The Set Covering 📔 Problem Using Genetic Algorithms

Sidharrth Nagappan, 2022

Background

In this notebook, we will take a GA approach to solving the set-covering problem. As a background, let's assume we have 500 potential lists that should form a complete subset.

The final product should be a list of 0s and 1s that indicate which lists should be included in the final set. We use a genetic approach to obtain this list via:

1. Mutation: randomly change a 0 to a 1 or vice versa
2. Crossover: randomly select a point in the list and swap the values after that point

Representing the Problem

We will represent the problem as a list of 0s and 1s. The length of the list will be the number of lists we have. The 0s and 1s will indicate whether or not the list should be included in the final set.

The objective of the algorithm is to find an optimal (or at least as optimal as possible) set of 0s and 1s that will cover all the elements in the list.

Assessing Fitness

Based on knowledge obtained in previous labs, the heuristic function evolved and these were the factors I considered:

1. Potential duplicates
2. Number of undiscovered elements
3. Length of subset

The following equations were formulated:

1. `len(distinct_elements)`
2. `len(distinct_elements) / (num_duplicates + 1)`
3. `len(distinct_elements) / (num_duplicates + 1) - num_undiscovered_elements`
4. `len(distinct_elements) / (num_undiscovered_elements + 1)`

After multiple trials, the best fitness function is the simplest, which is simply the number of distinct elements.

Results

The following are the results of the algorithm after 1000 generations (only the best results are reported):

| N | W | | _ | _ | | 5 | . | | 10 | 10 | | 20 | 24 | | 50 | 100 | | 500 | 1639 | | 1000 | 3624 |

One thing to note is that lower values of N require a smaller value of N requires larger population values. Whereas, with larger values of N, a smaller population and offspring size is sufficient. Early

stopping is used to detect the plateau, so the algorithm doesn't run endlessly. However, the minima is often reached in less than 100 generations.

Potential Types of Mutations

Flip Mutation

This mutation randomly selects a point in the list and flips the value at that point.

```
def flip_mutation(genome, mutate_only_one_element=False):
    '''
    Flips random bit(s) in the genome.
    Parameters:
    mutate_only_one_element: If True, only one bit is flipped.
    '''
    modified_genome = genome.copy()
    if mutate_only_one_element:
        # flip a random bit
        index = random.randint(0, len(modified_genome) - 1)
        modified_genome[index] = 1 - modified_genome[index]
    else:
        # flip a random number of bits
        num_to_flip = choose_mutation_rate(fitness_log) *
len(modified_genome)
        to_flip = random.sample(range(len(modified_genome)),
int(num_to_flip))
        # to_flip = random.sample(range(len(modified_genome)),
random.randint(0, len(modified_genome)))
        modified_genome = [1 - modified_genome[i] if i in to_flip else
modified_genome[i] for i in range(len(modified_genome))]

    # mutate only if it brings some benefit to the weight
    # if calculate_weight(modified_genome) < calculate_weight(genome):
    #     return modified_genome

    return return_best_genome(modified_genome, genome)
```

Scramble Mutation

Randomly scrambles elements between 2 points in the list.

```
def scramble_mutation(genome):
    '''
    Randomly scrambles the genome.
    '''
    # select start and end indices to scramble
    modified_genome = genome.copy()
    start = random.randint(0, len(modified_genome) - 1)
    end = random.randint(start, len(modified_genome) - 1)
    # scramble the elements
```

```
    modified_genome[start:end] = random.sample(modified_genome[start:end],
len(modified_genome[start:end]))
    return return_best_genome(modified_genome, genome)
```

Swap Mutation

Randomly swaps 2 elements in the list.

```
def swap_mutation(genome):
    '''
    Randomly swaps two elements in the genome.
    '''
    modified_genome = genome.copy()
    index1 = random.randint(0, len(modified_genome) - 1)
    index2 = random.randint(0, len(modified_genome) - 1)
    modified_genome[index1], modified_genome[index2] =
modified_genome[index2], modified_genome[index1]
    return return_best_genome(modified_genome, genome)
```

Inversion Mutation

Randomly inverts elements between 2 points in the list.

```
def inversion_mutation(genome):
    '''
    Randomly inverts the genome.
    '''
    modified_genome = genome.copy()
    # select start and end indices to invert
    start = random.randint(0, len(modified_genome) - 1)
    end = random.randint(start, len(modified_genome) - 1)
    # invert the elements
    modified_genome = modified_genome[:start] + modified_genome[start:end]
[::-1] + modified_genome[end:]
    return return_best_genome(modified_genome, genome)
```

Potential Types of Selections

The best performing is still a standard tournament but other possibilities were also implemented and tested.

Roulette Wheel Selection

```
def roulette_wheel_selection(population):
    '''
    Selects an individual from the population based on the fitness.
    '''
```

```
    # calculate the total fitness of the population
    total_fitness = sum([individual.fitness[0] for individual in
population])
    # select a random number between 0 and the total fitness
    random_number = random.uniform(0, total_fitness)
    # select the individual based on the random number
    current_fitness = 0
    for individual in population:
        current_fitness += individual.fitness[0]
        if current_fitness > random_number:
            return individual
```

Rank Selection

```
def rank_selection(population):
    '''
    Select using Rank Selection.
    '''
    # sort the population based on the fitness
    population.sort(key=lambda x: x.fitness[0], reverse=True)
    # calculate the total rank
    total_rank = sum([i for i in range(len(population))])
    # select a random number between 0 and the total rank
    random_number = random.uniform(0, total_rank)
    # select the individual based on the random number
    current_rank = 0
    for i, individual in enumerate(population):
        current_rank += i
        if current_rank > random_number:
            return individual
```

Vanilla Tournament

Randomly select k elements and return the fittest one.

What's Next?

Improve plateau detection and randomised mutation thresholds.