T3 - A Study on the Performance of Both Genetic Algorithms and Heuristics in Solving the Traveling Salesman Problem

Daniș Ciprian
Oloieri Alexandru
Second Year, A2

December 6, 2020

1 Introduction

The **travelling salesman problem**[1] (also called the **travelling salesperson problem** or **TSP**) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?" It is an **NP-hard** problem in combinatorial optimization, important in operations research and theoretical computer science.

In this study we aim to analyze the performance of both a Genetic Algorithm and a heuristic in solving several instances of TSP, compare the results of the two methods and draw an inference as to which algorithm yields optimal values for a certain instance.

2 Method

2.1 Simulated Annealing

We will use a version of the algorithm in which, at each step, we choose as the next candidate the first neighbor that meets the search criteria (Simulated Annealing First Improvement).

```
begin
        candidate := generate_candidate()
        /*generate a random permutation of the numbers 0,1,...,numberOfCities-1*/
        t := 1000
        bestValue = Euclidian_2D(currentInstance, candidate)
        while (true) do begin
                foundNewCandidate := false
                for (i:=0; i<numberOfCities-1; ++i) do
                        for (j:=i+1; j<numberOfCities;++j) do
                                swap(candidate[i], candidate[j])
                                /*"candidate" stores a neighbor of the current candidate*/
                                currentValue := Euclidian_2D(currentInstance, candidate)
                                if currentValue < bestValue
                                || random(0,1) < exp(-abs(currentValue-bestValue)/t) then
                                        bestValue := currentValue
                                        foundNewCandidate := true
                                        break
                                swap(candidate[i], candidate[j])
                        if foundNewCandidate then
                                break
                t := 0.9*t;
                if foundNewCandidate = false then
                        break
        return bestValue
end
```

Legend:
candidate - current candidate (a permutation of numbers 0,1,...,numberOfCities)
Euclidian_2D() - function used to determine the "weight" of a candidate for a TSP instance
bestValue - the best weight of a permutation found

2.2 Genetic Algorithm

We will work in the following experiment with an improved Genetic Algorithm that makes use of hypermutation and another heuristic (in our case, Hill Climbing Best Improvement).

```
begin
        init_pop() /*initialize the population*/
        set_mold()/*create a list of all the "cities"*/
        currentGen := 1
        optimal_tour := decodeElem(pop[det_optimal_chromosome()])
        best_sol := Euclidian_2D(currentInstance, optimal_tour)
        counter := 1
        while (currentGen <= number_of_gens) do begin
                hypermutation(counter)
                crossover()
                selection()
                opt_poz := det_optimal_chromosome()
                possible_sol := Euclidian_2D(currentInstance, decodeElem(pop[opt_poz]))
                currentGen++
                if possible_sol < best_sol then
                        best_sol := possible_sol
                        if Euclidian_2D(currentInstance, optimal_tour) > best_sol then
                                optimal_tour := decodeElem(pop[opt_poz])
                        counter := 1
                else counter++
        HillClimbingBI(optimal_tour)
end
```

Legend:
pop - current population/array of solution
decodeElem() - function used to obtain the permutation of the "cities" from a chromosome; see the **Experiment** section for more
Euclidian_2D() - function used to determine the "weight" of a decoded chromosome; see the **Experiment** section for more
best_sol - the current best value of *Euclidian_2D*() in all the population, throughout the generations
possible_sol - a candidate for the title of *best_sol* in a generation
optimal_tour - the best value of the best chromosome throughout all generations
currentInstance - the instance of TSP with which we are currently working
opt_poz - the position of the best chromosome in the population in a generation
det_optimal_chromosome() - function used to determine the position of the best chromosome in a population in a generation
counter - the number of generations throughout which the value of *best_sol* has not changed
HillClimbingBI(*optimal_tour*) - implementation of Hill Climbing Best Improvement[2] in which the starting point, which was generated randomly, is replaced by *optimal_tour*
selection(), *hypermutation*()[3] and *crossover*() - see the **Experiment** section

3 Experiment

3.1 General

We will work with 4 instances of symmetric TSP (precisely with *eil51*[4], *ts225*[4], *krod100*[4] and *st70*[4]), in which the distance from node i to node j is the same as from node j to node i. Moreover, the instances have the edges' weight represented as Euclidian 2D distances. Therefore, in *Euclidian_2D*() we will compute the weight of a circuit of all the "cities".

3.2 Simulated Annealing

The candidates will be represented as arrays of integers, which represent permutations of the order in which the "cities" will be visited (n cities). A neighbor of the current candidate will be a permutation of size n in which $n-2$ elements have the same position, and 2 elements are swapped.

