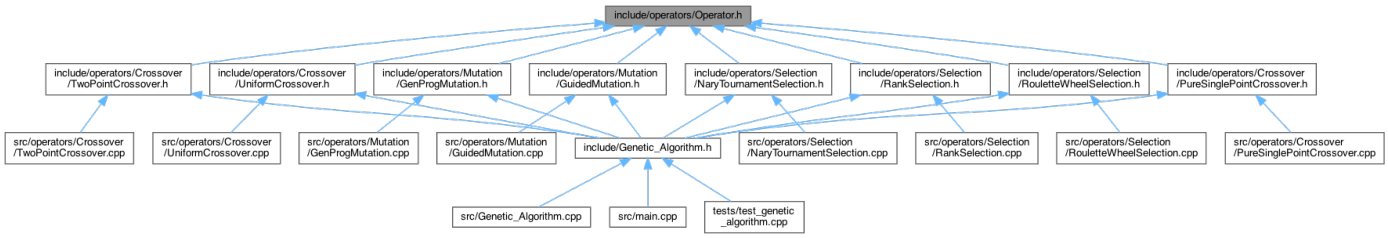
Genetic Algorithm Design

Overview

This project employs a genetic algorithm to optimize circuit configurations. Genetic algorithms are heuristic search algorithms inspired by the process of natural selection. They use operations such as selection, crossover, and mutation to find the optimal solution. This document provides a detailed explanation of the design structure and optimization strategies implemented in the genetic algorithm.

UML Graph

The following documentation provides an in-depth look at the design and implementation of the genetic algorithm components within our project. The structure of the project is depicted in the accompanying diagram, which illustrates the relationships between various header and implementation files.



Relationships

- The central file `Operator.h` is included by all specific operator header files (`Crossover`, `Mutation`, `Selection`), establishing a common base or interface for different types of genetic algorithm operators.
- Each operator header (`.h`) file is paired with its corresponding implementation (`.cpp`) file, where the declared methods and classes are defined.
- `Genetic_Algorithm.h` includes various operator headers to utilize the defined crossover, mutation, and selection strategies within the genetic algorithm.
- The main program (`main.cpp`) and test file (`test_genetic_algorithm.cpp`) include `Genetic_Algorithm.h` to execute and test the genetic algorithm functionalities.

The diagram effectively illustrates the modular structure of the genetic algorithm, showing clear separations between declarations and implementations, and the hierarchical inclusion of various components. This modularity facilitates easy maintenance, testing, and extension of the genetic algorithm with new operators or strategies. The project is organized to ensure that each component can be developed and tested independently, making it easier to manage and extend the genetic algorithm framework.

Directory Structure

The following is the directory structure relevant to the genetic algorithm:

```
├── include
│   ├── Genetic_Algorithm.h                    # Main genetic algorithm components and
functions.
│   ├── operators
```

```
│   │   ├── Crossover
│   │   │   ├── PureSinglePointCrossover.h    # Header for PureSinglePointCrossover
class.
│   │   │   ├── TwoPointCrossover.h           # Header for TwoPointCrossover class.
│   │   │   └── UniformCrossover.h            # Header for UniformCrossover class.
│   │   ├── Mutation
│   │   │   ├── GenProgMutation.h             # Header for GenProgMutation class.
│   │   │   └── GuidedMutation.h              # Header for GuidedMutation class.
│   │   ├── Selection
│   │       ├── NaryTournamentSelection.h     # Header for NaryTournamentSelection class.
│   │       ├── RankSelection.h               # Header for RankSelection class.
│   │       └── RouletteWheelSelection.h      # Header for RouletteWheelSelection class.
│   └── utils
│       ├── Helper.h                          # Utility functions for logging and current
time.
│       ├── Individual.h                      # Definition of the Individual class used in
the genetic algorithm.
│       └── Parameter.h                       # Definition of the parameter structures for
the algorithm.
└── src
    ├── Genetic_Algorithm.cpp                 # Implementation of the main genetic
algorithm components and functions.
    ├── operators
    │   ├── Crossover
    │   │   ├── PureSinglePointCrossover.cpp  # Implementation of
PureSinglePointCrossover class.
    │   │   ├── TwoPointCrossover.cpp         # Implementation of TwoPointCrossover
class.
    │   │   └── UniformCrossover.cpp          # Implementation of UniformCrossover class.
    │   ├── Mutation
    │   │   ├── GenProgMutation.cpp           # Implementation of GenProgMutation class.
    │   │   └── GuidedMutation.cpp            # Implementation of GuidedMutation class.
    │   └── Selection
    │       ├── NaryTournamentSelection.cpp   # Implementation of NaryTournamentSelection
class.
    │       ├── RankSelection.cpp             # Implementation of RankSelection class.
    │       └── RouletteWheelSelection.cpp    # Implementation of RouletteWheelSelection
class.
```

