COMP 472
MP2 Presentation

By Team of 2

Ricky Lam 40089502

Dongdong Zhang 40043995

Accepting configurations

- Values for n, b, s, d1, d2, t, a, player1, player2, p1e, p2e can be input through console through our function receive_inputs() or can be hard coded before running the program.

```
# n, b, s, d1, d2, t, a, player1, player2, recco_bool, p1e, p2e = receive_inputs()
# blocs = g.place_blocs(b=b, n=n)
n, b, s, d1, d2, t, a, player1, player2, recco_bool, p1e, p2e = 8, 6, 5, 6, 6, 1, True, Game.AI, Game.AI, True, Game.E2, Game.E1
```

```
g.play(algo=algo, player_x=player1, player_o=player2, n=n, s=s, d1=d1, d2=d2, t=t, p1e=p1e, p2e=p2e, f=f)
```

Placing blocs

- Blocs can be manually places through the console by using our game.place_blocs() function, where the user is prompted for either random placement or specific coordinates
- Blocs can be automatically placed randomly or by passing a list of coordinate tuples through the function game.auto_blocs()

```
blocs = []
g.auto_blocs(blocs=blocs, b=b, n=n)
```

E1 Heuristic

- Very simple heuristic
- Counts number of X and substracts from value
- Counts number of O and adds to value
- Does the above for every horizontal, vertical, and diagonal line

```
# Horizontal
for i in range(0, n):
    horizontal_string = ""
    for j in range(0, n):
        horizontal_string = horizontal_string + self.current_state[i][j]
    value = value - horizontal_string.count('X') + horizontal_string.count('O')
```

E2 Heuristic

- Considers advanced options
- Returns highest value if a win is detected
- Otherwise prioritizes stopping the opponent from winning…
- Otherwise prioritizes creating a situation where there’s more than one way to win
- Otherwise tries to simply place markers consecutively

```
for den in denial_x:
    if den in diagonal_string:
        return -9999
for den in denial_o:
    if den in diagonal_string:
        return 9999
for near_win in closest_to_win_x:
    if near_win in diagonal_string:
        if forkCount > 1:
            return -5000
        else:
            forkCount += 1
        value = value - 500
for near_win in closest_to_win_o:
    if near_win in diagonal_string:
        if forkCount > 1:
            return 5000
        else:
            forkCount += 1
        value = value + 500
```

Time and Depth

- The amount of time relative to maximum depth affects the performance of the heuristic functions!
- If it doesn’t have enough time, it will hastily return the first position it checks
- Unoptimal gameplay!

Most effective configurations

- We have found that our e2 heuristic plays most effectively on a $n = 5$ or smaller board with maximum depth allowed of $d = s + 1$ or less and time limit anywhere between 5 to 10
- The above similarly applies for the e1 heuristic
- This is because we don’t want the algorithm to run out of time looking too deep into a tree for the optimal move when it can often be found in shallower trees quickly

Meeting time and depth constraints

- We check the current depth of our node and force a heuristic evaluation if it is equal to or exceeds the depth limit
- At the same time, we check time elapsed and force a heuristic evaluation if it is cutting too close to the time limit

```
time_elapsed = round(time.time() - start_time, 7)
if iter >= d or time_elapsed >= t - (t * 0.0075):
```

Per move stats display

- Following assignment specifications, after each move there will be a list of information to be displayed on the console and on the gametrace files

```
Player O under AI control plays: x = D, y = 1

i   Evaluation time: 1.0s
ii  Total heuristic evaluations: 11062
iii Evaluations by depth: {6: 10689, 5: 357, 4: 12, 3: 1, 2: 1, 1: 1, 0: 1}
iv  Average evaluation depth: 6.0
v   Average recursion depth: 5.4

  ABCDEFGH  (move #25)
 +--------
0|*OXX....
1|X*OO...*
2|OXX..*..
3|XOO...*.
4|OXX.....
5|XOO*....
6|OXX.....
7|XOO.....
```

Game end stats

```
  ABCDEFGH  (move #51)
 +--------
0|*OXXOOX.
1|X*OOXXO*
2|OXXXO*X.
3|XOOOXO*.
4|OXXXOXO.
5|XOO*XOXO
6|OXXOOX..
7|XOOXXO..

The winner is O!

6(b)i   Average evaluation time: 0.97s
6(b)ii  Total heuristic evaluations: 1769528
6(b)iii Evaluations by depth: {6: 1673767, 5: 87740, 4: 7049, 3: 741, 2: 122, 1: 59, 0: 50}
6(b)iv  Average evaluation depth: 5.9
6(b)v   Average recursion depth: 5
6(b)vi  Total moves: 51
```

- After a game ends by a tie or a win, a list of information will be displayed for the entirety of the game

Scoreboard file

- Generated after a series of 2 x r games
- Players will play one game as normal, then switch starting turns after the first game, then repeat r times

```
for i in range(0, 10):
```

- For example, 10 is our r value in our main() function.

Scoreboard File

- The scoreboard file displays a list of averaged and total information across all 2 x r games
- If wins don't add up to 2 x r, it means some games ended in a tie!

```
n=8 b=5 s=5 t=5

Player 1: AI d=2 a=False
Player 2: AI d=6 a=False

20 games

Total wins for heuristic e1: 4 (23.529411764705884%) (simple)
Total wins for heuristic e2: 13 (76.47058823529412%) (complex)

i   Average evaluation time: 2.2947678312019626
ii  Total heuristic evaluations: 85724538
iii Evaluations by depth: {"2": 192602, "1": 4061, "0": 609, "6": 81895322, "5": 3396099, "4": 210185, "3": 17710}
iv  Average evaluation depth: 5.9
v   Average recursion depth: 3
vi  Average moves per game: 31.45
```