Genetic Algorithm For Image Puzzle

Introduction

In this project, a genetic algorithm is implemented to solve a tile-based image puzzle. The algorithm starts with a shuffled version of the original image, divided into smaller tiles, and evolves the tile arrangement over successive generations to reconstruct the original image. Each individual in the population represents a potential solution, and the algorithm optimizes the arrangement through selection, crossover, and mutation.

The primary goal is to achieve a tile arrangement that perfectly matches the original image, resulting in a fitness score of 1.0. The algorithm is designed to:

1. Load and preprocess the image to ensure consistency.
2. Initialize a population of shuffled tile arrangements.
3. Iteratively improve the population through evolutionary operations while tracking the best fitness scores.
4. Reconstruct and save the final solution and a plot of fitness evolution for visualization and analysis.

The following report provides a detailed breakdown of the individual functions, their roles, and how they contribute to the overall workflow of the genetic algorithm. It also highlights the logic behind key operations such as fitness evaluation, crossover, mutation, and next-generation creation. Finally, the results include the reconstructed image and the fitness evolution graph, showcasing the algorithm's efficiency and convergence over time.

By leveraging the principles of genetic algorithms, this solution demonstrates an efficient approach to solving combinatorial problems such as puzzle reconstruction, which can be extended to other domains like optimization, scheduling, and machine learning.

1. load_image

The `load_image` function is designed as the initial step in the image-processing workflow of the genetic algorithm. It prepares the input image by:

1. **Loading**: Fetching the image from a user-specified path.
2. **Converting**: Simplifying the image to grayscale mode to eliminate color-related complexity.
3. **Resizing**: Standardizing the dimensions of the image to a fixed size for consistent processing.
4. **Saving**: Storing the processed image for reference and further debugging.

This preprocessing is critical because the genetic algorithm works with image tiles to create and evaluate solutions. Ensuring all images have the same format, size, and characteristics allows the algorithm to function effectively and efficiently.

General Code Flow

1. The function accepts an **image path** as input to locate the image file, and an optional **size parameter** for resizing.
2. It opens the image and converts it to grayscale mode. Grayscale simplifies the problem by focusing solely on intensity values (brightness levels) instead of RGB color information.
3. The image is resized to the dimensions `(300, 300)` by default. This ensures that the image is computationally manageable while maintaining enough resolution to identify features.
4. Finally, the resized grayscale image is saved locally to a fixed location (`"C:/Users/User/Desktop/ML/resized_image.jpeg"`) and returned as an object.

2. split_image

The `split_image` function is a key step in the image-processing workflow of the genetic algorithm. After the `load_image` function prepares the input image, this function divides it into smaller rectangular sections (tiles), which serve as the "genes" for the algorithm. These tiles are saved for reference and further manipulation.

Purpose

The purpose of this function is to:

1. **Divide**: Break the input image into smaller, equally-sized rectangular tiles based on the specified matrix size.
2. **Save**: Store these tiles as individual images for debugging, analysis, or visualization.
3. **Prepare**: Return the tiles and their dimensions for use in subsequent steps of the genetic algorithm.

This function is essential because the genetic algorithm rearranges these tiles to reconstruct the original image as part of its evolutionary process.

General Code Flow

1. **Matrix Size**:
    - The `matrix_size` parameter determines how many rows and columns the image will be divided into. For example:
        - If `matrix_size=3`, the image is divided into a `3x3` grid, resulting in `9` tiles.
2. **Tile Dimensions**:
    - The width and height of each tile are calculated by dividing the image's total width and height by the `matrix_size`.
3. **Tile Creation**:
    - Using a nested list comprehension, the image is split into smaller tiles. Each tile is defined by cropping the original image using its coordinates:
        - `(left, upper, right, lower)`:
            - `left`: Starting x-coordinate.
            - `upper`: Starting y-coordinate.
            - `right`: Ending x-coordinate.

        - `lower`: Ending y-coordinate.
4. **Save Tiles**:
    - Each tile is saved as an individual image file in a specified directory (`"C:/Users/User/Desktop/ML"`). This helps with debugging and allows you to visualize each part of the image.
5. **Return Values**:
    - The function returns:
        - `tiles`: A list of cropped image objects.
        - `tile_width`: The width of each tile.
        - `tile_height`: The height of each tile.

3. calculate_fitness

The `calculate_fitness` function is a crucial component of the genetic algorithm. It quantifies how close a given arrangement of tiles (individual) is to the correct arrangement (original_tiles). This fitness value serves as the basis for evaluating and comparing solutions, guiding the genetic algorithm to evolve better solutions over generations.

