Reyhane Shahrokhian 99521361

HomeWork6 of Computational Intelligence Course

Dr. Mozayeni

Q1:

1-1:

Creating a tool for web page design using genetic programming algorithms involves several steps, including user input, genetic programming, and defining a fitness function.

The genetic programming algorithm will iteratively generate and refine candidate solutions until it converges towards a web page design that maximizes the fitness score based on the defined criteria.

- User Input:

  Gather information from the user regarding the desired features, layout, color scheme, and any other design preferences and convert them into a format that can be used as input for the genetic programming algorithm.

- Genetic Programming (GP) Algorithm:

  Initialize a population of candidate solutions (represented as trees in the context of genetic programming).

  Define genetic operations such as mutation and crossover for evolving the population.

  Evaluate the fitness of each candidate solution using the fitness function.

  Select individuals from the current population based on their fitness to form the next generation.

- Fitness Function:

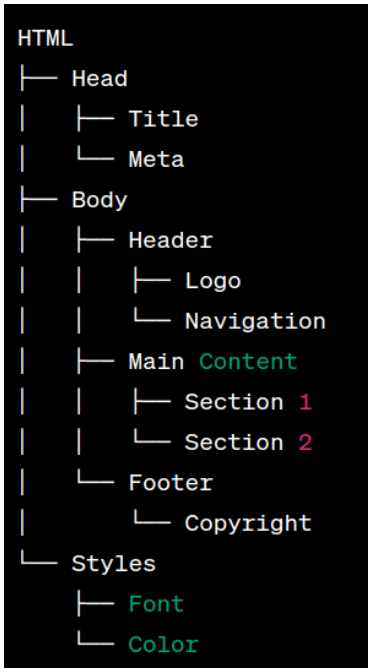
  The fitness function evaluates how well a candidate solution meets the user's preferences and design criteria.

- Evolutionary Process:

  Repeat the genetic programming cycle for several generations, allowing the population to evolve. Over time, the algorithm should converge towards solutions that better match the user's preferences.

Tree Representation:

Represent each candidate solution as a tree structure. Nodes in the tree represent different design elements, layout decisions, or style choices. Each node may have attributes like color, size, position, etc.



1-2:

An effective fitness function involves considering the specific goals and preferences of the user. In this case, let's design a fitness function that emphasizes aesthetics, usability, and adherence to user preferences. The goal is to encourage the algorithm to generate web page designs that are visually appealing, user-friendly, and aligned with the specified criteria.

Fitness Function Design Principles:

- Color Harmony:

  Assign higher scores to designs that use the specified color scheme effectively.

  Encourage a balanced and aesthetically pleasing color distribution.

- Layout Hierarchy:

  Reward designs that exhibit a clear hierarchy, such as Header, Main Content, and Footer.

  Encourage a visually intuitive flow that enhances user experience.

- Typography:

  Consider the readability and appropriateness of font choices. Reward designs with legible and well-chosen typography and penalize designs with poor font choices or inconsistent typography.

- Spacing and Alignment:

  Encourage consistent spacing and alignment throughout the design. Reward designs with well-proportioned and aligned elements and penalize designs with irregular spacing or misalignment.

- Element Proportions:

  Consider the proportions of different elements on the page. Encourage a balanced distribution of content and penalize designs with disproportionately sized elements.

- User Interaction:

  Consider user engagement factors, such as the placement and visibility of interactive elements (buttons, links). Reward designs that promote easy navigation and interaction and penalize designs with confusing or obstructed interactive elements.

- Adherence to User Preferences:

  Introduce weights for each criterion based on the importance of user preferences. Assign higher weights to color harmony, layout hierarchy, and other critical aspects specified by the user.

Scoring Mechanism:

- Each design is assigned a score based on the fitness function.
- The overall score is a weighted sum of scores from individual criteria.
- The weights are determined based on the relative importance of each criterion.
- Designs that better align with user preferences and exhibit better design principles receive higher scores.

The algorithm will aim to maximize this overall score by evolving candidate designs through genetic programming operations over multiple generations. This fitness function provides a comprehensive evaluation, promoting designs that not only meet user preferences but also adhere to good design principles for optimal web page aesthetics and usability.

