EE566 Report

MATH OF TURN-BASED STRATEGY GAMES

Checkers_

CODE link: https://github.com/Anupam0401/Checkers-AI

Harsh Vardhan	11940460
Aditya Keshari	11940070
Anupam Kumar	11940160
Deepak Chouhan	11940330
Utkarsh Alpuria	11941290

GAME DESCRIPTION_

Checkers is a two-player strategy board game played on an 8x8 board with 64 squares of alternating colors (usually black and white). Each player starts with 12 pieces placed on the dark squares of the first three rows on their side of the board.

The game's objective is to capture all the opponent's pieces so they cannot move more. Pieces move diagonally forward, one square at a time, and can capture an opponent's piece by jumping over it sideways to an empty square on the other side.

When a checker reaches the farthest row on the opponent's side, it is crowned a king and gains the ability to move and capture diagonally in any direction. The game ends when one player captures the opponent's pieces or when a player cannot make a legal move, resulting in a draw.

OUR SIMPLIFICATION_

We have made this game simple for this project in the following way-

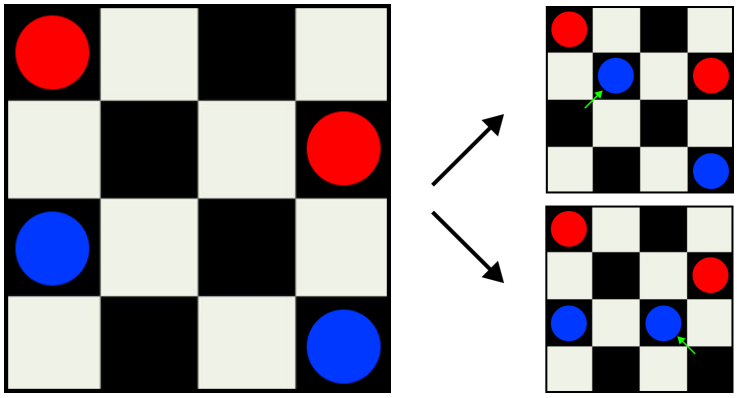
- Our game board is 4x4(16 squares of alternate colors).
- Each player starts with two pieces.
- All other rules and objectives of the game are the same.

MODEL - MARKOV DECISION PROCESS_

A Markov Decision Process(MDP) consists of a State, Action, Reward, Transition probabilities, and Discount factor.

States - Every dark square in the Checkers board will either be X, B, R, KB or KR. Here, X means empty square, B means blue piece, R means red piece, KB means blue king, and KR means red king. Thus, a state is simply a string of 8 characters, each representing the entry in that cell. The state is defined using the accessible cells (i.e., dark square) only. We will discuss this more later.

Action - Given a state, action would mean the set of playable cells. E.g.,



Rewards - There are a total of three rewards in this game. A very high reward is given to the winning state, whereas a meager reward is given to the losing state. No reward is given to the intermediate states.

```
def reward(self, board: checkers.Board):
    self.game.turn = BLUE
    if self.check_for_endgame(board):
        return -10000
    else:
        all_moves = []
        for piece_moves in self._generate_all_possible_moves(board):
            current_pos = piece_moves[0:2]
            for final_pos in piece_moves[2]:
                all_moves.append([current_pos,final_pos])
        for move in all_moves:
            new_board = deepcopy(board)
            self._action(move[0], move[1], new_board)
            self.game.turn = RED
            if self.check_for_endgame(new_board):
                self.game.turn = BLUE
                return 10000
            self.game.turn = BLUE
    return 0
```

Transition probabilities - Here, if player 1 makes a move, the resultant state is determined by changing the value of the concerned cells. This is an intermediate state following which player 2 makes his move which is equiprobable, i.e., the opponent can make any move from the possible number of moves with equal probability. This is our next playable state.

Discount factor - A discount factor of 0.9 is taken.

