AI 프로그래밍

Programming Assignment #7

정보컴퓨터공학부

202155546

Calderoni Echeverri Aldo Sigfrido

Implementation and Analysis of Genetic Algorithm for Problem Solving

Introduction:

Genetic Algorithm (GA) is a powerful optimization technique inspired by principles of natural selection and genetics. It is structured and implemented within the MetaHeuristics class to tackle both numerical optimization problems and the Traveling Salesman Problem (TSP) using various parameters and strategies to study efficiency and effectiveness. This report describes the development of GA implemented within the MetaHeuristics class for problem-solving purposes.

Problem Definition:

The implementation of GA aims to solve two types of problems: numerical optimization and the Traveling Salesman Problem (TSP). Numerical optimization, in this case, involves finding the minimum value of mathematical functions, while TSP aims to determine the shortest path visiting each location only once.

Algorithm Design:

Key components of GA within the MetaHeuristics class include:
- Initialization of the population.
- Evaluation and assignment of fitness to individuals.
- Parent selection using binary tournament selection.
- Cross-over and mutation operations.
- Elitism for maintaining the best individuals.
- Iterative execution across multiple generations.

Algorithm Implementation:

The Genetic Algorithm (GA) follows a structured process for exploring and utilizing potential solutions within the solution space.

1. Initialization: Creating a population of individuals

The algorithm generates an initial population of individuals, each representing a potential solution to the given problem. The size of the population is a crucial parameter determining diversity and exploratory capabilities. A larger population can help us obtain a better exploration, but it might increase computational complexity.

2. Evaluation: Assessing individual fitness

Each individual in the population undergoes evaluation using a specific evaluation function that is tailored to the problem domain. This function measures the fitness or suitability of individual solutions concerning optimization criteria. The evaluation process helps distinguish promising solutions from less suitable ones.

3. Selection: Applying binary tournament selection

The selection phase determines individuals to serve as parents for the next generation. Binary tournament selection, widely used in GAs, randomly selects two individuals from the population and compares their fitness. The fitter individual becomes a parent. This competitive process prefers good solutions while maintaining diversity within the population.

4. Crossover and Mutation: Genetic operations for offspring generation

After selecting parents, genetic operations like crossover and mutation are applied to produce new individuals or offspring. Crossover combines genetic information from two parents to create a new solution. Mutation introduces random changes to an individual's genetic information, increasing diversity and preventing premature convergence.

5. Elitism: Preserving high-quality solutions

The concept of elitism retains some of the best individuals from the previous generation, and that is why I decided to include this type of selection algorithm in my GA class. These elite individuals pass on their genetic information to the offspring generation, maintaining superior traits discovered in previous iterations. This mechanism sustains progress by maintaining superior traits found in previous iterations.

6. Iteration: Evolution across multiple generations

The GA operates through multiple generations or iterations. Each iteration involves steps such as the ones mentioned previously. The algorithm iterates to improve the overall population, preferring individuals with higher fitness, mimicking the natural selection process in evolution.

Analysis and Insights: Deep Understanding of the Algorithm

The effectiveness of the GA is influenced by various factors and parameters like population size, selection mechanisms, crossover and mutation rates, and the degree of elitism. Each component plays a critical role in shaping the algorithm's exploration, exploitation, and convergence characteristics.

The interaction between exploration (maintaining diversity) and exploitation (focusing on promising regions) is crucial. A balanced approach allows the algorithm to efficiently explore the solution space while exploiting promising areas. This harmony enables the algorithm to effectively navigate the solution space, striking a balance between exploration and exploitation, ultimately leading to discovering high-quality solutions.