1-3:

The fitness scores for each step based on the absence of styling, presence of elements, and functionality:

Step1:

Root

└── Container

  └── Form

    ├── Label (for "name")

    └── Input (type "text", id "name", name "name", placeholder "Enter your name")

  ➔ Presence Reward: Low (Basic structure but missing last name input)

  ➔ Styling Reward: None (No styling details)

  ➔ Functionality Reward: Low (No interactive elements)

  ➔ Overall Fitness Score: Low

Step2:

Root

└── Container

  └── Form

    ├── Label (for "name")

    ├── Input (type "text", id "name", name "name", placeholder "Enter your name")

    ├── Label (for "lastName")

    └── Input (type "text", id "lastName", name "lastName", placeholder "Enter your last name")

  ➔ Presence Reward: Moderate (Added last name input)

  ➔ Styling Reward: None (No styling details)

  ➔ Functionality Reward: Low (No interactive elements)

  ➔ Overall Fitness Score: Moderate

Step3:

Root

└── Container

    └── Form

        ├── Label (for "name", text "Your Full Name")

        ├── Input (type "text", id "name", name "name", placeholder "Enter your full name")

        ├── Label (for "lastName", text "Your Last Name")

        └── Input (type "text", id "lastName", name "lastName", placeholder "Enter your last name")

- ➔ Presence Reward: High (Complete structure)
- ➔ Styling Reward: None (No styling details)
- ➔ Functionality Reward: Low (No interactive elements)
- ➔ Overall Fitness Score: High

As the last two trees has the best fitness score so they have been chosen for other repeatations. with crossover and then mutation over them.

At the end it converges to a HTML like this:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Signup Form</title>
</head>
<body>
    <form>
        <label for="name">Your Full Name:</label>
        <input type="text" id="name" name="name" placeholder="Enter your full name">

        <label for="lastName">Your Last Name:</label>
        <input type="text" id="lastName" name="lastName" placeholder="Enter your last name">

        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

Q2:

Define Fitness Function: The fitness function measures how well a candidate solution satisfies the equation. It returns a fitness score, where higher values indicate better solutions.

```
def fitness(candidate, equation_func):
    return 1 / (1 + np.abs(equation_func(candidate)))
```

Initialize Population: The population of potential solutions (candidates) is randomly generated within a specified search range.

```
def initialize_population(population_size, search_range):
    return np.random.uniform(search_range[0], search_range[1], population_size)
```

Select Parents: Parents are selected from the population based on their fitness scores. The selection is done using a probability distribution that favors solutions with higher fitness.

```
def select_parents(population, fitness_scores):
    probabilities = fitness_scores / np.sum(fitness_scores)
    parents_indices = np.random.choice(len(population), size=2, p=probabilities)
    return population[parents_indices]
```

Crossover: The selected parents are combined to create new solutions (children) using an arithmetic crossover. The crossover introduces genetic diversity.

```
def arithmetic_crossover(parents):
    alpha = np.random.rand()  # Weighting factor
    return alpha * parents[0] + (1 - alpha) * parents[1]
```

Mutation: Mutation introduces small random changes to the offspring to explore new areas of the solution space.

```
def mutate(child, mutation_rate, mutation_range):
    if np.random.rand() < mutation_rate:
        return child + np.random.uniform(mutation_range[0], mutation_range[1])
    else:
        return child
```

Genetic Algorithm Main Loop: The genetic algorithm iteratively applies the selection, crossover, and mutation operations for a specified number of generations.

```python
def genetic_algorithm(equation, population_size=100, generations=1000, search_range=(-10, 10), mutation_rate=0.1, mutation_range=(-0.1, 0.1)):
    population = initialize_population(population_size, search_range)

    for generation in range(generations):
        fitness_scores = np.array([fitness(candidate, equation["equation"]) for candidate in population])

        parents = [select_parents(population, fitness_scores) for _ in range(population_size // 2)]
        children = [arithmetic_crossover(parents) for parents in parents]
        population = np.array([mutate(child, mutation_rate, mutation_range) for child in children])

    best_solution = population[np.argmax(fitness_scores)]
    return best_solution
```

Solving Equations: Finally, the genetic algorithm is applied to each equation, and the best solution (root) is printed.

```python
# Solve equations
for equation_info in equations:
    root = genetic_algorithm(equation_info, generations=1000)
    print(f"{equation_info['name']}, Root: {root}")
```

```
Equation 1, Root: 1.9640084359206944
Equation 2, Root: 0.50941567443927
Equation 3, Root: 1.2134190506258007
Equation 4, Root: 0.359253675232561
```

Q3:

- Chromosome Structure:

  A possible representation of a solution (chromosome) is a permutation of the numbers 1 to 36. Each number appears exactly once in the permutation, representing the order in which the numbers will be placed in the 6x6 square.