So we can model MDP as a tuple: (**S**, **A**, **P**, **R**, **γ**) where:

- **S** is the set of possible states
- **A** is the set of possible actions
- **P** is the set of transition probabilities
- **R** is the reward function
- **γ** is the discount factor

Hence, in Checkers, we have

- **S** = {all feasible 4x4 board configuration}
- **A** = {all possible unoccupied squares in the current state}
- **P**(s, a, s') = p
- **R**(s) = +10000 (for winning state)
  -10000 (for losing state)
  0 (otherwise).
- **γ**= 0.9

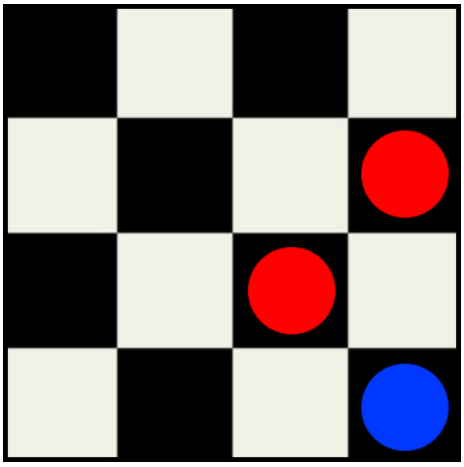
States_

Each board layout is mapped to one state in this game of checkers. Hence, different board layout means different states, terminal states being the one where no possible moves are left for either one of the player.

Here, we will determine Player 1's optimal strategy in opposition to Player 2's fixed policy.

States Representation_

A random state is represented as shown below.



The maximum number of possible states in a 4x4 board is 5^8, but many are not practically feasible. We use the DFS algorithm to generate unique, accessible states and store them in a dictionary. The code snippet to find these states is shown below.

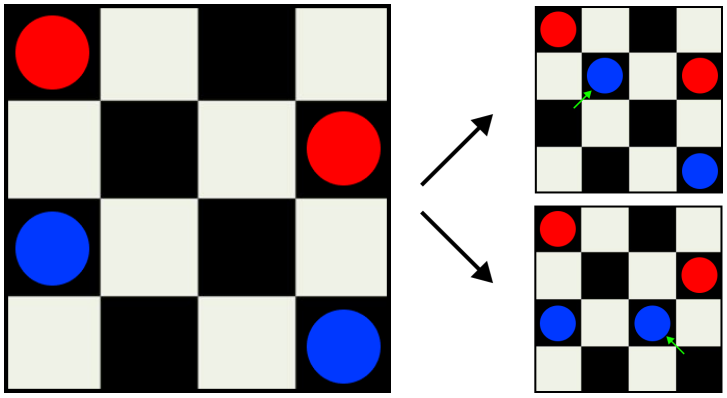
```
def generate_all_states(self,board):
    # DFS to generate all states
    all_states = set()
    all_states_tuple = set()
    stack = []
    stack.append(board)
    while stack:
        current_board = stack.pop()
        all_states.add(current_board)
        all_states_tuple.add(current_board.getMatrixAsTuple())
        self.game.turn = BLUE
        all_moves = []
        for piece_moves in self._generate_all_possible_moves(current_board):
            current_pos = piece_moves[0:2]
            for final_pos in piece_moves[2]:
                all_moves.append([current_pos,final_pos])
        for move in all_moves:
            new_board = deepcopy(current_board)
            self._action(move[0], move[1], new_board)
            self.game.turn = RED
            opponent_piece_moves = self._generate_all_possible_moves(new_board)
            self.game.turn = BLUE
            opponent_moves = []
            for piece_moves in opponent_piece_moves:
                current_pos = piece_moves[0:2]
                for final_pos in piece_moves[2]:
                    opponent_moves.append([current_pos,final_pos])

            for opponent_move in opponent_moves:
                s_next = deepcopy(new_board)
                self._action(opponent_move[0], opponent_move[1], s_next)
                if s_next.getMatrixAsTuple() not in all_states_tuple:
                    stack.append(s_next)
    return all_states
```

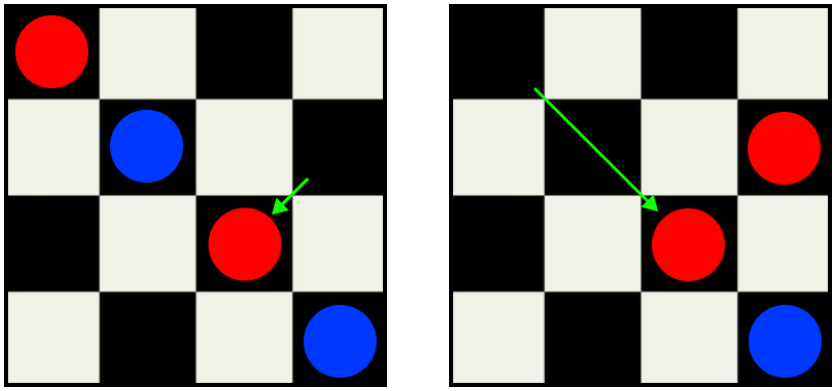
States Transition_

After player 1 plays a move, player 2 will also play its move, resulting in the new state. Hence, the transitions from one state to another are not independent of the actions of player 2.

