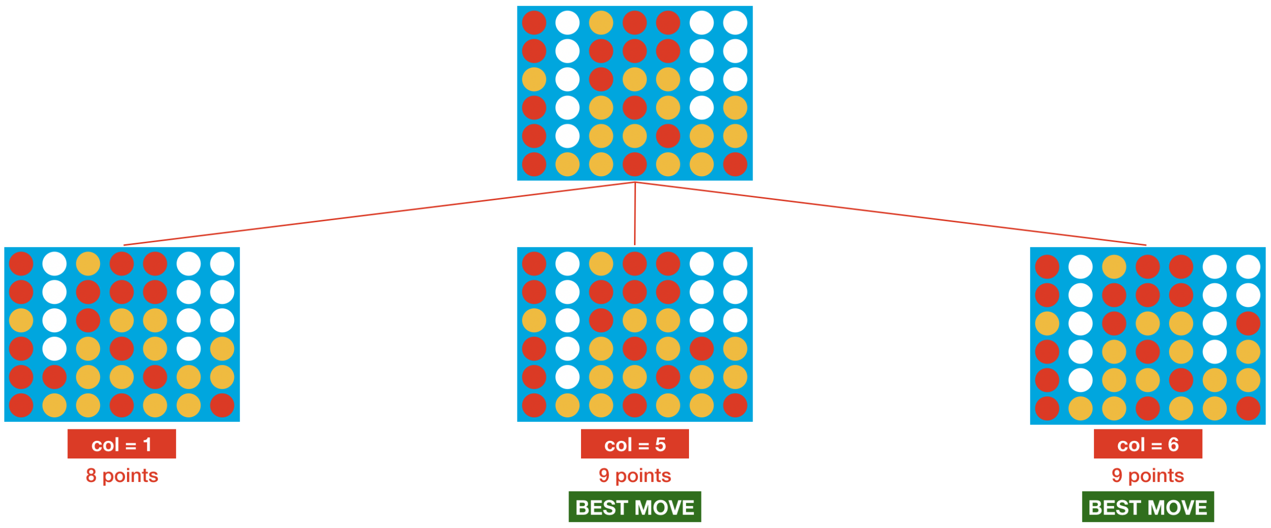
Introduction

In the previous tutorial, you learned how to build an agent with one-step lookahead. This agent performs reasonably well, but definitely still has room for improvement! For instance, consider the potential moves in the figure below. (*Note that we use zero-based numbering for the columns, so the leftmost column corresponds to* `col=0`*, the next column corresponds to* `col=1`*, and so on.*)



With one-step lookahead, the red player picks one of column 5 or 6, each with 50% probability. But, column 5 is clearly a bad move, as it lets the opponent win the game in only one more turn. Unfortunately, the agent doesn't know this, because it can only look one move into the future.

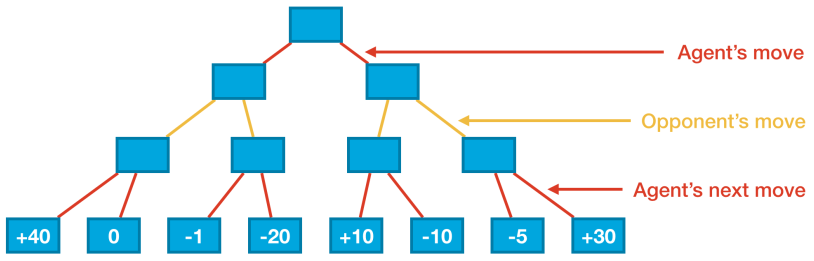
In this tutorial, you'll use the **minimax algorithm** to help the agent look farther into the future and make better-informed decisions.

Minimax

We'd like to leverage information from deeper in the game tree. For now, assume we work with a depth of 3. This way, when deciding its move, the agent considers all possible game boards that can result from

1. the agent's move,
2. the opponent's move, and
3. the agent's next move.

We'll work with a visual example. For simplicity, we assume that at each turn, both the agent and opponent have only two possible moves. Each of the blue rectangles in the figure below corresponds to a different game board.



We have labeled each of the "leaf nodes" at the bottom of the tree with the score from the heuristic. (*We use made-up scores in the figure. In the code, we'll use the same heuristic from the previous tutorial.*) As before, the current game board is at the top of the figure, and the agent's goal is to end up with a score that's as high as possible.

But notice that the agent no longer has complete control over its score -- after the agent makes its move, the opponent selects its own move. And, the opponent's selection can prove disastrous for the agent! In particular,

- If the agent chooses the left branch, the opponent can force a score of -1.

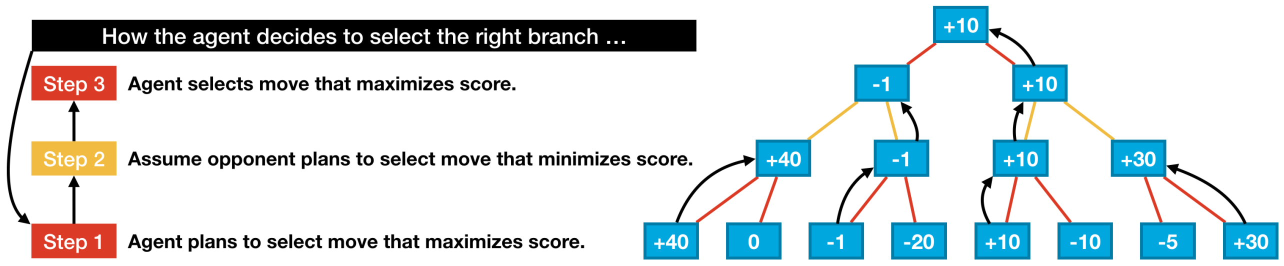
- If the agent chooses the right branch, the opponent can force a score of +10.

Take the time now to check this in the figure, to make sure it makes sense to you!

With this in mind, you might argue that the right branch is the better choice for the agent, since it is the less risky option. Sure, it gives up the possibility of getting the large score (+40) that can only be accessed on the left branch, but it also guarantees that the agent gets at least +10 points.

This is the main idea behind the **minimax algorithm**: the agent chooses moves to get a score that is as high as possible, and it assumes the opponent will counteract this by choosing moves to force the score to be as low as possible. That is, the agent and opponent have opposing goals, and we assume the opponent plays optimally.

So, in practice, how does the agent use this assumption to select a move? We illustrate the agent's thought process in the figure below.


How the agent decides to select the right branch ...
Step 3
Agent selects move that maximizes score.
Step 2
Assume opponent plans to select move that minimizes score.
Step 1
Agent plans to select move that maximizes score.
+10
-1
+10
+40
-1
+10
+30
+40
0
-1
-20
+10
-10
-5
+30

In the example, minimax assigns the move on the left a score of -1, and the move on the right is assigned a score of +10. So, the agent will select the move on the right.

Code

We'll use several functions from the previous tutorial. These are defined in the hidden code cell below. (*Click on the "Code" button below if you'd like to view them.*)

In [ ]:

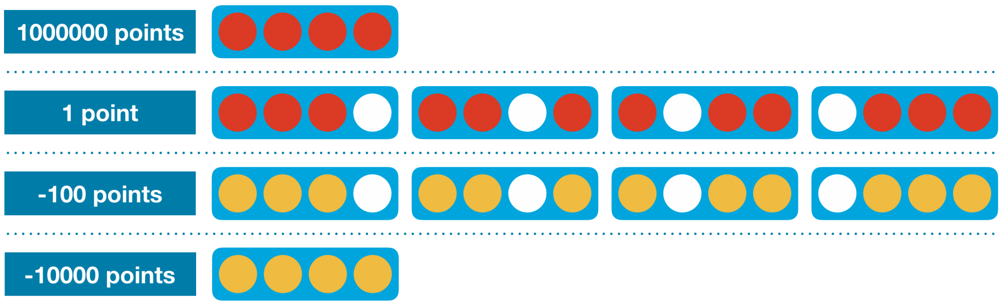
```
import random
import numpy as np

# Gets board at next step if agent drops piece in selected column
def drop_piece(grid, col, mark, config):
    next_grid = grid.copy()
    for row in range(config.rows-1, -1, -1):
        if next_grid[row][col] == 0:
            break
    next_grid[row][col] = mark
    return next_grid

# Helper function for get_heuristic: checks if window satisfies heuristic conditions
def check_window(window, num_discs, piece, config):
    return (window.count(piece) == num_discs and window.count(0) == config.inarow-num_di
scs)

# Helper function for get_heuristic: counts number of windows satisfying specified heuris
tic conditions
def count_windows(grid, num_discs, piece, config):
    num_windows = 0
    # horizontal
    for row in range(config.rows):
        for col in range(config.columns-(config.inarow-1)):
            window = list(grid[row, col:col+config.inarow])
            if check_window(window, num_discs, piece, config):
                num_windows += 1
    # vertical
    for row in range(config.rows-(config.inarow-1)):
        for col in range(config.columns):
            window = list(grid[row:row+config.inarow, col])
            if check_window(window, num_discs, piece, config):
                num_windows += 1
    # positive diagonal
    for row in range(config.rows-(config.inarow-1)):
        for col in range(config.columns-(config.inarow-1)):
            window = list(grid[range(row, row+config.inarow), range(col, col+config.inar
ow)])
            if check_window(window, num_discs, piece, config):
                num_windows += 1
    # negative diagonal
    for row in range(config.inarow-1, config.rows):
        for col in range(config.columns-(config.inarow-1)):
            window = list(grid[range(row, row-config.inarow, -1), range(col, col+config.
inarow)])
            if check_window(window, num_discs, piece, config):
                num_windows += 1
    return num_windows
```

We'll also need to slightly modify the heuristic from the previous tutorial, since the opponent is now able to modify the game board.



In particular, we need to check if the opponent has won the game by playing a disc. The new heuristic looks at each group of four adjacent locations in a (horizontal, vertical, or diagonal) line and assigns:

- **1000000 (`1e6`) points if the agent has four discs in a row (the agent won),**
- **1 point if the agent filled three spots, and the remaining spot is empty (the agent wins if it fills in the empty spot),**
- **-100 points if the opponent filled three spots, and the remaining spot is empty (the opponent wins by filling in the empty spot), and**
- **-10000 (`-1e4`) points if the opponent has four discs in a row (the opponent won).**

This is defined in the code cell below.

In [ ]:

```
# Helper function for minimax: calculates value of heuristic for grid
def get_heuristic(grid, mark, config):
    num_threes = count_windows(grid, 3, mark, config)
    num_fours = count_windows(grid, 4, mark, config)
    num_threes_opp = count_windows(grid, 3, mark%2+1, config)
    num_fours_opp = count_windows(grid, 4, mark%2+1, config)
    score = num_threes - 1e2*num_threes_opp - 1e4*num_fours_opp + 1e6*num_fours
    return score
```

In the next code cell, we define a few additional functions that we'll need for the minimax agent.

In [ ]:

```
# Uses minimax to calculate value of dropping piece in selected column
def score_move(grid, col, mark, config, nsteps):
    next_grid = drop_piece(grid, col, mark, config)
    score = minimax(next_grid, nsteps-1, False, mark, config)
    return score

# Helper function for minimax: checks if agent or opponent has four in a row in the windo
w
def is_terminal_window(window, config):
    return window.count(1) == config.inarow or window.count(2) == config.inarow

# Helper function for minimax: checks if game has ended
```

```
def is_terminal_node(grid, config):
    # Check for draw
    if list(grid[0, :]).count(0) == 0:
        return True
    # Check for win: horizontal, vertical, or diagonal
    # horizontal
    for row in range(config.rows):
        for col in range(config.columns-(config.inarow-1)):
            window = list(grid[row, col:col+config.inarow])
            if is_terminal_window(window, config):
                return True
    # vertical
    for row in range(config.rows-(config.inarow-1)):
        for col in range(config.columns):
            window = list(grid[row:row+config.inarow, col])
            if is_terminal_window(window, config):
                return True
    # positive diagonal
    for row in range(config.rows-(config.inarow-1)):
        for col in range(config.columns-(config.inarow-1)):
            window = list(grid[range(row, row+config.inarow), range(col, col+config.inar
ow)])
            if is_terminal_window(window, config):
                return True
    # negative diagonal
    for row in range(config.inarow-1, config.rows):
        for col in range(config.columns-(config.inarow-1)):
            window = list(grid[range(row, row-config.inarow, -1), range(col, col+config.
inarow)])
            if is_terminal_window(window, config):
                return True
    return False

# Minimax implementation
def minimax(node, depth, maximizingPlayer, mark, config):
    is_terminal = is_terminal_node(node, config)
    valid_moves = [c for c in range(config.columns) if node[0][c] == 0]
    if depth == 0 or is_terminal:
        return get_heuristic(node, mark, config)
    if maximizingPlayer:
        value = -np.Inf
        for col in valid_moves:
            child = drop_piece(node, col, mark, config)
            value = max(value, minimax(child, depth-1, False, mark, config))
        return value
    else:
        value = np.Inf
        for col in valid_moves:
            child = drop_piece(node, col, mark%2+1, config)
            value = min(value, minimax(child, depth-1, True, mark, config))
        return value
```

We won't describe the minimax implementation in detail, but if you want to read more technical pseudocode, here's the description from Wikipedia. (*Note that the pseudocode can be safely skipped!*)

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

Finally, we implement the minimax agent in the competition format. The `N_STEPS` variable is used to set the depth of the tree.

In [ ]:

```
# How deep to make the game tree: higher values take longer to run!
N_STEPS = 3

def agent(obs, config):
    # Get list of valid moves
    valid_moves = [c for c in range(config.columns) if obs.board[c] == 0]
    # Convert the board to a 2D grid
    grid = np.asarray(obs.board).reshape(config.rows, config.columns)
    # Use the heuristic to assign a score to each possible board in the next step
    scores = dict(zip(valid_moves, [score_move(grid, col, obs.mark, config, N_STEPS) for
col in valid_moves]))
    # Get a list of columns (moves) that maximize the heuristic
    max_cols = [key for key in scores.keys() if scores[key] == max(scores.values())]
    # Select at random from the maximizing columns
    return random.choice(max_cols)
```

In the next code cell, we see the outcome of one game round against a random agent.

In [ ]:

```
from kaggle_environments import make, evaluate

# Create the game environment
env = make("connectx")

# Two random agents play one game round
env.run([agent, "random"])

# Show the game
env.render(mode="ipython")
```

And we check how we can expect it to perform on average.

In [ ]:

```
def get_win_percentages(agent1, agent2, n_rounds=100):
    # Use default Connect Four setup
    config = {'rows': 6, 'columns': 7, 'inarow': 4}
    # Agent 1 goes first (roughly) half the time
    outcomes = evaluate("connectx", [agent1, agent2], config, [], n_rounds//2)
    # Agent 2 goes first (roughly) half the time
    outcomes += [[b,a] for [a,b] in evaluate("connectx", [agent2, agent1], config, [], n
_rounds-n_rounds//2)]
    print("Agent 1 Win Percentage:", np.round(outcomes.count([1,-1])/len(outcomes), 2))
    print("Agent 2 Win Percentage:", np.round(outcomes.count([-1,1])/len(outcomes), 2))
    print("Number of Invalid Plays by Agent 1:", outcomes.count([None, 0]))
    print("Number of Invalid Plays by Agent 2:", outcomes.count([0, None]))
```

In [ ]:

```
get_win_percentages(agent1=agent, agent2="random", n_rounds=50)
```

Not bad!

Your turn

Continue to check your understanding and submit your own agent to the competition.

Have questions or comments? Visit the Learn Discussion forum to chat with other Learners.