Started on	Thursday, 12 October 2023, 4:31 PM
State	Finished
Completed on	Thursday, 12 October 2023, 10:22 PM
Time taken	5 hours 51 mins
Marks	5.00/5.00
Grade	**1.00** out of 1.00 (**100**%)

Information

Introduction

This quiz starts with a number questions that use *explicit* game trees. Towards the end of the quiz you will see an example that requires *implicit* game trees. Explicit trees are useful for education purposes. However, in practice, game trees are usually very big and would take a lot of memory (and time) if they were to be stored in the memory. It is also inefficient to first generate a tree and then search it. The search and generation can be done at the same time. The search and generation are done according to the rules of the game without using much memory (linear in branching factor and the average number of moves to the end of the game). The time however still grows exponentialy with respect to the depth of the tree.

Information

Representation of explicit game trees

An explicit game tree is a tree that is already constructed and resides in the memory (as opposed to being constructed on the fly). We use the following recursive representation for explicit game trees. A game tree is either

- a number which represents the utility (payoff) of a terminal (end-game) state; or
- a list of one or more game trees.

Examples

The root of the following game tree has three children. The first child is a leaf node with a utility of 1. The second child has two children (leaf nodes with utilities 2 and 3). The third child of the root has a single child which has a single child (a leaf node with a utility of 4).

```
game_tree = [1, [2, 3], [[4]]]
```

The following trees are all different. The first one is game tree that is a single leaf node. The second one has a root with one child which is terminal. The third one has a root which has one child which has one terminal child.

```
game_tree1 = 7
game_tree2 = [7]
game_tree3 =[[7]]
```

Question 1

Correct

Mark 1.00 out of 1.00

Write **two** functions `max_value(tree)` and `min_value(tree)` that given a game tree, return the utility of the root of the tree when the root is a max node or min node correspondingly. Process the children of a node from left (lower index) to right (higher index).

For example:

Test	Result

`from student_answer import min_value, max_value`
 	
`game_tree = 3`
 	
`print("Root utility for minimiser:", min_value(game_tree))`
`print("Root utility for maximiser:", max_value(game_tree))`
	`Root utility for minimiser: 3`
	`Root utility for maximiser: 3`

`from student_answer import min_value, max_value`
 	
`game_tree = [1, 2, 3]`
 	
`print("Root utility for minimiser:", min_value(game_tree))`
`print("Root utility for maximiser:", max_value(game_tree))`
	`Root utility for minimiser: 1`
	`Root utility for maximiser: 3`

`from student_answer import min_value, max_value`
 	
`game_tree = [1, 2, [3]]`
 	
`print(min_value(game_tree))`
`print(max_value(game_tree))`
	`1`
	`3`

`from student_answer import min_value, max_value`
 	
`game_tree = [[1, 2], [3]]`
 	
`print(min_value(game_tree))`
`print(max_value(game_tree))`
	`2`
	`3`

Answer: (penalty regime: 0, 15, ... %)

```
def max_value(tree):
    if type(tree) is int:
        return tree
    return max([min_value(subtree) for subtree in tree])

def min_value(tree):
    if type(tree) is int:
        return tree
    return min([max_value(subtree) for subtree in tree])
```

	Test	Expected	Got	

✔	`from student_answer import min_value, max_value`
	
	`game_tree = 3`
	
	`print("Root utility for minimiser:", min_value(game_tree))`
	`print("Root utility for maximiser:", max_value(game_tree))`
		`Root utility for minimiser: 3`
		`Root utility for maximiser: 3`
			`Root utility for minimiser: 3`
			`Root utility for maximiser: 3`
				✔

✔	`from student_answer import min_value, max_value`
	
	`game_tree = [1, 2, 3]`
	
	`print("Root utility for minimiser:", min_value(game_tree))`
	`print("Root utility for maximiser:", max_value(game_tree))`
		`Root utility for minimiser: 1`
		`Root utility for maximiser: 3`
			`Root utility for minimiser: 1`
			`Root utility for maximiser: 3`
				✔

✔	`from student_answer import min_value, max_value`
	
	`game_tree = [1, 2, [3]]`
	
	`print(min_value(game_tree))`
	`print(max_value(game_tree))`
		`1`
		`3`
			`1`
			`3`
				✔

✔	`from student_answer import min_value, max_value`
	
	`game_tree = [[1, 2], [3]]`
	
	`print(min_value(game_tree))`
	`print(max_value(game_tree))`
		`2`
		`3`
			`2`
			`3`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question 2

Correct

Mark 1.00 out of 1.00

Write **two** functions `max_action_value(game_tree)` and `min_action_value(game_tree)` that given a game tree, return a pair where first element is the best action and the second element is the utility of the root of the tree when the root is a max node or min node correspondingly. For a leaf node the action is `None`; for an internal node, the action is the index of the subtree corresponding to the best action. Process the children of a node from left (lower index) to right (higher index). If there is a tie, return the left-most optimal action.

For example:

Test	Result

`from student_answer import min_action_value, max_action_value`
 	
`game_tree = [2, [-3, 1], 4, 1]`
 	
`action, value = min_action_value(game_tree)`
`print("Best action if playing min:", action)`
`print("Best guaranteed utility:", value)`
`print()`
`action, value = max_action_value(game_tree)`
`print("Best action if playing max:", action)`
`print("Best guaranteed utility:", value)`
	`Best action if playing min: 1`
	`Best guaranteed utility: 1`
	
	`Best action if playing max: 2`
	`Best guaranteed utility: 4`

`from student_answer import min_action_value, max_action_value`
 	
`game_tree = 3`
 	
`action, value = min_action_value(game_tree)`
`print("Best action if playing min:", action)`
`print("Best guaranteed utility:", value)`
`print()`
`action, value = max_action_value(game_tree)`
`print("Best action if playing max:", action)`
`print("Best guaranteed utility:", value)`
	`Best action if playing min: None`
	`Best guaranteed utility: 3`
	
	`Best action if playing max: None`
	`Best guaranteed utility: 3`

`from student_answer import min_action_value, max_action_value`
 	
`game_tree = [1, 2, [3]]`
 	
`action, value = min_action_value(game_tree)`
`print("Best action if playing min:", action)`
`print("Best guaranteed utility:", value)`
`print()`
`action, value = max_action_value(game_tree)`
`print("Best action if playing max:", action)`
`print("Best guaranteed utility:", value)`
	`Best action if playing min: 0`
	`Best guaranteed utility: 1`
	
	`Best action if playing max: 2`
	`Best guaranteed utility: 3`

Answer: (penalty regime: 0, 15, ... %)

```
def max_value(tree):
    if type(tree) is int:
        return tree
    return max([min_value(subtree) for subtree in tree])

def min_value(tree):
    if type(tree) is int:
        return tree
    return min([max_value(subtree) for subtree in tree])

def max_action_value(game_tree):
    if type(game_tree) is int:
        return None, game_tree
    min_values = [min_value(subtree) for subtree in game_tree]
    utility = max(min_values)
    return min_values.index(utility), utility

def min_action_value(game_tree):
    if type(game_tree) is int:
        return None, game_tree
    max_values = [max_value(subtree) for subtree in game_tree]
    utility = min(max_values)
    return max_values.index(utility), utility
```

	Test	Expected	Got	

✔	`from student_answer import min_action_value, max_action_value`
	
	`game_tree = [2, [-3, 1], 4, 1]`
	
	`action, value = min_action_value(game_tree)`
	`print("Best action if playing min:", action)`
	`print("Best guaranteed utility:", value)`
	`print()`
	`action, value = max_action_value(game_tree)`
	`print("Best action if playing max:", action)`
	`print("Best guaranteed utility:", value)`
		`Best action if playing min: 1`
		`Best guaranteed utility: 1`
		
		`Best action if playing max: 2`
		`Best guaranteed utility: 4`
			`Best action if playing min: 1`
			`Best guaranteed utility: 1`
			
			`Best action if playing max: 2`
			`Best guaranteed utility: 4`
				✔

✔	`from student_answer import min_action_value, max_action_value`
	
	`game_tree = 3`
	
	`action, value = min_action_value(game_tree)`
	`print("Best action if playing min:", action)`
	`print("Best guaranteed utility:", value)`
	`print()`
	`action, value = max_action_value(game_tree)`
	`print("Best action if playing max:", action)`
	`print("Best guaranteed utility:", value)`
		`Best action if playing min: None`
		`Best guaranteed utility: 3`
		
		`Best action if playing max: None`
		`Best guaranteed utility: 3`
			`Best action if playing min: None`
			`Best guaranteed utility: 3`
			