For each map, we will run the algorithm 30 times, and for each run we will have 100 iterations of the Simulated Annealing metaheuristic (so we will get 3000 results).

3.3 Genetic Algorithm

Each chromosome in the population will be represented as a list of positions that the nodes can have in a permutation. More exactly, each element in the chromosome will be an integer in the interval $[0, n-1-i]$, where n represents the number of nodes and i the position of the element in the chromosome.

In *hypermutation*(), *counter* keeps count of how many generations have gone by since the value of *best_sol* has been changed. If the value reaches the threshold then, instead of performing a generic mutation (a mutation with a small rate, e.g. 0.015), we will mutate the population with a rate of 0.5. This way, we "reset" the population so that the processing of the chromosomes does not stagnate around an accumulation point.

In *crossover*(), because of the structure of the chromosomes, which allows for repetitive values, we will perform a generic lossless cut-point crossover (both the "children" and their "parents" will be included in the population, resulting in an "extended" population).

In *selection*(), we will work with the extended population resulted from the execution of *crossover*(). We will firstly determine the determine the "fitness" of each chromosome/member of the population (a real positive number yielded by the expression $1/(0.00001 + Euclidian\_2D(decoded_chromosome))$). Then we will apply elitism on the population (we will ensure that a few best chromosomes appear in the future generation at least once) and finally, with the use of a "roulette wheel", we will select the remaining $n - k$ chromosomes, where n is the initial or standard size of the population and k is the number of chromosomes selected with elitism.

After executing the three operations above for a certain number of generations, we will use the optimal tour found throughout all generations as starting point for the Hill Climbing Best Improvement, which will be used to further improve the result yielded by the algorithm.

For each map, we will run the algorithm 30 times, with the following parameters: a population size of 100, 1500 allowed generation, a hypermutation threshold of 350, a mutation rate of 0.015, a crossover rate of 0.35 and an elitism rate of 0.1.

4 Results

4.1 Tables

Results						
Instance	Optimal Value	Lowest Value Obtained	Highest Value Obtained	Average	σ (standard deviation)	Average Time
eil51	426	452.39	504.52	484.1536	13.24391	2.921166 s
st70	675	857.882	949.251	908.1673	26.68194	8.730105 s
krod100	21294	41337.4	52108.6	47588.5	2501.634	4.066486 s
ts225	126643	543578	655030	621550.5	25043.06	14.2773 s

Results for Simulated Annealing

Results						
Instance	Optimal Value	Lowest Value Obtained	Highest Value Obtained	Average	σ (standard deviation)	Average Time
eil51	426	511.508	675.088	590.3399	43.02564	2.592588 s
st70	675	970.972	1251.44	1097.229	66.90697	3.920526 s
krod100	21294	35051.6	50003.3	41699.61	3799.186	7.01967 s
ts225	126643	273575	356486	319754	23678.86	45.09334 s

Results for Genetic Algorithm

4.2 Interpretation

As we can see in the tables above, when it comes to instances with a number of nodes lower than 100, the Simulated Annealing yields slightly better results than the Genetic Algorithm, in the case of *eil51* coming extremely close to the optimal value. On the hand hand, for instances with 100 or more

nodes, the Genetic Algorithm reaches better values, while the Simulated Annealing cannot even reach an optimal attraction pool. However, the results yielded by the Genetic Algorithm in this case are rather adequate than excellent.

As for execution time, for instances with less than 100 nodes, the Genetic Algorithm performs faster, while for instances with 100 or more nodes, it is the Simulated Annealing that computes the result quicker, in the case of *ts225* the difference between the 2 methods being more than 30 seconds in average.

5 Conclusions

All things considered, we can state that Genetic Algorithm produces good results for an **NP**-hard problem and, due to the short execution time, it is usable in practice. Moreover, because it is more complex than other heuristics, it has a higher number of parameters which can be modified, this giving us the possibility to change their value until we find the most suitable ones.

References

[1] Traveling Salesman Problem - From Wikipedia, the free encyclopedia – `https://en.wikipedia.org/wiki/Travelling_salesman_problem`

[2] Algoritmi Genetici - Metode traiectorie – `https://profs.info.uaic.ro/~pmihaela/GA/laborator2.html`

[3] Suvarna Patil, Manisha Bhende, *Comparison and Analysis of Different Mutation Strategies to improve the Performance of Genetic Algorithm* – `https://pdfs.semanticscholar.org/75eb/30781a18a63a89b09af6cea9c388accac0d4.pdf`

[4] MP-TESTDATA - The TSPLIB Symmetric Traveling Salesman Problem Instances – `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html`