- Initial Population:

  Generate an initial population of possible solutions (chromosomes). Each chromosome represents a unique arrangement of numbers in the 6x6 square.

- Fitness Function:

The fitness function evaluates how well a solution satisfies the given constraints. In this case, the fitness function should consider the number of odd and even numbers in each row and column, aiming for an equal distribution.

```python
def fitness(chromosome):
    # chromosome is a permutation of numbers from 1 to 36

    square = [[0] * 6 for _ in range(6)]
    for i in range(6):
        for j in range(6):
            square[i][j] = chromosome[i * 6 + j]

    # Calculate fitness based on the number of odd and even numbers in each row and column
    fitness_value = 0
    for i in range(6):
        row_count = sum(1 for num in square[i] if num % 2 == 0)
        col_count = sum(1 for row in square if row[i] % 2 == 0)
        fitness_value += abs(row_count - 3) + abs(col_count - 3)

    return fitness_value
```

- Selection:

  Select individuals (chromosomes) for the next generation based on their fitness. You can use various selection methods such as roulette wheel selection, tournament selection, or others.

- Crossover:

  Create new individuals (offspring) by combining the genetic material of two parents. In this context, you could perform a crossover operation that swaps a portion of the genetic material between two parent chromosomes.

- Mutation:

  Introduce small random changes to individuals to explore the search space. For example, you can randomly swap two numbers in a chromosome.

- Termination:

Define a stopping condition, such as reaching a certain number of generations or achieving a satisfactory fitness level.

- Genetic Algorithm Parameters:

  Population size: e.g. 100

  Crossover rate: e.g. 0.7 (70%)

  Mutation rate: e.g. 0.01 (1%)

  Number of generations: e.g. 10

Q4:

According to my student number, I have to recognize the second version.

PSO might not be the most common choice for image classification tasks, as it's more commonly used for optimization problems. Standard classification algorithms like neural networks, decision trees, or support vector machines are often more suitable for image classification. PSO is typically used for optimization problems, but it can be adapted for classification tasks by defining an appropriate fitness function.

- Define the PSO Parameters:
  - ➔ Number of Particles (Swarm Size): This is the number of potential solutions in your search space. You can start with, say, 20 particles.
  - ➔ Maximum Number of Iterations: This is the number of iterations the PSO algorithm will run. You can set this based on how many iterations you think are reasonable for convergence. Start with, say, 100 iterations.
  - ➔ Inertia Weight (w): This parameter determines the trade-off between exploration and exploitation. A typical value is 0.5.

- ➔ Cognitive Coefficient (c1): This influences the particle's movement based on its personal best. Start with a value like 2.
- ➔ Social Coefficient (c2): This influences the particle's movement based on the global best. Start with a value like 2.
- ➔ Particle Position and Velocity Initialization: Initialize the position and velocity of each particle randomly.

- Representation of Solutions:

  Each particle represents a picture with 16 pixels, so the particle's position should be a vector of 16 elements.

- Fitness Function:

  Your fitness function should evaluate how close the given picture is to the target pattern (0 1 1 0, 0 1 1 0, 0 1 1 0, 0 1 1 0). A possible fitness function could be the sum of squared differences between each corresponding pixel value in the given picture and the target pattern.

```python
def fitness_function(particle_position):
    class2_pattern = [
        [0, 1, 1, 0],
        [0, 1, 1, 0],
        [0, 1, 1, 0],
        [0, 1, 1, 0]
    ]

    # Calculate sum of squared differences
    fitness = sum((particle_position[i][j] - class2_pattern[i][j]) ** 2 for i in range(4) for j in range(4))

    return fitness
```

- Initialization:

  Initialize the particles randomly with 16 pixel values.

- PSO Loop:

Use the PSO algorithm to update particle positions and velocities, evaluate fitness, and find the global best solution.

- Classification:

  After the PSO algorithm converges or reaches the maximum number of iterations, use the global best position (the best particle) as the solution. You can classify the solution based on some criteria, such as thresholding the fitness value or comparing it to a predefined threshold.