que ya han sido rellenadas.

1.0.2 Alpha-Beta prunning

Es una versión modificada del algoritmo minimax. Es una técnica de optimización para el algoritmo minimax. Los dos parámetros se pueden definir como:

Alfa: la mejor opción (de mayor valor) que hemos encontrado hasta ahora en cualquier punto del camino de Maximizer. El valor inicial de alfa es -∞ .

Beta: la mejor opción (valor más bajo) que hemos encontrado hasta ahora en cualquier punto del camino de Minimizer. El valor inicial de beta es + ∞ .

1.1 Historia

1.1.1 Minimax Alan Turing

La meta es construir un algoritmo (Algoritmo de minimax) que a partir de una posición en que juega el computador pueda escoger entre las diversas alternativas a su disposición.

John Von Newmann y Osear Morgenstern (1945) propusieron la idea básica de Minimax con la suposición fundamental de que la misma función de evaluación está disponible para ambos jugadores.

La idea de función de evaluación es perfeccionada por Claude Shannon (1950). Alan Turíng en 1951 encuentra que la profundidad del árbol no necesariamente debe ser homogénea.

Shanon lo sugirió originalmente (50), basado en teoría de juegos de Von Neumann y O. Morgenstern, aunque Turing presentó el primer programa (51). La idea: Maximizar mis tiradas considerando que el oponente va a minimizar.

Para decidir qué jugada hacer, el árbol se divide por niveles:

- Max: el primer jugador (nivel) y todas las posiciones (niveles) donde juega.
- Min: el oponente y todas las posiciones en donde juega

1.1.2 Alpha-Beta prunning

Donald Knuth y Ronald Moore (1975) fueron los primeros en analizar a fondo la eficiencia de la poda alfa-beta, en su libro "An analysis of alpha-beta pruning". Es posible calcular en un determinado juego una decisión correcta sin tener que explorar todos los nodos de un árbol de búsqueda. A la técnica que consideramos en particular se le conoce como poda alfa-beta. Aplicada a un árbol minimax estándar, produce la misma jugada que se obtendría con minimax, pero elimina todas las ramas que posiblemente no influirán en la decisión final. La eficiencia de alfa-beta dependerá del orden como se exploren los sucesores dentro de un árbol, es mejor primero explorar aquellos sucesores que aparentemente tienen más posibilidades de ser los mejores.

1.2 Funcionamiento del Algoritmo Ejemplo Práctico

Para ver como funcionan vamos a revisar el juego de tres en raya mediante el analisis de cada algoritmo.

1.3 Juego OXO - Tres en Raya

1.3.1 Minimax

Para resolver este juego utilizaremos la siguiente heurística:

E(n)=X(n) - O(n)

donde:

E(n) es la función.

X(n) son el número de movimientos posibles para el Max, en este caso el jugador de las X.

O(x) son el número de movimientos posibles para el Min, en este caso el jugador de las O.

1.3.2 Aplicación

1.3.3 Alpha-Beta Prunning

Este algoritmo es una version mejorada de Minimax.

1.3.4 Aplicación

1.4 Propuesta de como emplear el Algoritmo en su problema del laberinto

Seria usar le tecnica de heurística ya vista que consiste en calcular la distancia recta entre los puntos de inicio y fin, osea la distancia euclideana. Ejemplo:

Por lo tanto encuentra el óptimo.

1.5 Papers del Uso del Algoritmo y su variante.

He encontrado los siguientes papers: 1. Experiments with Game Tree Search in Real-Time Strategy Games de Santiago Ontañón, USA. 2. Efficiency of parallel minimax algorithm for game tree search de Plamenka Borovska y Milena Lazarova Bulgaria. 3. Rminimax: An optimally randomized MINIMAX algorithm de Silvia García Díez, Jérome Laforge y Marco Saerens Belgica. 4. Alpha-Beta Pruning for Games with Simultaneous Moves de Abdallah Saffidine, Hilmar Finnsson y Michael Buro

2 Implementación Juego con EasyAI

2.0.1 Juego Cram con EasyAI

Movimientos Cram se juega con un tablero cuadriculado de n × m casillas. Dos jugadores juegan en turnos alternos poniendo un dominó de manera horizontal o vertical sobre dos casillas libres. El ganador es el último jugador que pone un dominó en el tablero. Ejemplo: Jugador segun el tablero puede ingresar: A1 B1 si esta disponible combinaciones de esa forma segun el tablero.

Puntajes No es un juego de puntaje.

¿Quién gana la partida? Gana la partida quien coloque la ultima pieza, osea en este caso el ultimo par de coordenadas por decirlo así.

[ ]:
```
from easyAI import TwoPlayersGame, Human_Player, AI_Player, Negamax
import numpy as np

# directions in which a knight can move
DIRECTIONS = list(map(np.array, [[1, 2], [-1, 2], [1, -2], [-1, -2],[2, 1], [2,␣
 ↪-1], [-2, 1], [-2, -1]]))

#functions to convert "D8" into (3,7) and back...
pos2string = lambda a: "ABCDEFGH"[a[0]] + str(a[1] + 1)
string2pos = lambda s: ["ABCDEFGH".index(s[0]), int(s[1]) - 1]

mov2string = lambda m: pos2string((m[0], m[1])) + " " + pos2string((m[2], m[3]))
```

```
def string2mov(s):
    poss = [string2pos(p) for p in s.split(" ")]
    return poss[0] + poss[1]

class Cram(TwoPlayersGame):
    def __init__(self, players, board_size = (6, 6)):
        self.players = players
        self.board_size = board_size
        self.board = np.zeros(board_size, dtype = int)
        self.nplayer = 1 # player 1 starts.

    def possible_moves(self):
        moves = []
        for i in range(self.board_size[0]):
            for j in range(self.board_size[1]):
                if self.board[i, j] == 0:
                    if (i + 1) < self.board_size[0] and self.board[i + 1, j] ==␣
↪0:
                        moves.append([i, j, i + 1, j])
                    if (j + 1) < self.board_size[1] and self.board[i, j + 1] ==␣
↪0:
                        moves.append([i, j, i, j + 1])
        return list(map(mov2string, moves))

    def make_move(self, move):
        move = string2mov(move)
        self.board[move[0], move[1]] = 1
        self.board[move[2], move[3]] = 1

    def unmake_move(self, move):
        move = string2mov(move)
        self.board[move[0], move[1]] = 0
        self.board[move[2], move[3]] = 0

    def show(self):
        print('\n' + '\n'.join(['  1 2 3 4 5'] + ['ABCDEFGH'[k] +' ' + ' '.
↪join(['.*'[self.board[k, i]]
                                                    for i in range(self.
↪board_size[0])])
                                                    for k in range(self.
↪board_size[1])] + ['']))

    def lose(self):
        return self.possible_moves() == []

    def scoring(self):
        return -100 if (self.possible_moves() == []) else 0
```

```
    def is_over(self):
        return self.lose()

if __name__ == "__main__":
    ai_algo = Negamax(6)
    game = Cram([Human_Player(), AI_Player(ai_algo)], (5, 5))
    game.play()
    print("Jugador {} Pierde".format(game.nplayer))
```

```
  1 2 3 4 5
A . . . . .
B . . . . .
C . . . . .
D . . . . .
E . . . . .
```

[ ]: