Introduction

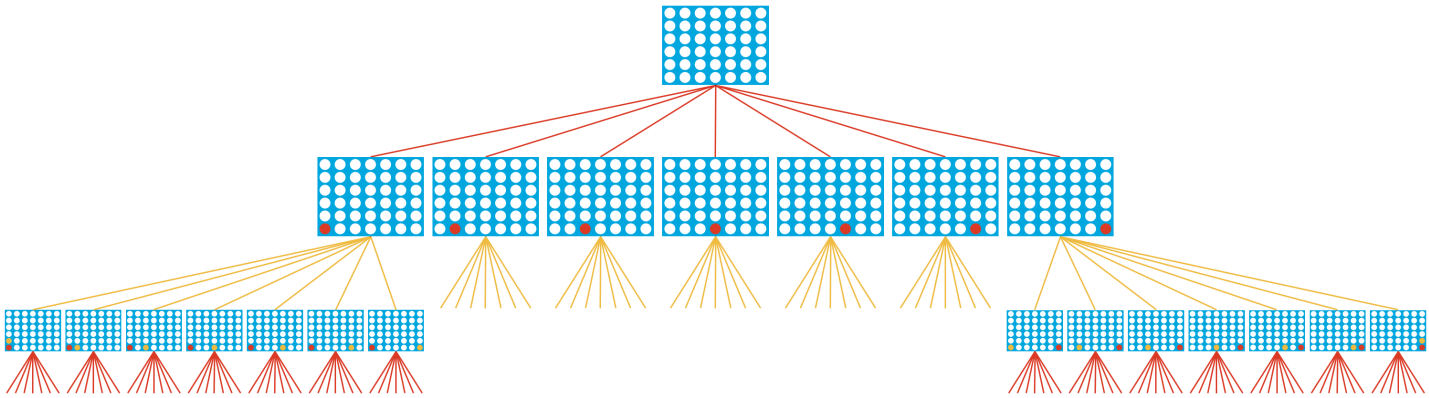
Even if you're new to Connect Four, you've likely developed several game-playing strategies. In this tutorial, you'll learn to use a heuristic to share your knowledge with the agent.

Game trees

As a human player, how do you think about how to play the game? How do you weigh alternative moves?

You likely do a bit of forecasting. For each potential move, you predict what your opponent is likely to do in response, along with how you'd then respond, and what the opponent is likely to do then, and so on. Then, you choose the move that you think is most likely to result in a win.

We can formalize this idea and represent all possible outcomes in a (complete) game tree.



The game tree represents each possible move (by agent and opponent), starting with an empty board. The first row shows all possible moves the agent (red player) can make. Next, we record each move the opponent (yellow player) can make in response, and so on, until each branch reaches the end of the game. (*The game tree for Connect Four is quite large, so we show only a small preview in the image above.*)

Once we can see every way the game can possibly end, it can help us to pick the move where we are most likely to win.

Heuristics

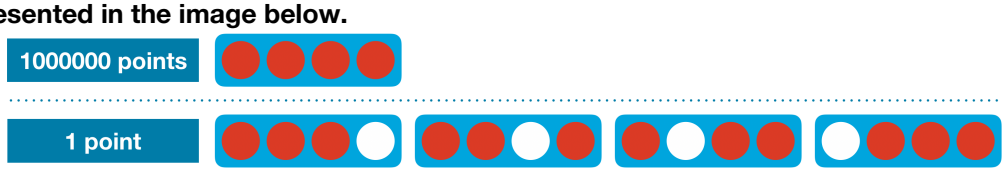
The complete game tree for Connect Four has over 4 trillion different boards! So in practice, our agent only works with a small subset when planning a move.

To make sure the incomplete tree is still useful to the agent, we will use a heuristic (or heuristic function). The heuristic assigns scores to different game boards, where we estimate that boards with higher scores are more likely to result in the agent winning the game. You will design the heuristic based on your knowledge of the game.

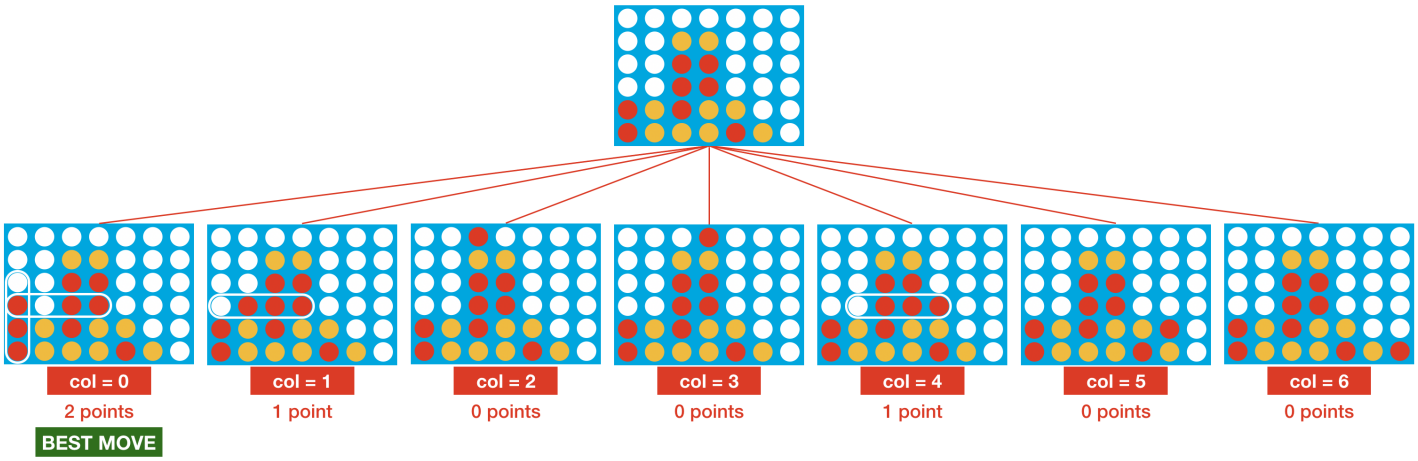
For instance, one heuristic that might work reasonably well for Connect Four looks at each group of four adjacent locations in a (horizontal, vertical, or diagonal) line and assigns:

- **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), and**
- **-100 points if the opponent filled three spots, and the remaining spot is empty (the opponent wins by filling in the empty spot).**

This is also represented in the image below.