			`Best action if playing max: None`
			`Best guaranteed utility: 3`
				✔

✔	`from student_answer import min_action_value, max_action_value`
	
	`game_tree = [1, 2, [3]]`
	
	`action, value = min_action_value(game_tree)`
	`print("Best action if playing min:", action)`
	`print("Best guaranteed utility:", value)`
	`print()`
	`action, value = max_action_value(game_tree)`
	`print("Best action if playing max:", action)`
	`print("Best guaranteed utility:", value)`
		`Best action if playing min: 0`
		`Best guaranteed utility: 1`
		
		`Best action if playing max: 2`
		`Best guaranteed utility: 3`
			`Best action if playing min: 0`
			`Best guaranteed utility: 1`
			
			`Best action if playing max: 2`
			`Best guaranteed utility: 3`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

α-β pruning

In an alpha-beta pruning question you are given an explicit game tree (where either max is playing at root or min) and then asked to prune the tree. You need to provide two variables: `pruned_tree` which is the pruned game tree and `pruning_events` which is a list of pairs of alpha and beta when a pruning event was triggered. Please keep the following points in mind:

- It might be easier to answer this question by drawing the tree.
- Process the children of a node from left to right.
- A pruning event is triggered when alpha becomes greater than or equal to beta.
- Children of a node are processed from left to right.
- A pruning event might be triggered without any branches getting pruned. This happens when the event is triggered after seeing the last child. Such events must also be included in the list.
- For some problems the pruned tree is the same as the original tree (i.e. no pruning) even if there have been some pruning events.
- You do not need to provide any function in your answer, however, if you wish, you can write a program to compute the requested variables automatically.

Question 3

Correct

Mark 1.00 out of 1.00

Consider the following explicit game tree.

```
[2, [-1, 5], [1, 3], 4]
```

Assuming that the player at the root of the tree is Max, prune the tree (if necessary). Provide two variables: `pruned_tree` which is the pruned game tree and `pruning_events` which is a list of pairs of alpha and beta, one for each time a pruning event was triggered.

.

For example:

Test	Result

`import student_answer`
 	
`check_it_is_a_gametree(student_answer.pruned_tree)`
`check_it_is_a_pruning_events_list(student_answer.pruning_events)`
	OK
	OK

Answer: (penalty regime: 0, 15, ... %)

Reset answer

```
from math import inf

pruned_tree = [
    2,
    [-1],
    [1],
    4
    ]


pruning_events = [
    (2, -1),
    (2, 1)
    ]
```

	Test	Expected	Got	

✔	`import student_answer`
	
	`check_it_is_a_gametree(student_answer.pruned_tree)`
	`check_it_is_a_pruning_events_list(student_answer.pruning_events)`
		OK
		OK
			OK
			OK
				✔

✔	`import student_answer`
	
	`check_pruned_tree(student_answer.pruned_tree)`
		OK	OK	✔

✔	`import student_answer`
	
	`check_pruning_events(student_answer.pruning_events)`
		OK	OK	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question 4

Correct

Mark 1.00 out of 1.00

Consider the following explicit game tree.

```
[0, [-2, 1], 5]
```

Assuming that the player at the root of the tree is Min, prune the tree (if necessary). Provide two variables: `pruned_tree` which is the pruned game tree and `pruning_events` which is a list of pairs of alpha and beta, one for each time a pruning event was triggered.

.

For example:

Test	Result

`import student_answer`
 	
`check_it_is_a_gametree(student_answer.pruned_tree)`
`check_it_is_a_pruning_events_list(student_answer.pruning_events)`
	OK
	OK

Answer: (penalty regime: 0, 15, ... %)

Reset answer

```
from math import inf

pruned_tree = [
    0,
    [-2, 1],
    5
    ]


pruning_events = [
    (1, 0),
    ]
```

	Test	Expected	Got	

✔	`import student_answer`
	
	`check_it_is_a_gametree(student_answer.pruned_tree)`
	`check_it_is_a_pruning_events_list(student_answer.pruning_events)`
		OK
		OK
			OK
			OK
				✔

✔	`import student_answer`
	
	`check_pruned_tree(student_answer.pruned_tree)`
		OK	OK	✔

✔	`import student_answer`
	
	`check_pruning_events(student_answer.pruning_events)`
		OK	OK	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question 5

Correct

Mark 1.00 out of 1.00

Consider the following explicit game tree.

```
[3, [[2, 1], [4, [7, -2]]], 0]
```

Assuming that the player at the root of the tree is Max, prune the tree (if necessary). Provide two variables: `pruned_tree` which is the pruned game tree and `pruning_events` which is a list of pairs of alpha and beta, one for each time a pruning event was triggered.

