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** (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

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.

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;++i) do
                        otherPos = random() % numberOfCities;
                        swap(candidate[i], candidate[otherPos]);
                        /*now \"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[otherPos]);
                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 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[1] in which the starting point, which was generated randomly, is replaced by *optimal_tour*
selection(), *hypermutation*() and *crossover*() - see the **Experiment** section

3 Experiment

3.1 General

We will work with instances of TSP that have the edges' weight represented as Euclidian 2D distances (precisely with *eil51*[2], *gr202*[2], *krod100*[2], *st70*[2] and *ulysses22*[2]). Therefore, in *Euclidian_2D*() we will compute the weight of a circuit of all the "cities".

3.2 Simulated Annealing

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

In each iteration, we will work with a population of . The stopping condition will be that the current generation does not have its index number greater than the number of allowed generations.

Regarding the structure of the Genetic Algorithm, we will refer to 3 important elements: *mutation*(), *crossover*() and *selection*(). The first 2 are "genetic operators" and are responsible for the changes that the population undergoes in a certain generation. The last one is tasked with selecting, based on a *fitness* function, which candidates are fit to be taken by the next generation.

In *mutation*(), we will assign for each bit in the population a real number between 0 and 1 representing the mutation probability. If the number of the current bit is lower than 0.01, then we will "mutate" that bit by negating it.

In *crossover*(), we will assign to each member of the population a random real number between 0 and 1 representing the crossover probability. We will then sort the members in ascending order based on the numbers assigned and those whose number is below 0.3 will be considered fit for crossover. If the number of the crossover candidates is even, we will simply group the candidates in pairs (the first 2 in a group, the following 2 in another group and so on), then generate a random number between 0 and $n \cdot 32 - 2$. That number will be the "crossover point", and all the bits between it and $n \cdot 32 - 1$ will be swapped between the members of a pair. If the number of crossover candidates is instead odd, we will generate a random boolean variable in a Bernoulli distribution. If the variable yields true, then we will pair the last crossover candidate with the first population member with the crossover probability ≥ 0.3. Otherwise, we will ignore the last candidate. Nonetheless, we will then proceed to execute the rest of the steps as for when the number of candidates is even. In all scenarios, we will keep both the parents and the children in the population, thus implementing a non-destructive crossover.

In *selection*(), we will firstly determine the "fitness" of each chromosome/member of the population. The "fitness" will be a real positive number yielded by the expression $1.1C - f(chromosome)$, where C is the worst member of the population (the one for which a test function returns the worst result) and $f()$ is the current test function. After determining each member's "fitness", we will then create a "roulette wheel" where each section will be assigned to a chromosome and delimited by $rouletteVal(previous\ chromosome)$ and $fitness(current\ chromosome) + rouletteVal(previous\ chromosome)$ $(= rouletteVal(current\ chromosome))$. Since the population that reaches the selection phase can have a size greater than 100, we will select only the first 100 fit candidates. To select the members of the next generation, we will generate for 100 times a random real number between 0 and 1 in a uniform distribution, then multiply the number by $rouletteVal(last\ chromosome)$. The section of the roulette wheel in which the resulted value falls will give us the chromosome that is fit to appear in the next generation. Thus, we will give a chance even to those chromosomes whose fitness value is extremely little.

Regarding the parameters, the Genetic Algorithm is mostly influenced by the number of allowed generation, the dimension of a chromosome (meaning the dimension of a solution in the population), the size of a population and the starting population. Other significant parameters would be the fitness function, the mutation probability mark and the crossover probability mark (any value assigned to a chromosome that is under these values will deem the chromosome fit for mutation/crossover), but since all these will be considered the same for each test function and are as described above, their influence is negligible. In *hypermutation*()[**?**], we introduce a new parameter, *counter*. *counter* keeps count of how many generations have gone by since the value of *best_sol* has been changed. If the value reaches a given mark (in our case 75) then, instead of performing a generic mutation (a mutation with a small rate, e.g. 0.01), 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. Otherwise, the hypermutation is implemented similarly to the mutation in a basic Genetic Algorithm.

In the case of Hill Climbing, in terms of implementation, apart from the details above and the reference, we will also refer to another written article[**?**] that draws a comparison between the performances of different heuristics. The heuristic will be used to process the best individual out of the entire final population (resulted after altering the final generation), so as to reach even better optimal results.

4 Results

4.1 Tables

4.2 Charts

4.3 Interpretation

5 Conclusions

References

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

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