CS471: Introduction to Artificial Intelligence

Assignment 2: Hill-climbing (10 points)

Shane Dyrdahl

https://colab.research.google.com/drive/1MvU3_Qda3wRY3z2Sr0fJzXi-ocagMjCh?usp=sharing

Hill-Climbing:

I chose to store the formulas as functions that could be called:

```
def f(x1):
    return 2 - (x1 ** 2)

def g(x2):
    return (0.0051 * x2 ** 5) - (0.1367 * x2 ** 4) + (1.24 * x2 ** 3) - (4.456 * x2 ** 2) + (5.66 * x2) - 0.287
```

The hill-climbing function, *climb()*, takes in the formula, step-size, lower and upper bounds, and finally it passes the variable tolerance=1e-10. This can be used to help with floating point errors.

The function itself first handles what direction the climbing should go.

```
# This function performs the hill-climbing algorithm
def climb(formula, stepSize, start, lowerB, upperB, tolerance=1e-10):
    current = start
    if start == lowerB:
        direction = 'right'
    elif start == upperB:
        direction = 'left'
    else:
        direction = check_neighbors(formula, (start - stepSize), start, (start + stepSize))
```

If the start is at either the upper or lower bounds, it will go the opposite direction. If not at the bounds, it calls the function *check_neighbors()* which checks the neighbors to the left and right of the starting position.

```
def check_neighbors(equation, l, m, r):
    left = equation(l)
    middle = equation(m)
    right = equation(r)

    # Choose direction to climb
    if left == right and middle < left:  # If both left & right are equal & higher than middle, climb random direction
        climb_start = random.choice(["right", "left"])
    elif left > middle:  # Climb Left
        climb_start = "left"
    elif middle < right:  # Climb Right
        climb_start = "right"
    else:
        climb_start = "middle"  # If no direction is clear, stay in the middle

    return m, climb_start
```

The *check_neighbors* function returns the direction to climb.
Back in the climb function, I set the starting positions y-value to the max value encountered. Then depending on the direction, I begin the climb. I continue climbing until I either get to the end of the bound, or I encounter a local maximum.

```
# This function performs the hill-climbing algorithm
def climb(formula, stepSize, start, lowerB, upperB, tolerance=1e-10):
    current = start     # Set the starting x position
    if start == lowerB:
        direction = 'right'
    elif start == upperB:
        direction = 'left'
    else:
        # Evaluate the neighbors to determine the climbing direction
        direction = check_neighbors(formula, (start - stepSize), start, (start + stepSize))

    upperBound = upperB
    climb_max_y = formula(current)   # Calculate the initial y value at the start position
    match direction:
        case 'right':
            upperBound = upperB             # Set upper bound for climbing right
        case 'left':
            upperBound = lowerB             # Set lower bound for climbing left
            stepSize = -stepSize            # Reverse step size for left direction
        case 'middle':
            return climb_max_y, current     # If staying in the middle, return the current maximum

    while (current <= upperBound) if direction == 'right' else (current >= upperBound):
        y = formula(current)  # Calculate y value at the current x position

        # # To handle floating-point precision errors
        # if abs(current) < tolerance:
        #     current = 0.0

        # If a local maximum is found, return the previous max
        if climb_max_y > y:
            return climb_max_y, current - stepSize

        current += stepSize     # Update the current x position
        climb_max_y = y         # Update the maximum y value

    return climb_max_y, current  # Return the found maximum y value and its corresponding x position
```

I have the option to handle floating-point precision error but decided to comment it out.

In the end, the climb function returns both the maximum and the x position that it was at.