Major Components

1. Genetic_Algorithm.h and Genetic_Algorithm.cpp

These files define the core module of the genetic algorithm, including the main flow and operations of the algorithm, such as population initialization, parent selection, crossover, mutation, and generation of new populations.

Key Functions

- `initializeRandomSeed`: Initializes the random seed.

- `initializePopulation`: Initializes the population.
- `elitism`: Implements the elitism strategy, preserving the best individuals.
- `optimize`: The main optimization function, executing the main loop of the genetic algorithm.

2. Selection Operators

Selection operators are responsible for selecting individuals from the current population to serve as parents for the next generation.

Files

- `NaryTournamentSelection.h` / `NaryTournamentSelection.cpp`: n-ary tournament selection, randomly selects n individuals to compete, and selects the best one.
- `RankSelection.h` / `RankSelection.cpp`: Rank-based selection, individuals are sorted by fitness, and selection probability is based on rank.
- `RouletteWheelSelection.h` / `RouletteWheelSelection.cpp`: Roulette wheel selection, selection probability is proportional to fitness.

3. Crossover Operators

Crossover operators combine the genes of parent individuals to create new offspring.

Files

- `PureSinglePointCrossover.h` / `PureSinglePointCrossover.cpp`: Single-point crossover, swaps genes between two parents at a random point.
- `TwoPointCrossover.h` / `TwoPointCrossover.cpp`: Two-point crossover, swaps genes between two random points.
- `UniformCrossover.h` / `UniformCrossover.cpp`: Uniform crossover, randomly selects genes from each parent.

4. Mutation Operators

Mutation operators introduce random changes to the genes of an individual, increasing the diversity of the population.

Files

- `GenProgMutation.h` / `GenProgMutation.cpp`: General-purpose mutation, randomly modifies the genes of an individual.
- `GuidedMutation.h` / `GuidedMutation.cpp`: Guided mutation, modifies genes based on specific rules or heuristics.

5. Utilities

Auxiliary functions and structures definitions.

Files

- `Helper.h` / `Helper.cpp`: Contains auxiliary functions like logging results and getting the current time.
- `Individual.h`: Defines the individual structure, including the gene sequence and fitness.
- `Parameter.h`: Defines the structure for algorithm parameters, including population size, crossover rate, mutation rate, etc.

Optimization Strategies

1. Elitism Strategy

The elitism strategy in genetic algorithms ensures that the best-performing individuals from the current generation are preserved and carried over to the next generation. This helps in maintaining the quality of solutions over generations and prevents the loss of the best solutions found so far.

Implementation Details

The implementation of the elitism strategy is done in the `elitism` function in `Genetic_Algorithm.cpp`. This function sorts the population based on fitness and selects the top individuals to be directly transferred to the new population.