And how exactly will the agent use the heuristic? Consider it's the agent's turn, and it's trying to plan a move for the game board shown at the top of the figure below. There are seven possible moves (one for each column). For each move, we record the resulting game board.



Then we use the heuristic to assign a score to each board. To do this, we search the grid and look for all occurrences of the pattern in the heuristic, similar to a word search puzzle. Each occurrence modifies the score. For instance,

- **The first board (where the agent plays in column 0) gets a score of 2. This is because the board contains two distinct patterns that each add one point to the score (where both are circled in the image above).**
- **The second board is assigned a score of 1.**
- **The third board (where the agent plays in column 2) gets a score of 0. This is because none of the patterns from the heuristic appear in the board.**

The first board receives the highest score, and so the agent will select this move. It's also the best outcome for the agent, since it has a guaranteed win in just one more move. Check this in figure now, to make sure it makes sense to you!

The heuristic works really well for this specific example, since it matches the best move with the highest score. It is just one of many heuristics that works reasonably well for creating a Connect Four agent, and you may find that you can design a heuristic that works much better!

In general, if you're not sure how to design your heuristic (i.e., how to score different game states, or which scores to assign to different conditions), often the best thing to do is to simply take an initial guess and then play against your agent. This will let you identify specific cases when your agent makes bad moves, which you can then fix by modifying the heuristic.

Code

Our one-step lookahead agent will:

- **use the heuristic to assign a score to each possible valid move, and**
- **select the move that gets the highest score. (*If multiple moves get the high score, we select one at random.*)**

"One-step lookahead" refers to the fact that the agent looks only one step (or move) into the future, instead of deeper in the game tree.

To define this agent, we will use the functions in the code cell below. These functions will make more sense when we use them to specify the agent.

In [ ]:

```
import random
import numpy as np
```

In [ ]:

In [ ]:

```
# Calculates score if agent drops piece in selected column
def score_move(grid, col, mark, config):
    next_grid = drop_piece(grid, col, mark, config)
    score = get_heuristic(next_grid, mark, config)
    return score

# Helper function for score_move: gets board at next step if agent drops piece in selecte
d 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 score_move: 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)
    score = num_threes - 1e2*num_threes_opp + 1e6*num_fours
    return score

# 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
```

The one-step lookahead agent is defined in the next code cell.

In [ ]:

```
# The agent is always implemented as a Python function that accepts two arguments: obs an
d config
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 turn
    scores = dict(zip(valid_moves, [score_move(grid, col, obs.mark, config) for col in v
alid_moves]))
    # permet de créer un dictinnaire dans lequel chaque clé correspond à un move valid et
chaque move valid est associé à son score
    # Exemple:
        # a = ("John", "Charles", "Mike")
        # b = ("Jenny", "Christy", "Monica")
        # x = zip(a, b)
        # result : (('John', 'Jenny'), ('Charles', 'Christy'), ('Mike', 'Monica'))
        # dict(x)
        # result : {'John': 'Jenny', 'Charles': 'Christy', 'Mike': 'Monica'}
    # 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 code for the agent, we begin by getting a list of valid moves. *This is the same line of code we used in the previous tutorial!*

Next, we convert the game board to a 2D numpy array. For Connect Four, `grid` is an array with 6 rows and 7 columns.

Then, the `score_move()` function calculates the value of the heuristic for each valid move. It uses a couple of helper functions:

- **`drop_piece()` returns the grid that results when the player drops its disc in the selected column.**
- **`get_heuristic()` calculates the value of the heuristic for the supplied board (`grid`), where `mark` is the mark of the agent. This function uses the `count_windows()` function, which counts the number of windows (of four adjacent locations in a row, column, or diagonal) that satisfy specific conditions from the heuristic. Specifically, `count_windows(grid, num_discs, piece, config)` yields the number of windows in the game board (`grid`) that contain `num_discs` pieces from the player (agent or opponent) with mark `piece`, and where the remaining locations in the window are empty. For instance,**
  - **setting `num_discs=4` and `piece=obs.mark` counts the number of times the agent got four discs in a row.**
  - **setting `num_discs=3` and `piece=obs.mark%2+1` counts the number of windows where the opponent has three discs, and the remaining location is empty (the opponent wins by filling in the empty spot).**

Finally, we get the list of columns that maximize the heuristic and select one (uniformly) at random.

(Note: For this course, we decided to provide relatively slower code that was easier to follow. After you've taken the time to understand the code above, can you see how to re-write it, to make it run much faster? As a hint, note that the `count_windows()` function is used several times to loop over the locations in the game board.)

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")
```

We use the `get_win_percentage()` function from the previous tutorial to 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")
```

This agent performs much better than the random agent!

Your turn

Continue to the exercise to improve the heuristic.

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