Let's take the previous example,



Here, player 1 can choose any of the above two steps shown. Say, it chooses the first one. Now, player two will have the following options.



Here, both choices are equiprobable. Let's say player 2 chooses option 2 with probability p (= 0.5). Now, this is when player one will play its move. Hence, this is the new state of s'.

This means that Player 1 played option one from state s and ended up at state s1' with a probability of p. This is because after Player 1 played option 1, player two played option 2 with probability p resulting in state s'. This is how the effect of player two can be modeled into the state transitions of

player 1. Here, since s' is not an end/terminal state, the reward for the move is 0. Rewards are 10000 for the win and -10000 for the loss.

Moving forward, we can model Player 1 as an MDP as follows:

1. States: All possible states of player 1, as found above.
2. Transitions with probability and reward. Following this approach, we can simulate all moves between player one and player two to find all the transition probabilities between all the states of player one along with their rewards.
3. We can set the gamma (discount factor) suitably to 0.9

Using this MDP, we can use the value-iteration algorithm to find an optimal response policy to the set fixed policy of player 1.

Optimal Policy through Value Iteration_

Value iteration is a special case of policy iteration. It does a single iteration of policy evaluation at each step and then, for each state, takes the maximum action value to be the estimated state value.

We need to compute the optimal state-value function V(s) for all the states s and then use the V* function to compute its optimal policy.

The algorithm is defined as follows:

- State-Value function V(s) is initialized to 0 for all states s.
- Repeat until convergence reaches:
    - For each state s, update the state-value function using the Bellman equation:

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s') \right)$$

Here,
a is an action.
s' is the next state.
P(s, a, s') is the probability of transitioning from s to s' through action a.

- If the changes in V(s) across all the states is smaller than some particular small threshold, terminate the algorithm.