Random-Restart Hill-Climbing:

In the random-restart algorithm, I make sure to keep track of the randomly generated positions so that all generated positions are unique.

```
# Random-restart hill-climbing algorithm that tries a specified number of restarts to find the global maximum
def random_restart(restarts, formula, step_size, lowerBound, upperBound):
    randomNums = []         # List to store unique randomly generated starting positions
    global_max = 0.0        # Store the maximum y-value found so far
    global_max_xpos = 0.0   # Store the x-position corresponding to the global maximum
```

I then just iterate for the number of restarts required. I generate a unique x position to pass to the *climb()* function. I perform the hill-climbing algorithm, and store the maximum and x position that it gets.

```
for n in range(1, restarts + 1):
    while True:  # Loop until a unique x_pos is generated
        restart_x_pos = round_nearest(random.uniform(lowerBound, upperBound), step_size)
        if restart_x_pos not in randomNums:  # Check for uniqueness
            randomNums.append(restart_x_pos)  # Store the unique x_pos
            # Perform hill-climbing from the randomly generated x_pos
            restart_max_y, restart_x_pos = climb(formula, step_size, restart_x_pos, lowerBound, upperBound)
```

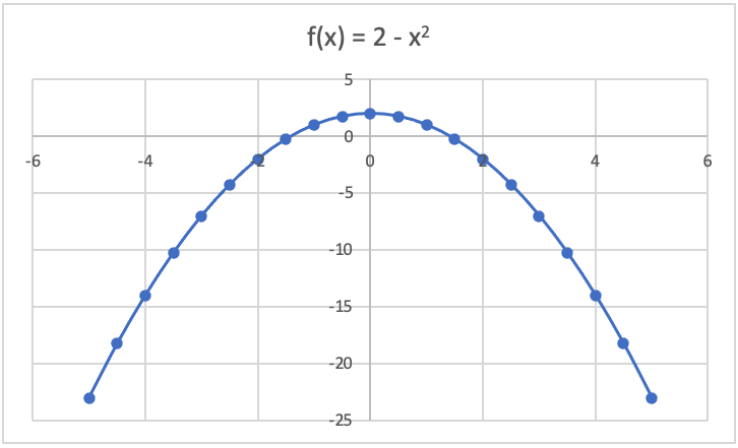
When I am generating a new unique x position, I round this number to match the step-size that was stated with a custom function *round_nearest()*.

```
# This function rounds a given number to the nearest step size (1, 0.5, or 0.01)
def round_nearest(num, step_size):
    match step_size:
        case 1:
            return round(num)  # Round to nearest whole number
        case 0.5:
            return round(num * 2) / 2  # Round to nearest 0.5
        case 0.01:
            return round(num * 100) / 100  # Round to nearest 0.01
```

I then update global max if the current maximum found is greater

```
            # Update global max
            if restart_max_y > global_max:
                global_max = restart_max_y
                global_max_xpos = restart_x_pos
            break  # Break the loop once a unique position is found and hill-climbing is performed
return global_max, global_max_xpos  # Return the global maximum value and its corresponding x position
```

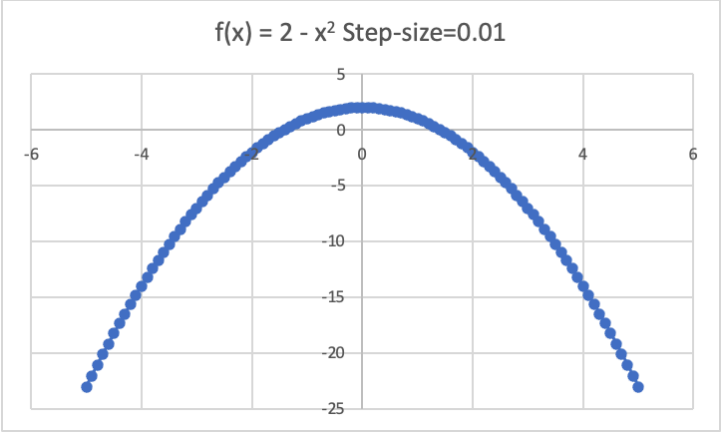
1. **Hill-climbing**
    a. Consider a function $f(x) = 2 - x^2$ in the following discrete state-space, where $x \in [-5, 5]$, step-size 0.5. Implement the hill-climbing algorithm in python to find the maximum value for the above function.