Code Explanation

`elitism`

The `elitism` function takes the current population and a reference to a new population vector where the elite individuals will be stored. It also takes the algorithm parameters which include the proportion of the population to be preserved as elites.

```
void elitism(const std::vector<Individual> &population, std::vector<Individual>
&newPopulation, const Algorithm_Parameters &params) {
    // Create a copy of the population to sort
    std::vector<Individual> sortedPopulation = population;

    // Sort the population based on fitness in descending order
    std::sort(sortedPopulation.begin(), sortedPopulation.end(), [](const Individual &a,
const Individual &b) {
        return a.fitness > b.fitness;
    });

    // Calculate the number of elite individuals to be preserved
    int numElites = static_cast<int>(params.populationSize * params.elitePercentage);

    // Add the top elite individuals to the new population
    for (int i = 0; i < numElites; i++) {
        // If the fitness of the individual is invalid, stop adding more elites
        if (sortedPopulation[i].fitness == INVALID_FITNESS) {
            break;
        }
        newPopulation.push_back(sortedPopulation[i]);
    }
}
```

Steps in the `elitism` Function

1. **Copy the Population**: A copy of the current population is created to avoid modifying the original population directly.

```
std::vector<Individual> sortedPopulation = population;
```

2. **Sort the Population**: The copied population is sorted based on the fitness values of the individuals in descending order. This means the individuals with the highest fitness values come first.

```
std::sort(sortedPopulation.begin(), sortedPopulation.end(), [](const Individual &a,
const Individual &b) {
    return a.fitness > b.fitness;
});
```

3. **Calculate the Number of Elites**: The number of elite individuals to be preserved is calculated based on the elite percentage parameter.

```
int numElites = static_cast<int>(params.populationSize * params.elitePercentage);
```

4. **Add Elite Individuals to the New Population**: The top elite individuals are added to the new population. If an individual's fitness is invalid, the process stops to avoid adding unfit individuals.

```
for (int i = 0; i < numElites; i++) {
    if (sortedPopulation[i].fitness == INVALID_FITNESS) {
        break;
    }
    newPopulation.push_back(sortedPopulation[i]);
}
```

Integration with the Main Optimization Loop

In the main optimization function (`optimize`), the `elitism` function is called to add the best individuals from the current population to the new population before proceeding with the selection, crossover, and mutation operations.

```
std::vector<Individual> newPopulation;
elitism(population, newPopulation, params);
```

By preserving the best individuals, the elitism strategy ensures that the genetic algorithm does not lose the highest quality solutions found in each generation, thereby maintaining or improving the overall quality of the population over time.

2. Selection Strategies

Selection strategies are critical in genetic algorithms as they determine how individuals are chosen to create

the next generation. The implemented selection strategies in this project include Nary Tournament Selection, Rank Selection, and Roulette Wheel Selection. Each strategy has its advantages in maintaining diversity and preventing premature convergence.

Nary Tournament Selection

Nary Tournament Selection is a method where 'n' individuals are randomly chosen from the population, and the one with the highest fitness among them is selected as a parent. This process helps in maintaining a healthy diversity in the population and prevents premature convergence to suboptimal solutions.

Rank Selection

Rank Selection assigns selection probabilities to individuals based on their rank rather than their raw fitness values. This ensures that even individuals with lower fitness have a chance of being selected, thereby maintaining diversity in the population.

Roulette Wheel Selection

Roulette Wheel Selection (also known as fitness proportionate selection) assigns selection probabilities proportional to the individuals' fitness values. Individuals with higher fitness have a higher chance of being selected, but there is still a probability for lower fitness individuals to be chosen, thus maintaining diversity.

3. Crossover Strategies

Crossover strategies play a vital role in genetic algorithms by combining the genetic information of parent individuals to create offspring for the next generation. This process helps in exploring the search space and finding better solutions.

PureSinglePointCrossover

PureSinglePointCrossover involves selecting a random crossover point and exchanging the subsequences after this point between two parent individuals. This strategy helps to combine different features from the parents, potentially leading to better offspring.

TwoPointCrossover

TwoPointCrossover is similar to the single-point crossover but involves selecting two random points. The segments between these two points are exchanged between the parents, leading to potentially more diverse offspring.

