Final Project

311554026 施泓丞 智能所 博一

https://github.com/roy-shih/2023-AI-final-project

Design philosophy

Intriduction

本專題的設計理念是利用Minimax演算法來找出最佳的下一步策略。

`make_your_move(board)` 函式接收一個 `board` 作為參數，將當前的棋盤狀態傳入。它會使用Minimax演算法來計算最佳的下一步行動，並回傳選擇的行數和減去的數量。

Methodology

我在函式內部定義了另一個名為 `minimax` 的遞迴函式，該函式根據當前的遊戲狀態進行遞迴搜索，以找出最佳行動的分數。

在 `make_your_move` 函式中，我們將他初始化最佳分數設為負無窮，並遍歷所有 `row_or_col` 和減去的數量的組合。

對於每個組合，檢查是否是有效的行動，並執行相同的操作：

1. 複製當前的棋盤狀態，進行行動
2. 以最小化玩家的角色呼叫遞迴的 `minimax` 函式，並將深度設為3，alpha設為負無窮，beta設為正無窮。
3. 比較得到的分數和目前的最佳分數，並更新最佳分數和選擇的行數和減去的數量。

程式如下：

```
best_score = float('-inf')
chosen_row_or_col = 0
chosen_subtract = 0
for row_or_col in range(SIZE*2):
  for subtract in range(1, 4):
    if check_valid(board, row_or_col, subtract):
      # Step 1 : 複製當前的棋盤狀態，進行行動
      board_copy = [row[:] for row in board]
      board_subtract(board_copy, row_or_col, subtract)
      # Step 2 : 以最小化玩家的角色呼叫遞迴的minimax函式
      score = minimax(board_copy, 3, float('-inf'), float('inf'), False, player)
      # Step3 : 比較得到的分數和目前的最佳分數，並更新最佳分數和選擇的行數和減去的數量。
      if score > best_score:
        best_score = score
        chosen_row_or_col = row_or_col
        chosen_subtract = subtract
return chosen_row_or_col, chosen_subtract
```

Minmax

在`minimax`函式中，首先檢查遞迴的終止條件。如果遞迴的深度為0或遊戲已經結束（使用`check_game_end`函式進行檢查），則返回當前遊戲狀態的分數。這裡根據`player`的值返回正負的`total_cost`。

```
if depth == 0 or check_game_end(board, False)[0]:
  if player == 0:
    return -total_cost[0]
  else:
    return -total_cost[1]
```

如果遞迴還需要繼續進行，則根據當前的玩家是最大化玩家（`maximizing_player`為True）還是最小化玩家（`maximizing_player`為False）來選擇適當的動作。

如果是最大化玩家，則初始化最佳分數為負無窮。然後，遍歷`row_or_col`和減去的數量的組合，檢查是否是有效的行動（使用`check_valid`函式進行檢查）。

如果是有效的行動，則複製當前的棋盤狀態，並在複製的棋盤上執行該行動（使用`board_subtract`函式）。

接著，以最小化玩家的角色呼叫遞迴的`minimax`函式，並將深度減1。最後，更新最佳分數和alpha值，並檢查是否可以進行beta剪枝（如果beta小於等於alpha，則跳出迴圈），最後返回最佳分數。

```
if maximizing_player:
  best_score = float('-inf')
  for row_or_col in range(SIZE*2):
    for subtract in range(1, 4):
```

```
      if check_valid(board, row_or_col, subtract):
        board_copy = [row[:] for row in board]
        board_subtract(board_copy, row_or_col, subtract)
        score = minimax(board_copy, depth - 1, alpha, beta, False, player)
        best_score = max(best_score, score)
        alpha = max(alpha, best_score)
        if beta <= alpha:
          break  # Beta cutoff
  return best_score
```

如果是最小化玩家，則初始化最佳分數為正無窮，並按照與最大化玩家相同的邏輯進行遞迴搜索。不同之處在於更新最佳分數和beta值，並檢查是否可以進行alpha剪枝。

```
else: # 接續上段 if maximizing_player
  best_score = float('inf')
  for row_or_col in range(SIZE*2):
    for subtract in range(1, 4):
      if check_valid(board, row_or_col, subtract):
        board_copy = [row[:] for row in board]
        board_subtract(board_copy, row_or_col, subtract)
        score = minimax(board_copy, depth - 1, alpha, beta, True, player)
        best_score = min(best_score, score)
        beta = min(beta, best_score)
        if beta <= alpha:
          break  # Alpha cutoff
  return best_score
```

Alpha-Beta Pruning

當進行Minimax演算法時，Alpha-Beta Pruning 是一個用於減少搜索空間的技巧，以提高運算效率。

- Alpha

  在每次迭代時，我們都會檢查是否能做alpha剪枝：如果beta的值小於等於alpha，這表示最小化玩家（對手）已經找到了一個比alpha更好（更小）的分數。因此，最大化玩家不再需要繼續搜索這個分支，因為它已經找到了一個比對手更好的選擇，並且對手不會選擇這個分支。

  因此，在遇到beta小於等於alpha的情況時，我們可以提前結束迭代並跳出迴圈，避免對該分支的進一步搜索。這種方式有效地減少了搜索空間，因為我們可以排除一些明顯不會選擇的選項。

- Beta

  相似的，在每次迭代時，我們都會檢查是否能做beta剪枝。如果beta的值小於等於alpha，這表示最大化玩家已經找到了一個比beta更好（更大）的分數。因此，

最小化玩家不再需要繼續搜索這個分支，因為它已經找到了一個比對手更差的選擇，並且對手不會選擇這個分支。

因此，在遇到beta小於等於alpha的情況時，我們可以提前結束迭代並跳出迴圈。

透過Alpha-Beta Pruning，Minimax演算法能夠更快地達到最佳解，尤其在搜索深度較深且遊戲樹較大的情況下，它的優勢尤為明顯。

Discussion

在撰寫本專案時，我一開始是只用了`minimax`，但在測試的時候有發現，有時候會出現一些動作時間特別長的現象，為了解決這個問題，因此做了 Alpha-Beta Pruning，從而減少搜尋空間，提高計算效率。

事實上在一開始學這個的時候，並沒有想像到說他會有這種時間上的問題存在，過去我們所做的所有專題都是解出來或解不出來，而很少遇到解出來，卻會遇到時間上太長的問題。我想執行效率這件事是 AI 在設計上很需要考量的東西。