COMPX216 Assignment 3
Adversarial search in
EinStein würfelt nicht!

Abstract

In this assignment, you will complete two implementations of the EinStein würfelt nicht! game. You will also implement a weighting function for a Monte Carlo tree search playout policy.

1 EinStein würfelt nicht!

EinStein würfelt nicht! (EWN) is a stochastic two-player zero-sum board game. Examples of the game can be found online, although there are minor changes to the game's rules in this assignment.

The official game rules can be found via the link provided above, but we describe the rules in this assignment that are simplified and generalised to boards of different sizes:

- The game board is square with a side width of k tiles ($k \geq 3$).
- Two players, Red and Blue, start in two opposite corners of the board.
- Red starts in the top left corner, and Blue starts in the bottom right corner. Red moves first. Red and Blue take turns to move.
- Each player starts with $(k-2)$ pieces on their starting row, $(k-3)$ pieces on the next row, and so on. This described order is used to number the pieces. Refer to the examples given below. Each player starts with a total of n pieces, with $n = \frac{(k-1)(k-2)}{2}$.
- The piece to move is determined by throwing a balanced n-sided die. If the piece associated with the rolled number is no longer on the board, a piece with the next-lower or next-higher number can be moved instead. A player has to make exactly one move during their turn.
- A piece can be moved to a vertically, horizontally, or diagonally adjacent tile closer to the opponent's corner, i.e., Red can move south, east, or southeast, and Blue can move north, west, or northwest. A piece cannot be moved over the edge of the board.

- A piece is captured and removed from the board if another piece, be it the opponent's or the player's own, moves to the tile it is currently occupying.
- A player wins when moving one of their pieces to the furthest tile in their opponent's corner or capturing their opponent's last piece on the board.

1.1 One-piece example

First consider a 3×3 board with one piece for each player. In this case, the element of chance disappears: there is no stochasticity in which piece to move and the game becomes deterministic. The starting position is as follows.

```
|R| | |
| | | |
| | |B|
```

Red starts at `(0, 0)`, and Blue starts at `(2, 2)`. Red can move to `(1, 0)`, `(0, 1)`, or `(1, 1)`. Suppose Red moves to `(1, 1)`.

```
| | | |
| |R| |
| | |B|
```

Blue can move to `(1, 2)`, `(2, 1)`, or `(1, 1)`. In this case Blue is best served by moving to `(1, 1)`, capturing Red's piece and winning the game.

```
| | | |
| |B| |
| | | |
```

Because of this, an adversarial search algorithm like minimax search should recognise that moving to `(1, 1)` as the first move is not in Red's interest and try another move such as `(1, 0)`, shown below, in the hope of reaching Blue's corner or capturing Blue's piece before Blue can do the same to Red.

```
| | | |
|R| | |
| | |B|
```

1.2 Three-piece example

Now consider a 4×4 board with three pieces for each player. The starting position is as follows.

```
|R0|R1|  |  |
|R2|  |  |  |
|  |  |  |B2|
|  |  |B1|B0|
```

Red moves first, and the piece to move is determined by sampling uniformly from `{0, 1, 2}`. Suppose the sampling outcome is `0`. Red can choose to move their `0`-th piece to `(1, 0)`, `(0, 1)`, or `(1, 1)`. We examine the differences between the first two choices.

Applying the first choice, the `0`-th piece is moved to `(1, 0)`, capturing and removing Red's own `2`-nd piece.

```
|  |R1|  |  |
|R0|  |  |  |
|  |  |  |B2|
|  |  |B1|B0|
```

At Red's next turn, the `0`-th piece can be moved if the sampling outcome is `0`, and the `1`-st piece can be moved if the sampling outcome is `1` or `2`, because the `2`-nd piece is no longer on the board, and it only has a next-lower piece, i.e., the `1`-st piece.

Applying the second choice to the starting position, the `0`-th piece is moved to `(0, 1)`, capturing and removing Red's own `1`-st piece.

```
|  |R0|  |  |
|R2|  |  |  |
|  |  |  |B2|
|  |  |B1|B0|
```

At Red's next turn, the `0`-th piece can be moved if the sampling outcome is `0` or `1`, and the `2`-nd piece can be moved if the sampling outcome is `1` or `2`, because the `1`-st piece is no longer on the board, and it has a next-lower piece and a next-higher piece, i.e., the `0`-th piece and the `2`-nd piece.

