AIAB lab report

CandNo: 252777

May 23, 2023

Abstract

This report compares the performance of a Genetic Algorithm (GA) with a hillclimbing population in solving optimisation issues. The purpose of this research is to compare the performance of these two search algorithms in terms of convergence rate, solution quality, and computational efficiency. The findings shed light on the advantages and disadvantages of each strategy, assisting in the selection of the best algorithm for various problem domains.

1 Introduction

Genetic Algorithms (GAs) and hillclimbers are two popular optimisation techniques with distinct search algorithms. GAs employ ideas derived from natural evolution, whereas hillclimbers concentrate on local search. The purpose of this research is to compare the performance of these two algorithms in solving optimisation issues and to comprehend the trade-offs between exploration and exploitation.

2 Methodology

2.1 Problem Definition

The Knapsack Problem is a classic combinatorial optimisation problem in which a subset of items with the highest value must be chosen while remaining within the limits of a given capacity. The name comes from the analogy of packing a rucksack with goods of varying weights and values, with the purpose of maximising the total value of the items carried.

2.2 Genetic Algorithm Configuration:

2.2.1 Population Initialization:

```
def initialise_pop(self):
    pop = np.random.choice([0, 1], (self.pop_size, self.num_items))
    return np.squeeze(pop)}
```

this code generates a random initial population for a genetic algorithm by creating a 2D array of binary strings, where each binary string represents a candidate solution. The dimensions of the array are determined by the population size (self.popsize) and the number of items or genes (self.numitems). The function returns the generated population as the output.

2.2.2 Selection Mechanism:

```
def select(self):
    selected = random.sample(self.population, self.tournament_size)
    return max(selected, key=self.fitness)
```

this code implements tournament selection within a genetic algorithm. It randomly selects a subset of individuals from the population, evaluates their fitness using the self.fitness function, and returns the individual with the highest fitness as the winner of the tournament. This winner will be used for further genetic operations, such as crossover and mutation, in the evolutionary process of the genetic algorithm.

2.2.3 Crossover Operator:

```
def crossover(self, parent1, parent2):
    if random.random() < self.crossover_rate:
        point = random.randint(0, self.num_items)
        return parent1[:point] + parent2[point:], parent2[:point] + parent1[point:]
    else:
        return parent1, parent2
```

this code performs the crossover operation between two parent individuals based on a given crossover rate. It randomly selects a crossover point, swaps genetic information from the parents at that point, and generates two offspring individuals. If the randomly generated value is not less than the crossover rate, no crossover is performed, and the original parent individuals are returned.

2.2.4 Mutation Operator:

```
def mutate(self, parent):
    return [gene if random.random() > self.mutation_rate else 1 - gene for gene in parent]
```

the mutate method takes a parent gene sequence as input and generates a new sequence by randomly flipping some of the genes based on a mutation rate. Genes are flipped by subtracting them from 1 if a randomly generated number is less than or equal to the mutation rate.

2.2.5 Fitness Operator:

```
def fitness(self, parent):
    return sum(parent)
```

the "fitness" method calculates the fitness value of a gene sequence by summing up all the elements in the sequence. The fitness value represents some measure of the gene sequence's quality or suitability, with higher values indicating better fitness.

2.2.6 Running the GA:

```
def run(self, generations):
    for _ in range(generations):
        parent1 = self.select()
        parent2 = self.select()
        child1, child2 = self.crossover(parent1, parent2)
        self.population.append(self.mutate(child1))
        self.population.append(self.mutate(child2))
        self.population.sort(key=self.fitness, reverse=True)
        self.population = self.population[:self.tournament_size]
        self.best_fitnesses.append(self.fitness(self.population[0]))
```

the run method runs a genetic algorithm for a specified number of generations. In each generation, it selects two parent gene sequences, performs crossover to create two child gene sequences, mutates the child sequences, updates the population list, sorts it based on fitness, selects the top gene sequences, and stores the fitness of the best gene sequence for each generation in the best fitnesses list. The specifics of the selection, crossover, and mutation operations are not provided in the code snippet and may be defined elsewhere in the class.

2.3 HillClimbers Configuration:

2.3.1 Population Initialization:

```
    def initialise_pop(self):
            parents = []
            for _ in range(self.population):
                parent = []
                for _ in range(self.num_items):
                    k = random.randint(0, 1)
                    parent.append(k)
                parents.append(parent)
            return parents
```

the method generates an initial population of gene sequences. It creates self.population number of gene sequences, each of length self.num items. For each gene sequence, it iterates over the length and randomly assigns a value of either 0 or 1 to each gene. The completed gene sequences are stored in a list called "parents", which is then returned as the initial population.

2.3.2 Evaluation:

Evaluating the quality or fitness of the current candidate solution using an objective function or evaluation metric specific to the problem being optimized. The objective function assesses how effectively the solution meets the limitations of the problem or achieves the desired goals.

2.3.3 Neighborhood Exploration:

Creating neighbouring solutions by making minor changes to the current solution. Local perturbations, such as flipping a bit or changing a value within a predetermined neighbourhood, can be included among these changes.

2.3.4 Selection of Improved Solution:

Examine the alternatives and choose the one that improves the objective function or fitness value the greatest. The present solution and its neighbours' fitness values are compared in this step.

2.3.5 Iteration:

Replacing the current solution with the selected improved solution from the previous step and repeat steps 2 to 4. Iterate until a termination condition, such as a local optimum, a maximum number of iterations, or a preset runtime, is fulfilled.

3 Results & Discussion

-Hillclimber: The Hillclimber method operates on a single solution at a time. It iteratively improves the current solution until convergence or termination criteria are met.

-GA: Genetic Algorithms maintain a population of solutions and evolve them over generations. They utilize genetic operators like crossover and mutation to explore and exploit the population, facilitating better search and adaptation.

4 Conclusions

The Hillclimber method is a simple and efficient algorithm for local optimization problems, but it may struggle with exploration and getting stuck in local optima. Genetic Algorithms provide a more robust approach for global optimization by balancing exploration and exploitation, handling constraints, and maintaining a diverse population. However, GAs typically require more computational resources and have a slower convergence rate compared to the Hillclimber method. The choice between the two methods depends on the problem characteristics, the trade-off between exploration and exploitation, and the desired solution quality.