UniformCrossover

UniformCrossover randomly selects genes from each parent, independent of their positions, resulting in a higher level of genetic diversity in the offspring. This method can be particularly effective in preventing premature convergence.

4. Mutation Strategies

Mutation strategies introduce variability in the genetic algorithm population, helping to explore the search space more thoroughly and avoid local optima.

GenProgMutation

GenProgMutation introduces random changes to the genes of an individual. This randomness helps the algorithm to avoid being trapped in local optima by exploring new areas of the search space.

GuidedMutation

GuidedMutation uses heuristic information to guide the mutation process. This approach increases the effectiveness of the mutations by focusing on areas more likely to improve the individual's fitness.

5. Simulated Annealing

Simulated Annealing (SA) is a probabilistic technique used for approximating the global optimum of a given function. In the context of genetic algorithms, SA is utilized to accept or reject new individuals (offspring) based on a temperature-controlled probability. This helps in balancing the exploration and exploitation of the search space.

Simulated Annealing helps the genetic algorithm escape local optima by occasionally accepting worse solutions with a certain probability. This probability decreases as the algorithm progresses, controlled by a parameter called "temperature."

Key Components of Simulated Annealing in Genetic Algorithms

1. **Acceptance Probability**: Determines whether to accept a new solution based on the difference in fitness and the current temperature.
2. **Temperature Schedule**: Controls how the temperature decreases over time.
3. **Integration with Genetic Algorithm**: Used in conjunction with crossover and mutation to guide the search process.

Code Explanation

Acceptance Probability

The acceptance probability is calculated using the formula:

$$P = exp(\frac{\Delta E}{T}) \tag{1}$$

where:

- ΔE is the change in fitness (new fitness - current fitness).
- T is the current temperature.

```
// Genetic_Algorithm.cpp

// Calculate acceptance probability of Simulated Annealing functions
double acceptanceProbability(double currentEnergy, double newEnergy, double temperature) {
    if (newEnergy > currentEnergy) {
        return 1.0;
    }
    return std::exp((newEnergy - currentEnergy) / temperature);
}
```

If the new solution is better ($\Delta E>0$), the probability is greater than 1, and the new solution is accepted. If the new solution is worse ($\Delta E<0$), it might still be accepted based on the calculated probability, allowing the algorithm to escape local optima.

The temperature is gradually decreased according to a cooling schedule. This reduces the likelihood of accepting worse solutions as the algorithm progresses, focusing more on exploitation of the best solutions found so far. The cooling schedule can be a simple linear decrement or a more complex function, but in our implementation, a straightforward decrement is used:

$$T = T - \Delta T \tag{2}$$

where ΔT is a small decrement value.

Simulated Annealing Process

During each iteration of the genetic algorithm, after generating offspring through crossover and mutation, the `applySimulatedAnnealing` function is applied to decide whether to accept the offspring. This mechanism ensures that even suboptimal solutions can occasionally be accepted, promoting diversity and preventing premature convergence to suboptimal solutions.

```
// Genetic_Algorithm.cpp

bool applySimulatedAnnealing(Individual &offspring, Individual &parent1, Individual
&parent2, int vector_size,
                             double (&func)(int, int *, struct CircuitParameters), bool
(&validity)(int, int *),
                             double &Temp, const Algorithm_Parameters &params, const
CircuitParameters c_params,
                             std::mt19937 &generator) {
    // Calculate the fitness (energy) of the offspring
    if (validity(vector_size, offspring.genes.data())) {
        offspring.fitness = func(vector_size, offspring.genes.data(), c_params);
    } else {
        offspring.fitness = INVALID_FITNESS; // Give a negative fitness for invalid
individuals
        return false; // Reject invalid offspring
    }
    double parents_fitness = (parent1.fitness + parent2.fitness) / 2;

    // Simulated annealing acceptance criterion
    std::uniform_real_distribution<double> distribution(0.0, 1.0);
```

```
    // Check the threshold
    if (Temp > 1) {
        if (acceptanceProbability(parents_fitness, offspring.fitness, Temp) >
distribution(generator)) {
            return true; // Accept the offspring
        }
        // Cool down the temperature
        Temp -= params.deltT;
    } else {
        // Keep the original parents
        if (parent1.fitness > parent2.fitness) {
            offspring = parent1;
        } else {
            offspring = parent2;
        }
        return true;
    }
    return false; // Reject the offspring
}
```