.

For example:

Test	Result

`import student_answer`
 	
`check_it_is_a_gametree(student_answer.pruned_tree)`
`check_it_is_a_pruning_events_list(student_answer.pruning_events)`
	OK
	OK

Answer: (penalty regime: 0, 15, ... %)

Reset answer

```
from math import inf

pruned_tree = [
    3,
    [[2, 1]],
    0
    ]


pruning_events = [
    (3, 2),
    ]
```

	Test	Expected	Got	

✔	`import student_answer`
	
	`check_it_is_a_gametree(student_answer.pruned_tree)`
	`check_it_is_a_pruning_events_list(student_answer.pruning_events)`
		OK
		OK
			OK
			OK
				✔

✔	`import student_answer`
	
	`check_pruned_tree(student_answer.pruned_tree)`
		OK	OK	✔

✔	`import student_answer`
	
	`check_pruning_events(student_answer.pruning_events)`
		OK	OK	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

If you are interested in implementing a complete game-playing agent for a small game (noughts and crosses) work on the next question. The question is challenging but does **not** carry any marks.

Question 6

Not answered

Not graded

In this question you need to implement an optimal player for the game *noughts and crosses* (aka Tic-tac-toe). We generalise the game to an n-by-n board where a player wins if some of their pieces form a complete row, column, or one of the two diagonals. We represent the board by a list of lists (list of rows) where the elements are either `X`, `O`, or `'.'`.

Write a function `optimal_move(board, player)` that takes a board and a player (characters 'X' or 'O') and returns a pair *(row, col)* which is the position at which the player must place her next piece in order to guarantee the best possible result. If multiple moves are equally good, the function must return one that has the lowest row number and then the lowest column number. The function will be tested with various board sizes (n larger than zero). The size of the search space will be roughly about that of an empty board with n=3. The function must compute the optimal move in less than a second.

Notes

- Since we are using a mutable object (list of lists) to represent the board, make sure the board does not get accidentally modified in your function.
- You are required to submit only one function. Other functions you see in the test cases are defined on the server (but not accessible by your code). It is up to you how you decompose the problem into several functions (or take an OO approach).
- You will need to do pruning in order to meet the required time constraint.
- If allocating and deallocating memory (creating new boards in each stack frame) is slowing down your program, you can keep track moves in stack frames and work with only one board (i.e. move, make the recursive call, and then 'unmove' all on the same board).
- It is recommended that you do some local testing with some games before submitting your solution.

For example:

Test	Result

`from student_answer import optimal_move`
 	
`board = [['.', '.'], ['.', '.']]`
`player = 'X'`
 	
`print(optimal_move(board, player))`
	`(0, 0)`

Test	Result

from student_answer import optimal_move as student_optimal
 	
board_str = """\
...
...
...
"""
 	
board = board_from_string(board_str)
player = student = 'X'
 	
display(board)
while not is_terminal(board):
  move_function = student_optimal if player == student else server_optimal
  move = move_function(board, player)
  print(f"{player} plays {move}:")
  apply(move, board, player)
  display(board)
  player = opponent(player)
	+---+
	|...|
	|...|
	|...|
	+---+
	
	X plays (0, 0):
	+---+
	|X..|
	|...|
	|...|
	+---+
	
	O plays (1, 1):
	+---+
	|X..|
	|.O.|
	|...|
	+---+
	
	X plays (0, 1):
	+---+
	|XX.|
	|.O.|
	|...|
	+---+
	
	O plays (0, 2):
	+---+
	|XXO|
	|.O.|
	|...|
	+---+
	
	X plays (2, 0):
	+---+
	|XXO|
	|.O.|
	|X..|
	+---+
	
	O plays (1, 0):
	+---+
	|XXO|
	|OO.|
	|X..|
	+---+
	
	X plays (1, 2):
	+---+
	|XXO|
	|OOX|
	|X..|
	+---+
	
	O plays (2, 1):
	+---+
	|XXO|
	|OOX|
	|XO.|
	+---+
	
	X plays (2, 2):
	+---+
	|XXO|
	|OOX|
	|XOX|
	+---+

Answer:

```
1
```