Assignment 3.

Metaheuristics

José Manuel Izquierdo Ramírez

Contents

List of Figures

1 Introduction

In this practice we have been asked to implement an algorithm of our choice to solve the problem given.

2 Problem approach

With a given dataset we have to implement an algorithm that finds patterns that repeat throughout it.

2.1 Decide on the encoding

At first I though of using a tree-based encoding, but after some deliberation I decided to use an alphabetic encoding.

```
solution = ['R', 'D', 'H', 'R', 'B', 'F', 'A']

solution = ['R', 'D', ['H', 'J'], 'R', 'B', 'F']
```

When creating random solutions I allow that two events can happen at the same time with an arbitrarily set probability.

```
...
for i in range(nSolutions):
solutions = []

#Probability of having more than one event at the same time

prob_occur_same_time = 0.9

it = 0

#Generate solutions between the min and max length randomly
while it < random.randint(min_length,max_length):

    #It generates solutions with events that occur at the
    same time with a probability of 90%.

    if random.randint(1,100) <= (prob_occur_same_time*100):

        ...
    else:

        ...
    it += 1

population.append([solutions,evaluateSolution(solutions,
                    events, list_of_dictionaries,
                        len(solutions) )])
```

2.2 Decide on the heuristic

To start with I really liked the idea of using artificial ant colony algorithm to solve it, but as much as I thought about it I didn't see that it fit completely with the problem we are handling.

Finally I decided to implement a genetic algorithm; having done it in practice 2 I thought of adding something more, that is why I have made a few implementations that I will detail later on.

2.3 How to evaluate the solutions?

In my opinion the evaluation function is one of the most important parts of a genetic algorithm, because if it is not accurate the population will not evolve correctly, will not converge, and will make it difficult to reach a global or even a local maximum.

To do this I start by using an auxiliary function *readFile()*, which returns a list of dictionaries, where each dictionary has as key the events and as value a list of the times in which the event has happened.