Explanation of the Code

- **Fitness Calculation**: The fitness (energy) of the offspring is calculated using the provided fitness function `func`. If the offspring is valid, its fitness is evaluated; otherwise, it is assigned a negative fitness value, and the offspring is rejected.
- **Parents' Fitness**: The average fitness of the two parents is computed. This serves as the baseline for comparing the fitness of the offspring.
- **Acceptance Criterion**: A uniform random number generator determines whether the offspring is accepted based on the calculated acceptance probability.
- **Temperature Check and Cooling**: The temperature is checked to see if it is above a certain threshold. If so, the acceptance probability is compared with a random number to decide if the offspring should be accepted. The temperature is then decreased according to the cooling schedule. If the temperature is too low, the offspring is not accepted unless it is better than the worse parent.
- **Updating Offspring**: If the offspring is accepted, it becomes part of the population. If not, the better of the two parents is retained.

Integration with the Genetic Algorithm

In the main loop of the genetic algorithm, simulated annealing is applied after the crossover and mutation steps. This ensures that the offspring are accepted based on the temperature-controlled probability.

```
// Genetic_Algorithm.cpp

// Main function to optimize the solution using a genetic algorithm
int optimize(int vector_size, int *vector, double (&func)(int, int *, struct
CircuitParameters),
             bool (&validity)(int, int *), Algorithm_Parameters params, CircuitParameters
c_params) {
```

```
                // ...

                // Parameters for Simulated Annealing
                double Temp = params.initialTemp;

                // Create offspring1 with Simulated Annealing Acceptance
                do {
                   // ...
                if (localNewPopulation.size() < params.populationSize) {
                    do {
                        crossover->setParents(offspring2, selectedParents[parent1_index],
selectedParents[parent2_index]);
                        crossover->apply(generators[omp_get_thread_num()]);

                        mutation->setIndividual(offspring2);
                        mutation->apply(generators[omp_get_thread_num()]);
                    } while (!applySimulatedAnnealing(offspring2,
selectedParents[parent1_index], selectedParents[parent2_index], vector_size, func,
validity, Temp, params, c_params, generators[omp_get_thread_num()]));

                    localNewPopulation.push_back(offspring2);
                }
            }
    // ...
}
```

In summary, simulated annealing is an essential component of the genetic algorithm in this project, allowing the algorithm to balance between exploration and exploitation by accepting new individuals based on a temperature-controlled probability. This helps the algorithm to escape local optima and find better solutions over time.

6. Progress Tracking with tqdm

In our genetic algorithm implementation, we use the self-implemented `tqdm.hpp` to provide real-time progress tracking and visualization of the optimization process. This helps developers and users monitor the progress of the algorithm, gain insights into the computational workload, and estimate the time remaining for completion.

Integration of tqdm in Genetic Algorithm

Tqdm is integrated into the main loop of the genetic algorithm to track and display the progress of each generation. This provides a dynamic and user-friendly way to observe the algorithm's performance.

1. **Initialization of tqdm**

We initialize tqdm to set up the progress bar before entering the main loop of the genetic algorithm. The progress bar is configured to track the number of generations.

2. **Progress Update**

Within the main loop, tqdm updates the progress bar at each iteration, showing the current generation number and other relevant metrics, such as the maximum fitness of the population.