Purpose

The purpose of this function is to:

1. **Evaluate**: Measure how well a given arrangement of tiles matches the original (correct) arrangement.
2. **Score**: Assign a fitness value between `0` and `1` to each individual, where:
    - `1.0` means the arrangement is perfect (all tiles are in the correct position).
    - A value closer to `0` means the arrangement is far from correct.
3. **Guide**: Provide a metric for selecting and prioritizing better individuals for the next generation.

General Code Flow

1. **Inputs**:
    - `individual`: A specific arrangement of tiles (list of tiles) being evaluated.
    - `original_tiles`: The correct arrangement of tiles (list of tiles in the right order).
2. **Tile-by-Tile Comparison**:
    - Using the `zip` function, the `individual` tiles are paired with the corresponding `original_tiles`.
    - For each pair of tiles (`ind_tile`, `orig_tile`), the function checks if the two tiles are the same:
        - If `ind_tile == orig_tile`, it means the tile is in the correct position.
3. **Count Correct Tiles**:
    - The function uses a generator expression to count how many tiles are in the correct position.
4. **Calculate Fitness**:

    - The fitness is computed as the ratio of correctly placed tiles to the total number of tiles:
        - Fitness=Correct TilesTotal Tiles\text{Fitness} = \frac{\text{Correct Tiles}}{\text{Total Tiles}}Fitness=Total TilesCorrect Tiles
5. **Return Fitness**:
    - The function returns this fitness value as a floating-point number between `0` and `1`.

4. uniform_crossover

The `uniform_crossover` function is a core component of the genetic algorithm, enabling the exchange of "genetic material" between two parent solutions. It simulates the biological crossover process by combining the "genes" of two parents to produce two new offspring solutions. This mechanism is essential for introducing variation in the population, which drives the algorithm toward better solutions over generations.

Purpose

The purpose of this function is to:

1. **Combine**: Mix the tiles (genes) from two parent individuals to create two new offspring individuals.
2. **Introduce Diversity**: Ensure that offspring inherit a mix of traits (tiles) from both parents, providing variation in the population.
3. **Preserve Genetic Material**: Avoid losing critical "good genes" by uniformly distributing genes from both parents.

General Code Flow

1. **Inputs**:
    - `parent1`: The first parent individual (a list of tiles representing an arrangement).
    - `parent2`: The second parent individual (another list of tiles of the same length as `parent1`).
2. **Iterate Over Genes**:
    - Using the `zip` function, the genes (tiles) from both parents are paired together and iterated.
3. **Random Gene Exchange**:
    - For each pair of genes (`gene1` from `parent1` and `gene2` from `parent2`):
        - A random number between `0` and `1` is generated using `random.random()`.
        - If the random number is less than `0.5`:
            - `gene1` is added to `offspring1`, and `gene2` is added to `offspring2`.
        - Otherwise:
            - `gene2` is added to `offspring1`, and `gene1` is added to `offspring2`.
4. **Return Offspring**:

- Two new offspring individuals are returned, each inheriting a randomized combination of genes from the two parents.

5. combine_image

The `combine_image` function is responsible for reconstructing a full image from an arrangement of tiles (an individual). This function allows the genetic algorithm to visualize the current solution or save the progress of the puzzle reconstruction.

Purpose

The purpose of this function is to:

1. **Reconstruct**: Assemble individual tiles back into a complete image based on their given order in an individual.
2. **Visualize**: Provide a visual representation of a specific solution in the genetic algorithm, which is useful for analysis and debugging.
3. **Save or Display**: Enable the reconstructed image to be saved or displayed as needed, helping users observe the progress of the algorithm.

General Code Flow

1. **Inputs**:
   - `individual`: A specific arrangement of tiles (list of tile objects) that will be assembled into a full image.
   - `tile_width`: The width of each tile.
   - `tile_height`: The height of each tile.
   - `image_size`: The total dimensions of the output image (e.g., `(300, 300)`).
2. **Create Blank Canvas**:
   - A new blank image is created using `Image.new('L', image_size)`, where:
     - `'L'`: Specifies grayscale mode.
     - `image_size`: The dimensions of the complete image.
3. **Calculate Tile Positions**:
   - Using a loop, each tile is placed in its corresponding position:
     - `x_offset`: Horizontal position based on the column index `(i % 3)`.
     - `y_offset`: Vertical position based on the row index `(i // 3)`.
4. **Paste Tiles**:
   - Each tile is pasted onto the blank canvas at the calculated `(x_offset, y_offset)` coordinates using `combined_image.paste(tile, (x_offset, y_offset))`.
5. **Return Assembled Image**:
   - The fully assembled image is returned as a `Pillow Image` object.

6. create_next_generation

The `create_next_generation` function is a critical component of the genetic algorithm. It creates a new population (next generation) of potential solutions by:

1. Retaining the best individuals from the current generation (elitism).
2. Producing new individuals through crossover and mutation operations.