Methods Used in the GA Class:

```
def evalAndFindBest(self, pop, p):
    bestInd = None
    bestFitness = float('inf')

    for ind in pop:
        p.evalInd(ind)
        fitness = ind[0]
        if fitness < bestFitness:
            bestFitness = fitness
            bestInd = ind

    return bestInd
```

evalAndFindBest(self, pop, p):

- Evaluates individuals within the population pop using the evalInd() method of the problem class p.
- Identifies and returns the best individual in the population based on fitness evaluations.

```
def selectParents(self, pop):
    parent1 = self.binaryTournament(pop)
    parent2 = self.binaryTournament(pop)
    return parent1, parent2
```

selectParents(self, pop):

- Selects two parents using the binaryTournament() method and returns them.

```
def selectTwo(self, pop):
    return random.choice(pop), random.choice(pop)
```

selectTwo(self, pop):

- Randomly selects two individuals from the given population pop.

```python
def binaryTournament(self, pop):
    # Select the winner between two individuals
    ind1, ind2 = self.selectTwo(pop)
    if ind1[0] < ind2[0]:  # Assuming fitness is stored at index 0 of an individual
        return ind1
    else:
        return ind2
```

binaryTournament(self, pop):

- Conducts a binary tournament selection among two randomly chosen individuals.
- Determines the winner based on their fitness, which is stored at index 0 of an individual.

```python
def selectNewPop(self, oldPop, p):
    elitism_rate = 0.05
    num_elites = int(elitism_rate * len(oldPop))

    if((num_elites % 2 != 0 and len(oldPop) % 2 == 0) or (num_elites % 2 == 0 and len(oldPop) % 2 != 0)):
        num_elites -= 1

    oldPop.sort(key=lambda x: x[0])
    elites = oldPop[:num_elites]
    newPop = elites[:]

    while len(newPop) < len(oldPop):
        parent1, parent2 = self.selectParents(oldPop)
        child1, child2 = p.crossover(parent1, parent2, self._pC)
        child1 = p.mutation(child1, self._pM)
        child2 = p.mutation(child2, self._pM)
        newPop.extend([child1, child2])

    newPop.extend(elites)
    return newPop
```

selectNewPop(self, oldPop, p):

- Implements elitism by preserving a certain percentage of the best individuals (elites) from the previous population.
- Creates a new population by applying crossover, mutation, and elitism strategies based on the problem class p.

```python
def run(self, p):
    num_generations = 50
    pop = p.initializePop(self._popSize)
    bestInd = self.evalAndFindBest(pop, p)
    f = open('genetic_algorithm.txt', 'w')

    for generation in range(num_generations):
        pop = self.selectNewPop(pop, p)
        bestInd = self.evalAndFindBest(pop, p)
        valueC = bestInd[0]
        f.write(str(round(valueC, 1)) + '\n')

    bestSolution = p.indToSol(bestInd)
    bestFitness = bestInd[0]
    f.close()
    p.storeResult(bestSolution, bestFitness)
```

run(self, p):

- Runs the Genetic Algorithm for a fixed number of generations (num_generations).
- Initializes the population using the initializePop() method of the problem class p.
- Iteratively selects a new population, evaluates fitness, writes the best fitness value to a file, and stores the best solution and fitness using the storeResult() method of the problem class p.

Methodology and Experiments:

The GA is tested with various parameters and settings to analyze its performance:
- Adjustments in population size, cross-over rate, mutation rate, and elitism rate.
- Evaluation of convergence behavior, computational efficiency, and solution quality.
- Experiments involve solving instances of numerical optimization problems and TSP of various complexities.

Results and Analysis:

The results demonstrate the algorithm's capability to find approximate optimal solutions within a reasonable number of generations. The analysis focuses on the impact of parameters on convergence speed and solution quality. Elitism proves effective in preserving high-quality solutions, and it also improves the quality of the genetic algorithm when it is applied to TSP problems.

Conclusion:

The GA implementation within the MetaHeuristics class shows flexibility in solving several types of optimization problems, such as the Ackley and Griewank functions. It demonstrates effectiveness in finding solutions for numerical optimization and TSP instances. However, further experimentation and parameter tuning could enhance its performance in solving complex problems.