Here's the code snippet demonstrating the integration of tqdm:

```
#include "tqdm.hpp"  // Include the tqdm library

// Main function to optimize the solution using a genetic algorithm
int optimize(int vector_size, int *vector, double (&func)(int, int *, struct
CircuitParameters),
             bool (&validity)(int, int *), Algorithm_Parameters params, CircuitParameters
c_params) {
    // ...
    // Initialize tqdm for progress tracking
    auto iter = tqdm::trange(params.maxGenerations);
    iter.set_prefix("Iterating ");  // Set the prefix for the progress bar
    for (int generation: iter) {
        // ...
        iter << "Max Fitness: " << maxFitness;
        // ...
    }
    // ...
 }
```

Detailed Explanation

- **Initialization**: The `tqdm` progress bar is initialized using `tqdm::trange` with the maximum number of generations as the parameter. The prefix "Iterating " is set to make it clear what the progress bar represents.
- **Progress Update**: Inside the main loop of the genetic algorithm, the progress bar is updated at each generation. The current maximum fitness value is also displayed, providing a real-time metric for the algorithm's performance.
- **Benefits**: This integration provides the following benefits:
    - **Real-time Monitoring**: Users can monitor the progress and performance of the algorithm in real-time.
    - **User-Friendly Visualization**: The progress bar offers a user-friendly way to track the computational process.
    - **Performance Insights**: Displaying metrics like maximum fitness helps in understanding the algorithm's behavior and performance trends.

The use of `tqdm` significantly enhances the usability and transparency of the genetic algorithm, making it easier to track and understand the optimization process.

7. Optimization with OpenMP

In addition to tqdm, the genetic algorithm implementation leverages OpenMP for parallel processing. OpenMP is used to parallelize the evaluation of the population's fitness and other computationally intensive tasks, significantly improving the algorithm's performance on multi-core systems.

Integration of OpenMP in Genetic Algorithm

OpenMP is integrated into the main loop of the genetic algorithm to parallelize the evaluation of fitness and other operations. This provides a substantial performance boost by utilizing multiple CPU cores.

Here's the code snippet demonstrating the integration of OpenMP:

```
#include <omp.h>  // Include the OpenMP library

// Main function to optimize the solution using a genetic algorithm
int optimize(int vector_size, int *vector, double (&func)(int, int *, struct
CircuitParameters),
             bool (&validity)(int, int *), Algorithm_Parameters params, CircuitParameters
c_params) {
    // ...

    // Initialize population in parallel
    #pragma omp parallel for
    for (int i = 0; i < params.populationSize; i++) {
        // ... Population initialization code ...
    }

    // Main loop for generations
    auto iter = tqdm::trange(params.maxGenerations);
    iter.set_prefix("Iterating ");  // Set the prefix for the progress bar
    for (int generation: iter) {
        // Evaluate fitness in parallel
        #pragma omp parallel for
        for (int i = 0; i < population.size(); i++) {
            population[i].fitness = func(vector_size, population[i].genes.data(),
c_params);
        }

        // Selection, crossover, and mutation operations
        // ...
        #pragma omp parallel for
        for (int i = 0; i < params.populationSize; i++) {
            // ... Genetic operations code ...
        }

        // Update progress bar
        iter << "Max Fitness: " << maxFitness;
    }
    // ...
}
```

Detailed Explanation

- **Parallel Initialization**: The population initialization is parallelized using `#pragma omp parallel for`, allowing multiple individuals to be initialized simultaneously.

- **Parallel Fitness Evaluation**: The evaluation of the population's fitness is parallelized, significantly speeding up the process, especially for large populations.
- **Parallel Genetic Operations**: Selection, crossover, and mutation operations are also parallelized to maximize performance.

Benefits of Using OpenMP

- **Improved Performance**: Leveraging multiple CPU cores significantly reduces the time required for fitness evaluation and genetic operations.
- **Scalability**: The algorithm can handle larger populations and more generations within a reasonable time frame.
- **Efficiency**: Parallel processing makes the genetic algorithm more efficient, enabling faster convergence to optimal solutions.

The integration of OpenMP with tqdm provides a powerful combination of performance and usability, making the genetic algorithm both fast and user-friendly.