Red wins if they manage to move one of their pieces to Blue's corner, i.e., `(3, 3)`, and Blue wins if they manage to move to `(0, 0)`. In addition, one player wins if their opponent loses all their pieces.

Generally in EWN, the fewer pieces a player has, the more mobile these remaining pieces become. A player needs to balance the benefit of having fewer but more mobile pieces and the risk of losing all pieces and the game.

2 Tasks

There are three tasks you need to perform for this assignment. Download the provided `assignment3.py` file from Moodle and move it into the `aima-python` directory. This file contains the skeleton code for this assignment. You can follow the comments in this code to complete the assignment. Test code for each task is provided in the lower half of the file. After completing a task, uncomment the corresponding block of test code by removing the `'''` marks around it to test it in your run. Do not modify the names of classes, functions, or variables provided in the file. This is the file you will submit for marking. It is permitted to import additional modules required by your implementation.

Once you have written the code for a task, uncomment its test code and run the assignment code in a command-line terminal as shown below. Remember to ensure that your working directory is `aima-python`, and your virtual environment is activated.

```
python assignment3.py
```

2.1 Task 1

Your first task is to complete the `EinStein` ("`OneStone`" in German) class that represents a 3×3 board starting with one piece per player. It operates on instances of `GameState`, which is a `namedtuple` defined in `aima-python`'s `games4e.py`. A `GameState` contains four values:

- `to_move` indicating which player's turn it is to move,
- `utility` indicating the utility of the board,
- `board` representing the state of the board, and
- `moves` containing all legal moves in this state.

The board in particular is a `dict` mapping a player (`'R'` or `'B'`) to a tuple of coordinates of their piece on the board (`(i, j)` for the `i`-th row and `j`-th column). If a piece is captured and removed, it is mapped to `None` instead of a tuple of coordinates.

Some functions are provided, including the following:

- `__init__()` initialises a `GameState` with Red to move, one piece per player in its starting position, and a list of legal moves for Red,
- `compute_moves()` returns a list of legal moves on input `board` for the player indicated by the variable `to_move`, where each move is represented by a tuple of destination coordinates, e.g., `(1, 2)` means moving to the `1`-st row and `2`-nd column,
- `display()` prints an input `state` to the terminal,
- `terminal_test()` returns whether an input `state` is a terminal game state, and
- `actions()` returns a list of legal moves.

You need to implement three functions: `result()`, `utility()`, and `compute_utility()`.

2.1.1 Task 1.1

The `result()` function takes a `GameState` object `state` and a `move` as input and returns a resulting `GameState`. In particular, it needs to do the following:

1. determine the player and the opponent in the `state`,
2. make a deep copy of the `state`'s board,
3. execute the `move` on the copy and, if it results in a capture, remove the opponent's piece by setting its value to `None`, and
4. return a `GameState` where it is the opponent's turn on the post-`move` copy of the board, with utility computed using `compute_utility()` and legal moves computed using `compute_moves()`.

2.1.2 Task 1.2

The `utility()` function takes a `state` and a `player` as input and returns `state`'s utility to `player`. As the following task will implement, the utility stored in a state is its utility to Red, so this stored utility is returned if `player` is `'R'`, and the negated value is returned if `player` is `'B'`.

2.1.3 Task 1.3

The `compute_utility()` function takes a `board` as input and returns its utility to Red. If Red's piece has been removed, or Blue's piece has reached the top left tile, the function returns `-1`. Conversely, if Blue's piece has been removed, or Red's piece has reached the bottom right tile, the function returns `1`. It returns `0` in all other scenarios.

2.2 Task 2

Your second task is to complete the `MehrSteine` ("`MoreStones`" in German) class that represents a $k \times k$ board (k stored in `self.board_size` for later access, $k \geq 3$) starting with $n = \frac{(k-1)(k-2)}{2}$ pieces per player (n is stored in `self.num_piece` for later access). It operates on instances of `StochasticGameState`, which is a `namedtuple` defined in `aima-python`'s `games4e.py`. In addition to the four values of `GameState`, `StochasticGameState` contains an additional `chance` value that represents the outcome of a stochastic decision process, e.g., the outcome of a dice roll.

