Advanced Optimization lecture

Université Paris-Saclay – M2 AIC

Exercise: Evolutionary Algorithms for the Traveling Salesperson Problem

Anne Auger, Dimo Brockhoff
firstname.lastname@inria.fr
http://researchers.lille.inria.fr/~brockhof/advancedOptSaclay/

Introduction

For discrete and combinatorial problems, evolutionary algorithms and other randomized search heuristics are often less competitive than problem-specific algorithms (in terms of the quality of the solutions found and the algorithm's convergence speed towards the optimum). However, they are simple enough to implement them quickly. Compared to problem-specific algorithms which need a deep (and time consuming) understanding of the problem characteristics, a simple evolutionary algorithm can already produce reasonably performant solutions in short time (counting both the algorithm development/implementation time and the actual optimization time). Once implemented, an evolutionary algorithm can then also be adapted and enhanced easily with problem-specific components as soon as a further understanding of the problem is available.

The goal of this exercise is to showcase how easy it is to implement an evolutionary algorithm from scratch for the traveling salesperson problem (TSP) and understand its basic working principles and their influence on the algorithm performance. The Traveling Salesperson Problem is thereby asking for the shortest tour through a given set of n cities with their pairwise (symmetric) Euclidean distances.

Problem Formulation

- ⋆ Each city is represented by a unique integer number between 1 and the total number of cities n.
- ⋆ A map of n cities is an $n \times 2$ matrix, indicating the coordinates (x_i, y_i) of each city i $(1 \leq i \leq n)$ as the two column entries:

$$\text{map: } \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \dots & \dots \\ x_n & y_n \end{pmatrix}$$

  For simplicity, we assume all coordinates to be integer.
- ⋆ The distances between the n cities are represented by entries in a symmetric matrix D of size $n \times n$ with 0s on the diagonal:

$$D(i, j) = \text{distance between city } i \text{ and city } j$$

In the following, we will investigate (and implement) the different parts of an evolutionary algorithm (EA) for the above TSP problem step-by-step.

Representation, Search Space, and Objective Function

1. One possible representation for the problem is to have a solution encoded as a permutation of length n. Why are permutations a natural representation for the TSP?
2. How would you represent a permutation in Python, i.e., which data structure do you suggest for storing a permutation in Python?
3. What is the size of the search space for a given set of n cities? In other words how many solutions does an exhaustive search have to touch to find the optimal tour?
4. Write down (on paper / in textual form) the objective function for the TSP, given a specific permutation and a distance matrix.

Creating Problem Instances

Before to start implementing any algorithmic aspect, we first create some (random) instances of the TSP.

5. Write a function in Python that takes as arguments the number n of cities and two integer numbers `xmax` and `ymax`. This function, called `placeCitiesRandomly`, shall return the (integer) x and y coordinates of n randomly placed cities in $[0, x_{\max}] \times [0, y_{\max}]$ as a two-dimensional array.
6. Write another function in Python that takes the above created random locations of n cities and returns the distance matrix for the corresponding TSP instance, i.e., an $n \times n$ matrix D in which entry (i, j) corresponds to the Euclidean distance between cities i and j in the given set.

Implementation of the Objective Function

A second important task to implement before the algorithm is the objective function.

7. Write a function that takes a permutation and a distance matrix as input and returns the length of the represented tour as objective function value.

Graphical Interface

The last issue that needs to be solved before we can start to code the algorithm, is a way to display a solution.

8. Write a function `plotSolution` that takes a permutation and a list of city locations (given by the above `placeCitiesRandomly`) as arguments and which plots the solution, given by the permutation, as a 2-dimensional map, indicating the cities with markers and the chosen path as straight lines.

Initialization

The first part of the actual algorithm for solving the TSP, we will implement, is the initialization of the algorithm's population. Typically, this is done uniformly at random in the search space. Hence, think about how to sample μ permutations randomly from the set of all permutations.

One trick to achieve random permutations has to do with sorting random numbers.

9. Write a function that samples μ permutations uniformly at random from the set of all permutations, given the parameters μ (number of samples) and n (dimension of the problem).

Variation Operators

To generate new solutions from already visited ones, an evolutionary algorithm performs crossover and mutation.

Mutation

10. Implement a mutation operator of your choice, for which you think it is well suited for the TSP. Why do you think so?

Crossover

For recalling the ideas behind crossover operators,
see `http://en.wikipedia.org/wiki/Crossover\_(genetic\_algorithm)`

The main idea behind good crossover operators is to create a new solution from its parents by combining different parts from the parents while keeping the good properties of them.

11. Propose possible crossover operators for the TSP.

Classical crossover operators that we have seen in the lecture like the 1-point crossover will not result in feasible solutions for the representation of permutations for the TSP case. Hence, other operators such as the Partially Mapped Crossover or the order 1 crossover have been developed to keep the order of cities in part from each parent. We detail in the following the working principles of the latter.

Order crossover: The idea behind the order crossover is that we keep the orders of the cities from either the first or the second parent: from one parent, we choose a (random) sub-path and copy it to the child. The newly generated solution will then be filled up by adding the remaining cities in the order, they appear in the other parent. In the following example, we generate two children at once from the following parents:

Parent 1 [1 2 | 3 4 5 | 6 7 8]
Parent 2 [2 4 | 6 8 7 | 5 3 1]

We start by picking a random sub-path in both parents, here the middle three cities (3, 4, 5 from the first parent, and 6, 8, 7 from the second parent:

Child 1 [⋆ ⋆ | 3 4 5 | ⋆ ⋆ ⋆]
Child 2 [⋆ ⋆ | 6 8 7 | ⋆ ⋆ ⋆]

And then we complete both children by adding the remaining cities in the order they appear in the other parent after cities 5 and 7 respectively (without repeating cities of course):

Child 1 [8 7 | 3 4 5 | 1 2 6]
Child 2 [4 5 | 6 8 7 | 1 2 3]

Note again that both the start and end indices of the initial sub-paths are chosen uniformly at random.

12. Write a function for the order 1 crossover, taking as arguments the two parent solutions and returning two children.

Overall Algorithm

We eventually need to combine all the above written parts to obtain our final evolutionary algorithm. See below an example of an “elitist” algorithm with plus selection.

A) Initialize the population uniformly at random with μ solutions.

B) Successively create $\lambda = 2 * \mu$ children by applying the following steps:

- choose uniformly at random (without replacement) two parent solutions from the population and use crossover to create two new children solutions
- apply the mutation operator to both children with a probability of p_m (choose the probability not too high).
- evaluate the solutions

C) select the best μ solutions among the λ children and the μ parents. These μ best solutions become the potential parents of the next iteration. Go to step B).

13. Write a function in Python that runs your evolutionary algorithm. As parameters, it shall take a (random) map of cities and the number of function evaluations ("budget"). The algorithm run should return both the best solution found as well as its length (i.e. its objective function value).

    For investigating how the algorithm's best solution evolves over time, it might also be interesting to keep track of the best ever achieved function value at each iteration.

14. Study the influence of the different parameters of your algorithm. In particular check how important the crossover operator and your chosen mutation operators are on the algorithm performance.