assignment 3

Fredrik Robertsen, Erlend Ulvund Skaarberg

2025-10-21

(a) code output

halving game output

```
The number is 5 and it is P1's turn
P1's action: --
The number is 4 and it is P2's turn
P2's action: --
The number is 3 and it is P1's turn
P1's action: /2
The number is 1 and it is P2's turn
P2's action: --
The number is 0 and P1 won
```

this is as expected (provided by the example code)

bucket game output

```
The state is (0, ['A', 'B', 'C']) and it is P1's turn
P1's action: B
The state is (1, [3, 1]) and it is P2's turn
P2's action: 1
The state is (0, [1]) and P1's utility is 1
```

the results follow the expected run-through of the minimax algorithm

tic tac toe output

```
   |   |
---+---+---
   |   |
---+---+---
   |   |

It is P1's turn to move
P1's action: (0, 0)

 x |   |
---+---+---
   |   |
---+---+---
```

```
   |   |

It is P2's turn to move
P2's action: (0, 1)

 x | o |
---+---+---
   |   |
---+---+---
   |   |

It is P1's turn to move
P1's action: (0, 2)

 x | o | x
---+---+---
   |   |
---+---+---
   |   |

It is P2's turn to move
P2's action: (1, 1)

 x | o | x
---+---+---
   | o |
---+---+---
   |   |

It is P1's turn to move
P1's action: (1, 0)

 x | o | x
---+---+---
 x | o |
---+---+---
   |   |

It is P2's turn to move
P2's action: (2, 0)

 x | o | x
---+---+---
 x | o |
---+---+---
 o |   |

It is P1's turn to move
P1's action: (2, 1)
```

```
 x | o | x
---+---+---
 x | o |
---+---+---
 o | x |

It is P2's turn to move
P2's action: (1, 2)

 x | o | x
---+---+---
 x | o | o
---+---+---
 o | x |

It is P1's turn to move
P1's action: (2, 2)

 x | o | x
---+---+---
 x | o | o
---+---+---
 o | x | x

The game is a draw
```

this is an optimal game, ending in a draw

(b) runtime improvement with alpha-beta pruning

results of our benchmarks

- 8.7112 seconds without alpha-beta pruning
- 0.0001 seconds with alpha-beta pruning

the results are measured at the first step of the algorithms

we can see a clear speed-up from using the alpha-beta pruning

benchmarking code

```
game = Game()
state = game.initial_state()
game.print(state)
record = False
while not game.is_terminal(state):
    player = game.to_move(state)
    start_time = time.time()
    action = minimax_search(game, state) # The player whose turn it is is the MAX
player
    if not record:
```

```
        print(f"Elapsed time (minimax): {time.time() - start_time:.4f} seconds")
        record = True
    print(f'P{player+1}\'s action: {action}')
    assert action is not None
    state = game.result(state, action)
    game.print(state)
    print()

game = Game()
state = game.initial_state()
game.print(state)
record = False
while not game.is_terminal(state):
    start_time = time.time()
    player = game.to_move(state)
    action = alpha_beta_search(game, state) # The player whose turn it is is the
MAX player
    if not record:
        print(f"Elapsed time (alpha-beta-pruning): {time.time() - start_time:.9f}
seconds")
        record = False
    print(f'P{player+1}\'s action: {action}')
    assert action is not None
    state = game.result(state, action)
    game.print(state)
    print()
```

non-mandatory

in the provided game state

```
 x | o | o
---+---+---
 x |   |
---+---+---
   |   |
```

the minimax algorithm will choose the path that leads to victory, which could be the middle point, or the lower left one. although, due to our implementation, it will always pick the middle point because it sees it first (it is on a higher row). this path also provides two paths to win. it is in such a sense even more optimal than greedily choosing the lower left option, but both options lead to a win and are as such considered equal.

to make the algorithm choose the immediate path on the lower left, we can for example have it greedily scan the board for such an option every recursive call so as to discover the option of a quicke win. though this may not be more performant due to the overhead of checking for this condition all the time, but on this rare occasion we will at least avoid performing unecessary minimax calls.

another way to achieve this, which could also improve the performance, would be to keep track of path lengths. this would also let it discover such an option.

yet another way would be to implement the algorithm differently, as a BFS algorithm that checks for the winning state before expanding it.