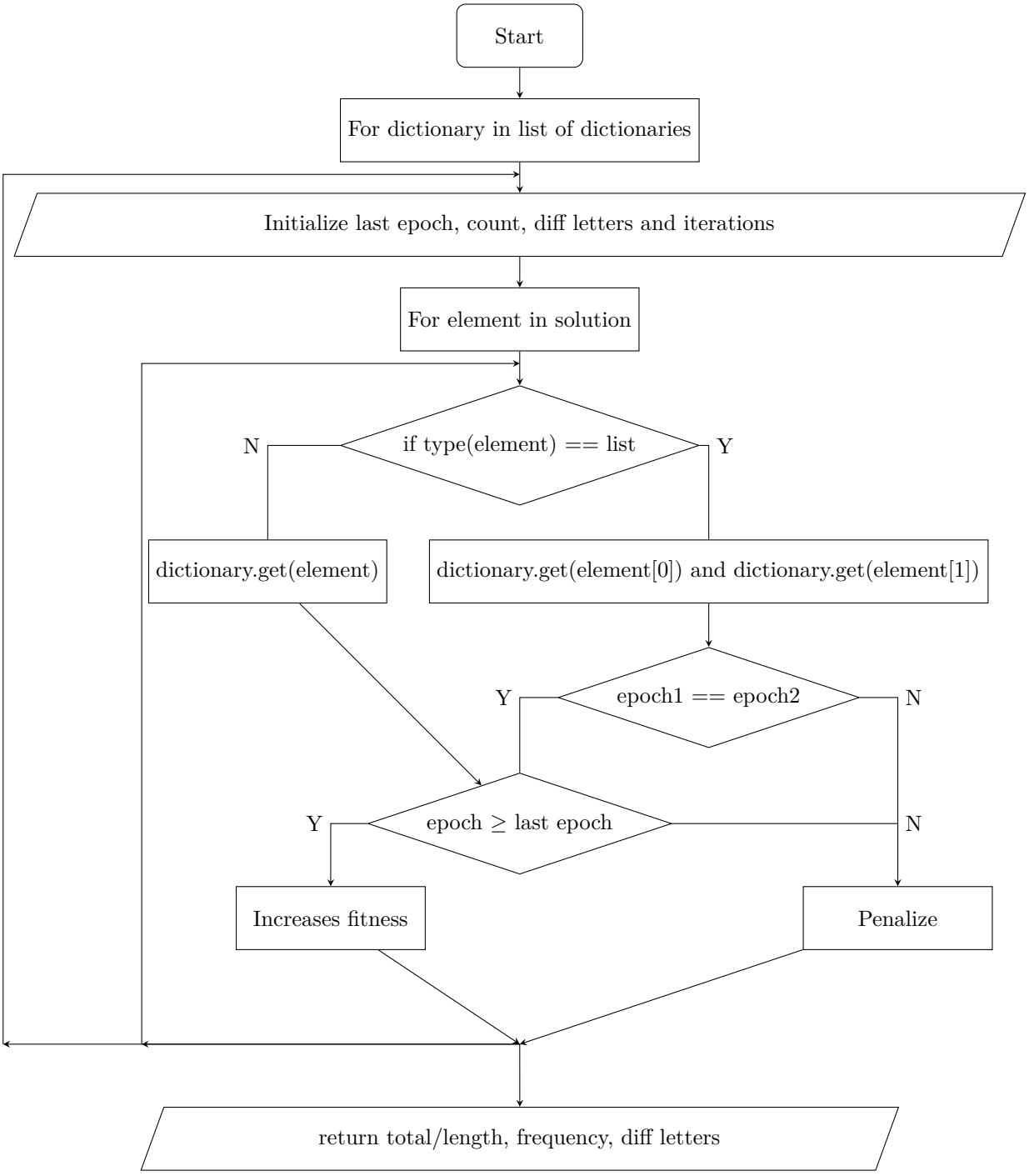

Figure 1: Class Diagram of Evaluation Function

3 Implementation

I started by doing a basic genetic algorithm, but as I said before I wanted to do something different from the implementation done in assignment 2.

One of the biggest problems of this algorithm is the computational time needed by the evaluation function, so reducing the number of calls it recieves is very worth it. I tried to parallelise the evaluation of the solutions, but the result I got was a longer evaluation time.

Also to have more intensification I thought on applying Simulated Annealing to change the degree to which worse solutions are accepted or, what finally I applyed, Hill Climbing to search the best neighbour of good enough solutions.

3.1 Parallelism

This was my first time working with python threads, and I think this was one of the reasons why the result is so bad.

I had thought about testing this implementation but the time it takes makes it prohibitive, we would be talking about hours.

```
def geneticAlgorithmMultiprocessing(nSolutions,
    maxGenerations,mProb,cProb,k,elitism,
            min_length, max_length):

    ...
    while it < maxGenerations:

        nSolutions = applyGeneticOperator(population,k,
                                          cProb,mProb,
                                              events)

        #Generational model
        population = []

        process = []
        for solution in nSolutions:

            # initialize total and frequency
            total = multiprocessing.Value('d',0.0)
            frequency = multiprocessing.Value('d',0.0)
            diff_letters = multiprocessing.Value('d',0.0)
            # creating a lock object
            #lock = multiprocessing.Lock()

            # creating new processes
            p = multiprocessing.Process(
                target=evaluateSolution_thread,
                    args=(solution[0],events,
                        list_of_dictionaries,len(solution[0]),
                        total, frequency, diff_letters))

            process.append(p)
```

```
            # starting processes
            p.start()

            # wait until processes are finished
            for p in process:
                p.join()

            population.append([solution[0], (total.value,
                frequency.value, diff_letters.value)])

        it+=1
    ...
```

3.2 Applying Caching

By applying caching we can see how the time and computational cost is practically halved.

```
    def geneticAlgorithmMultiprocessing(nSolutions,
        maxGenerations,mProb,cProb,k,elitism, min_length,
            max_length):

        ...
        while it < maxGenerations:

            nSolutions = applyGeneticOperator(population,k,
                cProb,mProb,events)

            #Generational model
            population = []

            process = []
            for solution in nSolutions:

                if solution in evaluated_solutions:
                    index =
                        evaluated_solutions.index(solution)
                    population.append(
                        evaluated_solutions[int(index)])

                else:
                    population.append(
                        [solution[0],evaluateSolution(
                            solution[0],events,
                                list_of_dictionaries,
                                    len(solution[0]) )])

        it+=1
        ...
```

3.3 Applying Hill Climbing

Implementing hill climbing has pros and cons. On one hand, the population converges much earlier, resulting in better results in fewer generations. On the other hand, the number of evaluations the algorithm performs while searching for the best neighbour makes it slower.

```
def getBestNeighbor(solution, data, events,
    list_of_dictionaries):

    ##Get the neighbors
    neighbors = []
    l=len(solution[0])

    for i in range(l):
        for j in range(i+1, l):
            n = solution[0].copy()
            n[i] = solution[0][j]
            n[j] = solution[0][i]
            neighbors.append(n)

    ##Get the best neighbor
    bestNeighbor = neighbors[0]
    ##evaluateSolution(solutions,events,
        list_of_dictionaries,
            len(solutions) )

    bestTotal,bestFrequency,bestNEvents =
        evaluateSolution(bestNeighbor, events,
         list_of_dictionaries, len(bestNeighbor))

    for neighbor in neighbors:

        Total,Frequency,NEvents =
            evaluateSolution(neighbor, events,
                list_of_dictionaries, len(bestNeighbor))

        if Total > bestTotal:
            bestTotal = Total
            bestFrequency = Frequency
            bestNEvents = NEvents

            bestNeighbor = neighbor

    return bestNeighbor,
        (bestTotal, bestFrequency, bestNEvents)

def hillClimbing(data, events, list_of_dictionaries):

    solution = []
```

```
        solution.append(data[0][0])

        total = data[0][1][0]

        neighbour = getBestNeighbor(solution, data,
            events, list_of_dictionaries)

        while neighbour[1][0] > total:
            solution = neighbour
            total = neighbour[1][0]
            neighbour = getBestNeighbor(solution, data,
                events, list_of_dictionaries)

        return solution[0]

    def geneticAlgorithmMultiprocessing(nSolutions,
        maxGenerations,mProb,cProb,k,elitism,
            min_length, max_length):

            ...
            while it < maxGenerations:

                ...

                population.sort(reverse=True,key=lambda
                    population: (population[1][0],
                        population[1][2]) )

                if population[0][1][0] > 60:
                    bestNeighbour = list(hillClimbing(
                        population, events,
                            list_of_dictionaries))

                    population.pop()

                    population.append(
                        [bestNeighbour,evaluateSolution(
                            bestNeighbour,events,
                                list_of_dictionaries,
                                    len(bestNeighbour) )])

        it+=1
        ...
```