The board is now a `dict` mapping a player to a list of coordinate tuples, as each player can have multiple pieces. A piece's index in the list is its associated number, and a removed piece is set to `None`.

Like before, some functions are provided: `__init__()`, `compute_moves()`, `display()`, `terminal_test()`, and `actions()`. The function `compute_moves()` is different from its previous iteration in `EinStein`. It takes a `board`, a player `to_move`, and the `index` of a piece as input and returns a list of moves. Each

move is a tuple of the `index` and a destination's coordinates, e.g., `(3, (1, 2))` means moving the `3`-rd piece to the `1`-st row and `2`-nd column.

You need to implement six functions: `result()`, `utility()`, `compute_utility()`, `chances()`, `outcome()`, and `probability()`.

2.2.1 Task 2.1

The `result()` function is similar to its previous iteration in `EinStein`. It takes a `state` and a `move` as input and returns a `StochasticGameState`. Note `move` is now a nested tuple containing both an index and a pair of coordinates. Remember to check whether any piece is captured and needs to be removed from the board. The `moves` and `chance` values of the returned state do not need to be computed here and should both be set to `None`.

2.2.2 Task 2.2

The `utility()` function is the same as its previous iteration in `EinStein`.

2.2.3 Task 2.3

The `compute_utility()` function is similar to its previous iteration in `EinStein`. The `board`'s utility to Red is returned. It is `-1` if Red loses all pieces, or one of Blue's pieces reaches the top left tile. Conversely, it is `1` if Blue loses all pieces, or one of Red's pieces reaches the bottom right tile. It is `0` in all other cases.

2.2.4 Task 2.4

The `chances()` function returns a list of possible outcomes of a dice roll. With n starting pieces per player, it should return a list counting from 0 to $(n-1)$.

2.2.5 Task 2.5

The `outcome()` function takes a `state` and a dice roll outcome `chance` as input and returns a `StochasticGameState` containing legal moves based on `chance`. Only the `moves` value of `state` needs to be changed to form the output state, and the other values can remain the same. Use the `compute_moves()` function for easy computation of the moves. Note that if the piece associated with `chance` is no longer on the board, the next-lower or next-higher piece can be selected for movement instead.

2.2.6 Task 2.6

The `probability()` function returns the probability of a given chance outcome. As the chances are uniformly distributed, it should always return $\frac{1}{n}$ with n being the number of starting pieces per player.

Table 1: The weight of an action based on its resulting state's relative difference in the two players' Schwarz scores

$\frac{diff}{max\_schwarz}$	weight
$(-\infty, -0.5)$	1
$[-0.5, -0.375)$	2
$[-0.375, -0.25)$	4
$[-0.25, -0.125)$	8
$[-0.125, 0)$	16
$[0, 0.125)$	32
$[0.125, 0.25)$	64
$[0.25, 0.375)$	128
$[0.375, 0.5)$	256
$[0.5, \infty)$	512

2.3 Task 3

Your third task is to implement a weighting function `schwarz_diff_to_weight()` for a Monte Carlo tree search playout policy.

A player's Schwarz score in EWN is an estimate of the number of turns still needed to reach their goal assuming no capture happens. Trying to increase the opponent's Schwarz score while decreasing their own generally improves a player's chance of winning. A playout policy based on the Schwarz score has been partially implemented. You need to complete it by implementing the `schwarz_diff_to_weight()` function to return the weight of an action based on its resulting state's relative difference in the two players' Schwarz scores as indicated in Table 1.

We notate the function that takes an action as input and returns the Schwarz-based weight of the state resulting from the action as f. The probability of an action a being selected from a set of actions A for the categorical random variable *Action* during playout is its weight divided by the sum of the weights of all actions in A:

$$P(Action = a) = \frac{f(a)}{\sum_{a'}^{A} f(a')}. \tag{1}$$

Use the provided test code to verify that a Schwarz-based policy is more likely to win than lose against a random policy in Monte Carlo tree search.