The code snippet for value iteration implementation is shown below.

```python
def value_iteration():
    global V, pi, V_initial, states, gamma, delta, max_iter
    for i in range(max_iter):
        max_diff = 0  # Initialize max difference
        V_new = deepcopy(V_initial)
        for s in states:      You, 8 hours ago • MDP implementation almost done …
            max_val = 0
            game = checkers.Game(loop_mode=False)
            game.setup()
            bot = gamebot.Bot(game, RED, mid_eval='piece_and_board',
                    end_eval='sum_of_dist', method='alpha_beta', depth=3)
            r = bot.reward(s)
            all_moves = []
            for piece_moves in bot._generate_all_possible_moves(s):
                current_pos = piece_moves[0:2]
                for final_pos in piece_moves[2]:
                    all_moves.append([current_pos,final_pos])
            for move in all_moves:
                new_board = deepcopy(s)
                bot._action(move[0], move[1], new_board)

                # Compute state value
                val = r  # Get direct reward
                opponent_piece_moves = bot._generate_all_possible_moves(new_board)
                opponent_moves = []
                for piece_moves in opponent_piece_moves:
                    current_pos = piece_moves[0:2]
                    for final_pos in piece_moves[2]:
                        opponent_moves.append([current_pos,final_pos])
                for opponent_move in opponent_moves:
                    s_next = deepcopy(new_board)
                    bot._action(opponent_move[0], opponent_move[1], s_next)
                    if s_next.getMatrixAsTuple() in V:
                        val += (1.0/len(opponent_moves)) * (
                            gamma * V[s_next.getMatrixAsTuple()]
                        )  # Add discounted downstream values
                # Store value best action so far
                max_val = max(max_val, val)
                # Update best policy
                if V[s.getMatrixAsTuple()] < val:
                    pi[s.getMatrixAsTuple()] = move  # Store action with highest value
```

```
            V_new[s.getMatrixAsTuple()] = max_val  # Update value with highest value

            # Update maximum difference
            max_diff = max(max_diff, abs(V[s.getMatrixAsTuple()] - V_new[s.getMatrixAsTuple()]))

        # Update value functions
        V = V_new

        V_str = {}
        for key in V:
            V_str[str(key)] = str(V[key])

        pi_str = {}
        for key in pi:
            pi_str[str(key)] = str(pi[key])

        current_time = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")

        # add the data of V to a file in JSON format
        with open(f'value_data/value_itr_data_{current_time}.json', "w") as outfile:
            json.dump(V_str, outfile)

        with open(f'policy_data/policy_itr_data_{current_time}.json', "w") as outfile:
            json.dump(pi_str, outfile)

        print("Iteration: ", i, ">> Max diff: ", max_diff)

        # If diff smaller than threshold delta for all states, algorithm terminates
        if max_diff < delta:
            break       You, 9 hours ago • MDP implementation almost done …
```

```
(mts) C:\StudyMaterials\Data-Structures-and-Algorithms\Checkers-AI>python valueIteration.py
pygame 2.3.0 (SDL 2.24.2, Python 3.7.6)
Hello from the pygame community. https://www.pygame.org/contribute.html
4029
Iteration:  0 Max diff:  10000.0
Iteration:  1 Max diff:  9000.0
Iteration:  2 Max diff:  8100.0
Iteration:  3 Max diff:  7290.0
Iteration:  4 Max diff:  6561.0
Iteration:  5 Max diff:  5904.900000000015
Iteration:  6 Max diff:  5314.4100000000035
Iteration:  7 Max diff:  4782.969000000012
Iteration:  8 Max diff:  4304.672100000011
Iteration:  9 Max diff:  3874.204890000008
Iteration:  10 Max diff:  3486.7844010000117
Iteration:  11 Max diff:  3138.1059609000076
Iteration:  12 Max diff:  2824.2953648100083
Iteration:  13 Max diff:  2541.8658283290133
Iteration:  14 Max diff:  2287.6792454961105
Iteration:  15 Max diff:  2058.9113209464995
Iteration:  16 Max diff:  1853.0201888518495
Iteration:  17 Max diff:  1667.718169966669
Iteration:  18 Max diff:  1500.9463529700006
Iteration:  19 Max diff:  1350.8517176729947
Iteration:  20 Max diff:  1215.7665459057025
Iteration:  21 Max diff:  1094.1898913151235
Iteration:  22 Max diff:  984.7709021836199
Iteration:  23 Max diff:  886.2938119652536
Iteration:  24 Max diff:  797.6644307687238
Iteration:  25 Max diff:  717.8979876918602
Iteration:  26 Max diff:  646.1081889226625
Iteration:  27 Max diff:  581.4973700303963
Iteration:  28 Max diff:  523.3476330273697
Iteration:  29 Max diff:  471.0128697246255
Iteration:  30 Max diff:  423.91158275216003
Iteration:  31 Max diff:  381.52042447695567
Iteration:  32 Max diff:  343.36838202926447
Iteration:  33 Max diff:  309.0315438263351
Iteration:  34 Max diff:  278.12838944369287
Iteration:  35 Max diff:  250.31555049933377
Iteration:  36 Max diff:  225.28399544939748
Iteration:  37 Max diff:  202.75559590446937
Iteration:  38 Max diff:  182.4800363140239
Iteration:  39 Max diff:  164.23203268261568
```

EXISTING POLICIES_

1. **Min-Max Algorithm:** In this algorithm, the AI evaluates each possible move by simulating the game to a certain depth and assigning a score to each possible outcome. The score of each move is the minimum score the opponent can achieve, assuming they play perfectly, and the AI maximizes its score. The AI selects the move with the highest score as its next move.

2. **Alpha-beta pruning:** This algorithm is a more efficient version of Min-max. It works by cutting off parts of the game tree irrelevant to the final result. When a node in the game tree is being evaluated, if the AI has found a move that is guaranteed to lead to a better outcome than any previously estimated moves, it can "prune" the rest of the subtree, as it is not necessary to explore those branches further.

3. **Random policy:** In this algorithm, we generate all the possible states given an existing board state, which includes moving all the possible pieces that can move, and any one of the next states is selected as the next board state.`