4 Evaluation

To compare the algorithms I used the google colab environment and the following hyperparameters:

- Size of the population = 50
- Maximum generations = 150[1] for [7−9], 200 for [10−14], 400 for [15−30]
- Mutation probability = 0.4
- Crossover probability = 0.8
- Tournament size = 3
- Elitism = True

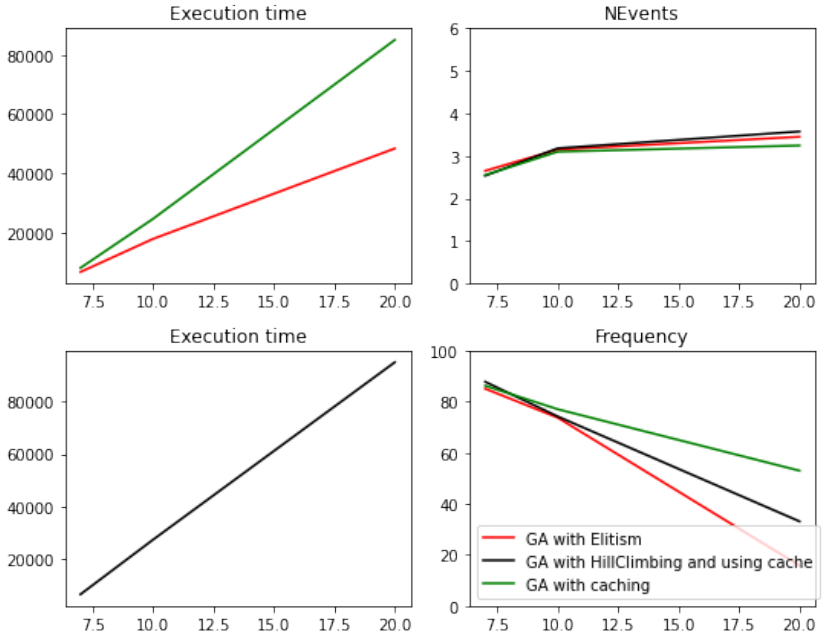

Figure 2: Comparison of different implementations

4.1 Execution time

I must have made some mistake that I can't find in the measurement of the data and its presentation in the graph because in other tests that I have carried out you can see how the one using hill climbing is much slower, during the execution this can be seen thanks to Google Colab in its lower part shows which function is running and the results are that it takes an hour running the repetitions of the algorithm, while the other two only take twenty minutes.

Another thing that does not make sense is that the GA with caching is slower than GA with elitism, I don't know if this is because the randomly generated solutions at the beginning are

[1]Except for the implementation using parallelism that I used 100 because of the time complexity.

worse in the case of implementation using cache or that caching has no effect because solutions are not repeated, in which case the execution time should be the same and not vary so much.

4.2 Number of Events

In the case of the number of events nothing remarkable happens, only that the greater the length of the solution there is a greater probability of obtaining results with a greater number of events.

4.3 Frequency

It makes sense that the worst one is the basic genetic algorithm. The implementation using Hill Climbing is better because when the solution already has a moderately good fitness we look for its best neighbor and add it to the population, which generates a population with very good solutions. What has surprised me is that using caching improves the frequency so much, because the only thing I am doing is to avoid re-evaluating previously evaluated solutions, which has an improvement in time, but I do not quite understand why in the quality of the solutions as well.

5 Conclusions

In my opinion and from the tests I have carried out I think that for this problem it is very beneficial to use an average mutation probability, because the space to explore is very large, we would be talking about N^{20} where N is the length of the solution, and the probability of being trapped in a local maximum is very high.

For the evaluation section I would have liked to carry out a more exhaustive analysis with a larger number of repetitions, not only twenty, and a larger number of generations for the intervals $[10-14]$ and $[15-30]$, but the time needed to carry out this task is prohibitive, just as it has been impossible to carry out the measurements for the implementation that uses parallelism, doing some calculations to carry out the analysis I would like, would entail a time cost of 155 hours.

Increasing the number of generations for the above-mentioned intervals would significantly increase the quality of the population and therefore the frequency.