Output:

```
Question 1:
  a) With step-size of 0.5:
       Global Max: 2.0 at x = 0.0
```

    b. Change the step-size to 0.01. Run the hill-climbing algorithm and share your observations.



Output:

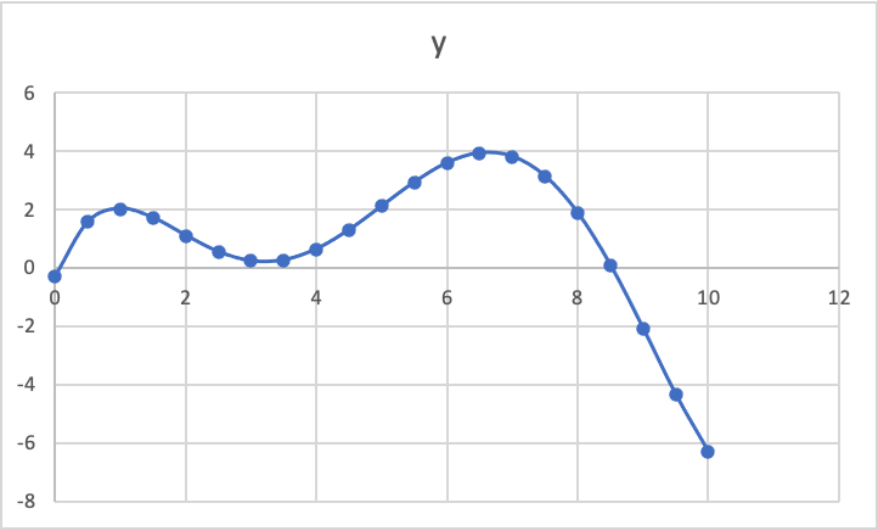
```
  b) With step-size of 0.01:
       Global Max: 2.0 at x = 1.6410484082740595e-15
```

I noticed that because the step-size is quite small, it results in floating-point precision errors because we are using Python. As we get closer to the value of 0, it gets really close but does not get exactly to zero. We can resolve this with a tolerance check to basically set it to zero if we get a very small number below our set tolerance.

Output that accounts for floating-point precision:

```
b) With step-size of 0.01:
     Global Max: 2.0 at x = 0.0
```

2. **Random-restart hill-climbing**
    a. Consider a function
    $g(x) = (0.0051x^5) - (0.1367x^4) + (1.24x^3) - (4.456x^2) + (5.66x) - 0.287$ in the following discrete state-space, where $x \in [0, 10]$, step-size 0.5. Implement the random-restart hill-climbing algorithm for 20 random restarts in python to find the global maximum value for the above function.



Output:

```
Question 2:
  a) Random-Restart:
       Global Max: 3.9287781250000213 at x = 6.5
```

    b. Run the hill-climbing algorithm for the function g(x). Compare and analyze the results of hill-climbing with the random-restart hill-climbing algorithm.

Output:

```
b) Hill-Climb:
     Global Max: 2.0254 at x = 1.0
```

When doing the random-restart hill-climb, with a step-size of 0.5, we are able to get very close to the true global maximum that is approximately 3.94614 at x = 6.63416. However when we do just the regular hill-climbing, depending on the starting point, we might get stuck at a local maximum. When I start at x = 0, I get stuck at the local maximum at x = 1; failing to get the global maximum. The key difference between the two is that random-restart can avoid local maxima by restarting, making it more reliable for functions with multiple peaks.

Grading rubric

Implementation of hill-climbing algorithm (1a): 3 points
Observations with respect to the difference in the step-size (1b): 2 points

Implementation of random-restart hill climbing (2a): 2 points
Comparison of hill-climbing vs random-restart hill-climbing (2b): 2 points
Proper commenting and clear formatting of code: 1 point

Total score = 10 points