This ensures the algorithm balances **exploitation** (using the best solutions) and **exploration** (generating new, diverse solutions).

Purpose

The purpose of this function is to:

1. **Carry Forward the Best**: Include the best-performing individuals (elite) directly in the next generation.
2. **Generate New Offspring**: Create new individuals by performing crossover and mutation on the elite individuals.
3. **Maintain Population Size**: Ensure the next generation has the same number of individuals as the current generation.

General Code Flow

1. **Inputs**:
    - `elite_individuals`: A list of the top-performing individuals from the current generation, along with their fitness scores.
    - `population_size`: The total number of individuals required in the next generation.
    - `mutation_rate`: The probability of applying mutations to individuals.
2. **Retain Elite Individuals**:
    - Extract only the "individuals" (not their fitness values) from `elite_individuals`.
    - Add these elite individuals directly to the `next_generation` list to preserve their superior traits.
3. **Crossover and Mutation**:
    - While the size of the `next_generation` is less than the required `population_size`:
        - Randomly select two elite individuals (`parent1` and `parent2`).
        - Perform `uniform_crossover` to create two offspring.
        - Apply `swap_mutation` to each offspring to introduce variation.
4. **Add Offspring to the Next Generation**:
    - Append `offspring1` and (if space remains) `offspring2` to the `next_generation`.
5. **Return Next Generation**:
    - Once the population reaches the required size, return the `next_generation`.

7. plot_fitness_evolution

The `plot_fitness_evolution` function visualizes the improvement of the best fitness score over generations. This function helps track the progress of the genetic algorithm, ensuring that the algorithm is evolving toward better solutions and converging to the correct arrangement of tiles.

Purpose

The purpose of this function is to:

1. **Visualize Progress**: Plot how the fitness value of the best individual changes over generations.
2. **Evaluate Algorithm Behavior**: Identify trends, such as stagnation, rapid improvement, or oscillations in fitness values.

8. genetic_algorithm

The `genetic_algorithm` function is the central part of the program that orchestrates all the steps of the genetic algorithm. It uses the other helper functions (like `load_image`, `split_image`, `calculate_fitness`, etc.) to evolve a population of solutions over multiple generations, with the goal of reconstructing the original image.

Purpose

The purpose of this function is to:

1. **Initialize**: Prepare the population, image tiles, and tracking variables.
2. **Iterate**: Evolve the population by repeatedly applying selection, crossover, and mutation until a perfect solution (fitness = 1.0) is found.
3. **Track Progress**: Record and save the fitness scores and improved solutions during the process.
4. **Conclude**: Return the final solution and plot the fitness evolution over generations.

Code Flow Explanation

1. **Inputs**:
    - **`image_path`**: File path of the input image.
    - **`matrix_size`**: Number of rows and columns for splitting the image (e.g., `3` for a `3x3` grid).
    - **`population_size`**: Number of individuals in each generation.
    - **`mutation_rate`**: Probability of mutation for each individual.
    - **`max_no_improvement`**: Maximum number of generations without improvement before reinitializing the population.
2. **Image Preprocessing**:

- The input image is loaded, converted to grayscale, resized, and split into tiles using the `load_image` and `split_image` functions.
- These tiles form the "genes" of the genetic algorithm.

3. **Initialize Population**:
   - A random initial population of individuals is created by shuffling the tiles.
4. **Initialize Tracking Variables**:
   - These variables track the progress of the algorithm:
     - `generation`: Current generation number.
     - `best_fitness`: Fitness value of the best individual so far.
     - `no_improvement_generations`: Number of generations without improvement.
     - `fitness_values`: List of best fitness scores for plotting.
     - `generations`: List of generation numbers for plotting.
5. **Iterative Evolution**:
   - **Fitness Evaluation**:
     Each individual's fitness is calculated using `calculate_fitness`, and the population is sorted by fitness.

   - **Track Best Fitness**:
     The best fitness in the current generation is recorded.

   - **Save Improved Solutions**:
     If the best fitness improves, the best individual's arrangement is reconstructed and saved.

   - **Handle Stagnation**:
     If no improvement occurs for `max_no_improvement` generations, the population is reinitialized to avoid stagnation.

   - **Create Next Generation**:
     The top two individuals are selected for elitism, and the rest of the population is generated using crossover and mutation.

   - **Termination**:
     The loop continues until a perfect solution (fitness = 1.0) is found.

   - **Save Final Solution**:
     The best solution is reconstructed and saved as the final result.

   - **Plot Fitness Evolution**:
     The fitness scores over generations are visualized using `plot_fitness_evolution`.

   - **Return**:
     Returns the paths to the saved final solution and the fitness evolution plot.